Escuela Politécnica Superior

Generación de jugadores automáticos de Atari 2600 con Machine Learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Guillermo Fernández Vizcaíno

Tutor/es:
Francisco José Gallego Durán

Junio 2016

Universitat d'Alacant
Universidad de Alicante

Preámbulo

¿Quién no ha soñado alguna vez con inventar algún algoritmo capaz de aprender como un cerebro humano y resolver así el problema de la inteligencia? Bueno... igual no es el sueño más común, pero estoy seguro de que mucha gente comparte conmigo esas inquietudes. La inteligencia artificial siempre ha llamado mi atención, pero fue el descubrimiento del *machine learning* lo que de verdad me cautivó. ¿Máquinas capaces de aprender cosas para las que no estuvieron específicamente programadas? ¡Yo quiero!

Como mucha otra gente, descubrí el *machine learning* a través del curso de *Andrew Ng* en *Coursera*. Desde entonces no he podido dejar de estudiarlo en mis ratos libres (que desgraciadamente son pocos durante el curso). Otro de los cursos de *Coursera* en el que he invertido algo de tiempo libre fue el que me dio la idea precursora del tema actual de este TFG. El curso del que os hablo se llama *General Game Playing* y trata la creación de programas capaces de jugar a diferentes juegos de tipo tablero definidos usando una sintaxis específica. La idea inicial de este TFG empezó siendo la construcción de un entorno de *General Game Playing* capaz de resolver varios tipos de juegos. Hacer eso cuando no tienes la experiencia suficiente en este campo es como empezar una casa por el tejado, así que decidí centrarme inicialmente en un juego concreto y a partir de ahí ir progresando pasito a pasito.

Por lo tanto, el objetivo de este TFG es aplicar distintas técnicas de *machine learning* para que el ordenador juegue a un videojuego de *Atari 2600* de forma autónoma. El videojuego elegido es el *Breakout*, cuyo objetivo es romper unos bloques de la pantalla con la ayuda de una raqueta y una pelota.

Este trabajo representa mi inicio en el mundo del *reinforcement learning*, un tipo específico de *machine learning* con el que hasta la fecha de inicio del TFG no había trabajado antes. Con este proyecto pretendo tener esa primera toma de contacto con ese tipo de aprendizaje, conocer sus técnicas más básicas y aplicar algunas de ellas. El punto de vista del aprendizaje por refuerzo me parece realmente desafiante y a la vez muy atractivo. Nosotros como humanos no solemos considerar conocimiento al simple hecho de observar y reconocer cosas (*supervised learning*), hay que actuar. El aprendizaje por refuerzo es el tipo de aprendizaje más relacionado con la forma con la que los humanos interactuamos con el entorno. No obstante, el *supervised learning* será protagonista de una buena parte de este trabajo.

Como se verá en la introducción, la creación de agentes inteligentes para resolver juegos se trata de un campo en el que mucha gente ha trabajado antes y en el que, sin embargo, queda mucho por descubrir. Por lo tanto, la investigación aquí realizada puede ser (y seguro que será :) profundizada en futuros trabajos.

Agradecimientos

No podría iniciarse el trabajo sin antes agradecer a Fran por el equilibrio perfecto que ha conseguido entre dejarme libertad y guiarme.

Es también de agradecer el apoyo recibido por parte de mi familia. En especial a mi madre, por aguantar día tras día mis explicaciones sobre el funcionamiento de los perceptrones, las redes neuronales, SVM...

A mi portátil. Esta pequeña criaturilla electrónica ha sufrido durante muchos años mi deliberada explotación, así que he creído necesaria su mención en este apartado.

También me gustaría agradecer a todos aquellos que han hecho posible que me haya iniciado en el mundo de la informática. Son ya muchos años los que llevo estudiando de forma autodidacta a través de Internet, así que me gustaría aprovechar bien estas líneas agradeciendo a cada una de las personas que han compartido sus conocimientos de forma desinteresada en la red. Les agradezco de todo corazón sus aportes, aportes que entre la gran cantidad de datos que hay en Internet son insignificantes, pero que para mí han marcado la diferencia.

A mi gata Elsa,

por dormir encima del teclado forzándome a descansar la vista de vez en cuando.

No trates a los ordenadores como si fueran personas...

no les gusta.

Yo no, autor desconocido.

Índice general

Índice de figuras **13**

Índice de tablas **17**

Índice de símbolos **19**

1. Introducción **21**

1.1. Estado del arte 23

1.2. Objetivos 25

1.3. Metodología 25

2. The Arcade Learning Environment **27**

2.1. Puesta en marcha 27

2.2. *Breakout* 29

2.3. Jugador humano 31

2.4. Agentes 32

2.4.1. *Noop agent* 32

2.4.2. *Random agent* 33

2.4.3. *If agent* 33

2.5. Conclusiones 36

3. Supervised Learning - Perceptron **39**

3.1. Recolectando datos 40

3.2. Perceptrón 40

3.3. Preprocesando datos 41

3.4. Entrenamientos 42

3.5. Pruebas 43

3.6. Conclusiones 44

4. Reinforcement Learning - *Q-Learning* **47**

4.1. De MDP a RL . . . 48

4.2. *Q-Learning* . . . 50

4.3. *Q-Table agent* . . . 51

4.4. Discretización . . . 51

4.5. Ajustando α y γ . . . 53

4.6. Pruebas . . . 56

4.7. Conclusiones . . . 56

5. Características de entrada **59**

5.1. Pruebas perceptrón . . . 60

5.2. Pruebas *Q-Learning* . . . 60

5.3. Conclusiones . . . 60

6. *KNN* y *Neural Networks* **65**

6.1. Perceptrón . . . 65

6.2. *KNN* . . . 65

6.3. *Neural Networks* . . . 66

6.4. Conclusiones . . . 67

7. Mejoras *Q-Learning* **71**

7.1. ϵ *decay* . . . 71

7.2. Modificaciones en la función de *reward* . . . 72

7.3. *Learning rate decay* . . . 73

7.4. Discretización selectiva . . . 73

7.5. Conclusiones . . . 74

8. Conclusiones **77**

8.1. Futuras investigaciones . . . 77

A. *HappyML* **79**

A.1. Compilación e instalación . . . 79

A.2. *Datasets* 80
A.2.1. `happyplot` 82
A.2.2. `happydatacreator` 82
A.3. Modelos 83
A.4. Modelos lineales 84
A.4.1. Perceptron 84
A.4.2. Regresión lineal 86
A.4.3. Regresión logística 87
A.5. KNN 88
A.6. Redes neuronales 88
A.7. SVM 92
A.8. Reducción de la dimensionalidad 93
A.9. Futuro 94

B. Implementación **95**
B.1. Compilación 95
B.2. El ejecutable *Runner* 96
B.3. *Trainers* 96
B.4. Agentes 97
B.5. Data agent 98
B.6. Q-Table Agent 99

Índice alfabético **100**

Referencias **101**

Índice de figuras

1.1. Videoconsola *Atari 2600* a la izquierda[Wikipedia] y cartuchos de diversos juegos a la derecha[VintageToySeller]. Entre los cartuchos se puede divisar el del *Breakout* arriba a la izquierda y el de su secuela *Super Breakout* en medio a la derecha. 22

1.2. Captura de pantalla del inicio del juego *Breakout*. Todos los bloques se encuentran intactos. . . 22

1.3. Logo del *Arcade Learning Enviroment*. 23

2.1. Inicialización básica de ALE y carga de la ROM. 28

2.2. Definición de ALE de los tipos de acciones disponibles. 29

2.3. Captura del juego Breakout con la información extraída de la RAM. 29

2.4. Código para leer datos de la RAM. 30

2.5. Captura del juego Breakout con la información extraída de la RAM, la extraída gracias al ciclo anterior y una característica creada por nosotros llamada $Diff_x$. Crear otra característica llamada $Diff_y$ carece de sentido debido a que la altura del jugador es siempre la misma. . . . 31

2.6. Esquema de un agente inteligente interactuando con su entorno. 32

2.7. Si no tocamos la pelota, los 5 lanzamientos que se realizan durante un episodio son los indicados con las flechas (siempre son realizados en el mismo orden). 33

2.8. Histograma del *reward* obtenido en los 1000 episodios jugados por el **Random agent**. 33

2.9. Histogramas del *reward* obtenido en los 1000 episodios jugados por el ***If Agent*** con diferentes valores de ϵ. 35

2.10. Histograma del *reward* obtenido en los 1000 episodios jugados por el **If Agent** con **ruido entero** ± 1. 36

2.11. Histogramas del *reward* obtenidos tras 1000 episodios jugados por el **If Agent** con diferentes valores de **ruido real**. 37

2.12. Explicación gráfica de la mayoría de los fallos del *If agent*. Suponiendo que la pelota está cayendo hacia la derecha, en el primer caso el jugador empieza a moverse a la derecha justo cuando la pelota sobrepasa la mitad de la barra. En el segundo caso, cuando $Ball_x > Player_x + 10$, el jugador comienza a moverse a la derecha, pero en ese momento la pelota ya casi le sobrepasa. 38

3.1. Puntuación media obtenida en cada *dataset* por un ***Perceptron agent*** con ruido ± 0.01 (izquierda) y con ϵ-***greedy policy*** con $\epsilon = 0.05$ (derecha). Las barras de error indican las medias máximas y mínimas de los diferentes entrenos. 44

3.2. Pesos de los perceptrones de uno de los entrenos que mejores resultados daban (10/20 con una puntuación media de 16). A la izquierda el perceptrón que predice las acciones izquierda y a la derecha el que predice las de la derecha. 45

4.1. Esquema que representa la diferencia entre un MDP (arriba) y RL (abajo). 49

4.2. Curvas de aprendizaje de un *Q-Table agent* con 4 factores de discretización distintos: 1, 4, 10 y 16. Se muestra una media móvil del *reward* de los primeros 2000 episodios ejecutados con $\epsilon = 0.05$, $\alpha = 0.2$ y $\gamma = 1$. El ancho de la ventana usada para realizar la media móvil es de 200 episodios. La línea roja marca la tendencia. 52

4.3. Puntuación obtenida por un *Q-Learning agent* con diferentes factores de discretización, durante 2000 episodios y con $\epsilon = 0.05$. 53

4.4. Número de entradas de la función $Q(s, a)$ en función del factor de discretización utilizado. . . 53

4.5. Número de acciones realizadas (en millones) durante las 5 repeticiones de las ejecuciones de 2000 episodios. 54

4.6. *Heat map* con una *policy* ϵ-greedy con $\epsilon = 0.05$. 55

4.7. *Heat map* del número de entradas que posee $Q(s, a)$ en función del valor de α y γ. 55

4.8. Curvas de aprendizaje de 3 de las pruebas de la tabla 4.2. En ellas se usa $\alpha = 0.2$, $\gamma = 0.99$ y se varía el factor de discretización. 57

4.9. La mejor *Q-Table* obtenida es la sacada del punto máximo de la curva de aprendizaje de este agente. Que la línea esté en 30 no significa que ese vaya a ser su *reward* medio puesto que la curva es una media móvil con un ancho de ventana de 200. El *reward* medio obtenido con la tabla del episodio 16000 se muestra en la tabla 4.3. 57

5.1. Gráficas que representan las puntuaciones del perceptrón entrenado con los diferentes estados. 61

5.2. *Heat maps* de la selección de parámetros de un *Q-Table* con diferentes estados. 62

6.1. *Reward* medio de un *KNN agent* con diferentes valores de k. 66

6.2. *Reward* medio de las diferentes redes neuronales sin regularización. En la leyenda se indican las capas ocultas y el número de neuronas de cada capa. 67

6.3. *Reward* medio de las diferentes redes neuronales con regularización L2 con $\lambda = 0.1$ y $\lambda = 0.2$. En la leyenda se indican las capas ocultas. 68

6.4. Errores de las redes usadas por el *ANN agent*. Cada gráfica corresponde a un valor de regularización diferente. 69

7.1. Comparación entre dos entrenos con diferentes valores de α en los que se ha utilizado ϵ *decay*. Inicialmente $\epsilon = 1$, tras cada episodio ϵ se ha disminuído en 0.001 hasta parar en 0.05. 72

7.2. Modificación del valor por defecto de la función $Q(s, a)$. $\alpha = 0.1$ y $\gamma = 0.9999$. 73

7.3. Curva de aprendizaje de un *Q-Table agent* con $\alpha = 0.2$, $\gamma = 0.99$ y un factor de discretización de 10. En la curva con *decay*, cada 1000 episodios se ha disminuido α a la mitad para conseguir que la curva oscile menos. 74

7.4. Visualización de las zonas discretizadas. En la imagen la pelota se encuentra en $Diff_x = 15$ y $Ball_y = 180$. En todo el recuadro en el que se encuentra la pelota las coordenadas serían las mismas. Las líneas de la imagen son aproximadas, no representan las medidas reales. 75

7.5. Curva de aprendizaje con discretización selectiva usando el estado `dx`. $\alpha = 0.1$, $\gamma = 0.9999$. 75

A.1. Logo de la librería. Como no podría ser de otra manera, el logo está muy feliz. 79

A.2. Contenido del archivo *train.csv*. 81

A.3. Carga de archivos CSV para usar como dataset. 81

A.4. Representación gráfica del *dataset* mostrado en A.2. 82

A.5. Todos los puntos de un mismo color indican que ya no se trata de un problema de clasificación, sino que es de regresión. 83

A.6. Diagrama de todos los modelos implementados en la librería. 83

A.7. Representación gráfica de un modelo lineal. 84

A.8. Diagrama de herencia del perceptrón. Diagramas de este tipo de cada una de las clases de la librería, pueden ser consultados en la documentación. 85

A.9. Creación y entreno de un perceptrón. 85

A.10. A la izquierda el *dataset* utilizado y a la derecha el mismo *dataset* con el *decision boundary* formado por el vector de pesos de un perceptrón que menos error consigue. El perceptrón ha llegado a esa solución de error 0 con sólo 4 iteraciones. 85

A.11. Método de `LinearRegression` que obtiene el vector de pesos que minimiza el error cuadrático en el *dataset* indicado. La ecuación matemática implementada en este método se encuentra en A.5. 86

A.12. A la izquierda los pesos del regresor lineal. A la derecha la visualización del ajuste junto a los datos utilizados. 86

A.13. Regresión logística. 87

A.14. Ejemplo de regresión logística. A la izquierda se observa el *dataset* orginal y el borde de decisión generado en el espacio $\mathcal{X}$. A la derecha el *dataset* y el borde de decisión se encuentran en el espacio $\mathcal{Z}$. 88

A.15. Código para generar un clasificador KNN. 88

A.16. Los bordes de decisión de un KNN en 2 dimensiones coinciden con los polígonos de *Voronoi*. 88

A.17. Creación y entreno de una red neuronal con *batch* y *stochastic gradient descent* (SGD). El primer paso consiste en estandarizar el *dataset*. 89

A.18. Representación gráfica de una red neuronal con una arquitectura 2-3-3-1. 90

A.19. Frontera de decisión de una red neuronal con arquitectura 2-1 (izquierda) y 2-10-1 (derecha). Se observa que en la derecha se obtiene una frontera de decisión diferente a la recta gracias a la presencia de una capa oculta. 90

A.20. Una buena forma de probar el correcto funcionamiento de una red neuronal es comprobar que sea capaz de hacer *overfitting* sobre un conjunto de datos [Li et al.]. 90

A.21. Creación y entreno de una red neuronal con *batch gradient descent*. 91

A.22. Ajuste de una red neuronal. El α usado es más bajo en comparación al usado en las redes neuronales de clasificación, debido a que altos valores de α satura los pesos y hacen que el error tienda a infinito. Para evitar esto podría introducirse alguna constante a la hora de actualizar los pesos [Li et al.]. 91

A.23. SVM con *kernel* lineal. En la imagen se encuentra representado el margen maximizado, de tamaño $\frac{1}{||\mathbf{w}||}$. Los dos puntos que se encuentran sobre los márgenes son los llamados vectores de soporte. Tras resolver el problema de programación cuadrática, son aquellos puntos cuyo $\alpha > 0$. 92

A.24. SVM con *kernel* gaussiano y diferentes valores de σ. 93

A.25. Código de como aplicar PCA y LDA. Se asume la existencia de una variable `dataset` del tipo `DataSet` con sus características debidamente normalizadas o estandarizadas. 94

A.26. En rojo los 0s y en azul los 1s. PCA (izquierda) y LDA (derecha) usando los 2 primeros vectores propios que más varianza mantienen (cerca del 45 %). 94

A.27. Algunas de las imágenes contenidas en el MNIST. 94

B.1. Diagrama de clases de los agentes implementados. 97

B.2. Definición del tipo de dato `Q` usado para representar la función $Q(s,a)$. 99

Índice de tablas

2.1. Estadísticas de un **humano** no experto creadas a partir de los datos de 10 episodios. 32

2.2. Resultados del ***Random agent*** recogidos tras la ejecución de 1000 episodios. 33

2.3. Resultados de un ***If agent*** tras 1000 episodios con distintos valores de ϵ. 864 es la puntuación máxima que se puede alcanzar en este juego. Una vez alcanzada esa puntuación la ejecución se termina. 34

2.4. Resultados de un **If agent** recogidos tras la ejecución de 1000 episodios con **ruido entero** ± 1 en las lecturas de la RAM. 35

2.5. Resultados de un **If agent** recogidos tras la ejecución de 1000 episodios con diferentes niveles de **ruido** en las lecturas de la RAM. 36

3.1. Errores de los perceptrones sobre el conjunto de entrenamiento. 43

3.2. Resultados de un ***Perceptron agent*** recogidos tras 10 entrenos con cada juego y ejecutándolos durante 100 episodios con **ruido** ± 0.01 en las lecturas de la RAM. 43

3.3. Resultados de un ***Perceptron agent*** recogidos tras 10 entrenos con cada juego y ejecutándolos durante 100 episodios con $\epsilon = 0.05$. 43

4.1. Representación de la tabla que guarda el *Q-Table agent*. 51

4.2. Resultados de un ***Q-Table agent*** tras ser entrenado con 20000 episodios con $\epsilon = 0.05$. 56

4.3. Resultados de la ejecución de 1000 episodios de un *Q-Table agent* con una tabla sacada tras 16000 episodios de entrenamiento con $\alpha = 0.1$, $\gamma = 0.9999$ y discretización 10. 56

5.1. Diferentes estados que usaremos en las pruebas. De ahora en adelante los referenciaremos con los nombres aquí descritos. Los nombres son muy descriptivos, los 4 primeros señalan las características que no son incluídas y el último señala la única característica incluída. 59

Índice de símbolos.

ALE	*Arcade Learning Enviroment*.
ANN	*Artificial Neural Network*, red neuronal artificial.
DL	*Deep Learning*, aprendizaje profundo.
DP	*Dinamic Programming*, programación dinámica.
KNN	*K-Nearest Neighbors*, K-Vecinos Cercanos.
MDP	*Markov Decision Process*.
ML	*Machine Learning*, aprendizaje automático.
RL	*Reinforcement Learning*, aprendizaje por refuerzo.
SL	*Supervised Learning*, aprendizaje supervisado.

Capítulo 1

Introducción

Por todos es sabido que la tecnología está muy presente entre nosotros, pero no todos saben que día a día usan de una forma u otra ***machine learning*** (de ahora en adelante ML). A grandes rasgos, ML trata de entender el mundo a través de los datos. Podemos pensar en cualquier campo de conocimiento, como la biología, la medicina, las finanzas, los automóviles, la robótica... en todos ellos hay lugar para el ML.

Desde que aparecieron, los ordenadores se han encargado de realizar de forma más rápida que nosotros muchas tareas tediosas y repetitivas. Un ordenador es capaz de calcular muchísimo más rápido que un humano, sin embargo, hay ciertas tareas que para nosotros son triviales y que ellos no han sido capaces de realizar con tanto éxito. Hablo de problemas como la visión, el reconocimiento del sonido, el autoaprendizaje... Los humanos aprenden a reconocer formas y sonidos desde una edad muy temprana, pero para los ordenadores eso supone un gran desafío. Es precisamente en ese tipo de tareas dónde se aplica el ML. Los algoritmos de ML aprenden a realizar tareas generalizando a partir de los ejemplos aportados. Es una buena forma de llevar a cabo tareas cuya programación manual sería muy costosa o inviable.

El tipo más común de ML es el llamado *supervised learning* (SL) que, a grandes rasgos, estudia la obtención de modelos capaces de predecir un valor a partir de datos correctamente etiquetados. El *reinforcement learning* (RL) es otra rama del ML que se dedica al estudio y creación de agentes inteligentes capaces de aprender a maximizar una recompensa que es resultado de sus acciones.

En el presente trabajo se usarán ambos tipos de ML para jugar al *Breakout*, un conocido juego de la videoconsola *Atari 2600*.

Atari 2600 es una videoconsola desarrollada en 1977 y vendida durante más de una década. Poseía una CPU de propósito general con una frecuencia de reloj de 1.19Mhz. Más de 500 juegos originales fueron publicados para este sistema y aún hoy siguen apareciendo nuevos títulos desarrollados por fans. La mayoría de los juegos *arcade* del momento fueron portados a esta videoconsola (*Pacman*, *Space Invaders*...). Una de las partes más características de esta videoconsola es el *joystick*, que se puede apreciar junto a la videoconsola en la figura 1.1. Los cartuchos ROM que contienen el código del juego suelen tener 2-4kB. La memoria RAM tiene muy poca capacidad, concretamente 128 bytes.

Figura 1.1: Videoconsola *Atari 2600* a la izquierda[Wikipedia] y cartuchos de diversos juegos a la derecha[VintageToySeller]. Entre los cartuchos se puede divisar el del *Breakout* arriba a la izquierda y el de su secuela *Super Breakout* en medio a la derecha.

Breakout es un juego de tipo *arcade*, como todos los juegos de *Atari 2600*, que tiene como objetivo romper todos los bloques que cubren la mitad superior de la pantalla (captura en la figura 1.2). El jugador es una pala o raqueta que puede moverse a la izquierda o a la derecha para golpear una pelota y romper con ella los bloques. Hay que impedir que la pelota se caiga, se cuenta con 5 oportunidades antes de perder el juego. Hay ciertas características del juego que merece la pena mencionar. Una de ellas es que la velocidad de la pelota aumenta en función de qué bloque hayas golpeado. Al romper alguno de los bloques superiores la pelota se acelera y se mantiene acelerada hasta que pierdes esa vida. Otro detalle importante es que la pala del jugador disminuye su tamaño cuando la pelota golpea la pared superior. Esto, unido a la mayor velocidad que suele llevar la pelota en esos momentos, convierte al *Breakout* en un juego que requiere de mucha habilidad para poder acabarlo. Cuando consigues romper todos los bloques, la pantalla se vuelve a llenar y pasas al nivel 2 (exáctamente igual que el nivel 1). El juego termina una vez acabas con todos los bloques del nivel 2. Cada vez que se rompe un bloque la puntuación crece hasta llegar a un máximo de 864 puntos tras completar ambos niveles.

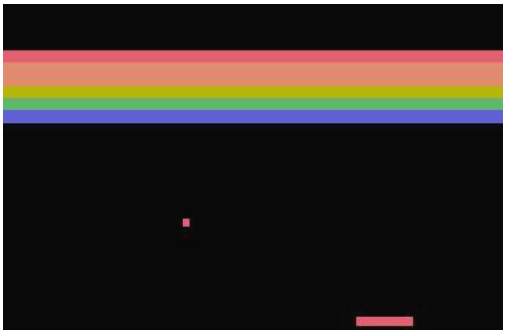


Figura 1.2: Captura de pantalla del inicio del juego *Breakout*. Todos los bloques se encuentran intactos.

Para ejecutar el *Breakout* usaremos el ***Arcade Learning Environment*** (ALE), que a su vez se apoya en el emulador *Stella* de una máquina *Atari 2600*. ALE nos proporciona una interfaz de programación que permitirá a nuestro código interactuar con el emulador de una forma muy sencilla. En concreto, haremos uso de la

interfaz de C++, que será el lenguaje utilizado en cada uno de los agentes programados durante el desarrollo del trabajo. El entorno soporta una multitud de juegos diferentes. La resolución de pantalla que nos proporciona es de 210x160 píxels, pero también nos permite ejecutar los juegos sin visualización gráfica. De esta manera, los juegos se ejecutan de forma muy rápida, lo cual nos va a ser muy útil a la hora de dejar que el ordenador entrene usando técnicas de RL.

Figura 1.3: Logo del *Arcade Learning Enviroment*.

1.1 Estado del arte

El ML se puede considerar como una rama de la inteligencia artificial (IA). Normalmente los programas de la IA más clásica estaban destinados a servir a un propósito concreto, habían sido programados para cumplir una única función. Por ejemplo, el conocido *DeepBlue* era capaz de jugar al ajedrez y sólo al ajedrez. Gracias a distintas técnicas de ML, a día de hoy se están consiguiendo modelos con un propósito más general que funcionan francamente bien en varios ámbitos.

Uno de los primeros estudios que hacía uso del ML tuvo lugar mucho antes de la aparición de *DeepBlue*. *Arthur Samuel* desarrolló un programa capaz de jugar a las damas [Samuel, 1959] en el que introdujo algunas técnicas de ML (*rote learning*). *Samuel* manifestó su pensamiento de que enseñar a los ordenadores a jugar era mucho más productivo que el hecho de desarrollar tácticas específicas para cada juego.

Quizás una de las aplicaciones más conocidas del RL sea el *TD-Gammon* [Tesauro, 1994]. Se trata del primer programa que consiguió jugar al *Backgammon* por encima del nivel de un humano profesional. El *TD-Gammon* usa una red neuronal que aproxima el valor de las jugadas, permitiendo obtener al algoritmo la mejor jugada aún en situaciones que nunca antes ha visto.

Los últimos estudios más exitosos sobre ML están relacionados con *deep learning* (DL). DL es una rama del ML que tiene el propósito de evitar la extracción de características de forma manual. La extracción de características es un proceso importantísimo a la hora de aplicar ML. En este trabajo se le ha dedicado un capítulo entero a la selección de las características de entrada. Con el DL pretende evitarse este paso dejando que sea el algoritmo el que extraiga esas características (*unsupervised learning*) a partir de, por ejemplo, una imagen. Esto se consigue utilizando modelos con muchas capas, de ahí el nombre *deep*. Volviendo al ejemplo

de las imágenes, el DL se ha aplicado con mucho éxito en ellas, generando redes neuronales que obtienen características de bajo nivel en las primeras capas (líneas y esquinas) y otras características de más alto nivel conforme nos acercamos a las capas de salida.

Los juegos siempre han sido una buena forma de poner a prueba los algoritmos de RL. Esto es lo que motiva la aparición del ALE. Hay bastantes trabajos publicados que lo utilizan. El primero de ellos es el que presenta el entorno [Bellemare et al., 2013]. En él se utiliza SARSA(λ), un algoritmo bien conocido en el mundo del RL. En su trabajo muestran como usando las imágenes del emulador son capaces de jugar a varios juegos con el mismo algoritmo. Las imágenes son previamente tratadas y se extraen de ellas ciertas características para simplificar así el proceso de aprendizaje. Una de las características utilizadas son las llamadas *Basic features*. Discretizan la pantalla y comprueban qué color se encuentra activo en cada una de las regiones. Otro tipo de características son las *DISCO features*. Éstas intentan detectar y clasificar los objetos que se encuentran en la pantalla con el objetivo de inferir sus posiciones y velocidades.

En *Google DeepMind* [Mnih et al., 2015] se ha realizado un estudio similar en la que la extracción de características es realizada por un algoritmo de DL. Su algoritmo, DQN, usa una CNN para extraer las características de las imágenes obtenidas del emulador. No es necesario poseer conocimiento específico de cada juego, por lo que permite poder aplicar el mismo algoritmo en distintos juegos sin ningún tipo de ajustes de parámetros. Con dicha técnica se ha conseguido que el ordenador juegue por encima del nivel de un humano en bastantes de los juegos probados, incluido el *Breakout*. Es una de las primeras combinaciones exitosas de DL con RL, llamado por muchos *deep reinforcement learning*.

Estudios más recientes muestran que se puede conseguir resultados iguales y superiores a los del DQN sin usar redes profundas [Liang et al., 2015]. En este estudio, sin embargo, no se consiguen tan buenos resultados en el *Breakout*.

Recientemente, al *TD-Gammon* y al *Deep Blue* se le suma *AlphaGo* de *Google DeepMind* [Silver et al., 2016], el primer programa capaz de ganar en el juego del *Go* a los mejores jugadores profesionales. Comparado con el ajedrez, el *Go* es un juego en el que el espacio de búsqueda es muy superior, por lo que las técnicas usadas en otros juegos no funcionaban tan bien en éste. Cuando se compara *AlphaGo* con *Deep Blue* o con máquinas como *Watson* de IBM, los algoritmos que componen *AlphaGo* son más de propósito general. Una combinación de búsquedas y algoritmos de ML ha conseguido que *AlphaGo* se proclame campeón de *Go*.

Muy recientemente se ha lanzado una web llamada *OpenAI Gym* que pretende ser un gimnasio en el que entrenar algoritmos de RL. Una interfaz en *Python* permite programar agentes que tienen acceso a diferentes tipos de entornos: problemas clásicos de RL, juegos de *Atari* (usando ALE)... Esta iniciativa permite subir los resultados obtenidos por los algoritmos y poder comprar su rendimiento con el de otra gente. Además, muchos de los trabajos subidos se encuentran publicados junto a información de cómo reproducirlos en tu propio PC. Esto facilita el acceso a algoritmos avanzados, lo que fomentará el progreso en este campo.

1.2 Objetivos

El objetivo principal del trabajo es usar algoritmos de ML que jueguen de forma autónoma al *Breakout*. Debido a los distintos tipos de aprendizaje que existen, las pruebas se enfocarán de dos maneras claramente diferenciadas. Usaremos algoritmos de SL con el objetivo de imitar el comportamiento de un humano y usaremos RL para conseguir el mejor jugador que podamos a partir de la experiencia de juego del algoritmo.

Se ha preferido que, quitando el uso de una librería de álgebra y el ALE, todos los algoritmos usados sean una implementación propia. Esto trae consigo un problema de rendimiento si se equipara con las optimizadas librerías que existen en el mercado, pero el problema es mínimo en comparación a los conocimientos adquiridos y a la satisfacción que supone usar tu propia implementación 😊

El trabajo supone mi inicio en el mundo de la investigación. Con él se espera aprender y mejorar distintas técnicas de realización de pruebas y muestra de resultados.

1.3 Metodología

La metodología usada en este trabajo está basada en experimentos. Cada capítulo de la memoria corresponde a un experimento. De cada uno de los experimentos realizados se extraerán una serie de conclusiones que darán lugar a los siguientes experimentos.

- Este primer capítulo contiene una descripción del estado del arte en el campo del ML así como las motivaciones y los objetivos perseguidos con este trabajo.
- Nuestro primer experimento constará de la **puesta en marcha del ALE** y la obtención de características de la RAM para el *Breakout*. Compararemos las puntuaciones obtenidas por un humano, un agente aletorio y uno basado en reglas programado expresamente para el *Breakout*.
- El segundo experimento será abordado desde el punto de vista del **SL**. En él se usará un perceptrón que intentará imitar el comportamiento de varios jugadores con diferentes niveles de habilidad.
- En este experimento se utilizará un enfoque basado en el **RL**. Utilizaremos el algoritmo *Q-Learning* para intentar conseguir la mayor puntuación posible.
- Llegados a este punto repetiremos varias de las pruebas realizadas anteriormente con diferentes **características de entrada** para nuestros algoritmos.
- Continuando con el SL, se pretendrá mejorar los resultados obtenidos usando otros modelos de SL diferentes al perceptrón. En concreto se hará uso del **KNN** y de **redes neuronales**.
- En este experimento aplicaremos una serie de **modificaciones** en ciertos parámetros del ***Q-Learning*** con el objetivo de mejorar su rendimiento.

- Terminaremos extrayendo **conclusiones** generales de todos los experimentos realizados e indicando las futuras investigaciones sobre el tema que podrían llevarse a cabo.

Tras finalizar el núcleo del trabajo se encuentran un par de apéndices que aportan detalles más técnicos sobre las implementaciones.

- ***HappyML***. Descripción del contenido de la librería de ML desarrollada, así como detalles sobre su uso.
- **Detalles de implementación** de los agentes inteligentes desarrollados y descripción de los medios utilizados para la automatización de tareas.

Capítulo 2

The Arcade Learning Environment

Nuestro primer experimento consiste en poner en marcha el entorno y escribir los primeros agentes para el juego del *Breakout*. Estos primeros agentes no van a utilizar ningún tipo de algoritmo de ML, simplemente son creados con el objetivo de comparar los resultados de éstos con los de los agentes que se creen posteriormente.

Como se ha comentado con anterioridad, se hará uso del ALE, que proporciona un entorno cómodo para la programación de agentes inteligentes para juegos de *Atari 2600*. ALE proporciona una interfaz que puede ser usada para jugar a cientos de juegos diferentes, pero por el momento nos centraremos solamente en el *Breakout*.

Comenzaremos explicando como hacer uso del ALE y mostrando las principales características del *Breakout*. Después mostraremos la puntuación media conseguida por un humano no experto jugando al *Breakout* y pasaremos a mostrar los resultados de los 3 agentes implementados:

- ***Noop agent***. Agente que no se moverá durante todo el juego.
- ***Random agent***. Agente que realizará acciones aleatorias.
- ***If agent***. Agente programado para que haga uso de un par de reglas. Este agente hace uso de nuestro conocimiento sobre el objetivo del *Breakout* para jugar.

Para poner a prueba a los agentes, introduciremos dos métodos que usaremos también en los próximos experimentos para comparar unos agentes con otros: ϵ*-greedy* y el ruido en las lecturas de la RAM. Las pruebas realizadas nos permitirán llegar a conclusiones interesantes sobre el funcionamiento del juego.

2.1 Puesta en marcha

El primer paso es obtener el código del ALE de la página web oficial o del repositorio de *GitHub*. Se ha compilado el entorno haciendo uso de **SDL** (*Simple DirectMedia Layer*). Esto nos permitirá visualizar en una ventana el desarrollo del juego cuando lo deseemos. Aún compilado con SDL, para el entrenamiento de algoritmos de RL o para la realización de tests, se puede desactivar la visualización gráfica para acelerar la ejecución.

Hay varias formas de utilizar ALE: FIFO (permite el uso de cualquier lenguaje de programación usando una interfaz de texto), CTypes (Python), RLGlue ó C++. Los agentes inteligentes de este trabajo serán todos realizados utilizando la interfaz que proporciona la *shared library* de ALE para C++.

Para poder acceder a la interfaz del sistema se utiliza un objeto de tipo `ALEInterface`.

```
ALEInterface ale;

ale.setInt("random_seed", 0);  // Difiere de la semilla de srand.
ale.setFloat("repeat_action_probability", 0);
ale.setBool("display_screen", false);
ale.setBool("sound", false);

ale.loadROM("/ruta/al/BinarioDelJuego.bin");
```

Figura 2.1: Inicialización básica de ALE y carga de la ROM.

El entorno ALE posee una serie de parámetros configurables. En el fragmento de código de la figura 2.1 se le da valor a algunos de los más importantes.

Si la **semilla aleatoria** que le pasamos es 0, el entorno utilizará el tiempo actual como semilla. ALE no usa `srand`, por lo que la tarea de inicializar la semilla de `srand` recae sobre nosotros.

Otra de las opciones que ofrece ALE es establecer una probabilidad de **repetición de acciones**. Con la probabilidad indicada, la acción ejecutada en el siguiente frame puede ser igual a la ejecutada en el anterior. Esta opción fue añadida con la intención de mejorar los entrenamientos con algoritmos de RL, en los que una acción recibe *rewards* a largo plazo. El valor por defecto de este parámetro es 0.25, pero nosotros por el momento dejaremos ese valor a 0 para que no interfiera en las decisiones de nuestros agentes.

En esos parámetros también le indicamos a ALE si debe **mostrar la ventana** con el juego y si tiene que reproducir el **sonido**.

Por último le indicamos la ruta al archivo **binario del juego** para que lo cargue.

La forma de jugar a los juegos es enviándole acciones al emulador. La máquina *Atari 2600* recibe 18 acciones asociadas al jugador A y otras 18 asociadas al jugador B. Además, hay otras acciones con objetivos ajenos al juego, como el reinicio de la máquina.

En la interfaz de C++ de ALE las acciones son una enumeración (ver figura 2.2). Para **ejecutar una acción** se utiliza `ale.act(a)`, donde `a` es una variable del tipo `Action`. Llamaremos **ciclo** a cada una de las llamadas a esta función. Es importante darse cuenta de que si no se realiza esta llamada el juego no avanzará, es decir, que si no queremos realizar ninguna acción en un momento dado debemos de enviarle una acción específica para no hacer nada o sino el juego no seguirá ejecutándose. De esta forma, el número de acciones por segundo de juego dependen del emulador y no de la máquina en la que se ejecuta.

Como veremos más adelante, la mayoría de los juegos no usan todas las acciones disponibles. El hecho de enviar una acción que no genera resultado no produce ningún error, será tomada como la acción `PLAYER_A_NOOP` que no hace nada.

```
// Define actions
enum Action {
    PLAYER_A_NOOP           = 0,
    PLAYER_A_FIRE           = 1,
    PLAYER_A_UP             = 2,
    PLAYER_A_RIGHT          = 3,
    PLAYER_A_LEFT           = 4,
    .
    .
    .
```

Figura 2.2: Definición de ALE de los tipos de acciones disponibles.

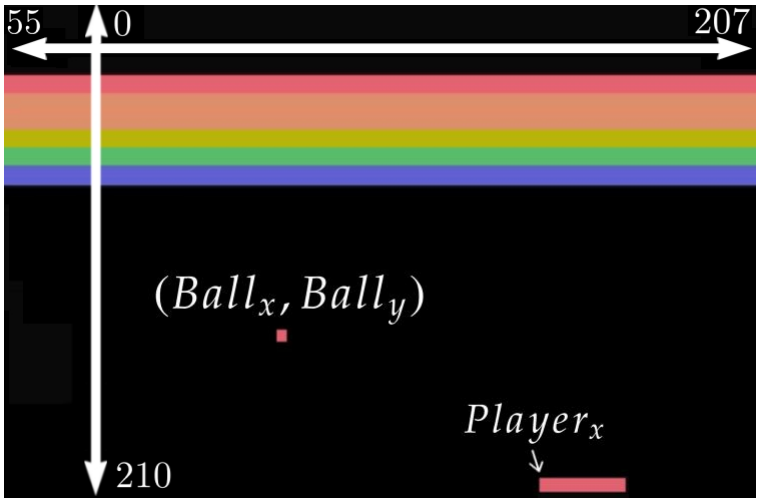


Figura 2.3: Captura del juego Breakout con la información extraída de la RAM.

Tras ejecutar un paso del juego, el método `ale.act(a)` nos devolverá un valor de *reward* asociada a la acción que acaba de realizar. Esto, además de sernos útil para los entrenos de algoritmos de RL, nos permite saber cuál es la puntuación obtenida por los agentes.

2.2 *Breakout*

En este estudio nos centraremos en resolver el juego *Breakout* con diferentes técnicas de ML. ALE no proporciona las ROM de los juegos emulados, por lo que hay que conseguirlos utilizando webs de terceros [AtariAge, 1998].

El juego original del *Breakout*, tal como se indica en su manual [Atari, Inc., 1978], tiene muchos modos de juego diferentes. ALE pone a nuestra disposición el modo más básico: un jugador (el jugador A) y todos los bloques visibles en pantalla.

En este juego un **episodio** consiste en agotar las 5 vidas iniciales.

```
const ALERAM ram = ale.getRAM();
int playerX = ram.get(72);
int ballX = ram.get(99);
int ballY = ram.get(101);
```

Figura 2.4: Código para leer datos de la RAM.

La **memoria RAM** de una máquina Atari 2600 está compuesta por 128 bytes. El número de vidas, la puntuación, la posición de la pelota y del jugador son datos que nos resultarían muy útiles para utilizar en nuestros algoritmos. Todos esos datos se encuentran almacenados en la RAM. Las vidas restantes y la puntuación ya nos lo proporciona el ALE, pero para averiguar las coordenadas de la pelota y del jugador hay que buscar en la RAM las posiciones en las que se guardan.

Para ello se comprobaron las posiciones de memoria que iban cambiando de un ciclo a otro y se logró encontrar un valor que varía en función del movimiento del jugador (posición 72 de la RAM) y también se han encontrado otros dos valores (posiciones 99 y 101) que cambian en función de la posición de la pelota. En la figura 2.3 se visualizan gráficamente las coordenadas que hemos recuperado de la RAM así como los rangos que toman las coordenadas. Aunque ya lo veremos más adelante, es importante darse cuenta de que la coordenada x del jugador ($Player_x$) se mide con respecto al extremo izquierdo de la pala. Los rangos de cada variable son

$$Player_x \in [55, 191]$$
$$Ball_x \in [55, 207]$$
$$Ball_y \in [0, 210].$$

El valor máximo que puede alcanzar $Player_x$ es menor que el que puede alcanzar $Ball_x$ debido al ancho de la pala y al hecho de que la coordenada x del jugador se mida desde el extremo izquierdo. Esto significa que por muy a la derecha que vayamos, el extremo izquierdo de la pala no podrá estar tocando la pared derecha.

Con esos datos tenemos implícitamente la velocidad de la pelota. Para obtenerla bastaría con restar las coordenadas $Ball_x$ y $Ball_y$ del ciclo anterior al valor del ciclo actual.

$$Ball_{vx}(t) = Ball_x(t) - Ball_x(t-1)$$
$$Ball_{vy}(t) = Ball_y(t) - Ball_y(t-1)$$

De las 18 acciones disponibles para el jugador A, *Breakout* cuenta con 4 acciones legales, es decir, acciones que al ser enviadas al emulador son aceptadas y no ignoradas. El número mínimo de acciones para un juego puede ser obtenido con `ale.getMinimalActionSet()` y para este juego son las siguientes:

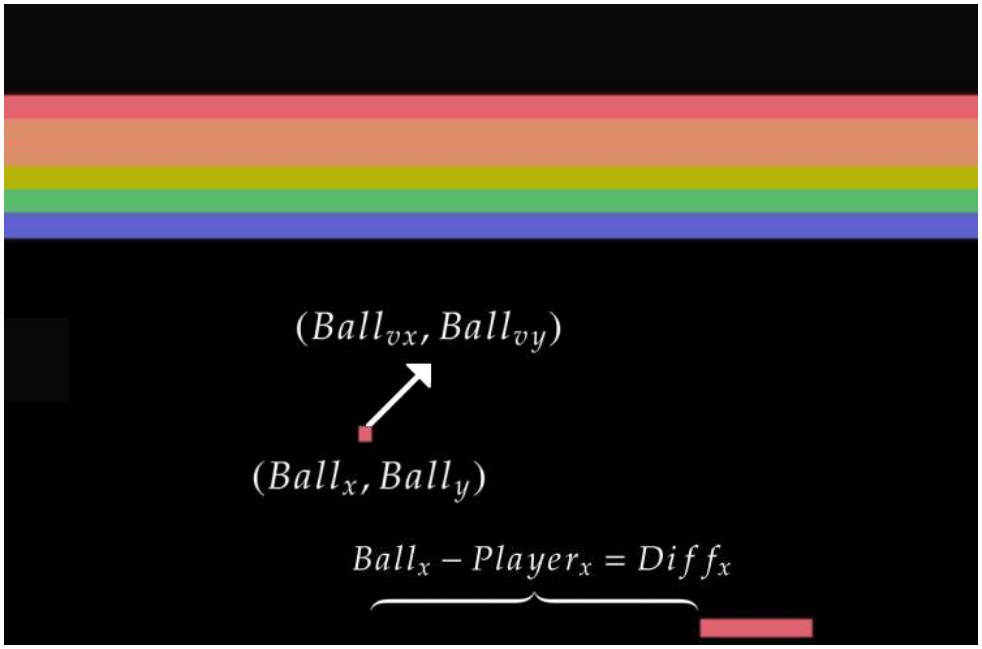


Figura 2.5: Captura del juego Breakout con la información extraída de la RAM, la extraída gracias al ciclo anterior y una característica creada por nosotros llamada $Diff_x$. Crear otra característica llamada $Diff_y$ carece de sentido debido a que la altura del jugador es siempre la misma.

- `PLAYER_A_FIRE`: Hace aparecer la pelota para que se pueda comenzar a jugar. Esta acción sólo tiene efecto cuando es usada al inicio de cada episodio y después de perder una vida. Para simplificar los algoritmos y hacer que tengan que elegir entre menos acciones diferentes, esta acción será ejecutada de forma automática en cuanto el jugador pierda una vida. De esta forma nuestros agentes sólo se tendrán que preocupar de ir a la derecha, izquierda o de no hacer nada.

- `PLAYER_A_NOOP`: No hace nada.

- `PLAYER_A_LEFT`: Mueve la pala a la izquierda.

- `PLAYER_A_RIGHT`: Mueve la pala a la derecha.

2.3 Jugador humano

Antes de empezar a crear diferentes tipos de agentes sería conveniente hacerse una idea de qué puntuación puede llegar a conseguir una persona no experta jugando a este juego. **He jugado** 10 partidas y los resultados obtenidos se muestran en la tabla 2.1. Se observa una gran desviación en los datos debida a despistes, cansancio... en definitiva, un humano nunca va a repetir exactamente las mismas acciones de una partida a otra.

Media	Desviación Típica	Máximo	Mínimo	*Reward* / 100 ciclos
256.00	104.40	394	96	3.70

Tabla 2.1: Estadísticas de un **humano** no experto creadas a partir de los datos de 10 episodios.

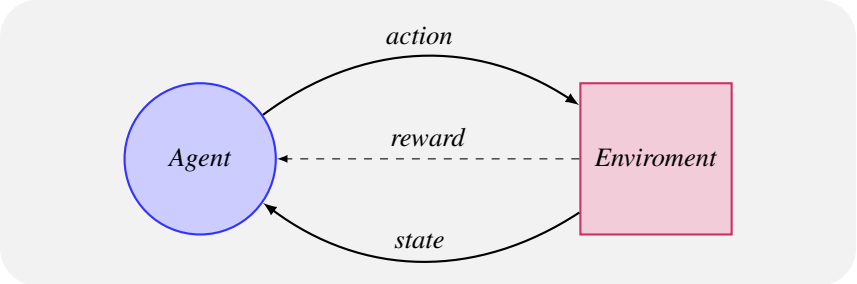


Figura 2.6: Esquema de un agente inteligente interactuando con su entorno.

2.4 Agentes

Un agente es una entidad que puede desenvolverse de manera autónoma en un entorno determinado. Los agentes tendrán la misión de, usando los datos que hemos obtenido del entorno, jugar al *Breakout* lo mejor que puedan. Las acciones realizadas por el agente tienen repercusión en el entorno: producen un cambio de estado y proporcionan al agente una recompensa. En la figura 2.6 se representa de manera gráfica la interacción de todo agente con su entorno.

En este capítulo crearemos tres agentes sin usar ninguna técnica de ML, simplemente con el objetivo de comparar los resultados de éstos con los de los agentes que se creen luego. Los agentes que usan algún tipo de inteligencia artificial se llaman comúnmente **agentes inteligentes**.

2.4.1. *Noop agent*

El agente más sencillo que podemos realizar es aquel que siempre realiza la acción `PLAYER_A_NOOP`, es decir, que no hace nada.

Comprobamos que tras la ejecución de 1000 episodios **siempre** obtiene de puntuación 0. Si visualizamos los episodios se observa que la pelota es tirada siempre desde los mismos sitios y en las mismas direcciones. Si el lugar de aparición de la pelota fuera aleatorio podría darse el caso de que golpeara a la pala inmóvil y se rompiera así algún bloque. Debido a que eso no ocurre en ningún momento, podemos deducir que se trata de un **juego determinista**.

En la figura 2.7 se observan por orden los 5 saques que son realizados en cada juego del *Noop agent*. Si en algún momento golpeamos la pelota, el siguiente saque no sigue la misma secuencia. Sin embargo, realmente solo hay 4 tipos diferentes de saques. En la figura se puede observar que el quinto saque es en realidad el mismo que el primero.

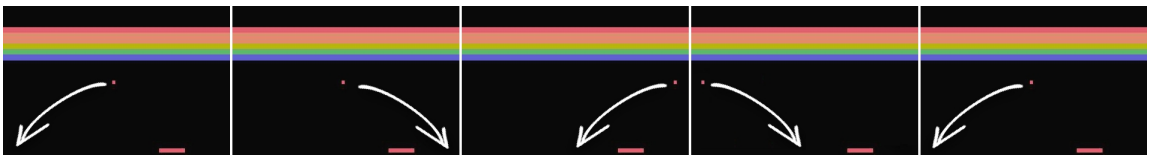


Figura 2.7: Si no tocamos la pelota, los 5 lanzamientos que se realizan durante un episodio son los indicados con las flechas (siempre son realizados en el mismo orden).

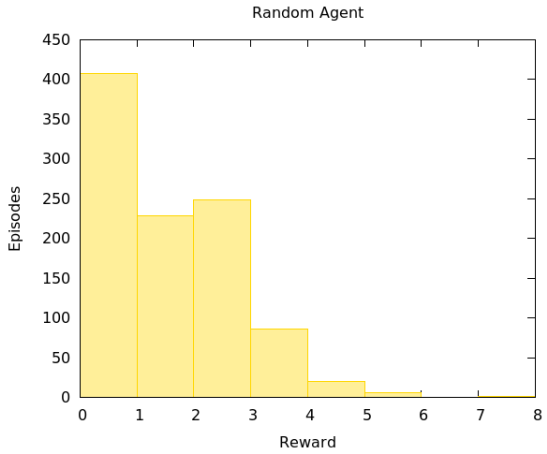


Figura 2.8: Histograma del *reward* obtenido en los 1000 episodios jugados por el **Random agent**.

2.4.2. *Random agent*

Agente que realiza aleatoriamente cualquiera de las 3 acciones del *Breakout*. Cada acción tiene la misma probabilidad de ser elegida ($\frac{1}{3}$, distribución uniforme). Para probar el rendimiento de este agente vamos a ejecutarlo durante 1000 episodios y calcularemos la puntuación media conseguida así como su desviación y la puntuación máxima y mínima. Los resultados se encuentran en la tabla 2.2 y se puede encontrar un histograma de las ejecuciones en la figura 2.8.

Media	Desviación Típica	Máximo	Mínimo
1.11	1.16	7	0

Tabla 2.2: Resultados del ***Random agent*** recogidos tras la ejecución de 1000 episodios.

2.4.3. *If agent*

Con los datos obtenidos de la RAM (Figura 2.3) vamos a crear otro agente al que le introduciremos conocimiento del entorno con un par de reglas programadas por nosotros. Lo único que realizará será leer las coordenadas x de la pelota ($Ball_x$) y del jugador ($Player_x$) y mover el jugador a la derecha si $Ball_x > Player_x$ o moverlo a la izquierda en caso contrario.

Al ejecutar este agente por primera vez nos damos cuenta de que $Player_x$ se mide desde el extremo izquierdo del jugador (tal y como se indica en la figura 2.3), debemos de sumarle la mitad de la longitud del jugador para que intente golpear la pelota con la mitad de la pala. La mitad es aproximadamente 10.

Debido a que en cada ciclo se realiza un movimiento, la pala se mueve muy rápidamente provocando que se le escape la pelota bastante a menudo. Si después de cada acción tomada realizamos una acción `PLAYER_A_NOOP` disminuimos a la mitad la toma de decisiones y la pala se tambalea menos mejorando así el acierto.

Un nuevo problema nos aparece al ejecutar este agente, se queda golpeando la pelota contra la pared en un bucle infinito. Si lo repetimos varias veces siempre llegamos a la misma situación. A diferencia de un humano, el ordenador sigue el código programado instrucción a instrucción y realiza siempre las mismas acciones. Con el *Noop agent* comprobamos que todos los inicios eran iguales, por lo que si siempre se aplica el mismo código cada juego es exactamente idéntico al anterior.

Para solucionar esto y poder medir mejor el rendimiento del agente vamos a aplicar dos métodos: ϵ-*greedy policy* e introducción de ruido en las lecturas de la RAM.

ϵ-*greedy policy*

La primera forma de abordar este problema es añadirle un componente de aleatoriedad a la toma de decisiones. Utilizaremos el método llamado ϵ-***greedy policy***, que consiste en que, en cada turno, con una probabilidad ϵ se vaya a realizar una acción aleatoria. Puesto que es una probabilidad, $\epsilon \in [0, 1]$.

La elección del valor de ϵ es muy importante. Si es demasiado grande podría desvirtuar el funcionamiento de los agentes y si es demasiado pequeño podría no afectar en absoluto al agente. Para que haya cambios en el juego, las acciones aleatorias deben de producirse justo en el momento de la colisión con el jugador. Si se producen cuando la pelota se encuentra lejos del jugador, las reglas del *If agent* harán que se vuelva a colocar justo en la posición en la que habría estado si no llega a realizar esa acción aleatoria.

ϵ	**Media**	**Desviación Típica**	**Máximo**	**Mínimo**	***Reward* / 100 ciclos**
0.01	**580.48**	190.31	**864**	**283**	3.68
0.05	454.92	123.84	**864**	84	6.15
0.1	335.71	**96.50**	670	46	**6.77**

Tabla 2.3: Resultados de un ***If agent*** tras 1000 episodios con distintos valores de ϵ. 864 es la puntuación máxima que se puede alcanzar en este juego. Una vez alcanzada esa puntuación la ejecución se termina.

En la tabla 2.3 se muestran los resultados de varias ejecuciones con diferentes valores de ϵ (histogramas en la figura 2.9). En ella se observa que la puntuación media máxima es mayor cuanto menor es el valor de ϵ. Aunque con $\epsilon = 0.01$ se obtienen unas puntuaciones muy buenas, el problema se observa en el *reward* obtenido por cada 100 ciclos. Se trata de un valor mucho más pequeño que el de las demás pruebas. Eso es debido a que en muchas ejecuciones la pelota ha rebotado creando algún bucle, llegado al caso de tener que parar la

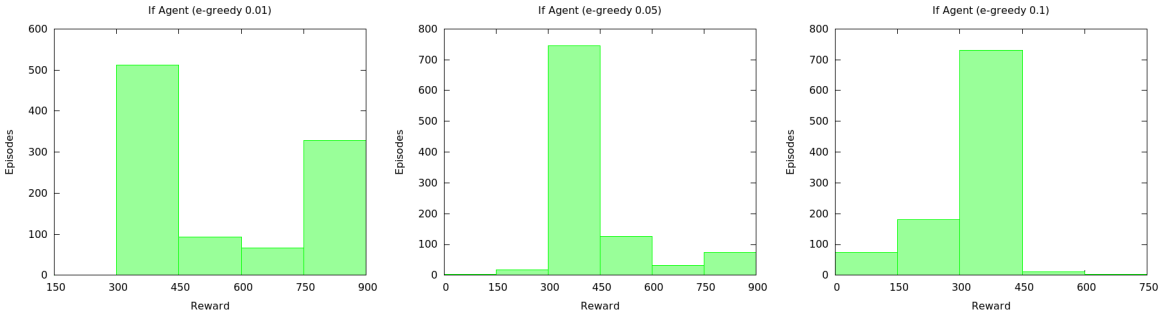


Figura 2.9: Histogramas del *reward* obtenido en los 1000 episodios jugados por el ***If Agent*** con diferentes valores de ϵ.

ejecución al superar el límite (establecido por nosotros) de 20000 ciclos.

Para probar el *If agent* usaremos $\epsilon = 0.05$. Estudios como los de Bellemare et al. [2013] y Mnih et al. [2015] utilizan ese mismo valor para sus pruebas y acabamos de ver que es un buen valor en comparación con los otros dos propuestos. El hecho de usar el mismo ϵ que otros estudios nos permitirá poder comparar resultados con aquellos que usen el mismo sistema de *rewards*.

Ruido

Hacer una acción aleatoria de vez en cuanto durante un juego puede parecer una idea bastante mala cuando se quiere poner a prueba el rendimiento de un agente, puesto que esa acción no ha sido elegida por el agente directamente. Vamos a repetir las mismas pruebas pero esta vez le introduciremos fallos de percepción, es decir, le introduciremos **ruido en la lectura de los datos** de la RAM. Los datos de la RAM son números enteros. El ruido elegido es de ± 1 (no hay valor entero más pequeño que se le pueda introducir), lo que significa que las lecturas de $Ball_x$, $Ball_y$ y $Player_x$ pueden ahora no ser correctas: pueden estar desviadas en una unidad a la derecha o a la izquierda, arriba o abajo. Esto produce partidas diferentes cada vez, en las que el agente siempre toma la decisión que cree más correcta y no una aleatoria.

Los resultados obtenidos con esta nueva técnica se encuentran en la tabla 2.4. El histograma de la puntuación se encuentra en la figura 2.10.

Ruido	**Media**	**Desviación Típica**	**Máximo**	**Mínimo**	***Reward* / 100 ciclos**
± 1	408.54	74.52	838	144	7.64

Tabla 2.4: Resultados de un **If agent** recogidos tras la ejecución de 1000 episodios con **ruido entero** ± 1 en las lecturas de la RAM.

El ruido de ± 1 parece afectar bastante al comportamiento del agente si lo comparamos con ϵ-*greedy*. Con la intención de mejorar esos resultados vamos a introducir de nuevo ruido, pero, esta vez utilizando números reales. Al igual que antes, le introduciremos ruido blanco uniforme a las lecturas de la RAM. Los resultados de diferentes niveles de ruido se encuentran en la tabla 2.5. Los histogramas, que son muy parecidos entre sí, se

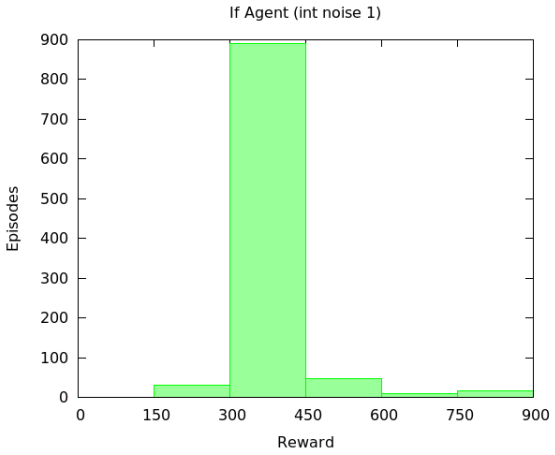


Figura 2.10: Histograma del *reward* obtenido en los 1000 episodios jugados por el **If Agent** con **ruido entero** ± 1.

Ruido	**Media**	**Desviación Típica**	**Máximo**	**Mínimo**	***Reward* / 100 ciclos**
± 0.01	**444.43**	105.83	**864**	190	**7.35**
± 0.05	442.86	**100.17**	842	**218**	7.25
± 0.1	440.46	104.03	860	144	7.22
± 1	439.39	100.27	857	179	7.26

Tabla 2.5: Resultados de un **If agent** recogidos tras la ejecución de 1000 episodios con diferentes niveles de **ruido** en las lecturas de la RAM.

encuentran en la figura 2.11.

2.5 Conclusiones

Hemos utilizado dos métodos para probar la efectividad de los agentes. Una observación interesante es la de porqué usando ruido real la puntuación media prácticamente no varía entre diferentes niveles de ruido (entre ruido 0.01 y 1 la puntuación media solo disminuye en 5 unidades). Añadiendo números reales en el ruido, lo que se consigue es que, por muy pequeño que sea el ruido, rara vez $Ball_x$ va a ser igual a $Player_x$. Esto provoca en el jugador controlado por el *If agent* un movimiento oscilatorio. Una mínima variación en esas coordenadas afecta a las reglas por las que se rige el *If agent*. En la mayoría de los casos da prácticamente lo mismo que una coordenada varíe 0.01 o que varíe 1: en ambos casos la variación puede a llevar al agente a tomar la misma acción.

La diferencia entre la puntuación obtenida con ruido entero ± 1 con respecto a la obtenida con el ruido real ± 1 es casi de 50 puntos. Esto es debido a que con ruido entero, 2 de cada 3 veces se introducía el ruido máximo que puede aplicarse. Usando ruido real es más difícil obtener un ruido de 1 ó -1, normalmente el ruido tomará valores más pequeños.

Sin ningún tipo de algoritmo de aprendizaje hemos conseguido obtener un agente que supera holgadamente

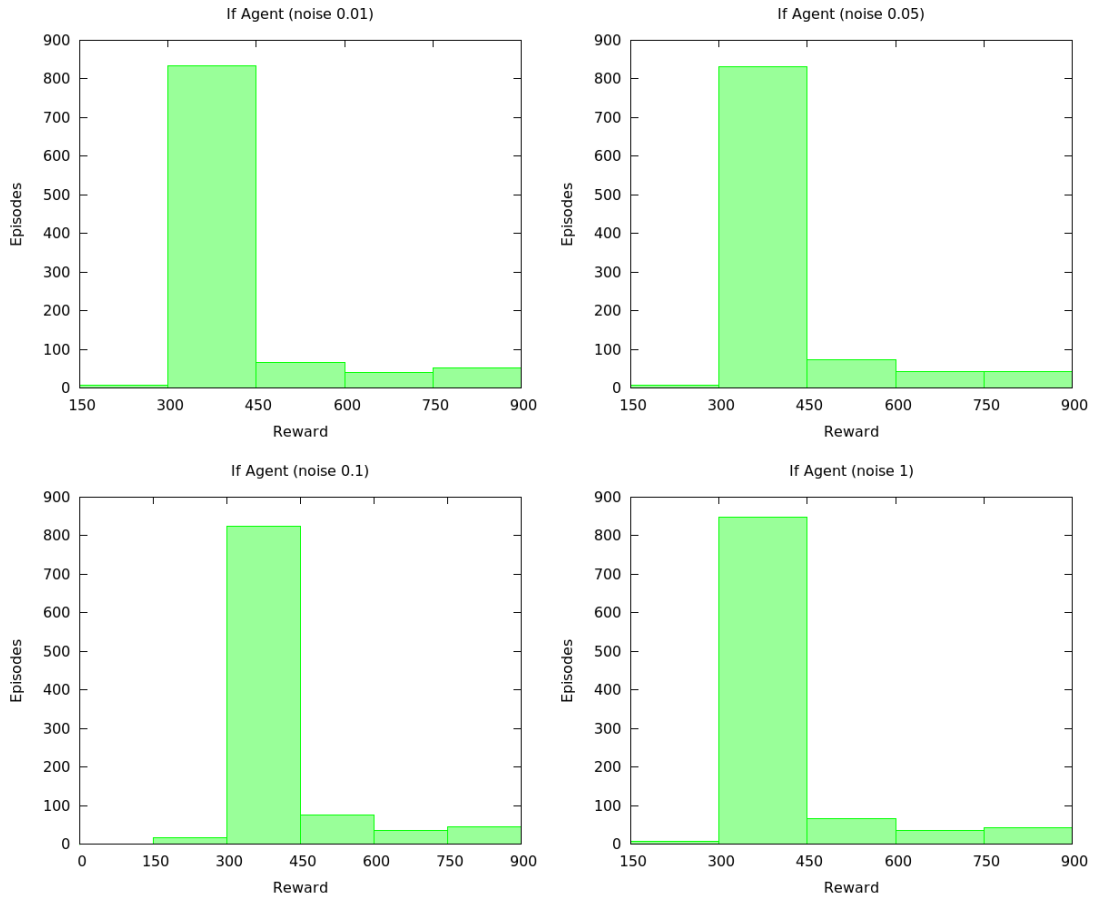


Figura 2.11: Histogramas del *reward* obtenidos tras 1000 episodios jugados por el **If Agent** con diferentes valores de **ruido real**.

a un jugador humano. El juego del *Breakout* es un juego realmente simple que puede ser resuelto con unas cuantas reglas. Sin embargo, el hecho de que sea un juego simple no lo convierte en un juego fácil. El *Random agent* nos demuestra que es difícil jugar sin saber qué es lo que tienes que hacer. De cara a futuros experimentos, cualquier agente que pueda superar el *reward* conseguido por un *Random agent* puede considerarse que ha aprendido algo.

Observando los fallos del *If agent* uno se da cuenta de que la mayoría son en los rebotes con las paredes cuando la pala se encoge (recordamos que la pala cambia su tamaño cuando la pelota rebota con la pared superior por primera vez en el turno). En la ilustración de la figura 2.12 se explica gráficamente el porqué del fallo. Se ha dado por hecho que la longitud del jugador es siempre 10 sin tener en cuenta el cambio de tamaño. Conseguir un agente utilizando el conocimiento aplicado por nosotros no es lo que se persigue en este trabajo. Si ese fuera el objetivo, se podrían aplicar ecuaciones de cinemática para poder predecir en todo momento donde caerá la pelota y conseguir así un jugador perfecto.

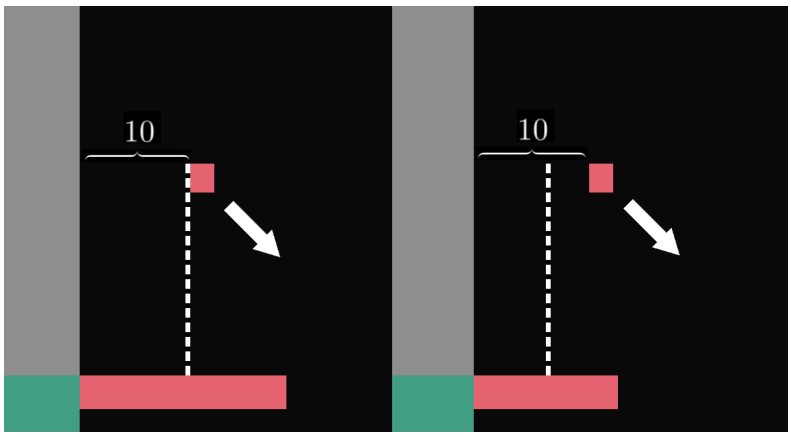


Figura 2.12: Explicación gráfica de la mayoría de los fallos del *If agent*. Suponiendo que la pelota está cayendo hacia la derecha, en el primer caso el jugador empieza a moverse a la derecha justo cuando la pelota sobrepasa la mitad de la barra. En el segundo caso, cuando $Ball_x > Player_x + 10$, el jugador comienza a moverse a la derecha, pero en ese momento la pelota ya casi le sobrepasa.

Este primer experimento nos ha sido muy útil para explorar ciertas características del juego (determinista, posiciones del jugador y la pelota). Los agentes desarrollados en este capítulo nos van a servir para compararlos con los basados en técnicas de ML.

Capítulo 3

Supervised Learning - Perceptron

Inicialmente vamos a abordar el problema desde el punto de vista del SL. En este tipo de aprendizaje, dado un vector de entrada $\mathbf{x}$ y la salida correcta y asociada a ese vector, se intenta encontrar una función f que los relacione. A grandes rasgos, intenta aproximar una función.

$$y = f(\mathbf{x}) \tag{3.1}$$

En el ámbito que nos ocupa, el objetivo de cualquier modelo de SL no es el de jugar lo mejor posible, sino el de **imitar el comportamiento** recogido por el juego de otro agente o jugador humano. Si los datos recogidos son de un jugador malo, para considerar que el modelo obtenido funciona bien, deberá de jugar igual de mal.

Para poner a prueba los modelos utilizados se han jugado 20 bolas de diferentes maneras: fallando las 20 (0/20), devolviendo 5 y fallando las 15 restantes (5/20), devolviendo 10 de 20 (10/20), devolviendo 15 de 20 (15/20) y, finalmente, acertando las 20 (20/20). Fallar más de 5 bolas significa finalizar un episodio, por lo que los 3 primeros juegos (0/20, 05/20 y 10/20) se componen de más de un episodio. Veamos más en detalle de qué constan estos 5 juegos:

- **0/20**. Se compone de 4 episodios en los que no se le da en ningún momento a la pelota (4 episodios $\cdot$ 5 bolas $= 20$). En lugar de quedarse quieto[1], lo que se ha hecho ha sido huir de la pelota, es decir, si ésta caía hacia la derecha, el jugador se pegaba a la pared de la izquierda y viceversa. De esta forma se obtienen pulsaciones que nos permitirán generar datos de los cuales aprenderán los algoritmos. El hecho de utilizar un patrón tan sencillo como es huir de la pelota nos permitirá ver, además de si consigue puntuación 0, qué tal lo imita. Por ejemplo, si nuestro algoritmo de SL obtiene la puntuación objetivo de 0 quedándose quieto no podemos considerar que ha aprendido correctamente debido a que no ha imitado el comportamiento que se encuentra en los datos.

- **5/20**. Se compone de 3 episodios: en el primero y el segundo se rompen 2 bloques en cada uno y en el tercero se rompe 1. Las pruebas que vamos a realizar van a mostar la puntuación media por episodio, lo que quiere decir que debería de estar en torno a 2 y no a 5 como el nombre del juego sugiere.

- **10/20**. En este juego pasa algo parecido al anterior. Se compone de 2 episodios con 5 puntos por episodio. Para hacer un buen trabajo, nuestro algoritmo de SL debe de obtener una puntuación de 5 y no de 10.

[1] Recordamos que el *Noop agent* programado en el capítulo 2 realizaba siempre la acción de quedarse quieto y conseguía 0 puntos.

- **15/20**. Este juego se compone de un solo episodio, por lo que esta vez sí que debe de coincidir la puntuación obtenida con el del nombre del juego: 15.

- **20/20**. Compuesto por un solo episodio que es cortado manualmente cuando se llega a una puntuación de 20. Los modelos entrenados de SL deben de ser capaces de hacer como mínimo 20 para imitar correctamente a este juego.

Lo que necesitamos para empezar es recolectar los datos de dichas partidas. Posteriormente, utilizando esos *datasets*, podremos entrenar diferentes modelos de SL con el objetivo de comprobar cómo se adaptan a los datos. En este experimento utilizaremos el **perceptrón**, uno de los primeros modelos de SL que aparecieron.

3.1 Recolectando datos

Para que los algoritmos nos imiten necesitan aprender de algo. Es por ello que nuestro primer objetivo será encontrar la forma de obtener datos de partidas jugadas por humanos. Posteriormente el perceptrón se encargará de encontrar un patrón en nuestro juego que más tarde utilizará para jugar él mismo.

Cuando iniciamos ALE con SDL nos indica que presionando la tecla 'm' el juego pasa de estar controlado por el agente inteligente programado a estar controlado por nosotros usando el teclado. El problema es que el control por un humano no nos sirve para ir recolectando los datos que nos interesan sobre la partida puesto que perdemos totalmente el control del flujo del programa, es decir, el código de nuestro agente deja de ejecutarse. Es por ello por lo que se ha programado un agente (llamado *Data agent*) que lee las teclas de la terminal, las transforma en acciones, las guarda en un *dataset* y las envía al emulador usando la interfaz del ALE.

El *Data agent* guardará los datos en un archivo de texto en formato CSV en el que cada línea contiene:

$$Player_x, Ball_x, Ball_y, Action$$

Las acciones vienen representadas por el orden del tipo de datos enumeración `Action` del ALE (2.2), es decir, `PLAYER_A_NOOP = 0`, `PLAYER_A_LEFT = 3` y `PLAYER_A_RIGHT = 4`. Puesto que los datos se van guardando en orden cronológico, se puede procesar fácilmente el archivo para obtener la velocidad de la pelota.

3.2 Perceptrón

El primer modelo elegido para este tipo de aprendizaje ha sido un perceptrón. Se trata de un modelo lineal cuya hipótesis tiene la forma

$$h(\mathbf{x}) = \operatorname{sgn}(\mathbf{w}^{\mathrm{T}}\mathbf{x}), \tag{3.2}$$

siendo $\mathbf{x}$ el vector de entradas[2], $\mathbf{w}$ un vector de pesos y sgn la función signo. La función signo utilizada viene dada por

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{si} \quad x \geq 0 \\ -1 & \text{si} \quad x < 0 \end{cases} \tag{3.3}$$

por lo que la salida de nuestra hipótesis queda restringida a dos valores, $h(\mathbf{x}) \in \{+1, -1\}$.

La obtención de un vector de pesos a partir de un conjunto de datos se realiza de una forma iterativa que sigue la siguiente regla de actualización

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y_i \mathbf{x}_i \tag{3.4}$$

en la cual $(\mathbf{x}_i, y_i) \in \mathcal{D}$ se trata de un punto mal clasificado por $\mathbf{w}_t$, es decir, $y_i \mathbf{w}_t \mathbf{x}_i < 0$, y el valor inicial del vector de pesos $\mathbf{w}_0 = \mathbf{0}$. Los mejores pesos obtenidos se guardan (*pocket algorithm*) y al final de un número determinado de iteraciones el algoritmo termina y devuelve el vector de pesos con menor error. La convergencia de este algoritmo está demostrada.

3.3 Preprocesando datos

Antes de pasarle los datos a un modelo para que entrene debemos de preprocesar los datos y adaptarlos a las características del modelo usado. A partir del conjunto de datos recolectados por el *Data agent* vamos a crear **dos *datasets***: uno de ellos para predecir las situaciones que mueven al jugador a la **derecha** y el otro para predecir las que lo mueven a la **izquierda**. Esto es necesario debido a que el perceptrón no puede decidir entre quedarse quieto, moverse a la izquierda o moverse a la derecha. El perceptrón proporciona una salida $h(\mathbf{x}) \in \{+1, -1\}$, por lo que solo puede predecir si se va a ejecutar una acción concreta o no.

Un ***dataset*** $\mathcal{D}$ está formado por pares de entradas y salidas. Los vectores de entrada $\mathbf{x}$ elegidos para entrenar los modelos estarán compuestos por 4 datos además de la característica artificial $x_0 = 1$. Esos cuatro datos son los mostrados en la figura 2.5. La característica $Diff_x$ ha sido seleccionada debido a que es la única que utiliza el *If agent* y a él le ha funcionado muy bien. Además, se le proporcionan 3 características más ($Ball_x$, $Ball_{vx}$, $Ball_{vy}$) con la idea de que puedan ayudar a mejorar las predicciones[3]. En la ecuación 3.5 definimos de forma matemática los *dataset*.

[2]Los vectores de entrada van a contener siempre $x_0 = 1$ con el objetivo de simplificar las ecuaciones. De esta forma el *bias* o umbral se encuentra en w_0.

[3]En posteriores experimentos se discutirá más en profundidad la elección de características. Por el momento usaremos los datos propuestos en esta página.

$$\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N), \text{ dónde } \mathbf{x}_i = \begin{bmatrix} x_0 = 1 \\ Ball_y \\ Ball_{vx} \\ Ball_{vy} \\ Diff_x \end{bmatrix} \in \mathbb{R}^5, y_i = Action \in \{+1, -1\} \tag{3.5}$$

Puesto que hemos decidido crear un conjunto de datos para predecir el movimiento a la derecha y otro para el movimiento a la izquierda, cuando la salida y de cada *dataset* sea $+1$ significará que se ha de aplicar ese movimiento y cuando sea -1 no se realizará dicha acción.

Si analizamos el número de pulsaciones que se han realizado en comparación con el número de veces en el que no se ha hecho nada, se observa una gran diferencia. En la mayor parte del tiempo de juego no se realiza ninguna acción. Si entrenáramos un modelo con esos datos, podría llegar a la hipótesis de clasificar todas las entradas como si fueran de la clase de la que más ejemplos hay (porque es la forma de producir menos errores). En dicho caso no se produce ningún tipo de aprendizaje.

Por lo tanto tenemos un *dataset* **sesgado** que debemos de equilibrar. El resultado de este proceso es que ahora los dos *dataset* contienen el mismo número de clases positivas y negativas. Entre las clases negativas del conjunto de datos de la acción izquierda se encuentran casos en los que hay que ir a la derecha y viceversa. Esto supone desechar muchos de los ejemplos que poseemos. Dependiendo de qué ejemplos concretos eliminemos del *dataset*, los entrenamientos de cualquier modelo podrían variar. Es por ello que para mostrar nuestros resultados utilizaremos la media de los entrenos sobre distintos *datasets*.

Por último, normalizamos los *datasets* para facilitar la tarea a muchos algoritmos de aprendizaje que funcionan mucho mejor si las características se mueven en un mismo rango. Para normalizar utilizamos *z-score*, les restaremos la media μ y dividiremos el resultado por la varianza σ. El proceso debe de repetirse por cada característica i de entrada (excluyendo obviamente $x_0 = 1$).

$$z_i = \frac{x_i - \mu_i}{\sigma_i}$$

3.4 Entrenamientos

Puesto que a partir de los mismos datos podemos generar muchos *datasets* no sesgados, las pruebas serán la media de los resultados de 10 *datasets* diferentes generados a partir de cada datos de juego: 10 con 0/20, 10 con 5/20...

En la tabla 3.1 se muestra la media de los errores de los perceptrones en cada juego. Cabe destacar que el error es muy similar en todos los entrenos de un mismo juego, lo que podría hacernos pensar que las puntuacio-

Dataset	% Error perceptrón izquierda	% Error perceptrón derecha
0/20	18.57	21.33
5/20	29.95	24.08
10/20	29.42	31.38
15/20	30.00	23.96
20/20	23.91	21.21

Tabla 3.1: Errores de los perceptrones sobre el conjunto de entrenamiento.

Dataset	Reward
0/20	0.30
5/20	1.83
10/20	11.73
15/20	14.71
20/20	5.78

Tabla 3.2: Resultados de un ***Perceptron agent*** recogidos tras 10 entrenos con cada juego y ejecutándolos durante 100 episodios con **ruido** ± 0.01 en las lecturas de la RAM.

nes obtenidas deberían de ser similares. Como veremos más adelante, la variación en las puntuaciones sí que va a ser mucho mayor.

3.5 Pruebas

Tras poseer los perceptrones entrenados con 100000 iteraciones cada uno, debemos de crear un agente que haga uso de los pesos encontrados para clasificar entradas $\mathbf{x}$ nunca vistas antes. El ***Perceptron agent*** se encargará de crear los vectores de entrada leyendo datos de la RAM y realizará clasificaciones con los perceptrones. Se moverá a la izquierda cuando el perceptrón izquierda devuelva $+1$ y el perceptrón derecha devuelva -1. Se moverá a la derecha justo en la situación contraria. Cuando los dos perceptrones devuelvan el mismo valor se realizará `PLAYER_A_NOOP`.

Los resultados de las pruebas se encuentran en las tablas 3.2 (agente con ruido ± 0.01), y 3.3 (ϵ-*greedy* con $\epsilon = 0.05$). En la figura 3.1 se representan gráficamente los resultados mostrados en cada tabla.

Dataset	Reward
0/20	0.17
5/20	2.05
10/20	8.58
15/20	9.56
20/20	4.19

Tabla 3.3: Resultados de un ***Perceptron agent*** recogidos tras 10 entrenos con cada juego y ejecutándolos durante 100 episodios con $\epsilon = 0.05$.

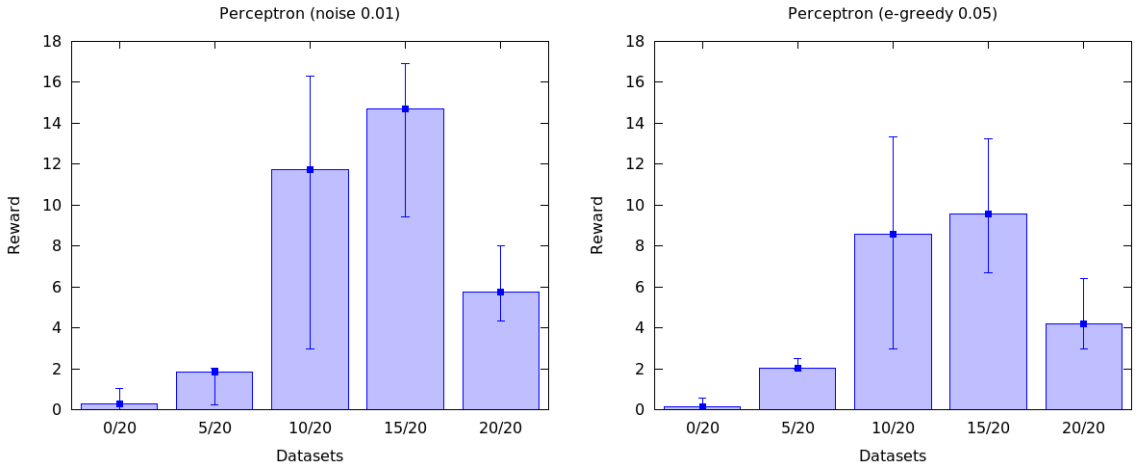


Figura 3.1: Puntuación media obtenida en cada *dataset* por un ***Perceptron agent*** con ruido ± 0.01 (izquierda) y con ϵ-***greedy policy*** con $\epsilon = 0.05$ (derecha). Las barras de error indican las medias máximas y mínimas de los diferentes entrenos.

3.6 Conclusiones

En los juegos 0/20 y 5/20 se obtiene exactamente la puntuación deseada. En el 10/20 se obtienen más puntos de los esperados, puesto que, como ya comentamos anteriormente, se espera una media de 5 por episodio y alcanza unos 10. En el juego 15/20 se alcanza el valor deseado cuando lo probamos usando ruido, pero el rendimiento es peor con ϵ-*greedy*. El resultado más sorprendente tiene lugar en el último juego, en ambos casos se queda lejos de alcanzar una puntuación de 20 o más.

Se observa que el error no es determinante a la hora de averiguar si el perceptrón se ajusta bien o no al *dataset*. En los *dataset* del juego 20/20 los errores son del 20 % y 22 %, uno de los errores más bajos, sin embargo, hemos comprobado que está muy lejos de realizar su función. Además, todos los entrenos consiguen prácticamente el mismo porcentaje de error, sin embargo las barras de los máximos y mínimos de cada *Perceptron agent* muestran una gran varianza.

Obviando esa varianza y haciendo caso a las puntuaciones medias obtenidas, se aprecia que la media de las puntuaciones posee una tendencia alcista a medida que se aumentan los aciertos en los *datasets* usados para el entrenamiento. Es necesario prestar atención en el último de todos, el de 20/20. ¿No debería conseguir mejores puntuaciones que los otros o al menos igualarlos?

El porqué del tan mal rendimiento en el juego de 20/20 reside en el patrón de juego. Visualizando los juegos y comparándolos, se puede percibir como en los otros se tendía más a seguir al balón. Sin embargo, en el juego de 20/20, a partir del golpe 7 la pelota coge mucho efecto [4] y en lugar de moverse hacia la pelota el jugador

[4] En el séptimo golpe $Ball_{vx}$ aumenta, lo que hace que la pelota rebote con las paredes varias veces antes de caer. Esto no es casualidad que ocurra en el golpe 7, es algo que ocurre siempre. En los demás juegos no se ha llegando nunca a devolver 7 seguidas, lo que hace que no se suela usar el rebote con las paredes.

se ha movido a donde acabaría la pelota tras rebotar con las paredes. Con los datos elegidos como vectores de entrada las paredes no pueden ser detectadas puesto que no contienen ni $Ball_x$ ni $Player_x$. $Diff_x$ nos proporciona la posición de la pelota relativa al jugador y sólamente a partir de esa característica no hay forma de saber dónde se encuentran las paredes.

Por lo tanto, la incapacidad de los datos de entrada de detectar las paredes unido a que los patrones más complejos son difíciles de imitar con modelos simples, son el principal motivo de la falta de rendimiento del perceptrón en ese último juego.

Para terminar, vamos a **observar e interpretar los pesos de los perceptrones** del *Perceptron agent* que mejor puntuación conseguía (figura 3.2). Se ve que los pesos que corresponden con $Ball_{vx}$ son los de valor absoluto más elevado. Eso quiere decir que se le da mayor importancia a esa entrada que al resto. El perceptrón de la izquierda le da un peso de -10 a $Ball_{vx}$, lo que indica que el perceptrón se activará cuanto más negativa sea la velocidad. En el perceptrón de la derecha pasa justo lo contrario, el peso asociado a $Ball_{vx}$ es positivo, por lo que el movimiento hacia la derecha de la pelota ayudará a la activación de la salida. Algo parecido pasa con la posición de la pelota con respecto al jugador ($Diff_x$). El hecho de que la pelota esté a la izquierda del jugador ($Diff_x < 0$) tiene importancia para el perceptrón de la izquierda, por lo que le asigna un peso negativo. Para el perceptrón de la derecha es importante que la pelota esté a la derecha del jugador ($Diff_x > 0$), por lo que se le da un peso positivo. En ambos perceptrones, cuanto más alto es el valor de $Ball_{vy}$ (más baja está la pelota) más ayuda a la activación de la salida. Es interesante darse cuenta de que el peso de la característica de entrada x_0 siempre va a ser un número entero en un perceptrón. Eso es debido a la regla de actualización del perceptrón, que suma/resta el vector de pesos con el de entradas que siempre poseen $x_0 = 1$. Eso, junto a que $\mathbf{w}_0 = \mathbf{0}$, implica que en todo momento $w_0 \in \mathbb{Z}$.

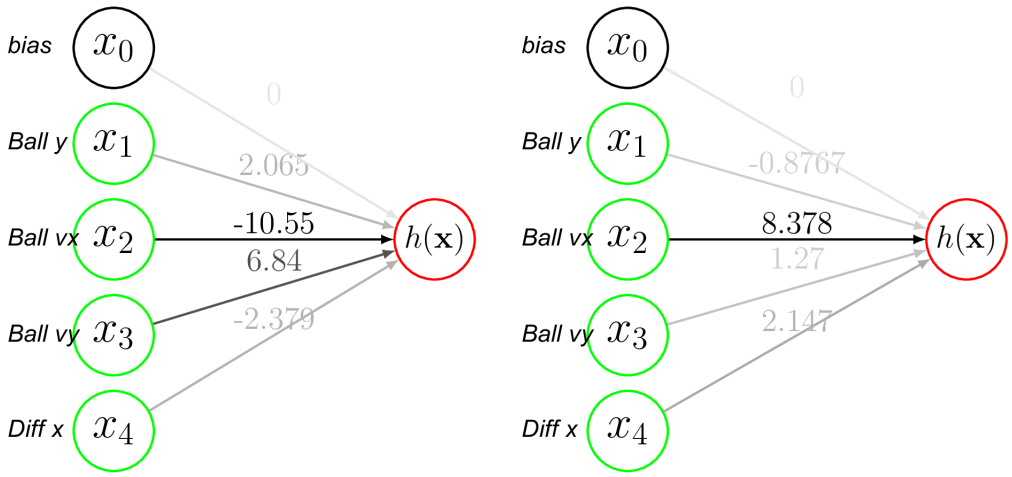


Figura 3.2: Pesos de los perceptrones de uno de los entrenos que mejores resultados daban (10/20 con una puntuación media de 16). A la izquierda el perceptrón que predice las acciones izquierda y a la derecha el que predice las de la derecha.

Capítulo 4

Reinforcement Learning - *Q-Learning*

Tras aportarle conocimiento previo a los agentes (*If agent* programado específicamente para ello, *Perceptron agent* usa los pesos de un percetrón entrenado a partir de ejemplos de juegos de humanos) vamos a abordar el problema usando técnicas de RL. De esta forma, la única información del juego que recibirá el algoritmo, además de los datos de entrada, será la puntuación (la recompensa o *reward*). A partir de ello el algoritmo se encargará de buscar las acciones que maximicen ese *reward*.

$$y = f(\mathbf{x}), r \tag{4.1}$$

En RL se reciben entradas $\mathbf{x}$ y valores r. Con eso debemos de predecir f para obtener la salida y. Los valores r son el *reward* que nos dice lo bien o mal que lo estamos haciendo. El enfoque del RL es particularmente desafiante en comparación con el SL debido a que el retraso entre una acción y su recompensa puede ser muy grande. En el SL la asociación entre una entrada y su salida es directa. El SL se podría considerar un tipo de RL en el que las acciones obtienen un *reward* inmediato y no existe una sucesión de acciones. Por lo tanto, esto hace al SL un tipo de aprendizaje que no es adecuado para obtener conocimiento a partir de la interacción con un entorno.

Un ejemplo de problema adecuado para aplicar RL sería hacer andar a un robot hacia delante. Si queremos hacer andar a un robot que tiene dos piernas no disponemos de la salida y, es decir, no sabemos cuales son las acciones exactas que el algoritmo debe de predecir para mover los motores del robot satisfactoriamente. ¿Durante cuántos segundos debemos de activar el motor de la rodilla de la pierna derecha? ¿En qué sentido? ¿Con qué intensidad? No tenemos una respuesta preestablecida para este tipo de problemas. En lugar de eso, le proporcionamos a nuestros algoritmos un valor de *reward*. En este ejemplo, el valor $reward$ sería positivo cuando el robot consiguiera moverse hacia delante y negativo cuando anduviera en la dirección equivocada o se cayera.

A diferencia de un modelo de RL, cualquier modelo de SL lo podrá hacer tan bien como bien estén los ejemplos de los que aprende. Es decir, introducirle conocimiento a mano es una buena forma de acelerar el proceso de aprendizaje, pero supone limitar el *reward*. Si el sistema es suficientemente capaz de aprender a través de su propia experiencia, puede darse el caso de que se consigan nuevos descubrimientos llegando incluso a sorprender a los creadores. En Mnih et al. [2015], su algoritmo *deep Q-network* (DQN) llega a encontrar la técnica óptima para resolver el juego del *Breakout*, cavar un túnel en uno de los lados y hacer

que la pelota quede atrapada en la parte superior rebotando con el techo y golpeando a múltiples bloques de detrás. Esa estratégica no podría haberse alcanzado si llega a ser entrenado usando datos de alguien que no haya reparado en esa forma de jugar.

Debido a las caracterísitas del RL, en este experimento el objetivo ha pasado de ser el de imitar el comportamiento de un humano al de conseguir la máxima puntuación posible.

La teoría que se mostrará a continuación ha sido extraída principalmente de Sutton and Barto [1998] y de Georgia Tech.

4.1 De MDP a RL

Un concepto que es imprescindible conocer antes de empezar a estudiar RL es el de ***Markov Decision Process*** (MDP). Los MDP son una manera de formalizar las secuencias de tomas de decisiones. No se trata de algo nuevo, ya por los años 50 *Richard Bellman* mostraba la forma de resolverlo con *dynamic programming* (DP). Sin embargo es bien sabido que resolver un MDP de gran dimensionalidad con DP (utilizando *policy iteration* o *value iteration*) es impracticable.

Algunos de los conceptos más importantes de un MDP son descritos a continuación. Todos ellos son también aplicables a los problemas de RL.

- Un sistema se encuentra en todo momento en un **estado** s. Llamamos S al conjunto que contiene todos los estados. En el juego que nos ocupa, el estado es lo que vemos en la pantalla en un instante determinado. Más adelante veremos que la representación que hemos elegido para nuestros estados no es la imagen, sino que hemos utilizado las características extraídas de la RAM. Esta forma de representar el estado está obviando la cantidad de bloques que hay en pantalla, pero facilita mucho la ejecución de los algoritmos de RL. Un estado puede ser identificado de muchas maneras diferentes: pueden usarse imágenes, vectores, usando nombres... La forma de identificar un estado no es lo que importa, lo importante es la situación que representan. Así mismo, es más normal utilizar vectores debido a la sencillez que supone generarlos y tratar con ellos.

- El paso de un estado a otro viene dado por la aplicación de una **acción** a. Llamamos A al conjunto de acciones que se pueden aplicar. Una vez decidida la acción a realizar, existe una probabilidad de que sea finalmente aplicada. Dependiendo del valor de la probabilidad hablamos de sistemas deterministas o indeterministas. En el experimento inicial comprobamos que el juego del *Breakout* se trata de un juego determinista. Esto significa que cuando se intenta realizar una acción, ésta es llevada a cabo siempre, es decir, con probabilidad 1.

- Una ***policy*** $\pi : S \rightarrow A$ es una función que relaciona un estado del conjunto S con una acción del conjunto total de acciones A. Dicho de otra forma, π dicta un comportamiento, es decir, indica las acciones que

se toman en cada uno de los estados. Llamamos ejecutar una *policy* al hecho de actuar en función de la salida proporcionada por dicha *policy*. Llamamos π^* a la *policy* óptima, es decir, a aquella que si es ejecutada maximiza el *reward*.

- *Discount factor* $\gamma \in [0, 1)$. Este factor permite al MDP centrarse más en maximizar los valores de *reward* próximos en el tiempo que en maximizar aquellos mucho más lejanos.

- Cumplen la **propiedad de *Markov***, es decir, la mejor acción a tomar en un estado s no depende de cómo hayamos llegado a dicho estado. En el *Breakout*, si no introducimos la velocidad en los estados estamos violando la propiedad de *Markov* debido a que las acciones óptimas en un instante dado dependen de a dónde se dirija la pelota.

El conocimiento extraído a partir de los datos y la falta de conocimiento del agente acerca del funcionamiento del entorno es lo que diferencia una MDP del RL. En un MDP se conocen todos los *rewards* R y todas las transiciones T (junto a sus distribuciones de probabilidad) del entorno *a priori*, por lo que encontrar la *policy* óptima requiere una planificación más que un aprendizaje. En la figura 4.1 se utiliza un esquema que condensa lo explicado en este párrafo.

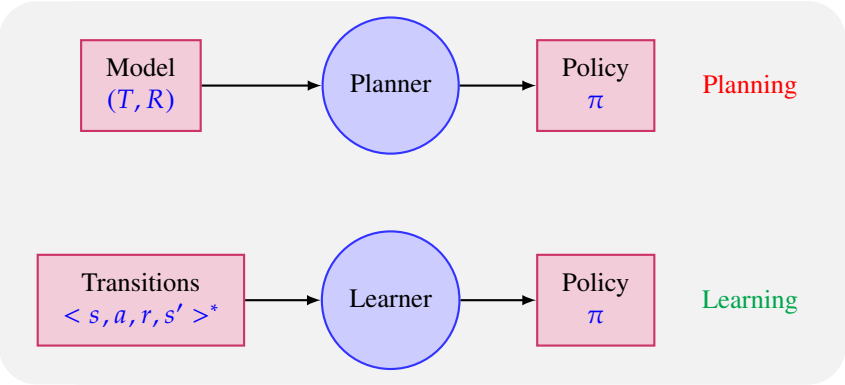


Figura 4.1: Esquema que representa la diferencia entre un MDP (arriba) y RL (abajo).

Un algoritmo de RL no conoce el modelo que describe el entorno en el que actúa, debe de intentar aproximarlo usando la experiencia. Cuando decimos que un agente aprende de la experiencia queremos decir que aprende gracias a un conjunto de observaciones

$$< s_t, a_t, r_{t+1}, s_{t+1} >^* \tag{4.2}$$

donde s_t es el estado actual, a_t la acción tomada, r_{t+1} el *reward* por tomar dicha acción y s_{t+1} el estado siguiente. El objetivo de los agentes de RL es el de interactuar con su entorno (en nuestro caso el ALE) realizando las acciones que maximicen su futuro *reward*.

4.2 *Q-Learning*

Empezaremos utilizado un conocido algoritmo de RL llamado ***Q-Learning*** [Watkins and Dayan, 1992]. Se trata de un método de resolución de problemas de RL clasificado dentro del conjunto de los algoritmos de *Temporal Difference* (TD).

Su funcionamiento se basa en el uso de la función $Q(s, a)$, conocida como la función *action-value* (acción-valor). Esto es porque devuelve el valor medio del *reward* al aplicar la acción a estando en el estado s.

$$Q(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right] \tag{4.3}$$

Llamamos $Q^*(s, a)$ a la función que maximiza el *reward*. Cuando el número de iteraciones se acerca a infinito, el algoritmo *Q-Learning* garantiza la convergencia. Puesto que esto es impracticable, el método usado para obtener el valor de dicha función es actualizarla de forma iterativa usando TD.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{4.4}$$

Este algoritmo es ***model-free***, o a veces también conocido como *value-function-based*, es decir, resuelve el problema del RL a partir de un conjunto de observaciones en lugar de construir un modelo completo a partir de ellas y usarlo para resolver el problema.

Otra de las principales características de *Q-Learning* es que se trata de un método ***off-policy***, es decir, que la *policy* π usada para la toma de decisiones durante el aprendizaje es diferente a la que se usa para actualizar el valor de la función $Q(s, a)$. Esa es de hecho la principal diferencia entre este algoritmo y *Sarsa*. Este otro conocido algoritmo de RL usa las acciones tomadas para actualizar el valor de la función *action-value*.

Es de vital importancia en estos tipos de algoritmos que se explore muy bien el conjunto de estados. Hay que tratar de buscar el equilibrio entre **exploración y explotación**. En RL se utiliza el término exploración al hecho de ejecutar acciones que no son las mejores, con el objetivo de producir actualizaciones de la función $Q(s, a)$. La exploración es lo que diferencia al RL de cualquier otro tipo de método de ML. En contrapartida, el término explotación hace referencia al hecho de hacer uso de la mejor acción que hemos encontrado hasta el momento (*greedy action*). El éxito de todo algoritmo de RL depende del equilibrio de estas dos caracteríticas. En Sutton and Barto [1998] se comentan muchos criterios de selección y lo difícil que es saber cual de ellos funciona mejor. Utilizaremos ϵ-*greedy* por su sencillez y buenos resultados que aporta.

Las tareas a las que se aplican los problemas de RL pueden ser **continuas o episódicas**. Las tareas que no tienen final son las consideradas continuas. El famoso problema del *cart-pole balancing* [Sutton and Barto, 1998] se trata de una tarea continua. Otro tipo de tareas, como el *Breakout* o cualquier juego arcade en general,

Estado s	**Acción** a	**Valor** $Q(s,a)$
⋮	⋮	⋮

Tabla 4.1: Representación de la tabla que guarda el *Q-Table agent*.

son llamadas episódicas puesto que acaban tras un número variable de pasos[1].

4.3 *Q-Table agent*

Antes de comenzar hay que definir qué és un estado para nosotros. Utilizaremos los mismos 4 componentes que utilizamos en las entradas de métodos de aprendizaje supervisado: $Ball_y, Ball_{vx}, Ball_{vy}, Diff_x$. Esta 4-tupla define completamente el estado de la pelota respecto al jugador en todo momento.

El *Q-Learning* implementado basa su funcionamiento en una tabla en la que cada combinación de estados y acciones tiene un valor asociado. Inicialmente, todo valor es 0. Posteriormente dicho valor se va actualizando utilizando la regla de actualización. La tabla 4.1 representa la forma en la que el *Q-Table agent* guarda la función $Q(s,a)$ para ir actualizando sus valores. Esa tabla es llamada ocasionalmente en la literatura *Q-Table*.

Debido a la implementación del *Q-Learning* con una tabla, no vamos a aplicar ningún tipo de normalización en este agente puesto que carece de sentido. A cada par estado-acción le correspondería otro par estado-acción normalizado, por lo que las entradas en la tabla serían las mismas. Como ya hemos comentado a la hora de definir lo que se entiende por estado en un MDP, da igual la forma de nombrar los estados, lo importante es lo que representan.

Relacionado con la forma de guardar los estados está el motivo por el cuál este tipo de agente no puede ser puesto a prueba con ruido en la RAM. Si los estados tienen ruido puede haber infinitos vectores que representen al mismo estado. En la tabla, cada uno de esos vectores serían guardados como si fueran todos estados diferentes. Esto es un problema debido a que a la hora de buscar un estado durante el juego en la tabla no se encontraría y devolvería el valor 0.

4.4 Discretización

La discretización es una técnica utilizada en entornos continuos para poder aplicar en ellos algoritmos de RL. Si el estado está compuesto por números reales, es casi imposible que se vuelva a estar justamente en dicho estado (deberían de coincidir todas las cifras decimales para que se considerara como el mismo estado). Sin volver a pasar por el mismo estado ni una vez, la función $Q(s,a)$ no se actualizaría nunca y no se aprendería nada.

[1] Hemos comprobado que sin ϵ-*greedy* o ruido los agentes podrían no terminar nunca. El límite máximo de pasos establecido es de 20000, más que suficiente para pasarse los dos niveles del juego.

Aunque el juego que nos ocupa no posee valores reales, posee una gran multitud de estados que son muy parecidos entre sí. Por ejemplo, no hay prácticamente diferencia entre que la pelota esté a 100 de distancia del jugador que si está a 101. Por ello vamos a introducir un factor de discretización que se encargará de dividir las distancias $Diff_x$ y $Ball_y$. Las velocidades no las discretizaremos debido a que sólo varían entre -5 y 5 aproximadamente. Una $Diff_x$ de 100 pasaría a valer 50 si usamos un factor de discretización 2. Del mismo modo, una $Diff_x$ de 101 pasaría a ser 50 también (obviamente se tratan de divisiones de números enteros).

En la figura 4.2 se muestran cuatro curvas de aprendizaje con diferentes valores de discretización. En 4.3 se muestra un *heat map* con los factores de discretización entre 1 y 30. El valor representado en el *heat map* es el valor de la línea de tendencia en el punto $x = 2000$, donde terminan las ejecuciones. Los datos son la media de 5 ejecuciones a 2000 episodios. Según los resultados, un valor de discretización entre 10 y 20 sería el más adecuado para conseguir un aprendizaje mayor en 2000 episodios.

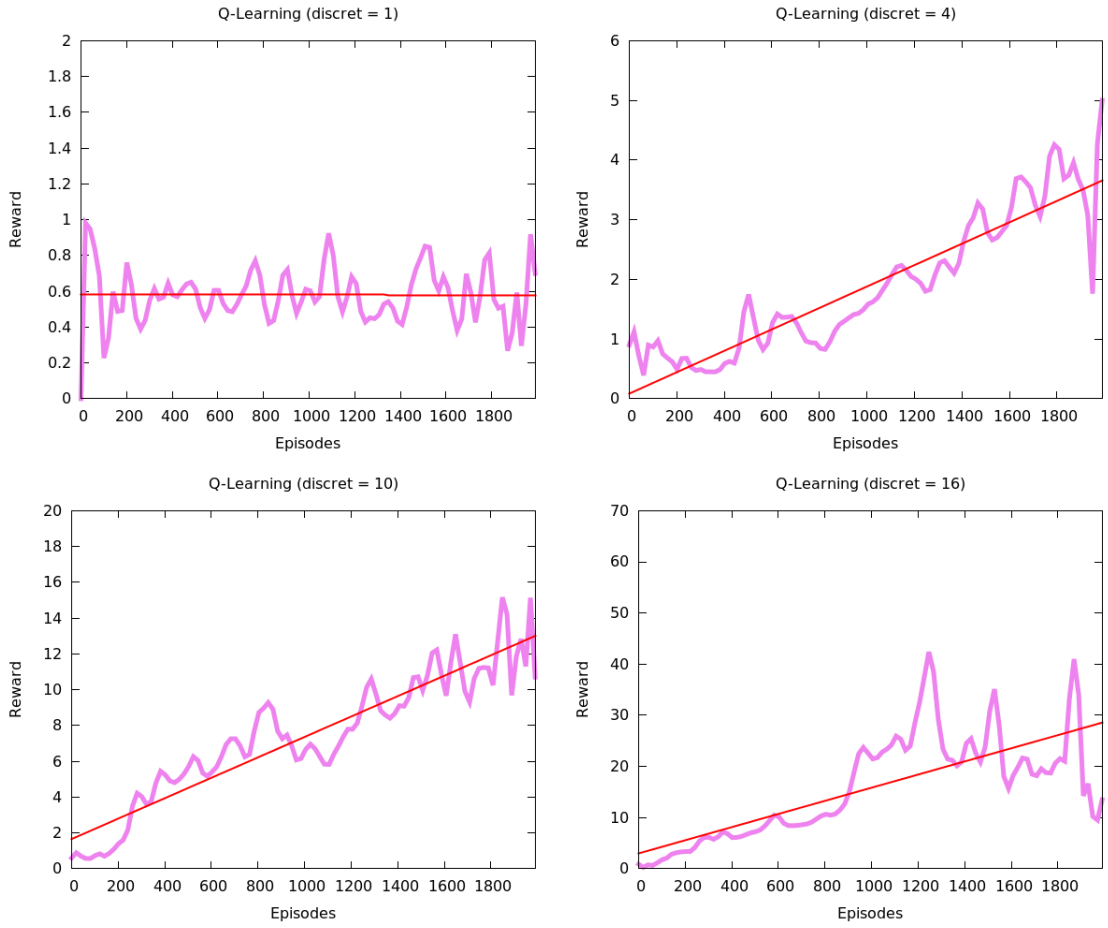


Figura 4.2: Curvas de aprendizaje de un *Q-Table agent* con 4 factores de discretización distintos: 1, 4, 10 y 16. Se muestra una media móvil del *reward* de los primeros 2000 episodios ejecutados con $\epsilon = 0.05$, $\alpha = 0.2$ y $\gamma = 1$. El ancho de la ventana usada para realizar la media móvil es de 200 episodios. La línea roja marca la tendencia.

Gracias a la discretización se consiguen reducir el número de estados de la tabla (ver figura 4.4), por lo que

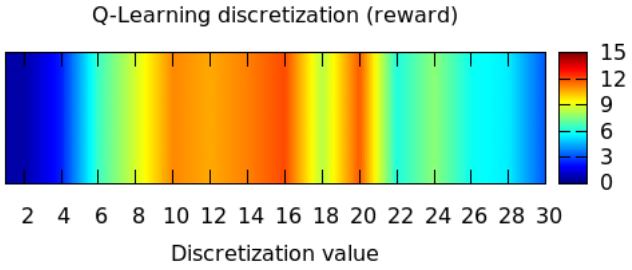


Figura 4.3: Puntuación obtenida por un *Q-Learning agent* con diferentes factores de discretización, durante 2000 episodios y con $\epsilon = 0.05$.

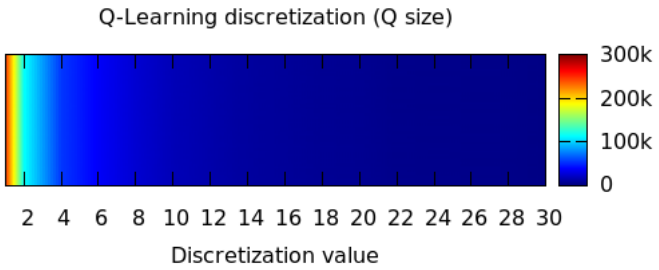


Figura 4.4: Número de entradas de la función $Q(s,a)$ en función del factor de discretización utilizado.

aumenta la probabilidad de volver a pasar por uno de esos estados y actualizar así el valor de la función $Q(s,a)$ con más frecuencia.

Como es de suponer, el tiempo que usan las pruebas es mayor cuanto más *reward* se consigue (ver figura 4.5). Cuando hablamos de tiempo nos referiremos al número de pasos (número de acciones) realizados durante la ejecución. De esta forma evitamos que puedan inferir las posibles tareas simultáneas que estuvieran ejecutándose en el ordenador a la hora de realizar los tests.

4.5 Ajustando α y γ

Q-Learning tiene dos parámetros a los cuales debemos de dar valor. Es necesario ajustar correctamente esos valores si queremos que se obtengan buenos resultados.

El parámetro α es el llamado ***learning rate***, es decir, la velocidad a la que aprende. Su valor debe de estar comprendido entre 0 y 1. Si lo ponemos a 0 estamos indicando que no queremos que nuestro algoritmo aprenda nada y los valores de la función $Q(s,a)$ nunca cambiarán. Si le damos el valor 1 le indicamos que debe de aprender mucho en cada actualización de $Q(s,a)$. Aunque lo de aprender rápido puede resultar atractivo, hay que tener cuidado porque corremos el riesgo de olvidar lo aprendido.

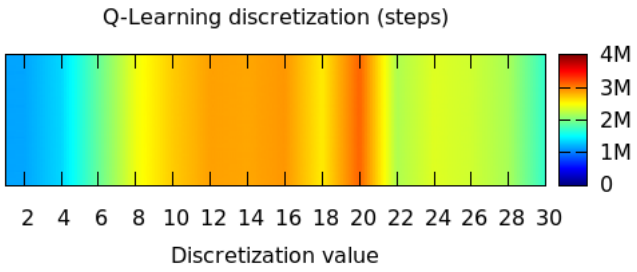


Figura 4.5: Número de acciones realizadas (en millones) durante las 5 repeticiones de las ejecuciones de 2000 episodios.

El factor γ es el llamado ***discount factor***, el factor de descuento. Al igual que α su valor se debe encontrar entre 0 y 1. Cuanto mayor sea el valor de γ más visión va a tener nuestro algoritmo, es decir, va a ser capaz de asociar acciones a futuros rewards (*long term reward*). Antes de realizar ningún experimento se puede esperar que los valores más óptimos de γ deberán de ser altos puesto que en el Breakout los *rewards* no se obtienen inmediatamente después de aplicar una acción, sino que es tras la ejecución de varias acciones posteriores cuando se sabe si esa acción era buena.

Vamos a realizar un estudio para averiguar qué rango de valores para ambos parámetros es el más prometedor. Este tipo de parámetros, que realmente no afectan al modelo más que en su entrenamiento, son conocidos como ***hyperparameters***. Para encontrar ese rango de valores utilizaremos el método llamado ***grid search***. El objetivo es probar nuestro modelo entre un rango de valores definidos por nosotros. Vamos a ir incrementando en 0.1 el valor de α desde 0 hasta 1. Por cada valor de α hacemos lo mismo con γ. Con cada par de valores ejecutaremos el algoritmo *Q-Learning* durante 2000 episodios (con una ϵ-*greedy policy* con $\epsilon = 0.05$ y un factor de discretización de 10). Los resultados los dispondremos en forma de matriz y la representaremos en un *heat map*.

Se pueden observar en la figura 4.6 los resultados. Las zonas más cálidas son los valores de esos parámetros que mayor *reward* consiguen. El valor óptimo de α se encuentra entre 0.1 y 0.3 y el de γ entre 0.9 y 1.

Si realizamos otro mapa que mida esta vez la cantidad de entradas que posee la función $Q(s, a)$ obtenemos lo mostrado en la figura 4.7. Las funciones $Q(s, a)$ que más entradas tienen coinciden con los parámetros que proporcionan mayor puntuación. Eso significa que durante el entrenamiento se han visitado más estados y por lo tanto ha aprendido mejor. Como es lógico, con valores $\alpha = 0$ o $\gamma = 0$ no se realiza ningún tipo de aprendizaje, de ahí el color azul del borde izquierdo e inferior.

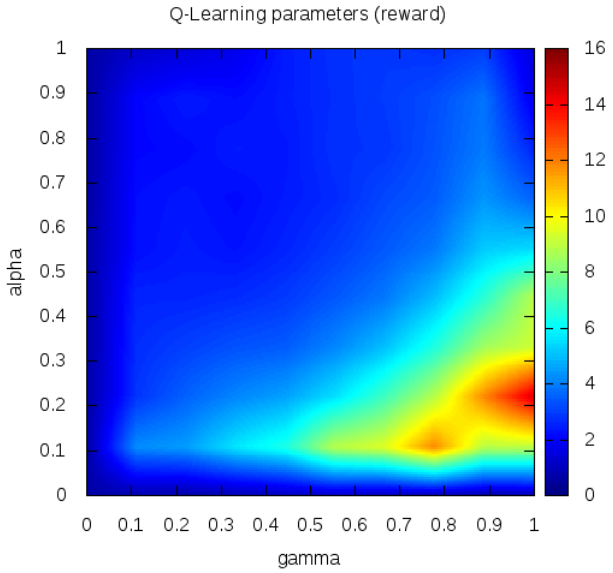


Figura 4.6: *Heat map* con una *policy* ϵ-greedy con $\epsilon = 0.05$.

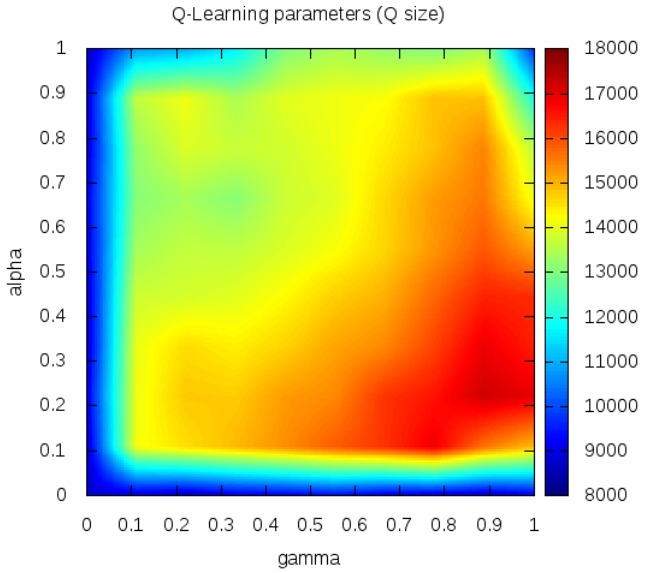


Figura 4.7: *Heat map* del número de entradas que posee $Q(s, a)$ en función del valor de α y γ.

4.6 Pruebas

Las pruebas realizadas en la selección de *hyperparameters* se han hecho utilizando únicamente 2000 episodios. El objetivo era encontrar los rangos óptimos de α, γ y el factor de discretización de manera que ahora podemos dedicarle mucho más tiempo a las pruebas que pensemos que van a ser más prometedoras.

Vamos a coger unos cuantos valores de parámetros y entrenarlos durante muchos más episodios: 20000. La tabla 4.2 muestra los parámetros elegidos y los resultados.

α	γ	**Discretización**	**Media**	**Desviación**	**Máx**	**Mín**	**Tendencia**
0.2	0.99	8	12.19	7.23	51	0	16.22
0.2	0.99	10	13.87	8.67	**218**	0	16.10
0.2	0.99	16	9.93	7.00	65	0	10.78
0.2	1	10	12.42	8.07	75	0	17.03
0.2	0.9999	10	14.10	8.88	73	0	19.99
0.1	0.99	10	**17.87**	11.24	81	0	**25.61**
0.1	0.9999	1	1.24	**1.22**	13	0	1.56

Tabla 4.2: Resultados de un ***Q-Table agent*** tras ser entrenado con 20000 episodios con $\epsilon = 0.05$.

Como se puede observar, en lugar de $\gamma = 1$, en muchos casos se ha utilizado un número inferior para que los *rewards* vayan disminuyendo con el tiempo. El valor 1 no hace decaer los *rewards* con el tiempo, es decir, que tiene en cuenta todos los futuros *rewards*. Vamos a utilizar otros valores como 0.99 y 0.99999, que se encuentran muy cercanos a 1, pero nos permitirán centrarnos en los futuros *rewards* más cercanos en el tiempo. Se observa que valores cercanos a 1 aportan mejores resultados que el propio 1.

El *Q-Table Agent* entrenado con $\alpha = 0.1$, $\gamma = 0.99$ y discretización 10 ha conseguido una de las tablas que mayor puntuaciones obtiene. En la figura 4.9 se muestra la curva de aprendizaje de este entreno. La tabla extraída es la que corresponde al máximo absoluto de la curva, momento cercano al episodio 16000. Haciendo uso de esa tabla se ha ejecutado el *Q-Table* durante 1000 episodios (desactivando las actualizaciones de la *Q-Table* para que no se modifique). Los resultados se encuentran en la tabla 4.3.

Media	**Desviación Típica**	**Máximo**	**Mínimo**
48.73	13.60	89	16

Tabla 4.3: Resultados de la ejecución de 1000 episodios de un *Q-Table agent* con una tabla sacada tras 16000 episodios de entrenamiento con $\alpha = 0.1$, $\gamma = 0.9999$ y discretización 10.

4.7 Conclusiones

Tras ejecuciones largas, las curvas de aprendizaje dejan de crecer y empiezan a oscilar. Es importante ver en qué momento guardar el valor de la función $Q(s, a)$ debido a que si tomamos el valor de la *Q-Table* en uno de los valles de la curva de aprendizaje, el rendimiento será peor que al cogerlo de la cumbre más alta de la

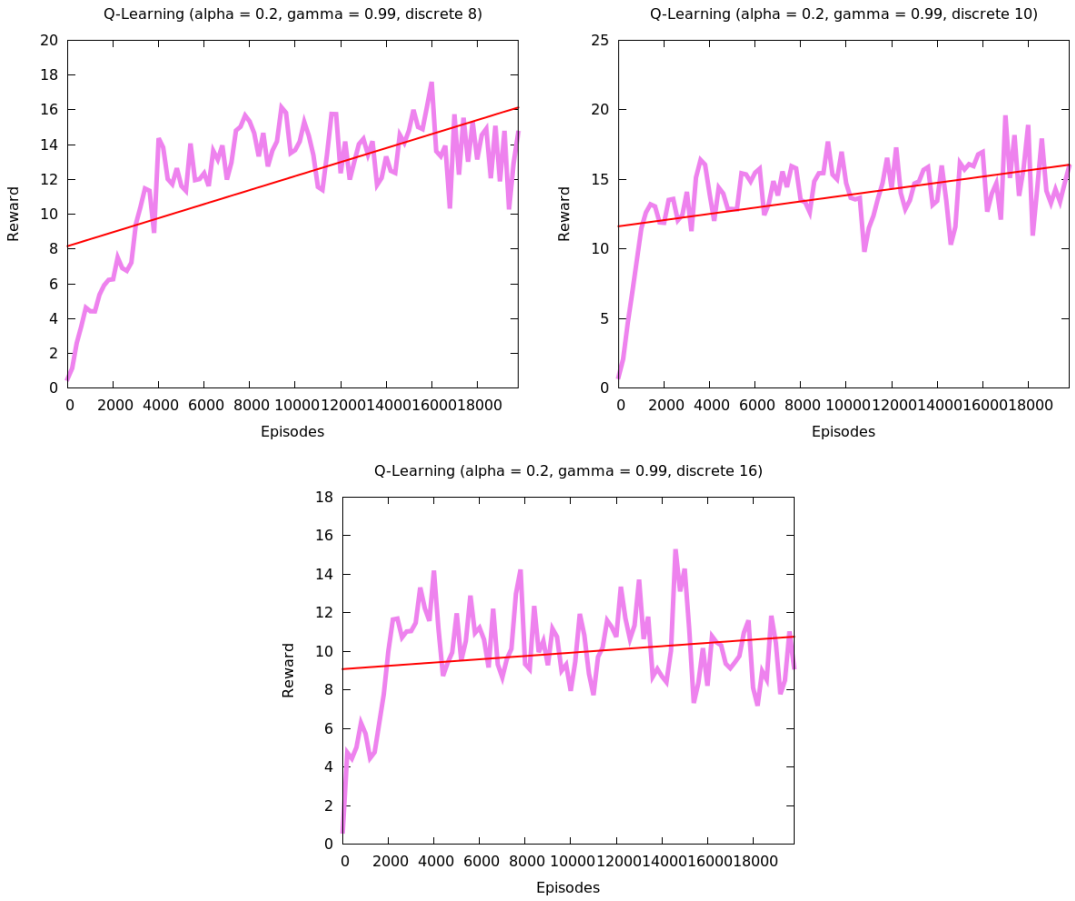


Figura 4.8: Curvas de aprendizaje de 3 de las pruebas de la tabla 4.2. En ellas se usa $\alpha = 0.2$, $\gamma = 0.99$ y se varía el factor de discretización.

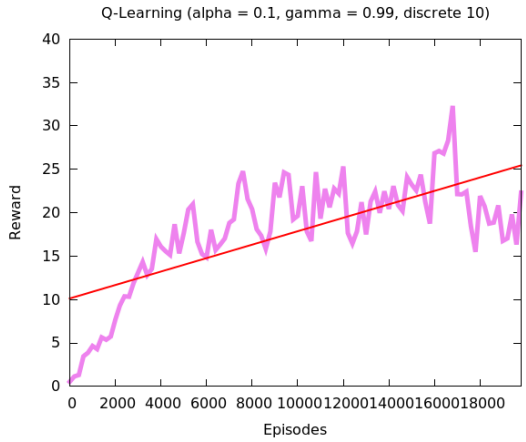


Figura 4.9: La mejor *Q-Table* obtenida es la sacada del punto máximo de la curva de aprendizaje de este agente. Que la línea esté en 30 no significa que ese vaya a ser su *reward* medio puesto que la curva es una media móvil con un ancho de ventana de 200. El *reward* medio obtenido con la tabla del episodio 16000 se muestra en la tabla 4.3.

curva.

La discretización ayuda bastante. De entre los valores probados, cuanto más grande es el factor de discretización antes mejora su puntuación. La contrapartida de esto es que a largo plazo no se consigue pasar de cierto límite. Por ejemplo, con valor de discretización 30, en menos de 200 episodios alcanza un *reward* medio de 3, pero tras los 2000 episodios su tendencia sigue siendo de 3. Con un valor de discretización de 6, en el episodio 200 solo tiene un *reward* medio de 1, pero a largo plazo, en el episodio 2000 alcanza una tendencia de 7. Sin usar discretización, aún utilizando más de 20000 episodios, no se ha conseguido superar a un *Random agent*.

Por el momento, el algoritmo de RL desarrollado está bastante lejos de las marcas obtenidas por el *If Agent*. Uno de los principales defectos que se encuentran en su juego es un balanceo constante. Los motivos que provocan ese balanceo son muy similares a los que producían ese balanceo en el *If agent*. Con los datos que le pasamos como entrada no puede detectar dónde se encuentran las paredes. Esa "falta de visión" le provoca algunos fallos en los rebotes y fallos debido al balanceo. Pero el principal problema se encuentra en el momento en el que se golpean bloques que hacen subir la velocidad de la pelota. En dichos casos, aún después de realizar entrenos largos de más de 20000 episodios, el jugador se suele quedar parado sin saber lo que hacer. Eso es debido a que nunca ha visto ese caso antes y los valores de la tabla para ese estado son 0 [2]. Los casos en los que se golpean bloques que suben la velocidad suele ser a partir de los 50 puntos. Puesto que la curva de aprendizaje del agente fluctúa entre 10 y 50 puntos, es raro que llegue a ver los casos en los que la pelota aumenta su velocidad.

Los resultados obtenidos gracias a estos experimentos nos deja abiertos varios caminos:

- Mejorar las características de entrada para que le permitan detectar las paredes.
- Evitar fluctuaciones tan elevadas en las curvas de aprendizaje.
- Buscar técnicas que nos permitan acelerar el aprendizaje (pero sin perder rendimiento de nuestro agente). Una de esas técnicas podría ser buscar aproximaciones de la tabla para que pueda funcionar igualmente con velocidades más elevadas.

[2] En caso de que un estado tenga valor 0 para cada una de las acciones se realiza `PLAYER_A_NOOP`.

Capítulo 5

Características de entrada

La elección de características es uno de los pasos más importantes a la hora de aplicar un algoritmo de ML. Dependiendo de los datos elegidos para ser utilizados en el proceso de aprendizaje, el resultado puede pasar de ser un fracaso a ser un éxito. Dependiendo del ámbito en el que se esté aplicando ML, el proceso de elección de características puede ayudar que sea llevado a cabo por un especialista en la materia. En nuestro caso, un simple juego, no requerirá de ningún experto, pero sí conocer bien el juego y reflexionar acerca de cómo lo intentamos resolver nosotros como humanos.

El vector de entrada **x** que le pasábamos al perceptrón y que usábamos como estado del *Q-Learning* ha sido elegido de forma poco justificada. ¿Porqué usar $Diff_x$ en lugar de $Player_x$ y $Ball_x$? ¿Sólo porque el *If agent* jugara tan bien usando solo $Diff_x$? ¿Por qué no usar entonces $Diff_x$ y ninguna otra característica más?

Dar respuesta a estas preguntas sin hacer pruebas es muy difícil, así que en este experimento vamos a probar unas cuantas combinaciones de características de entrada (aquellas que por intuición puedan parecer más prometedoras) y dejaremos que los resultados muestren cuáles son más favorables. Vamos a repetir algunas de las pruebas que ya hemos realizado usando ahora diferentes características de entrada. Repetiremos los experimentos del perceptrón y el *grid search* del *Q-Table agent*. Las combinaciones que usaremos se muestran resumidas en la tabla 5.1. A continuación las comentaremos con más detalle.

El estado `nobxpx` es exactamente el mismo que llevamos utilizando hasta ahora, carece de las coordenadas absolutas que nos permitirían saber dónde están las paredes. Los siguientes 3 sí que nos aportan esas coordenadas. En `nobx` se ha optado por eliminar la coordenanda de la pelota, pero mantenemos la coordenada del jugador y $Diff_x$, que sería suficiente para encontrar la posición absoluta de la pelota. En `nopx` es exáctamente lo mismo solo que ahora eliminamos la coordenada del jugador y dejamos la de la pelota. En `nodx` usamos la

Nombre	$Ball_x$	$Ball_y$	$Ball_{vx}$	$Ball_{vy}$	$Player_x$	$Diff_x$
nobxpx	☐	☑	☑	☑	☐	☑
nobx	☐	☑	☑	☑	☑	☑
nopx	☑	☑	☑	☑	☐	☑
nodx	☑	☑	☑	☑	☑	☐
dx	☐	☐	☐	☐	☐	☑

Tabla 5.1: Diferentes estados que usaremos en las pruebas. De ahora en adelante los referenciaremos con los nombres aquí descritos. Los nombres son muy descriptivos, los 4 primeros señalan las características que no son incluídas y el último señala la única característica incluída.

coordenada absoluta de la pelota y la del jugador. En este caso suprimimos la característica $Diff_x$ puesto que es redundante. En `dx` solo usamos $Diff_x$, tal y como hacía el *If agent*, sin tener en cuenta la velocidad de la pelota ni las posiciones absolutas del jugador y la pelota.

5.1 Pruebas perceptrón

Se han repetido las pruebas realizadas con el perceptrón. Los resultados se han dispuesto en forma de gráficos de barras (ver figura 5.1).

5.2 Pruebas *Q-Learning*

Vamos a repetir las pruebas del *Q-Table agent* para ver como se comporta el *Q-Learning* haciendo uso de las nuevas combinaciones de características. Se ha repetido el *heat map* usado para elegir valores de α y γ. El factor de discretización utilizado ha sido de 10. Para ahorrar tiempo de cómputo se han eliminado las pruebas cuando $\alpha = 0$ o $\gamma = 0$, puesto que con dichos valores ya se ha comentado que no se consigue aprendizaje. Los gráficos resultantes se encuentran en la figura 5.2.

5.3 Conclusiones

Tras unos días de pruebas vemos que se obtienen resultados en los que se consigue aprendizaje, pero que aún así no son comparables con los obtenidos con las características que utilizábamos anteriormente.

En el **perceptrón**, los resultados más nefastos son los conseguidos usando el estado `dx`. Esa sola característica no es capaz de describir correctamente el patrón de juego de una persona. Quizás con patrones como los del *If agent* sí que se puedan conseguir mejores resultados usando solo esa característica, pero nosotros como humanos nos quedamos mucho más tiempo sin hacer movimientos, y cuando los hacemos no nos regimos únicamente por la posición relativa de la pelota al jugador.

Con los estados `nobx`, `nopx` y `nodx` los resultados obtenidos son muy similares. En cada uno de ellos se ha añadido una nueva característica al vector de estados con la idea de tener la posición absoluta de la pelota y del jugador, por lo tanto, no sorprende que los resultados con esos estados sean tan parecidos. Cabe destacar que las puntuaciones del juego 15/20 son bastante bajas en comparación a las conseguidas con el estado `nobxpx`.

En el ***Q-Learning***, aún con los mismos episodios que antes, los *rewards* obtenidos son muy bajos. Debida a la mínima diferencia que hay entre las puntuaciones obtenidas con los diferentes estados, es difícil decir porqué uno es mejor que otro. Menos en `dx`, en todas las nuevas combinaciones de características utilizadas hemos añadido una nueva variable, lo que disminuye la probabilidad de volver a caer en un mismo estado.

La **maldición de la dimensionalidad** (*the curse of dimensionality*) es una expresión acuñada por *Bellman*

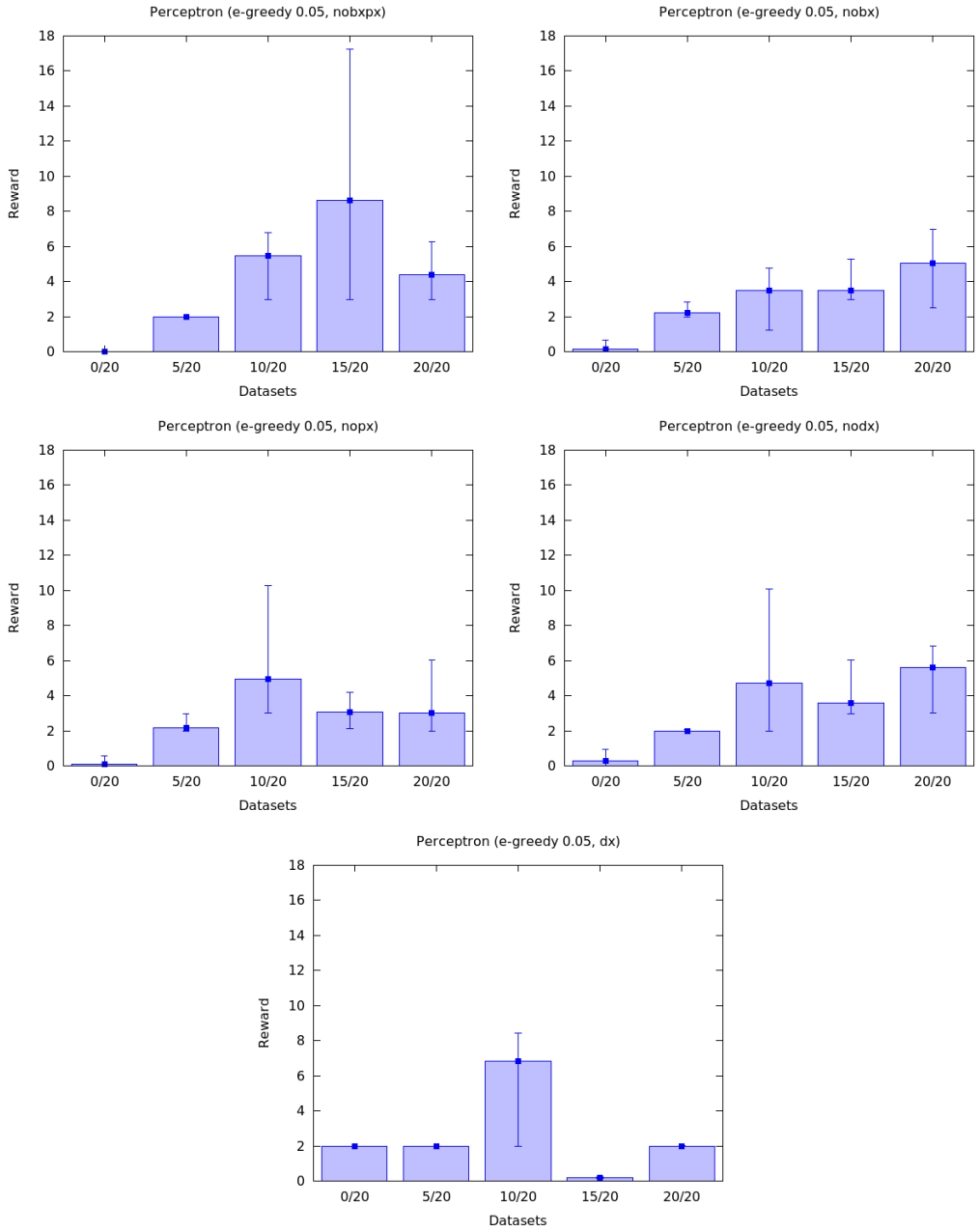


Figura 5.1: Gráficas que representan las puntuaciones del perceptrón entrenado con los diferentes estados.

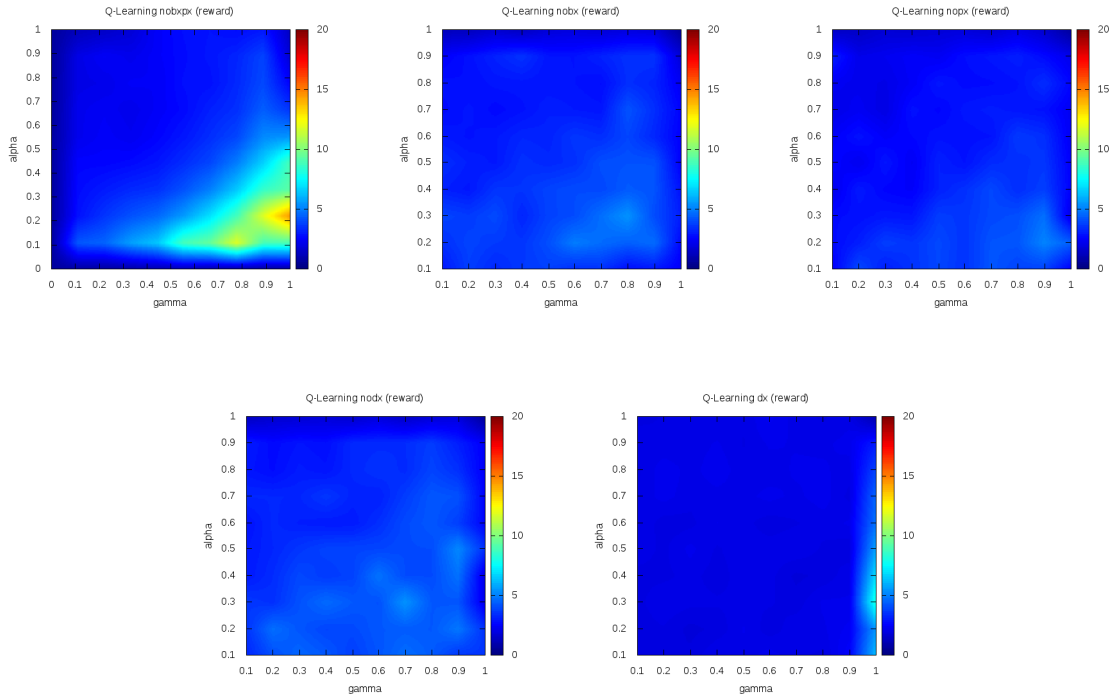


Figura 5.2: *Heat maps* de la selección de parámetros de un *Q-Table* con diferentes estados.

que describe el problema causado por el crecimiento exponencial del número de estados al añadir una nueva dimensión matemática a nuestro vector descriptor del estado.

En el juego del 3 en raya por ejemplo, el conjunto de estados sería todas las posibles configuraciones del tablero. Cada una de las 9 casillas del tablero pueden estar vacías, tener una $\times$ o tener un $\circ$, por lo que el número de estados es $3^9 = 19683$ (en realidad serían menos, puesto que entre todos esos estados los hay que nunca podrían llegar a darse, como, por poner un ejemplo extremo, todas las casillas rellenas por el mismo jugador). Ese número de estados es relativamente pequeño, permitiendo a un ordenador poder generarlos todos varias veces sin ningún tipo de problema. Esto produce una relativamente rápida convergencia del *Q-Learning* implementado con una tabla.

En el problema que nos ocupa, usando un estado `nobxpx` se obtiene un total aproximado de 3 millones de estados. El número de entradas de nuestra tabla es todavía mayor debido a que tendremos repetido cada estado una vez por cada acción, es decir, tendremos el triple de entradas. Redondeando, con un estado del tipo `nobxpx` se obtiene un número de estados cercano a los 10 millones. El cálculo aproximado es:

$$\begin{aligned} states &= |S_{\texttt{nobxpx}}| \cdot |A| \\ &= |Ball_y \times Ball_{vx} \times Ball_{vy} \times Diff_x| \cdot |A| \\ &\approx 210 \cdot 8 \cdot 8 \cdot 230 \cdot 3 \approx 10^7 \end{aligned}$$

Haciendo uso de un factor de discretización 10 se consigue reducir ese total a $\frac{210}{10} \cdot 8 \cdot 8 \cdot \frac{230}{10} \cdot 3 \approx 10^5$ estados. En la práctica, si observamos el tamaño de la función $Q(s,a)$ obtenida durante el entreno cuya curva de aprendizaje muestra la figura 4.9, vemos que tiene solo 25000 estados. Esto tiene sentido debido a que las velocidades de la pelota no llegaron a ser muy elevadas, puesto que de 50 puntos no se solía pasar. Tampoco llegó a enviarse la pelota por encima de los bloques, por lo que el rango de $Ball_y$ disminuyó prácticamente a la mitad.

En las nuevas representaciones elegidas para los estados, con la idea de proporcionar la información necesaria a los algoritmos para que detecten paredes, se le ha añadido una nueva característica. Esto aumenta considerablemente el número de estados totales del juego. Por ejemplo, para `nodx`, el número de estados viene dado por $150 \cdot 210 \cdot 8 \cdot 8 \cdot 150 \approx 300$ millones. El número de estados es enorme y para que converja el algoritmo debe de pasarse muchas veces por cada uno de esos estados. Esto hace impracticable el uso del *Q-Learning* basado en tablas.

Muchos de esos estados son similares entre sí, por lo que nos ayudaría una forma de encontrar similitud entre estados y utilizar así los valores de la función $Q(s,a)$ para varios estados. Las llamadas técnicas de *function approximation* ayudarían mucho para esa tarea.

Capítulo 6

KNN y *Neural Networks*

Anteriormente ya hemos utilizado el SL para medir el rendimiento de un perceptrón en diferentes *datasets* con diferentes aciertos. La frontera de decisión de un perceptrón es un hiperplano, por lo que solo sería capaz de encontrar un vector de pesos de error 0 en conjuntos de datos linealmente separables. En este experimento vamos a utilizar *K-Nearest Neighbors* (KNN) y *artificial neural networks* (ANN) con el objetivo de comprobar los beneficios del uso de fronteras de decisión más complejas.

En lugar de usar los datos que se usaron en el anterior experimento, vamos a usar un nuevo conjunto de datos que corresponden a un juego en el que se consiguieron aproximadamente 400 puntos. Para referirnos a este juego utilizaremos de aquí en adelante el nombre `400points`. Se trata de un conjunto de datos más grande que los anteriores, lo que será favorable para entrenar ANN con un gran número de parámetros. El juego fue jugado por mí, un humano, por lo que implica que posee un patrón complicado. Aunque se incluyen en los datos las 5 pérdidas de balón que componen un episodio entero, el objetivo de entrenar los algoritmos de SL con estos datos es comprobar cuál consigue una puntuación más alta.

6.1 Perceptrón

Las puntuaciones obtenidas por un perceptrón con los datos del juego `400points` son muy bajas, no pasa de 3. Observando su juego se aprecia que esos 3 puntos se consiguen yendo a la derecha cuando la pelota está a la derecha del jugador y cae hacia la derecha, y yendo a la izquierda cuando la pelota está a la izquierda del jugador y cae hacia la izquierda. De esta forma se golpea la pelota 3 veces, debido a que los saques tiran la pelota hacia las esquinas en 3 ocasiones durante un episodio (ver figura 2.7 para ver gráficamente los saques). Prácticamente todos los entrenos dan lugar a esa forma de juego.

6.2 *KNN*

El algoritmo KNN es un método no paramétrico de clasificación, lo que significa que no requiere de un entrenamiento para ajustar parámetros, únicamente requiere de un *dataset* que usará para clasificar las futuras entradas. La forma de clasificar es haciendo uso de una función de distancia que calcula la similitud entre el vector de entrada y los vectores del *dataset*. De los k vectores más cercanos se obtiene la clase y que más se repite. La función de distancia utilizada en la implementación es la distancia euclídea.

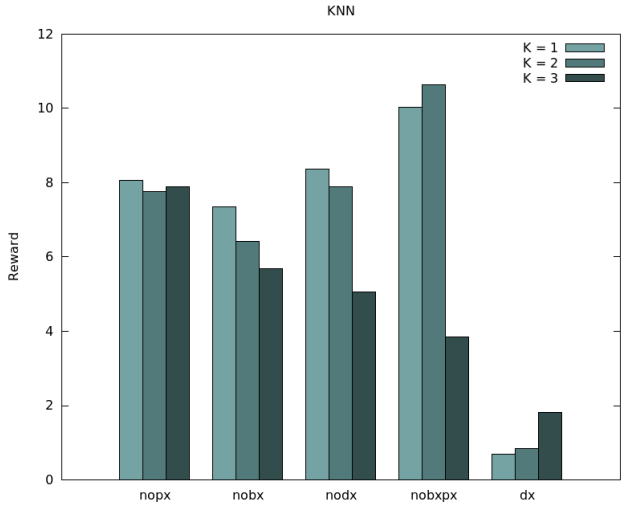


Figura 6.1: *Reward* medio de un *KNN agent* con diferentes valores de k.

Se ha creado un *KNN agent* que recibe un *dataset* que usa como conjunto de entrenamiento de un KNN. El único parámetro que puede modificarse en este agente es la k, el número de vectores cercanos a mirar para clasificar un nuevo punto. La acción realizada en cada ciclo es aquella devuelta por el algoritmo.

Los resultados obtenidos con 3 valores diferentes de k se encuentran en la figura 6.1.

6.3 *Neural Networks*

Las ANN son un modelo computacional que intenta emular el funcionamiento de las neuronas biológicas. Una red neuronal se compone de unidades más sencillas llamados neuronas. Dependiendo del número de neuronas y de capas en las que se encuentren dispuestas, tendrá más o menos capacidad de adaptarse a los datos. Es por ello que realizaremos varias pruebas con diversas estructuras en las redes. En concreto se han propuesto 9 estructuras:

- **Sin capas ocultas**. Se trata de una estructura idéntica a la de un perceptrón, por lo que se espera de esta configuración que consiga resultados muy similares.
- **Una capa oculta**. Variaremos el número de neuronas de la capa oculta: 10, 20, 30 y 40.
- **Dos capas ocultas**. Al igual que con una capa, variaremos el número de neuronas de ambas capas ocultas: 10-10, 20-20, 30-30 y 40-40.

Tratándose de 5 datos de entrada como mucho, se ha optado por no añadir más complejidad a las redes. Además, añadir más capas supondría enfrentarse al *vanishing gradient problem*.

En la implementación utilizada, la función de activación de cada una de las neuronas es $\tanh$. Para realizar el entrenamiento se utiliza *backpropagation* [Rumelhart et al., 1986], optimizado haciendo uso de *stochastic*

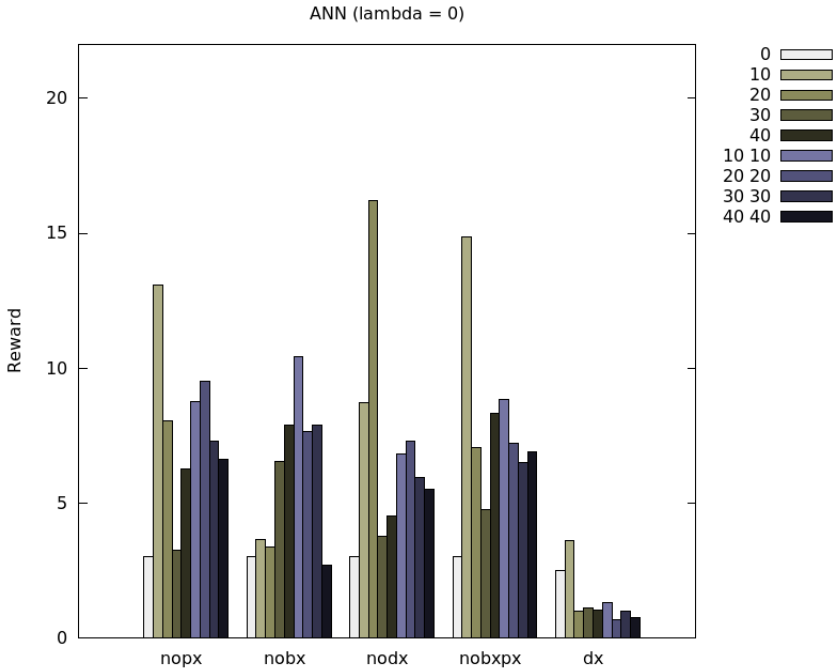


Figura 6.2: *Reward* medio de las diferentes redes neuronales sin regularización. En la leyenda se indican las capas ocultas y el número de neuronas de cada capa.

gradient descent. Para más detalles sobre la implementación consultar el apéndice A sobre *HappyML*.

Las pruebas se han realizado inicialmente sin ningún tipo de regularización. Los resultados del *reward* medio obtenido en esas pruebas se encuentra en la figura 6.2.

Se ha repetido el mismo proceso utilizando esta vez regularización L2 (*weight decay*). Llamamos λ al factor de regularización y probaremos a darle dos valores: 0.1 y 0.2. De esta forma se intenta evitar la memorización de los datos (*overfitting*), con la intención de generalizar mejor ante casos nunca vistos antes. Normalmente se suelen probar con valores de λ múltiplos de 10, pero hemos elegido esos valores con la idea de verificar los beneficios de la regularización y de mostrar los problemas de aplicar una regularización excesiva. El ajuste de λ, así como de otros *hyperparmeters* de una ANN, deberían de ser diferentes en cada una de las redes. Debido a la falta de tiempo para la realización de pruebas se han probado todas las redes con los mismos valores de λ sin realizar una búsqueda exhaustiva de cuál es el mejor de ellos. Los resultados tras esta serie de entrenos se encuentran representados en la figura 6.3.

6.4 Conclusiones

Las mejoras obtenidas con los dos nuevos métodos probados son claramente visibles. Fronteras de decisión más complejas ayudan a imitar patrones de juego más elaborados.

Se ha comprobado que el ordenador es capaz de jugar perfectamente haciendo caso únicamente de la ca-

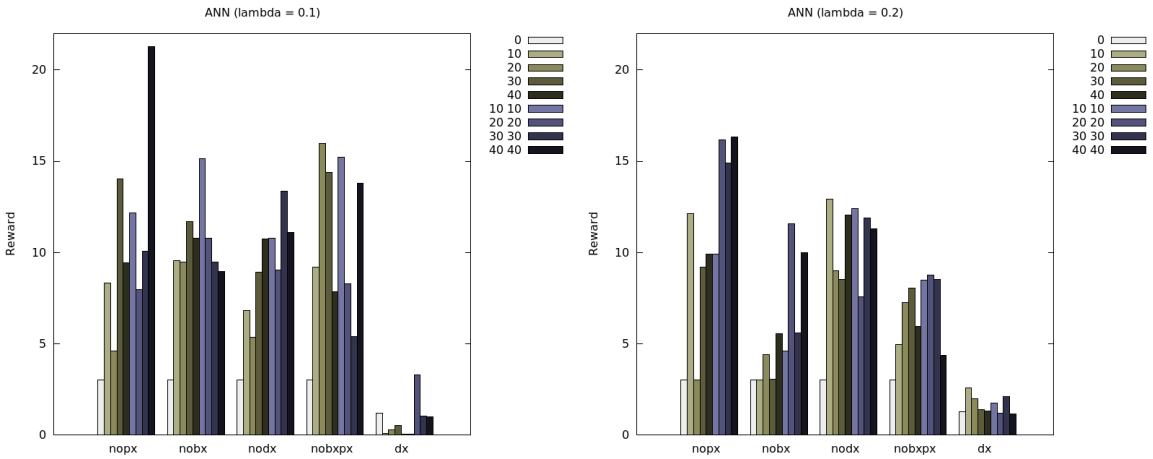


Figura 6.3: *Reward* medio de las diferentes redes neuronales con regularización L2 con $\lambda = 0.1$ y $\lambda = 0.2$. En la leyenda se indican las capas ocultas.

racterística $Diff_x$ (*If agent*), sin embargo, ha quedado demostrado que esa sola característica no es suficiente como para definir el patrón de juego de un humano. De ahí la gran cantidad de error que se ha conseguido usando el estado `dx`.

En el **KNN**, valores pequeños de k parecen favorecer el aprendizaje. Este algoritmo usa todo el *dataset* obtenido de los datos, es decir, contiene muchas acciones en las que no se mueve. Cuanto más grande es k, más acciones `PLAYER_A_NOOP` encuentra entre los primeros k, por lo que no se moverá y provocará más pérdidas.

Con respecto a las **ANN**, en la figura 6.4 se muestran gráficos de barras con los errores de las redes entrenadas. A continuación comentaremos detalladamente la información extraída de esas gráficas.

La estructura de una ANN **sin capas ocultas** es exáctamente la misma que un perceptrón y la puntuación obtenida es muy similar, en torno a 3. Los errores de las redes sin capas ocultas también son muy similares a los del perceptrón, cercanos al $25\,\%$, y los valores de regularización probados no parecen afectarle.

Si añadimos **una capa oculta**, los errores disminuyen conforme aumenta el número de neuronas. Sin el uso de regularización, las redes con una sola capa oculta y muchas neuronas parecen tener menos error que algunas de las redes de dos capas con pocas neuronas. Si miramos el *reward* conseguido por las redes de una capa, obtienen más puntuación aquellas que más error tienen. Aquellas redes que se encuentran por debajo del $10\,\%$ de error se han ajustado demasiado al ruido de los datos de entrenamiento y no son capaces de generalizar tan bien como lo hacen las redes con más errores. En este caso, el menor número de neuronas de las primeras capas impide que se pueda ajustar de más a los datos. Tras aplicarle regularización con los dos valores de λ propuestos, los errores han aumentado llegando al punto de ser prácticamente idénticos entre las distintas redes de una capa. La regularización ha bajado el *reward* de las redes con menos neuronas en su capa oculta, pero ha mejorado el rendimiento de aquellas con más neuronas.

En las redes con una arquitectura de **dos capas ocultas** ha ocurrido algo muy similar. Sin regularización se

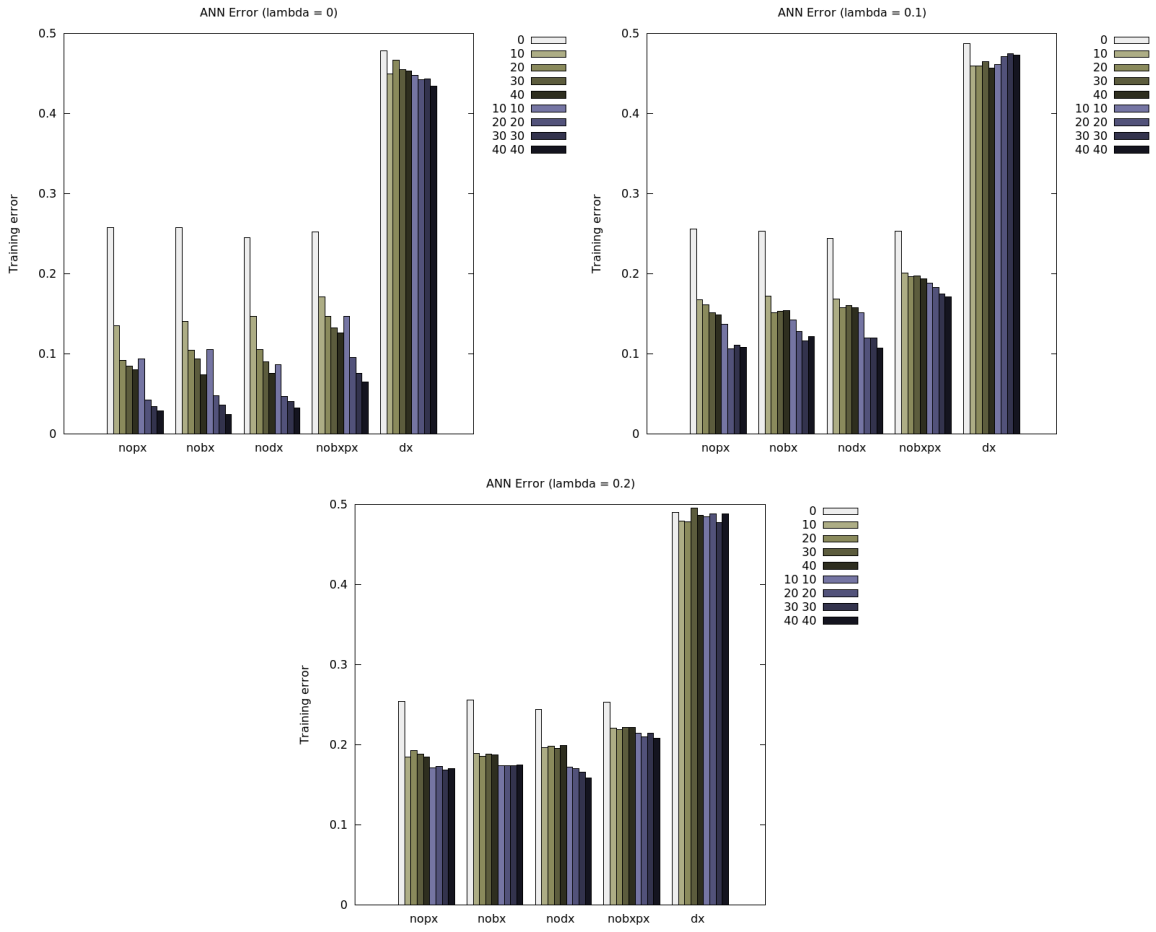


Figura 6.4: Errores de las redes usadas por el *ANN agent*. Cada gráfica corresponde a un valor de regularización diferente.

consiguen errores más pequeños cuanto más neuronas en sus capas ocultas. La regularización hace aumentar el error conforme crece el parámetro λ, llegando a conseguirse un error muy similar en todas las redes.

Con respecto a los valores de λ se puede ver que 0.2 es un valor demasiado alto, puesto que hace que los *rewards* conseguidos sean menores en comparación a los conseguidos con 0.1. El valor 0.1 evita un ajuste excesivo al ruido de los datos y favorece la generalización. Aún con la ayuda de la regularización, llama la atención la gran diferencia entre los puntos conseguidos en `400points` y los conseguidos por los agentes.

¿Por qué consiguen los agentes (tanto *KNN agent* como *ANN agent*) puntuaciones tan lejanas a las del juego con el que ha sido entrenado? Si bien es cierto que los agentes podrían mejorar su rendimiento si tuvieran a su disposición muchos más datos de juego, hay una característica importante que hay que tener en cuenta a la hora de poner a prueba los algoritmos de SL. Golpear a la pelota no supone acertar en predecir una acción, sino en la predicción de una secuencia de acciones. Cualquier fallo de predicción en un momento dado puede afectar gravemente al jugador haciéndole perder una vida. Observando los juegos se aprecian muchos fallos "tontos", es decir, que el jugador se mueve en la dirección en la que está la pelota, pero, o bien se mueve muy lento y no le da tiempo a llegar o va muy rápido y se pasa. Esto significa que va por el buen camino pero que aún así ha perdido una vida. Este tipo de comportamientos no se ve reflejado en las gráficas, donde sólo se aprecia la puntuación conseguida obviando las "buenas intenciones" del *KNN agent* y del *ANN agent*.

Capítulo 7

Mejoras *Q-Learning*

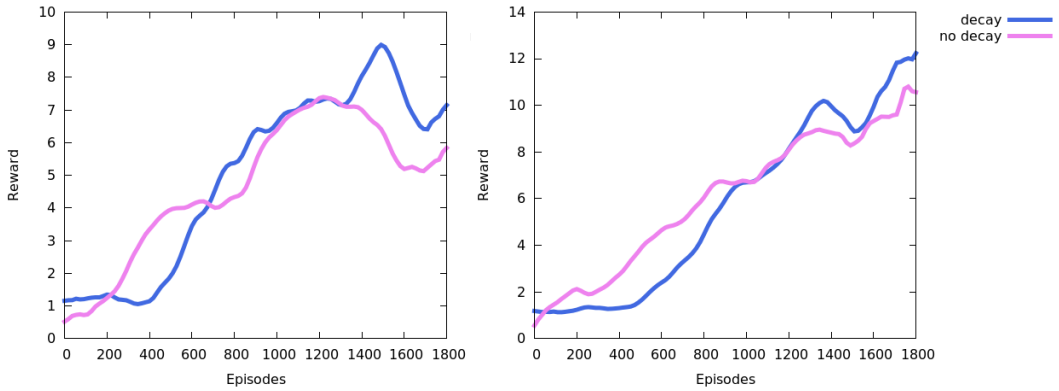
En este capítulo vamos a centrarnos en mejorar los resultados obtenidos por el *Q-Table agent*, el agente que implementa el *Q-Learning* usando una tabla. Las ideas propuestas para conseguir esas mejoras son:

- Empezar usando valores altos de ϵ e ir reduciéndolo gradualmente hasta llegar a un mínimo de 0.05, el mismo usado hasta ahora. Esto hará que los primeros episodios sean casi como un *Random agent*, pero en cuanto se fije ϵ a 0.05 se habrán explorado muchos más estados y se tendrán valores más actualizados de la función $Q(s, a)$.
- Realizar modificaciones de las funciones de *reward*. Probaremos a darle un valor por defecto a la función $Q(s, a)$ de forma que cuando no se encuentre un estado en la tabla se devuelva un valor diferente a 0. En concreto probaremos con darle un valor de 0.5 con la idea de que explore nuevos estados antes de pasar por estados conocidos que aportan poco *reward*. Otra modificación de los *rewards* consiste en otorgarle al agente un *reward* negativo cuando pierde una vida.
- Reducir el α con el paso del tiempo para intentar conseguir la convergencia de nuestros algoritmos. Esta técnica tiene el objetivo de evitar que se pase de una media de 30 a una media de 5 puntos en cuestión de episodios. Lo deseado es conseguir una curva de aprendizaje mucho más estable y que alcance un *reward* medio más alto de los ya conseguidos.
- Discretización selectiva. Esta mejora propone discretizar los estados en los que la pelota está lejos del jugador y no discretizar aquellos en los que la pelota está muy cerca del jugador. Además, también se discretizará la velocidad.

Salvo que se diga lo contrario, en este experimento seguiremos usando `nobxpx` como estado, para combatir el tiempo excesivo necesario para entrenar los algoritmos de RL usando otros estados.

7.1 ϵ *decay*

La idea de empezar con un valor alto de ϵ da lugar a una mayor exploración en los momentos iniciales. Un nuevo parámetro del agente controlará la velocidad a la que se va reduciendo ϵ hasta llegar a un mínimo fijado en 0.05.



(a) $\alpha = 0.1$ y $\gamma = 0.9999$. (b) $\alpha = 0.2$ y $\gamma = 0.9999$.

Figura 7.1: Comparación entre dos entrenos con diferentes valores de α en los que se ha utilizado ϵ *decay*. Inicialmente $\epsilon = 1$, tras cada episodio ϵ se ha disminuído en 0.001 hasta parar en 0.05.

En la figura 7.1 se muestra un par de ejemplos en los que se compara la curva de aprendizaje de dos *Q-Table agents* que usan y no usan ϵ *decay*.

En 7.1a, en la curva con ϵ-*decay* se observa un inicio con puntuaciones oscilando sobre 1 y a partir del episodio 500, donde ϵ comienza a tomar valores menores que 0.5, se va incrementando el *reward* obtenido. Tras el episodio 1000 la subida deja de ser tan pronunciada, puesto que ϵ se ha establecido ya en el valor de 0.05. A partir de ahí el aprendizaje sigue su curso tal y como hemos visto en los demás experimentos, pero con la ventaja de haber visitado más estados.

En 7.1b se muestra otra ejecución con diferente valor de α en la que la curva con ϵ-*decay* vuelve a superar a la curva sin *decay*.

7.2 Modificaciones en la función de *reward*

Si a cada estado aún no visitado se le da el valor inicial de 0.5, el agente va a explorar estados como hace con un valor elevado de ϵ pero no de forma aleatoria, visitará aquellos en los que no haya estado. Preferirá ir a un estado desconocido a aquellos que tengan un reward inferior al inicial. De esta forma si el *reward* de un estado es conocido y es superior a 0.5, explotará esa acción, si no es mayor, explorará nuevos estados.

Si el valor inicial propuesto es mayor de 1, el *reward* mínimo que puede conseguirse en el *Breakout*, se tiende a explorar demasiado consiguiendo un rendimiento menor.

La comparación entre dos curvas de aprendizaje que usan un *reward* por defecto de 0 y 0.5 se encuentra en la figura 7.2.

El hecho de aportarle un *reward* negativo a la hora de perder una vida no ha afectado en absoluto a la curva de aprendizaje del agente.

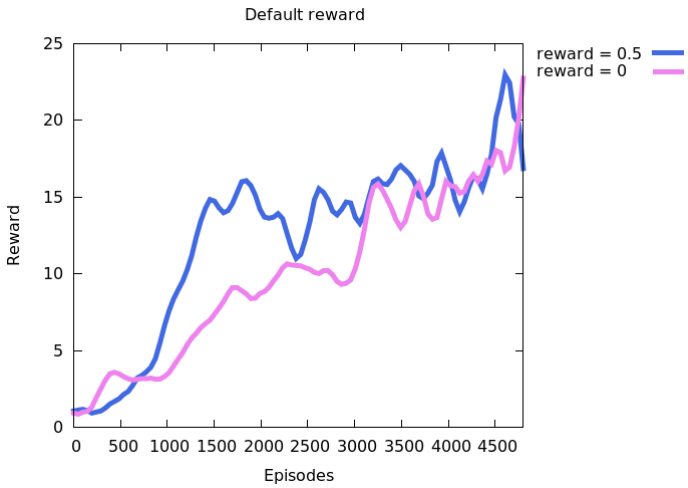


Figura 7.2: Modificación del valor por defecto de la función $Q(s,a)$. $\alpha = 0.1$ y $\gamma = 0.9999$.

7.3 *Learning rate decay*

Se ha introducido un *decay factor* que disminuirá α un porcentaje una vez pasados unos determinados episodios. Por ejemplo, en la figura 7.3 se ha reducido en un $50\,\%$ el *learning rate* cada 1000 episodios. Conforme α se aproxima a 0 la curva va disminuyendo su varianza, pero siempre oscilará un poco debido al uso de una ϵ-*greedy policy*. Si α esta muy cerca de 0 se deja de producir aprendizaje, por lo que llegados a ese punto hay que parar el entrenamiento.

En la prueba de la figura 7.3 se ha disminuido demasiado rápido el *learning rate*, impidiendo que esa primera subida llegue a su tope. Se observa que en la primera subida la curva deja de crecer al mismo ritmo cuando se alcanza el episodio 1000, que coincide con la reducción de α. A partir de ese episodio deja de valer 0.2 y pasa a valer 0.1.

Los verdaderos beneficios del *learning rate decay* se aprecian en entrenos con pocas características de entrada. En el experimento en el que analizamos distintas características de entrada, el estado `dx` no destacó por conseguir más puntuación que el resto, aún no existiendo en él el problema de la dimensionalidad. El uso de pocas dimensiones hace fluctuar la curva de aprendizaje puesto que se actualizan continuamente los mismos estados. La disminución de α ayuda mucho en estos casos.

7.4 Discretización selectiva

Se va a utilizar un tipo de discretización que modifica los valores de $Diff_x$ y $Ball_y$. Vamos a seguir con la misma idea de minimizar el número total de estados, pero esta vez discretizaremos de diferente manera los estados en los que la pelota esté más cerca del jugador que los estados en los que no lo está. De esta forma

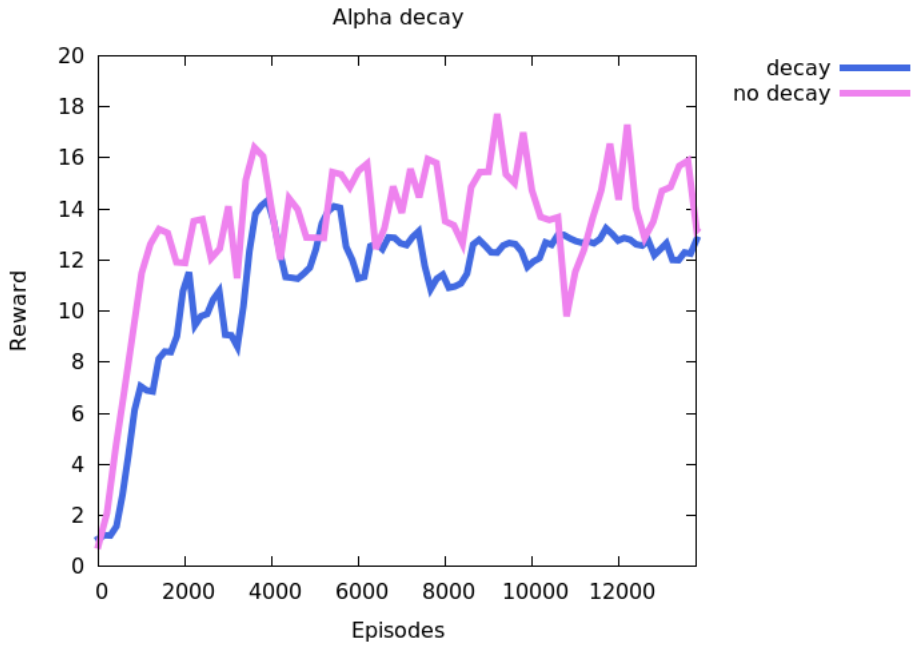


Figura 7.3: Curva de aprendizaje de un *Q-Table agent* con $\alpha = 0.2$, $\gamma = 0.99$ y un factor de discretización de 10. En la curva con *decay*, cada 1000 episodios se ha disminuido α a la mitad para conseguir que la curva oscile menos.

ganaremos en precisión cuando el jugador y la pelota se encuentran cerca y eliminaremos muchos estados realmente parecidos.

Todo lo que esté a más de 15 unidades a la izquierda del centro del jugador se le da el valor de -15 y lo que esté a más de 15 a la derecha se le da el valor de 15. Los valores intermedios no se modifican para no perder precisión. Por lo tanto, $Diff_x \in [-15, 15]$. Con la altura se hace lo mismo, todo aquello que esté por debajo de 180 se le asignará el valor de 180. De modo que ahora $Ball_y \in [180, 210]$. Recordamos que la altura 0 coincide con el techo. En la figura 7.4 se visualiza las divisiones realizadas.

Además vamos a discretizar también la velocidad, dándole valores de 1, 0 o -1. De esta forma seguimos teniendo información aproximada de a dónde se dirige la pelota pero minimizamos el número de estados posibles.

Vamos a ejecutar un *Q-Table* con el estado `dx`. Es en este momento en el que se aprecia mucho mejor los beneficios del *learning rate decay*. En la figura 7.5 se aprecia la enorme diferencia entre usarlo o no.

7.5 Conclusiones

La aplicación de ϵ ***decay*** proporciona algunas mejoras sobretodo cuanto mayor es la dimensionalidad del estado. Cuando se usa por ejemplo el estado `dx` no se han notado mejoras puesto que en sólo 100 episodios se pueden llegar a conseguir *rewards* de más de 50 sin decaer ϵ. El problema es mantener una buena media y

Figura 7.4: Visualización de las zonas discretizadas. En la imagen la pelota se encuentra en $Diff_x = 15$ y $Ball_y = 180$. En todo el recuadro en el que se encuentra la pelota las coordenadas serían las mismas. Las líneas de la imagen son aproximadas, no representan las medidas reales.

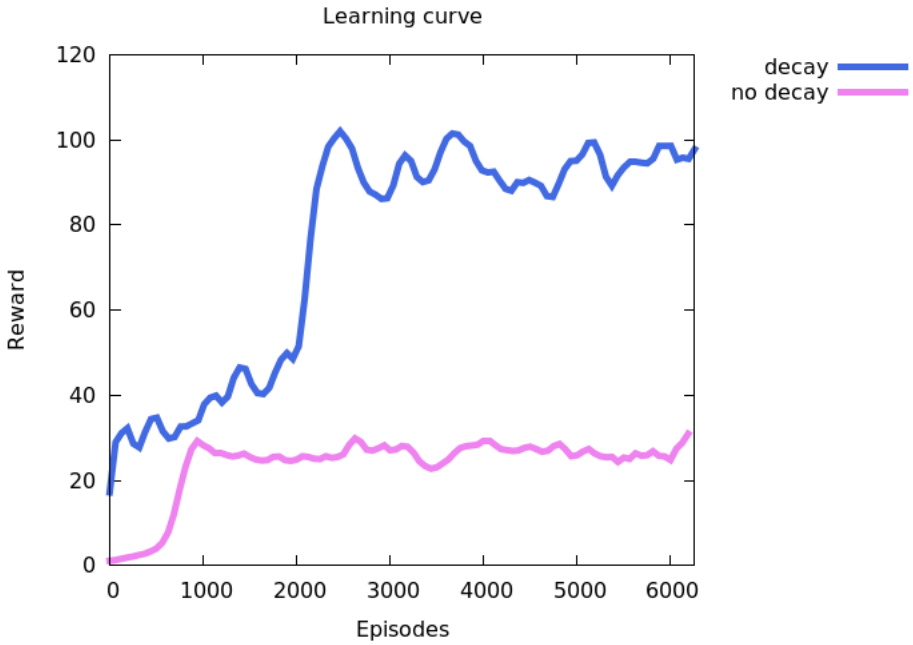


Figura 7.5: Curva de aprendizaje con discretización selectiva usando el estado `dx`. $\alpha = 0.1$, $\gamma = 0.9999$

evitar que en los próximos juegos se vuelvan a conseguir 0 puntos. Ese problema ha sido solucionado gracias al *learning rate decay*.

La **modificación de la función de *reward*** ha sido realizada de dos formas diferentes. Darle un valor por defecto como 0.5 a la función $Q(s,a)$ ayuda en el inicio, pero con el tiempo parece que la curva de aprendizaje con valor de *reward* inicial de 0 y de 0.5 convergen. Además, como ya hemos comentado antes, darle *rewards* negativos a los agentes cuando estos pierden una vida no afecta visiblemente a las curvas de aprendizaje.

La **discretización selectiva** ha proporcionado muy buenos resultados puesto que ha reducido notablemente el número de estados.

La **disminución del *learning rate*** ha sido una técnica importante tanto para evitar oscilaciones en la curva como para hacerla aumentar. Aunque la combinación de todas estas mejoras consiguen tablas capaces de obtener *rewards* medios de 130, no se ha conseguido que siga mejorando.

El tamaño del jugador ya hemos comentado que se reduce cuando se golpea la parte superior de la pantalla. Ninguna de las características de entrada indican cuál es el tamaño del jugador, por lo que se observan fallos "tontos" en el agente cuando el jugador es más pequeño. Cuando llega a puntuaciones altas en las que la barra del jugador ya ha encogido, el valor de α ya es muy pequeño, por lo que le cuesta aprender que no debe de pegarle con la parte derecha de la pala y debe hacerlo con la parte izquierda. Si se aumenta el *learning rate* en lo más alto de la curva con la esperanza de que aprenda esa nueva forma de darle al balón, la curva disminuye hasta puntuaciones cercanas a 0 y oscila.

Otro de los problemas que se observa es el continuo balanceo que sigue teniendo, puesto que lo hemos entrenado con el estado `nobxpx` que hace uso de $Diff_x$. El balanceo se justifica diciendo que cuando la pelota está alta, lejos del jugador, cualquier acción aleatoria realizada en ese momento no afecta en absoluto a que cuando baje la pelota pueda darle. Cuando se golpea un bloque las acciones aleatorias realizadas en ese momento son seleccionadas como acciones que proporcionan *reward* positivo.

Capítulo 8

Conclusiones

Uno de los problemas recurrentes que nos hemos encontrado a lo largo de los experimentos de **RL** ha sido la convergencia del *Q-Learning*. Hemos comprobado que la implementación de un *Q-Learning* con una tabla es aplicable solo a problemas de pequeño tamaño. En nuestro caso, cuando hemos querido añadirle una quinta componente al vector de estados, la cardinalidad del conjunto S ha aumentado de tal manera que no hemos conseguido buenos resultados. Por lo tanto, una buena forma de continuar este estudio sería utilizar una aproximación de la función $Q(s, a)$. El uso de técnicas como *funtion approximation* se convierte en algo obligatorio cuando hablamos de problemas complejos. Los algoritmos de RL con los que se han conseguido las mayores puntuaciones son algoritmos ciegos: no ven ni las paredes ni los bloques que hay en la pantalla. Utilizando técnicas de *function approximation* sería viable el pasarle todos esos datos al algoritmo.

Con respecto al **SL**, hemos comprobado la importancia de unos buenos datos así como la dificultad de valorar la calidad de los que se posee. Quizás la implementación de algoritmos de entrenamiento más potentes para redes neuronales o simplemente hacer uso de alguna de las librerías actuales hubiera ayudado a mejorar los resultados obtenidos.

En relación a la librería de ML implementada, ***HappyML***, se ha observado la relativa facilidad con la que pueden llegar a implementarse la mayoría de los algoritmos. Por lo general son muy sencillos de usar, sin embargo, otra cosa diferente es saber usarlos bien. ¿Qué algoritmo utilizar? ¿Qué parámetros elegir? ... Son muchas las preguntas que te asaltan a la hora de abordar un problema de ML.

Durante todo el proceso se ha echado en **falta un ordenador más potente** para la realización de pruebas. La divertida implementación es solo una pequeña parte del trabajo, la mayor parte consiste en la realización de pruebas. El número y la longitud de las pruebas realizadas se han visto afectadas por la potencia de cálculo del portátil en las que han sido realizadas. Los entrenos de redes neuronales podría haber sido acelerados haciendo uso de una GPU. Puede que algunos de los resultados que hemos considerado que tenían una gran dimensionalidad hubieran llegado a mostrar más aprendizaje en un ordenador más potente dejándolo entrenando un día entero.

8.1 Futuras investigaciones

Las futuras investigaciones están asociadas al RL, en concreto tratan sobre *function approximation*.

Se han hecho algunas pruebas utilizando las mejores tablas obtenidas por el *Q-Table agent* para entrenar una red neuronal. La red neuronal, a partir del vector del estado y la acción, predice el valor de la función $Q(s, a)$. Aunque no se han conseguido jugadores muy estables, se aprecia que aún habiendo siendo entrenados sin discretizar la velocidad, se alcanza a golpear la pelota en bastantes ocasiones cuando la velocidad es elevada. Ante esas nuevas situaciones el *Q-Table agent* se quedaba quieto.

Sin embargo, la forma más conocida de *function approximation* no se realiza de esa manera. Consiste en un aprendizaje *online* en el que se realiza un paso del descenso por gradiente en cada acción tomada. Si la aproximación es diferenciable con respecto a sus parámetros (como es el caso de las aproximaciones lineales o las redes neuronales), los parámetros pueden ser aprendidos tratando de minimizar el error TD (E_{TD}), que se puede definir como

$$E_{TD}(s_t, a_t) = y_t - Q(s_t, a_t) = \left[r_t + \gamma \max_a Q(s_{t+1}, a) \right] - Q(s_t, a_t). \tag{8.1}$$

El uso de aproximaciones a partir de funciones no lineales puede causar que algoritmos como *Q-Learning* y *Sarsa* se vuelvan inestables, es decir, que los parámetros y los valores devueltos tiendan a infinito. Es por ese motivo que se utilizan muy frecuentemente aproximaciones lineales del tipo

$$Q(s, a; \theta) = \theta^{\mathrm{T}} \phi_{s,a} \tag{8.2}$$

donde θ es un vector de pesos y $\phi_{s,a}$ es el vector de características que representa el estado junto a la acción.

La actualización del vector de pesos se haría de forma *online* tras realizar cada acción.

$$\theta_{t+1} = \theta_t - \frac{1}{2} \alpha \nabla_{\theta_t} \left[y_t - Q(s_t, a_t; \theta_t) \right]^2 \tag{8.3}$$

$$= \theta_t + \alpha \left[y_t - Q(s_t, a_t; \theta_t) \right] \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{8.4}$$

$$= \theta_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) - Q(s_t, a_t; \theta_t) \right] \phi_{s_t, a_t} \tag{8.5}$$

Con el objetivo de mejorar la convergencia de una red neuronal, en Mnih et al. [2015] se comenta la mejora que supone el ir guardando las transiciones en una *replay memory* con el objetivo de entrenar la red cada x tiempo. De esta forma se evita entrenar la red con datos consecutivos, lo que es sabido que nunca ayuda a su convergencia.

Con el objetivo de poder aplicar el mismo agente a varios juegos totalmente diferentes tendría que darse el paso de realizar la extracción de características a partir de las imágenes o bien a partir de toda la información de la RAM.

Apéndice A

HappyML

HappyML es una librería de C++ orientada al ML. Ha sido concebida con propósitos educacionales, por lo que no pretende que sea usada en entornos de producción. Se trata de un proyecto de código abierto que se encuentra disponible en *GitHub* (`https://github.com/guiferviz/happyml`) con licencia *MPL v2.0*.

Esta basada en *cmake* con la idea de ser multiplataforma, aunque hasta el momento solo ha sido probada en varias versiones de *Ubuntu*. Depende de la librería *armadillo*, que es la encargada de las operaciones de álgebra. El alto nivel de esta librería hace que el código se centre más en los algoritmos en lugar de en cómo implementarlos, lo que ayuda mucho a la hora de programarlos y facilita su comprensión por aquellos que los leen.

Figura A.1: Logo de la librería. Como no podría ser de otra manera, el logo está muy feliz.

A continuación vamos a hacer un recorrido por las herramientas y modelos que nos ofrece la librería.

A.1 Compilación e instalación

Una vez clonado el repositorio, desde la carpeta raíz del proyecto debemos de ejecutar la siguiente secuencia de comandos:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

```
$ sudo make install
```

Con ello tenemos instalado las cabeceras y la librería compilada en nuestro sistema. Además, se han instalado las herramientas `happyplot` y `happydatacreator` que veremos más adelante.

El proceso fallará en el caso de no tener instalada la librería *armadillo*. Puedes instalarla de forma rápida en *Ubuntu* usando el comando:

```
$ sudo apt-get install libarmadillo-dev
```

Si tienes instalado *Doxygen* se puede generar la documentación de la librería ejecutando:

```
$ make doc
```

Los archivos generados se encuentran en el directorio `<happyml_root_dir>/build/doc`. Existe una versión online de la documentación alojada en `https://guiferviz.github.io/happyml`.

HappyML usa *Google Tests* como librería de pruebas y *CTest* para ejecutarlas. *CTest* es una herramienta de testeo que forma parte de *CMake*, sin embargo, debes de asegurarte de tener la librería *Google Test* instalada si quieres ejecutar los tests. En `<happyml_root_dir>/test` se encuentra el código fuente de los tests. Para compilarlos se utiliza `cmake -Dbuild_tests=True ..` en lugar de `cmake ..` a la hora de construir el proyecto. De esta forma, una vez hecho el *make*, pueden lanzarse los tests con el comando

```
$ ctest
```

o bien algún test en concreto con

```
$ ctest -R nombre_test -verbose
```

A.2 *Datasets*

Los conjuntos de datos $\mathcal{D}$ que usan los algoritmos están encapsulados en un solo objeto llamado `DataSet`. Matemáticamente un *dataset* se define como

$$\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N), \text{ dónde } \mathbf{x}_i = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^d, \ \mathbf{y}_i = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \in \mathbb{R}^k. \tag{A.1}$$

Una forma más óptima de manejar los *datasets* es juntar todos los puntos en una matriz. Así es como guarda los datos el objeto `DataSet`. Éste contiene una matriz $\mathbf{X}$ con todas las características de entrada y una matriz $\mathbf{Y}$ en la cual se guardan las salidas que corresponden a cada entrada.

$$\mathbf{X} = \begin{bmatrix} - \mathbf{x}_1^{\mathrm{T}} - \\ - \mathbf{x}_2^{\mathrm{T}} - \\ \vdots \\ - \mathbf{x}_N^{\mathrm{T}} - \end{bmatrix} \in \mathbb{R}^{N \times d}, \ \mathbf{Y} = \begin{bmatrix} - \mathbf{y}_1^{\mathrm{T}} - \\ - \mathbf{y}_2^{\mathrm{T}} - \\ \vdots \\ - \mathbf{y}_N^{\mathrm{T}} - \end{bmatrix} \in \mathbb{R}^{N \times k} \tag{A.2}$$

Con el objetivo de simplificar los ejemplos que vamos a mostrar, de ahora en adelante se asume que $k = 1$. De esta forma se podrán visualizar cómodamente los *datasets* en un gráfico. Debida a esa asunción, la salida asociada a un vector $\mathbf{x}_i$ puede ser representada con y_i en lugar de con $\mathbf{y}_i$.

La mayoría de las veces, los datos que conforman el *dataset* no son generados en el mismo programa que en el que se entrena el modelo, por lo que los objetos `DataSet` deben de ser persistentes. Por ello, la clase es serializable, es decir, puede guardarse/cargarse en el disco usando los métodos `save`/`load`. Esos métodos esperan recibir el nombre de un archivo como argumento.

La escritura/lectura de *datasets* se realiza a partir de archivos CSV. Se ha optado por dicho formato por la facilidad que supone su edición con editores de texto plano o programas de hojas de cálculo. Un ejemplo de archivo podría ser el mostrado en la figura A.2.

```
1,1,1
-1,-1,-1
```

Figura A.2: Contenido del archivo *train.csv*.

Para cargar esos datos en un objeto `DataSet` basta con escribir algo como en la figura A.3. Por defecto se toma la última columna como la salida $\mathbf{Y}$. A la matriz $\mathbf{X}$ se le añade por defecto una primera columna llena de unos, con el objetivo de añadir la propiedad $x_0 = 1$ a cada uno de los puntos.

```
#include <happyml.h>


int main()
{
    happyml::DataSet dataset;
    dataset.load("train.csv");

    cout << "Samples: " << dataset.N << endl;  // Samples: 2
    cout << "Features: " << dataset.d << endl; // Features: 2
}
```

Figura A.3: Carga de archivos CSV para usar como dataset.

Existen otras maneras de crear *datasets*, así como otros métodos que facilitan el uso de esta clase. Para más información mirar la documentación.

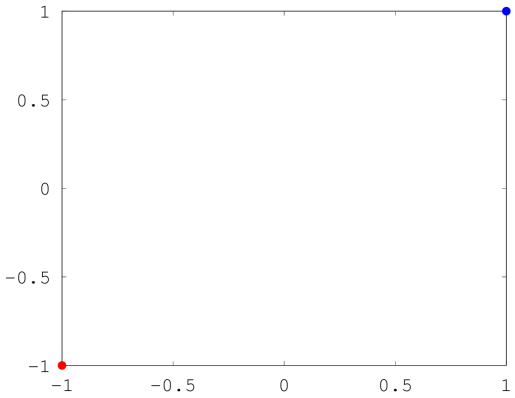


Figura A.4: Representación gráfica del *dataset* mostrado en A.2.

A.2.1. `happyplot`

HappyML pone a nuestra disposición una herramienta que facilita la visualización de *datasets*[1]. Para usar esta herramienta debe de tener instalado en su sistema un intérprete de *Octave*.

El archivo CSV debe de tener 3 columnas ($d = 2$) para que sea considerado como un problema de clasificación. Los números de la última columna deben de ser $y_i \in \{-1, +1\}$.

```
$ happyplot -d train.csv -o train.png
```

Si la opción `-o` (`o` de *output*) no se indica, la imagen generada será guardada en un archivo de nombre genérico llamado `output.png`. El resultado obtenido se muestra en la figura A.4.

Como se observa, los puntos que corresponden a clases negativas se pintan de color rojo y aquellos que pertenecen a clases positivas son de color azul.

Si por el contrario el archivo indicado con la opción `-d` (`d` de *dataset*) contiene 2 columnas ($d = 1$), será considerado un problema de regresión lineal y se representará de otra forma. En la figura A.5 se encuentra un ejemplo con un *dataset* algo más elaborado que dos simples puntos.

Quién crea que he ido escribiendo en un CSV punto a punto para formar el dataset mostrado en la figura A.5, es que no sabe de la existencia de `happydatacreator` 😎

A.2.2. `happydatacreator`

Esta segunda herramienta también basada en *Octave* sirve para generar de forma visual *datasets* de 1 (problemas de regresión) y 2 (problemas de clasificación) dimensiones. Basta con ejecutar

```
$ happydatacreator
```

[1]Como veremos más adelante, también es capaz de representar la frontera de decisión de los clasificadores, así como hipótesis cuya función no es la de clasificar sino la de aproximar un valor.

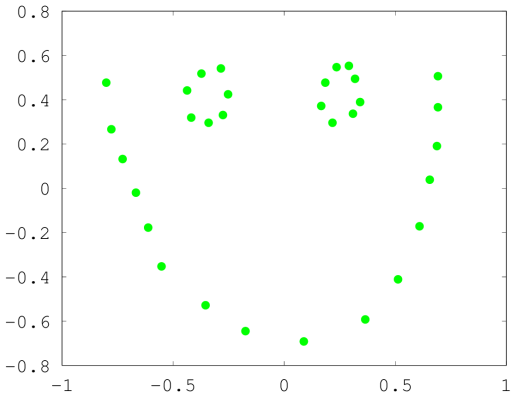


Figura A.5: Todos los puntos de un mismo color indican que ya no se trata de un problema de clasificación, sino que es de regresión.

para que se abra una venta de *Octave* que va pintando puntos conforme haces clic. Al finalizar el programa, el *dataset* es guardado en un archivo llamado `points.data` que puede ser cargado directamente en *HappyML*.

Los puntos creados con el botón derecho pertenecen a la clase -1 y los creados con el izquierdo a la clase $+1$. Si al cerrar `happydatacreator` todos los puntos creados son de la misma clase, el *dataset* es considerado de regresión.

A.3 Modelos

La clase `Predictor` es una clase abstracta que engloba a todas las hipótesis implementadas en la librería. Un tipo específico de `Predictor` son los clasificadores, que predicen la pertenencia de un vector de entrada a una o varias clases. Un esquema completo de la jerarquía de las clases que heredan de `Predictor` se encuentra en la figura A.6.

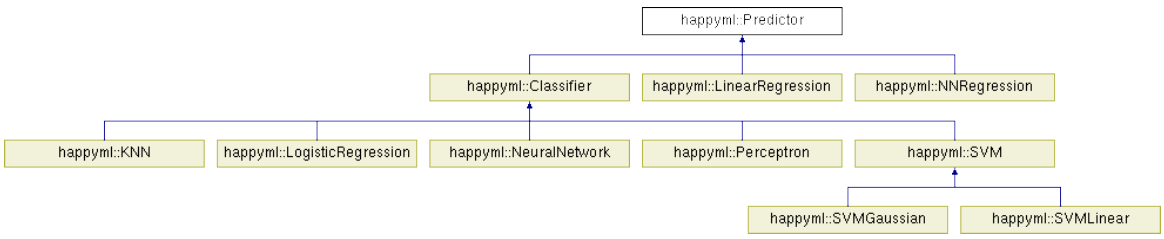


Figura A.6: Diagrama de todos los modelos implementados en la librería.

A continuación haremos un repaso rápido por cada uno de ellos viendo algún ejemplo de uso.

A.4 Modelos lineales

Los modelos lineales comparten la siguiente estructura

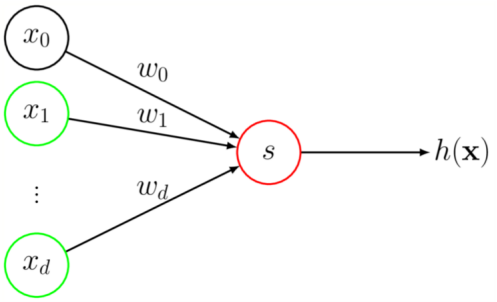


Figura A.7: Representación gráfica de un modelo lineal.

siendo $x_0 = 1$ y s lo que se conoce como señal y que se calcula con

$$s = \sum_{i=0}^{d} w_i x_i = \mathbf{w}^{\mathrm{T}}\mathbf{x} \tag{A.3}$$

Dependiendo de cómo se procese esa señal se obtienen los diferentes modelos lineales:

- Si la señal es transformada usando la función sgn (signo) se obtiene un **perceptrón**.
- Si la señal es transformada usando una función sigmoidea se obtienen los modelos de **regresión logística**.
- Si la señal no se procesa se obtiene un modelo de **regresión lineal**.

A.4.1. Perceptron

La hipótesis es calculada con $h(\mathbf{x}) = \mathrm{sgn}(s)$, donde s es la señal propia de los modelos lineales y sgn la función signo que viene dada por

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{si} \quad x \geq 0 \\ -1 & \text{si} \quad x < 0 \end{cases} \tag{A.4}$$

por lo que la salida de nuestra hipótesis queda restringida a dos valores, $h(\mathbf{x}) \in \{+1, -1\}$.

La forma de encontrar el mejor vector de pesos se realiza de forma iterativa utilizando el algoritmo del *pocket*. Cada vez que es modificado el vector de pesos se mide el error que produce y, en el caso de que su error sea el más bajo conseguido hasta ahora, se guarda como el mejor vector.

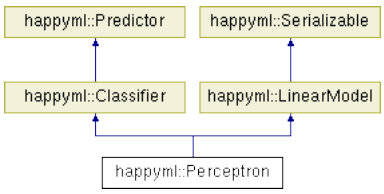


Figura A.8: Diagrama de herencia del perceptrón. Diagramas de este tipo de cada una de las clases de la librería, pueden ser consultados en la documentación.

Como se puede observar en la figura A.8, los perceptrones heredan de la clase `LinearModel` que es serializable. Esos significa que el vector de pesos de cualquier modelo lineal puede ser guardado/cargado en un archivo en el disco duro. En concreto, se trata de un archivo de texto en el que hay un número en cada línea que se corresponde con los pesos w_0, w_1, ...

En la figura A.9 se muestra el código necesario para crear un perceptrón y entrenarlo usando un objeto de la clase `DataSet`. En la última línea del ejemplo se guarda un archivo con información necesaria para que `happyplot` pueda representar un borde de decisión como el mostrado en la figura A.10. Para que `happyplot` lo encuentre basta con indicarle con la opción `-b` (`b` de *boundary*) la ruta al archivo `boundary.data`.

```
// Crear perceptrón con un vector de pesos con tantas dimensiones
// como dimensiones tiene el training set.
happyml::Perceptron p(dataset.d);

// Entrenamos usando pocket algorithm durante 10 iteraciones máximo.
int iterations = 10;
float error = p.train(dataset, iterations);

// Guardamos el borde de decisión para representarlo con happyplot.
p.saveSampling("boundary.data", -1, 1, 500, -1, 1, 500);
```

Figura A.9: Creación y entreno de un perceptrón.

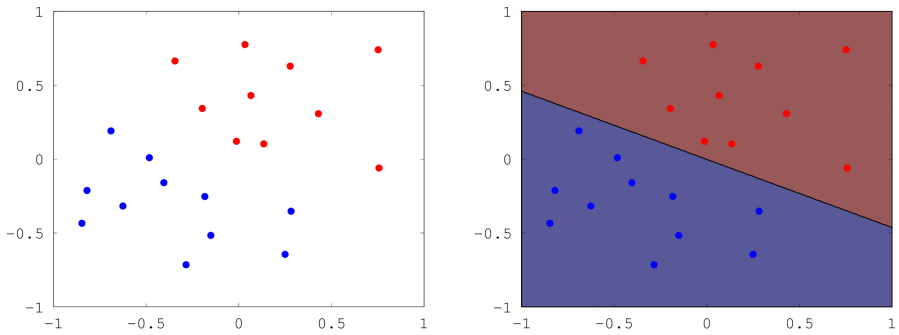


Figura A.10: A la izquierda el *dataset* utilizado y a la derecha el mismo *dataset* con el *decision boundary* formado por el vector de pesos de un perceptrón que menos error consigue. El perceptrón ha llegado a esa solución de error 0 con sólo 4 iteraciones.

A.4.2. Regresión lineal

En la implementación de la mayoría de operaciones de la librería se ha pretendido usar operaciones con vectores en lugar de bucles para acelerar así su ejecución. Veamos un ejemplo de lo sencilla que puede llegar la implementación de la obtención del vector de pesos de un regresor lineal. En la ecuación A.5 se muestra las operaciones necesarias para obtener un vector de pesos $\mathbf{w}$ que minimize el error cuadrático. En la figura A.11 lo sencilla que supone su implementación gracias a la librería *armadillo*.

$$\mathbf{w} = \mathbf{X}^{\dagger}\mathbf{y} \quad \text{donde} \quad \mathbf{X}^{\dagger} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}} \tag{A.5}$$

```
void LinearRegression::train(const DataSet& dataset)
{
    w = pinv(dataset.X) * dataset.y;
}
```

Figura A.11: Método de `LinearRegression` que obtiene el vector de pesos que minimiza el error cuadrático en el *dataset* indicado. La ecuación matemática implementada en este método se encuentra en A.5.

Al igual que en todos los modelos que se van a nombrar a continuación, la regresión lineal puede ser aplicada utilizando regularización. Para más información de cómo usarla mirar la documentación.

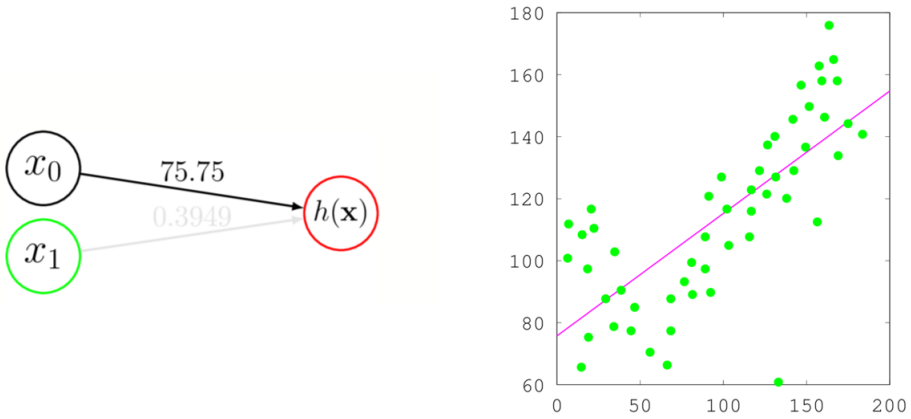


Figura A.12: A la izquierda los pesos del regresor lineal. A la derecha la visualización del ajuste junto a los datos utilizados.

HappyML contiene un método que genera archivos DOT a partir de modelos lineales y que pueden ser dibujados posteriormente con la herramienta *Graphviz*[2]. Las aristas que conectan los nodos del modelo están pintadas con diferente intensidad en función del valor del peso de esa conexión. Las funciones para la obtención

[2]Por el momento, la generación de las imágenes *png* no es algo que realice la librería, recae sobre el usuario final. Solamente está implementado para modelos lineales, aunque se contempla la posibilidad de en un futuro realizar algo parecido con las redes neuronales.

de gráficos a partir de modelos lineales se encuentran en el *namespace* `happyml::tools`.

A.4.3. Regresión logística

Con la regresión logística se pretende obtener una probabilidad de pertenencia de un vector de entrada a una clase dada. Las salidas obtenidas por este modelo se encuentran en el intervalo $(0, 1)$. Las salidas vienen dadas por $h(\mathbf{x}) = \sigma(s)$, donde s es la señal propia de los modelos lineales y σ es la función logística o sigmoidea definida como

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} \tag{A.6}$$

En este ejemplo de regresión logística vamos a mostrar otra de las posibilidades que nos ofrece *HappyML*: transformaciones no lineales. Las características de entrada $\mathbf{x} \in \mathcal{X}$, tras aplicarle una transformación $\phi(\mathbf{x}) = \mathbf{z} \in \mathcal{Z}$. De esta forma se puede entrenar cualquier clasificador en el nuevo espacio $\mathcal{Z}$.

En el ejemplo vamos a elevar al cuadrado tanto x_1 como x_2 para que nos permita separar los puntos con un óvalo. El código para realizar dichas trasformaciones, así como el entreno del clasificador logístico se encuentra en la figura A.13. Los bordes de decisión generados por el clasificador tanto en $\mathcal{X}$ como en $\mathbf{Z}$ se encuentran en la figura A.14.

```
// Transformaciones a aplicar.
happyml::TransformerCollection t;
t.add(new happyml::transforms::Pow(1, 2));  // z_1 = x_1^2
t.add(new happyml::transforms::Pow(2, 2));  // z_2 = x_2^2
t.apply(dataset);  // Aplicamos transformaciones por orden.
dataset.save("train_transformed.csv");

// Clasificador.
happyml::LogisticRegression lr(dataset.d);
int iterations = 1000;
float learning_rate = 0.1;
float error = lr.train(dataset, iterations, learning_rate);

// Guardamos boundary con las trasformaciones.
// Boundary en X.
lr.saveSampling("boundary.data", -1, 1, 500, -1, 1, 500, t);
// Boundary en Z.
lr.saveSampling("boundary.data", -1, 1, 500, -1, 1, 500);
```

Figura A.13: Regresión logística.

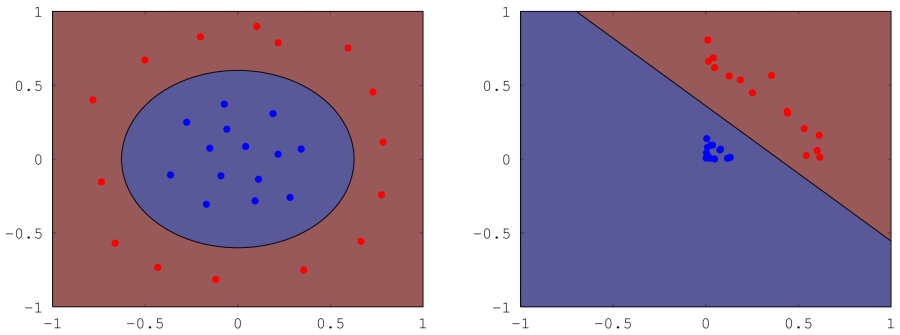


Figura A.14: Ejemplo de regresión logística. A la izquierda se observa el *dataset* orginal y el borde de decisión generado en el espacio $\mathcal{X}$. A la derecha el *dataset* y el borde de decisión se encuentran en el espacio $\mathcal{Z}$.

A.5 KNN

Este sencillo modelo es no paramétrico, por lo que no necesita de entrenamiento. El constructor espera un *dataset* que será usado para predecir las nuevas entradas.

```
happyml::KNN knn(dataset);
knn.saveSampling("boundary.data", -1, 1, 500, -1, 1, 500);
```

Figura A.15: Código para generar un clasificador KNN.

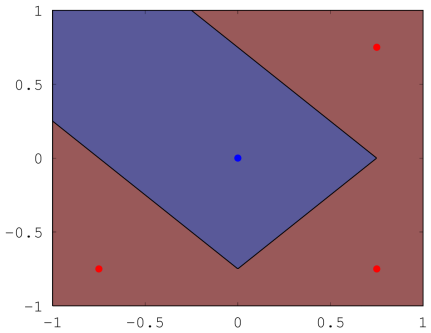


Figura A.16: Los bordes de decisión de un KNN en 2 dimensiones coinciden con los polígonos de *Voronoi*.

A.6 Redes neuronales

La clase `NeuralNetwork` implementa una red neuronal para clasificación. La forma de construir la red es a partir del tamaño de cada capa. En las redes orientadas a la clasificación (clase `NeuralNetwork`), todas las funciones de activación utilizadas son $\tanh$, lo que limita la salida $h(\mathbf{x}) \in (-1, 1)$. Los pesos de las conexiones que hay entre dos capas están representadas internamente con matrices. Se trata de una implementación sencilla

y poco flexible (como dirían algunos, *vanilla implementation*), lo que elimina la posibilidad de cambiar las funciones de activación de cada capa y solo ofrece *batch* o *stochastic gradient descent* para ser entrenada.

Las redes neuronales son también objetos serializables, pueden ser cargadas/guardadas de/en un archivo. El archivo posee un fomato binario y en él están contenidos todos los pesos de cada una de las capas.

Para la implementación se han seguido casi en su totalidad los capítulos extra del libro *Learning from Data* [Abu-Mostafa et al., 2012], así como algunos consejos de *Efficient Backprop* [LeCun et al., 1998] y de *CS231n* [Li et al.].

Algo imprescindible a la hora de usar redes neuronales es estandarizar las entradas de la red. En las redes de regresión lineal es muy favorecedor el hecho de normalizar también las salidas para evitar saturaciones en los gradientes (con salidas grandes las derivadas tendrían valores cercanos a cero y no se realizarían modificaciones en los pesos usando *backprop*). La forma de normalizar/estandarizar con *HappyML* es haciendo uso de objetos de tipo `Transformer`. La normalización genera números en el intervalo [0, 1] y la estandarización aplica *z-score*, por lo que genera valores negativos y positivos con media 0. En las redes neuronales la estandarización aporta mejores resultados debido a la variación de signo.

```
// Estandarización del dataset.
happyml::Standarizer std(dataset);  // Obtiene la media y stddev.
std.apply(dataset);  // Aplica la transformación al dataset.

// Creación de la red neuronal con pesos aletorios en sus conexiones.
int n_layers = 3;
// happyml::NeuralNetwork nn(n_layers, n_inputs, ... , n_outputs);
happyml::NeuralNetwork nn(n_layers, dataset.d, 100, dataset.k);

// Entreno de la red usando batch gradient descent.
int iterations = 1000;
float alpha = 0.1;      // Learning rate
float lambda = 0;       // Regularizer param
nn.train(dataset, iterations, alpha, lambda);

// Entreno de la red usando stochastic gradient descent.
int batch_size = 8;
nn.sgdTrain(dataset, iterations, alpha, lambda, batch_size);
```

Figura A.17: Creación y entreno de una red neuronal con *batch* y *stochastic gradient descent* (SGD). El primer paso consiste en estandarizar el *dataset*.

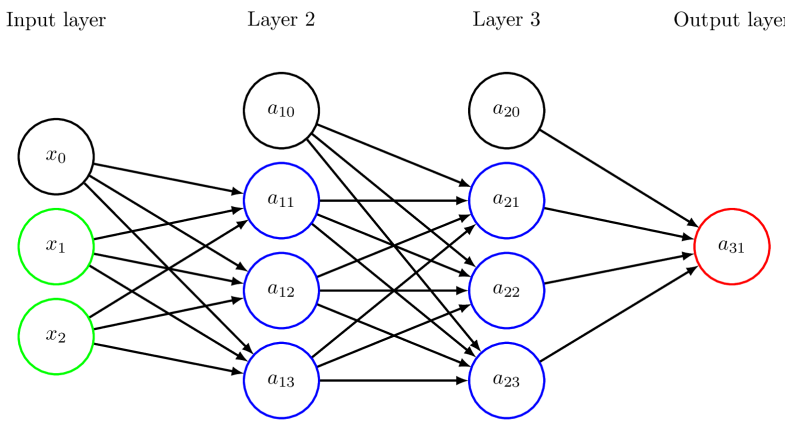


Figura A.18: Representación gráfica de una red neuronal con una arquitectura 2-3-3-1.

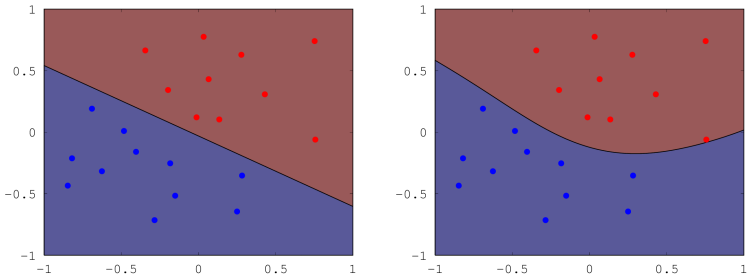


Figura A.19: Frontera de decisión de una red neuronal con arquitectura 2-1 (izquierda) y 2-10-1 (derecha). Se observa que en la derecha se obtiene una frontera de decisión diferente a la recta gracias a la presencia de una capa oculta.

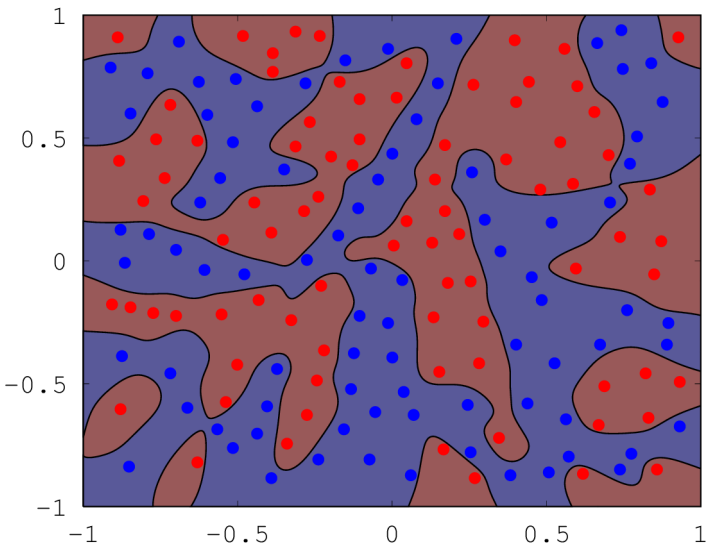


Figura A.20: Una buena forma de probar el correcto funcionamiento de una red neuronal es comprobar que sea capaz de hacer *overfitting* sobre un conjunto de datos [Li et al.].

Las redes neuronales también pueden ser usadas en problemas de regresión. *HappyML* cuenta con la clase `NNRegression` que nos ayuda en esa tarea. Todas las capas usan la función de activación $\tanh$ a excepción de la última de todas que utiliza una activación lineal.

```
// Estandariza las entradas y salidas.
happyml::StandarizerXY std(dataset);
std.apply(dataset);

// Creación de la red neuronal con pesos aletorios en sus conexiones.
int n_layers = 4;
happyml::NNRegression nn(n_layers, dataset.d, 1000, 1000, dataset.k);

// Entreno de la red usando batch gradient descent.
int iterations = 10000;
float alpha = 0.01;      // Learning rate
float lambda = 0;        // Regularizer param
nn.train(dataset, iterations, alpha, lambda);

// Guarda predicciones en el intervalo [-1, 1]. Tomando 500 muestras.
nn.saveSampling("line.data", -1, 1, 500);
```

Figura A.21: Creación y entreno de una red neuronal con *batch gradient descent*.

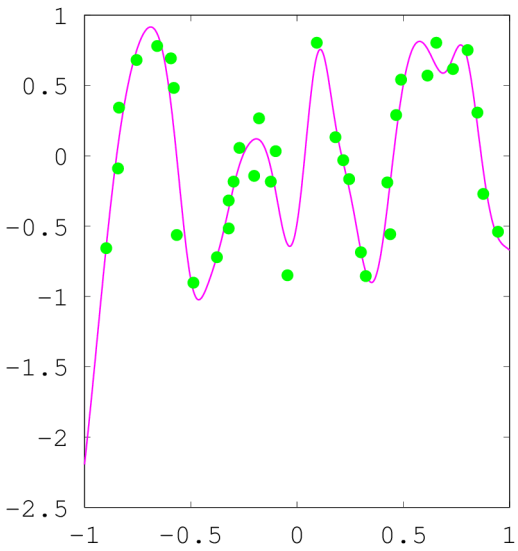


Figura A.22: Ajuste de una red neuronal. El α usado es más bajo en comparación al usado en las redes neuronales de clasificación, debido a que altos valores de α satura los pesos y hacen que el error tienda a infinito. Para evitar esto podría introducirse alguna constante a la hora de actualizar los pesos [Li et al.].

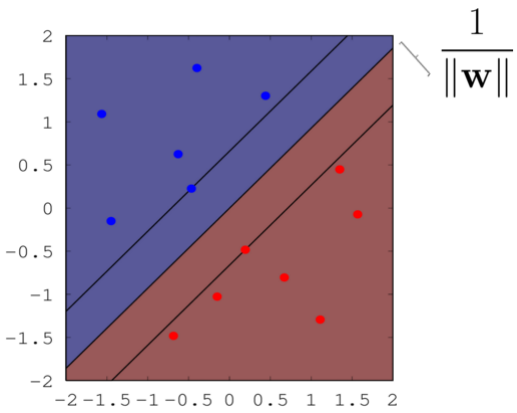


Figura A.23: SVM con *kernel* lineal. En la imagen se encuentra representado el margen maximizado, de tamaño $\frac{1}{\|\mathbf{w}\|}$. Los dos puntos que se encuentran sobre los márgenes son los llamados vectores de soporte. Tras resolver el problema de programación cuadrática, son aquellos puntos cuyo $\alpha > 0$.

A.7 SVM

Las SVM tienen como objetivo encontrar el borde de decisión que maximice el margen entre los puntos. Tras una derivación matemática se llega a un problema de programación cuadrática.

$$\mathcal{L}(\alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y_n y_m \alpha_n \alpha_m \mathbf{x}_n^{\mathbf{T}} \mathbf{x}_m \tag{A.7}$$

$$\text{sujeto a} \quad \alpha_n \geq 0 \quad \text{para } n = 1, 2, \ldots, N \tag{A.8}$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \tag{A.9}$$

El algoritmo usado para resolver el problema de programación cuadrática que se encuentra tras la maximización del margen, es el *Sequential Minimal Optimization* (SMO). No se ha usado la versión original del algoritmo [Platt, 1998], sino que se ha utilizado una versión simplificada [CS229].

Existen dos clases diferentes de SVM implementadas: `SVMLinear` usa un *kernel* lineal y `SVMGaussian` usa un *kernel* gausiano. La implementación de otro tipo de *kernels* es trivial, basta con extender una clase de `SVM` e implementar el método `double kernel(vec x1, vec x2) const`.

Las SVM cuentan con ún parámetro C que es el encargado de controlar la compensación entre los errores de clasificación y los márgenes (*soft margin*). Para más información acerca de los parámetros consultar documentación.

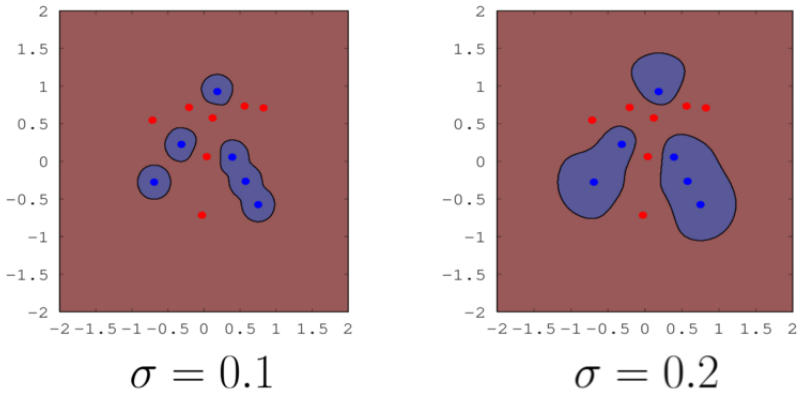


Figura A.24: SVM con *kernel* gaussiano y diferentes valores de σ.

A.8 Reducción de la dimensionalidad

HappyML cuenta con una implementación del *Principal Component Analysis* (PCA) y del *Linear Discriminant Analysis* (LDA). El funcionamiento de estos algoritmos es muy similar, su misión es la de reducir el número de dimensiones de una colección de vectores.

A día de hoy, la implementación del LDA en *HappyML* sólo funciona para *datasets* binarios. La aplicación en aquellos *datasets* que contengan más de dos clases dará error.

El PCA sin embargo, se trata de un algoritmo de aprendizaje no supervisado, por lo que no tiene en cuenta las clases. Puede ser aplicado en cualquier *dataset*.

Para ilustrar su funcionamiento de estos dos algoritmos se va a aplicar PCA y LDA a 40 datos del conocido *dataset* MNIST [LeCun and Cortes, 2010] (ver figura A.27). Este *dataset* está compuesto por números del 0 al 9. Se han extraído de él 20 unos y 20 ceros. Tras aplicar PCA a las imágenes de 28x28 píxeles, se ha reducido el número de dimensiones a 2, pudiendo representar el resultado en una gráfica. El resultado se observa en la figura A.26. Un extracto del código de este ejemplo (aquellas partes relativas al PCA y al LDA) se encuentra en A.25.

A ambos algoritmos se le puede indicar que utilicen las dimensiones necesarias para mantener un porcentaje deseado de varianza. Si queremos que con el PCA se retenga el 99 % de varianza se obtiene que son necesarias 32 dimensiones. Eso supone una reducción importante teniendo en cuenta que las dimensiones iniciales de los vectores de entrada eran $28 \cdot 28 = 784$.

```
// El constructor calcula los eigenvec pero no modifica el dataset.
happyml::PCA pca(dataset, 2);  // Conservar dos dimensiones
pca.apply(dataset);  // Aplica la transformación.
cout << "Var retenida: " << pca.getRetainedVariance() << endl;

// El constructor calcula los eigenvec pero no modifica el dataset.
happyml::LDA lda(dataset, 2);  // Conservar dos dimensiones
lda.apply(dataset);  // Aplica la transformación.
cout << "Var retenida: " << lda.getRetainedVariance() << endl;
```

Figura A.25: Código de como aplicar PCA y LDA. Se asume la existencia de una variable `dataset` del tipo `DataSet` con sus características debidamente normalizadas o estandarizadas.

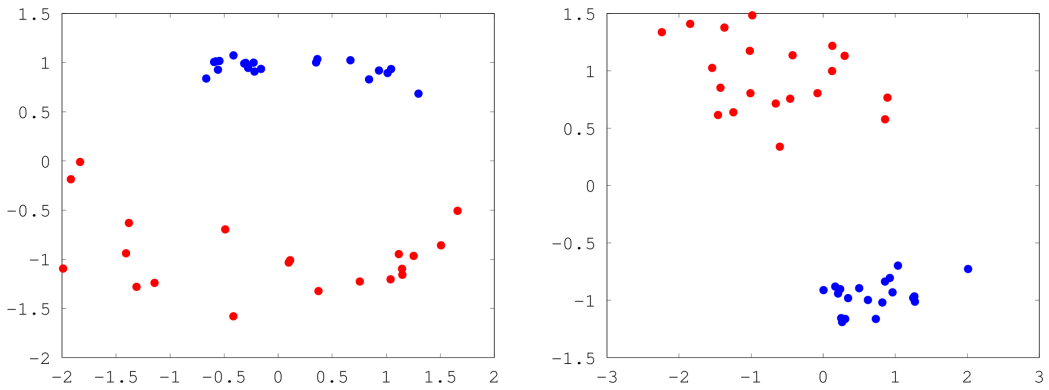


Figura A.26: En rojo los 0s y en azul los 1s. PCA (izquierda) y LDA (derecha) usando los 2 primeros vectores propios que más varianza mantienen (cerca del 45 %).

A.9 Futuro

La librería cuenta con bastantes aspectos mejorables pero cumple con el objetivo de ser comprensible a la lectura y sencilla de usar. La creación de métodos de minimización de errores que puedan ser aplicados a distintos modelos, o permitir que las funciones de activación y la estructura de una red neuronal sean modificables, complicarían más el proyecto. En principio no se piensa modificar mucho más la librería, únicamente para refactorizar algunos archivos y corregir posibles *bugs* que están esperando a ser descubiertos.

Figura A.27: Algunas de las imágenes contenidas en el MNIST.

Apéndice B

Implementación

En este apéndice se hará un recorrido rápido sobre cómo se han estructurado los archivos del proyecto, así como algunos detalles relevantes sobre la implementación de los agentes. El objetivo de este apéndice no es el de ser una guía completa que permita entender todo el código y cómo ejecutarlo, sino el de aportar una idea general que facilite mucho el uso de aquél que decida utilizarlo.

La estructura de directorios está organizada de la siguiente manera:

- **src**. Código fuente.
- **include**. Archivos de cabecera.
- **data**. Resultados de la ejecución de los experimentos.
- **scripts**. Código dedicado a la ejecución automatizada de pruebas y a la obtención de los gráficos con el uso de *gnuplot*.
- **report**. Directorios con los archivos LaTeX e imágenes usados para realizar la presente memoria. La memoria accede a los gráficos y a los resultados de las pruebas guardados en archivos CSV, de forma que cualquier cambio en los resultados de las ejecuciones produce un cambio en el documento.

B.1 Compilación

El primer paso es obtener el código del ALE de la página web oficial o del repositorio de *GitHub* y una vez en el directorio del proyecto ejecutamos:

```
$ cmake -DUSE_SDL=ON -DUSE_RLGLUE=OFF -DBUILD_EXAMPLES=ON .
$ make -j 4
```

Con eso ya tenemos compilado el entorno haciendo uso de **SDL** (*Simple DirectMedia Layer*). Esto nos permitirá visualizar en una ventana el desarrollo del juego cuando lo deseemos. Aún compilado con SDL, para el entrenamiento de algoritmos de RL o para la realización de tests, se puede desactivar la visualización gráfica para acelerar la ejecución.

A la hora de compilar un agente inteligente en C++, el compilador debe de contener la ruta hacia `src` entre las rutas en las cuales buscará los archivos a incluir. Además, el compilador también debe de saber dónde

encontrar la librería dinámica generada tras la compilación (`libale.so`). Cuando se ejecute el programa, la variable de entorno `LD_LIBRARY_PATH` debe de contener la ruta donde se encuentra la librería.

Para una mayor comodidad, se ha optado por instalar manualmente la librería renombrando el directorio `src` a `ale` y copiándolo en `/usr/local/include`. También debemos de copiar el archivo `libale.so` a `/usr/local/lib`. Tras mover la librería dinámica se ejecuta

```
# ldconfig
```

y de esta forma ya podemos compilar con únicamente añadir al final del comando de compilación un `-lale`. Tras la instalación manual ya no necesitamos hacer uso de la variable `LD_LIBRARY_PATH` para ejecutar nuestros agentes.

Para compilar el proyecto, asumiendo que se encuentran instalados ALE y *HappyML*, basta con ejecutar los siguientes comandos:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

B.2 El ejecutable *Runner*

El programa *runner* es el punto de entrada de cualquier prueba. Este programa tiene el método *main* y es el encargado de la creación del agente y de controlar el bucle principal de la ejecución. Se encarga de iniciar el entorno ALE, activar o desactivar la visualización gráfica y el sonido, ejecutar las acciones devueltas por los agentes, llevar la cuenta de las puntuaciones, establecer el número de episodios a ejecutar, limitar el número de *frames* por episodio, grabar los juegos...

La configuración de cada ejecución se indica con argumentos. El mismo puntero de argumentos que recibe el *runner* es pasado al constructor de cada agente, por lo que los argumentos que se le pasan al *runner* pueden contener configuración específica del agente a ejecutar. Al inicio del archivo `runner.cpp` se encuentran los parámetros que recibe. En el constructor de cada agente se pueden consultar los parámetros que acepta. Como es lógico, aquellos agentes que heredan de otros, aceptan también los parámetros de su clase superior.

B.3 *Trainers*

Existen otros archivos ejecutables (`<tfg_root_dir>/src/trainers`) que son los encargados de realizar los entrenos de los algoritmos de SL. La ejecución de cada uno de esos programas guarda en un

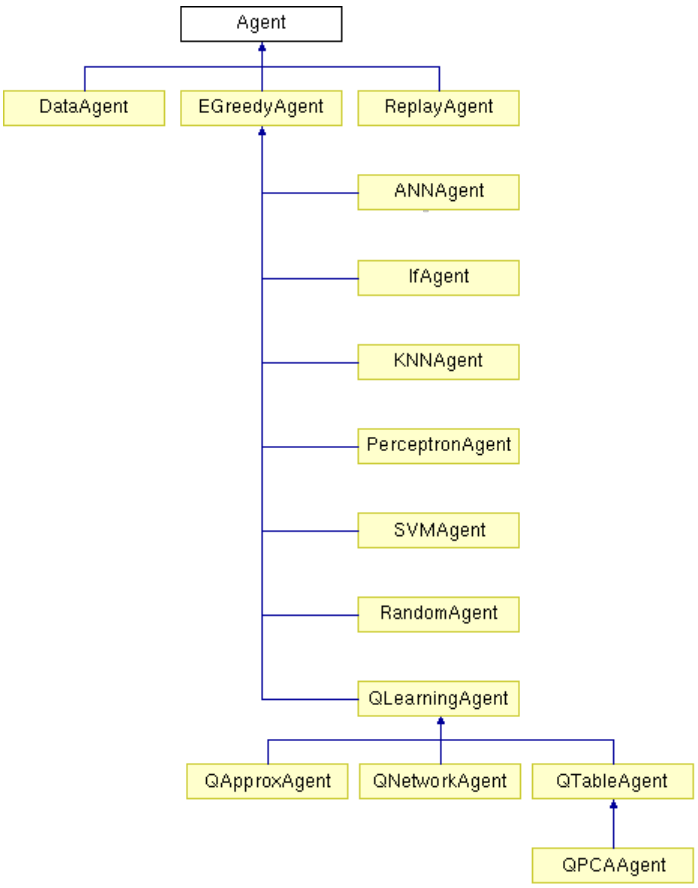


Figura B.1: Diagrama de clases de los agentes implementados.

archivo el resultado del modelo tras ser entrenado. De esta forma, el modelo puede ser cargado desde un agente sin la necesidad de realizar de nuevo el entreno.

B.4 Agentes

Durante el desarrollo del proyecto se han ido creando diferentes tipos de agentes. De la mayoría de ellos ya hemos ido hablando conforme íbamos haciendo los experimentos, pero hay algunos más de los que no se ha comentado nada. En la figura B.1 aparece la jerarquía de herencia de todos los agentes desarrollados.

A continuación vamos a hacer un breve repaso sobre el contenido de cada agente.

- La clase *Agent*. Es la encargada de la obtención de datos de la RAM así como de la gestión de los estados (`nobxpx`, `dx`, ...), se encarga de la introducción de ruido, contiene callbacks para saber cuando se empieza o acaba un episodio... El *runner* es el encargado de trabajar usando su interfaz sin tener en cuenta que tipo de agente es.

- *Data, Replay agent*. El *Data agent*, como ya se ha comentado durante el desarrollo de los experimentos,

se encarga de capturar la información de juego de un usuario y guardarla en un archivo. A partir de ese archivo, el *Replay agent* es capaz de reproducir el juego realizado por el humano.

- *Random agent*. Realiza movimientos aleatorios.
- *If agent*. Juega siguiendo unas reglas programadas por nosotros que son específicas para el *Breakout*.
- ϵ*-greedy agent*. Introduce una *policy* ϵ*-greedy*.
- *Perceptron, KNN, ANN, SVM agent*. Estos agentes utilizan el enfoque del SL: son entrenados a partir de unos datos e imitan el comportamiento encontrado en ellos. Entre ellos se encuentra el *SVM agent*, del que se no ha comentado nada en la memoria debido a que no se ha encontrado el tiempo suficiente para hacer buenas pruebas sobre su funcionamiento.
- *Q-Learning agent*. Implementa el algoritmo del *Q-Learning* sin tener en cuenta la forma de representar la función $Q(s,a)$. Es una clase abstracta.
- *QNetwork agent*. Utiliza una red neuronal que aproxima el valor de una función $Q(s,a)$. Después de cada ciclo los pesos de la red son ajustados para que se aproxime al *reward* obtenido. Por el momento no se han conseguido resultados con este agente debido a que diverge.
- *QApprox agent*. Utiliza una red neuronal que aproxima el valor de una función $Q(s,a)$. A diferencia del *QNetwork agent*, la aproximación se saca de una tabla obtenida usando un *QTable agent*.
- *QPCA agent*. Es un tipo de *QTable agent* que utiliza PCA con el objetivo de reducir las dimensiones de los vectores de entrada. El PCA, sin embargo, no consigue reducir ninguna dimensión sin perder una gran cantidad de información, por lo que no produce ninguna mejora.

En los siguientes apartados se comentarán algunos de los datos más interesantes sobre la implementación de alguno de los agentes.

B.5 Data agent

Este agente es utilizado para recolectar datos de un jugador humano y poder así crear *datasets* que puedan ser utilizados posteriormente por algoritmos de SL.

Uno de los principales problemas al construir este agente fue el leer los datos de entrada de forma en la que no bloquee la terminal. Usando *ncurses* se solucionó facilmente el problema. Para evitar el típico *delay* que tienen todos los teclados antes de repetir una tecla que mantienes presionada, se ha utilizado el siguiente comando:

```
$ xset r rate 30 100
```

De esta forma se aumenta el muestreo del teclado al inicio de la ejecución de este agente. Al finalizar el agente la configuración del teclado se vuelve a dejar tal y como estaba usando

```
$ xset r rate
```

B.6 Q-Table Agent

Este agente implementa el algoritmo *Q-Learning* basándose en una tabla.

La función $Q(s,a)$ está implementada con un `unordered_map`. Inicialmente se optó por utilizar un array multidimensional (una dimensión por cada una de las características que componen un estado), pero fue descartado. El motivo del cambio estuvo relacionado con el no poder utilizar números negativos o con decimales como índice. El uso de un `unordered_map` también facilita el hecho de poder añadir tantas dimensiones como queramos sin tener que modificar el código (sin modificar el número de dimensiones de un array multidimensional).

```
typedef unordered_map<string, double> Q;
```

Figura B.2: Definición del tipo de dato `Q` usado para representar la función $Q(s,a)$.

La clave de tipo `string` se obtiene de la representación en tipo texto del estado más la acción. Por ejemplo, para el estado `nobxpx` la clave sería el `string` formado por "$Ball_y, Ball_{vx}, Ball_{vy}, Diff_x, Action$".

La función $Q(s,a)$ puede guardarse fácilmente en archivos de texto en formato CSV, lo que nos puede facilitar la depuración.

Índice alfabético

Agente, 32
Arcade Learning Environment, 22, 27
Atari 2600, 21

Breakout, 22

Curse of dimensionality, *véase* Maldición de la dimensionalidad

Data agent, 40, 98
Dataset, 41
Deep Learning, 23

ϵ*-greedy*, 34
Episodio, 29

Function approximation, 63

If agent, 33

LDA, 93

Maldición de la dimensionalidad, 60

Neural Network, 88
Noop agent, 32

PCA, 93
Perceptrón, 40

Q-Learning, 50

Random agent, 33
Reinforcement Learning, 21, 47, 48
Runner, 96

Sequential Minimal Optimization, 92
Supervised Learning, 21
SMO, *véase* Sequential Minimal Optimization
Support Vector Machine, 92
SVM, *véase* Support Vector Machine

TD-Gammon, 23

Z-Score, 42

Referencias

Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin. *Learning From Data*. AMLBook, 2012. ISBN 1600490069, 9781600490064.

Atari, Inc. *Breakout*. Atari, Inc., 1978. Manual de usuario original del juego *Breakout*, © 1978 Atari, Inc. *https://atariage.com/manual_html_page.php?SoftwareID=889* Visitado por última vez el 17 de febrero de 2016.

AtariAge, 1998. URL `http://www.atariage.com/system_items.html?SystemID=2600&ItemTypeID=ROM`. Último acceso 23 de febrero de 2016.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

CS229. Simplified SMO Algorithm. URL `http://cs229.stanford.edu/materials/smo.pdf`.

Georgia Tech. Reinforcement learning. URL `https://www.udacity.com/course/reinforcement-learning--ud600`.

M. J. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015. URL `http://arxiv.org/abs/1507.06527`.

Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. URL `http://dl.acm.org/citation.cfm?id=645754.668382`.

F.-F. Li, A. Karpathy, and J. Johnson. Cs231n: Convolutional neural networks for visual recognition 2016. URL `http://cs231n.stanford.edu/`.

Y. Liang, M. C. Machado, E. Talvitie, and M. H. Bowling. State of the art control of atari games using shallow reinforcement learning. *CoRR*, abs/1512.01563, 2015. URL `http://arxiv.org/abs/1512.01563`.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 02 2015. URL `http://dx.doi.org/10.1038/nature14236`.

J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, January 1998. URL http://research.microsoft.com/apps/pubs/default.aspx?id=68391.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 10 1986. URL http://dl.acm.org/citation.cfm?id=65669.104451.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3):210–229, July 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210. URL http://dx.doi.org/10.1147/rd.33.0210.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

R. S. Sutton and A. G. Barto. *Reinforcement Learning, An Introduction*. MIT Press Cambridge, MA, USA © 1998, 1998. ISBN 0262193981.

B. Tanner and A. White. Rl-glue: Language-independent software for reinforcement-learning experiments. *The Journal of Machine Learning Research*, 10:2133–2136, 2009.

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, Mar. 1994. ISSN 0899-7667. doi: 10.1162/neco.1994.6.2.215. URL http://dx.doi.org/10.1162/neco.1994.6.2.215.

VintageToySeller. Atari 2600 video computer system game cartridges lot breakout defender pinball. URL https://www.vintagetoys.com/toys/classified/30977. Último acceso 3 de abril de 2016.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's Collage, 1989.

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

Wikipedia. List of Atari 2600 games. URL https://en.wikipedia.org/wiki/List_of_Atari_2600_games. Último acceso 3 de abril de 2016.