**INTRODUCCIÓN A LAS REDES NEURONALES**

INDICE

[**INTRODUCCIÓN: LA IMPLEMENTACIÓN DE LA FUNCIÓN XOR** 2](#_Toc172392097)

[**1.- DESARROLLO HISTÓRICO DE LA IA, PRINCIPALES HITOS: 1950 A 1980** 3](#_Toc172392098)

[1.1.- Un primer modelo técnico de redes neuronales: Maculloch - Pitts 4](#_Toc172392099)

[1.2.- La computación moderna. Hacia el congreso Dartmouth 6](#_Toc172392100)

[1.3.- La aparición del perceptrón 8](#_Toc172392101)

[1.4.- El primer invierno de la IA 10](#_Toc172392102)

[**2.- SUPUESTOS BÁSICOS DE LOS MODELOS DE REDES NEURONALES. EL ALGORITMO DE BACKPROPAGATION** 12](#_Toc172392103)

[2.1. El teorema de aproximación de funciones y las redes neuronales multicapas 13](#_Toc172392104)

[2.2. El algoritmo de backpropagation 15](#_Toc172392105)

[2.3. Usando *redes neuronales supervisadas* en la actualidad, algunas librerías útiles en R y python 21](#_Toc172392106)

[**3.- DISTINTOS TIPOS DE REDES NEURONALES** 22](#_Toc172392107)

[**4.- REDES NEURONALES NO SUPERVISADAS** 26](#_Toc172392108)

[**5.- INTRODUCCIÓN AL DEEP LEARNING** 32](#_Toc172392109)

[**6.- NOCIONES BÁSICAS SOBRE REINFORCEMENT LEARNING** 37](#_Toc172392110)

[6.1.- Un problema clásico: El viajero 37](#_Toc172392111)

[6.2.- Breves pinceladas de programación dinámica e introducción histórica al Reinforcement Learning 40](#_Toc172392112)

[6.3.- Las bases del Reinforcement Learning 43](#_Toc172392113)

[El agente 43](#_Toc172392114)

[Características de un entorno tipo MDP 44](#_Toc172392115)

[Funciones – valor de un entorno tipo MDP 45](#_Toc172392116)

[Políticas 45](#_Toc172392117)

[6.4.- Desarrollo de un caso típico de reinforcement learning: El K-armed bandit 46](#_Toc172392118)

[El dilema exploración – explotación 46](#_Toc172392119)

[Planteamiento y breve análisis del *K-armed bandit* 47](#_Toc172392120)

[Implementación y solución del *MAB*. Métodos greedy 48](#_Toc172392121)

[Las Ecuaciones de Bellman 50](#_Toc172392122)

[**BIBLIOGRAFÍA** 51](#_Toc172392123)

# **INTRODUCCIÓN: LA IMPLEMENTACIÓN DE LA FUNCIÓN XOR**

El problema de la función *XOR* aparece en los años 60, tal y como se detalla a continuación, tras unos años de éxitos e incluso de euforia en el campo de la *Inteligencia Artificial*, de repente dos autores punteros publican en 1969 el libro titulado “Perceptrons: An introduction to Computational Geometry” donde se demuestra que un *perceptrón* no puede aprender funciones linealmente como la *XOR* que resulta ser linealmente no separable, por tanto se requería algo más complejo que un simple *perceptrón* para integrar la anterior función.

En este sentido Marvin Minsky y Seymour Papert en definitiva no negaban que con la *inteligencia artificial* se pudiera asimilar la función *XOR*, el gran problema era que como se verá también los *algoritmos* *supervisados* (al igual que los *no supervisados*) no estaban aún suficientemente optimizados y la tecnología de la época para tales fines (aún basada en tuvos de vacío) no permitía llevar en tiempos óptimos entrenamientos de una *red multicapa* que sin ningún problema habría podido entrenar la anterior función, así pues una de las causas de lo que se conoce como “primer invierno de las redes neuronales” resultó no ser sólo la suficiencia de recursos tecnológicos sino también el hecho de que habría que esperar unos 10 años para que apareciese el *algoritmo de backpropagation* básico en este tipo de implementaciones y que permitiría nuevos avances en este campo,

Tal y como se puede observar en el PDF generado en RMARKDOWN y que fue explicado en un Grupo de Usuarios de R en Madrid cuya información estaría en <http://madrid.r-es.org/2016/01/> y cuyos resultados se comentan más adelante. Mientras un único *perceptron* no es capaz de aprender una capa *XOR*, si se le añade una simple capa oculta, claramente con la tecnología actual se alcanza el resultado esperado en cuestión de segundos de entrenamiento, lo que para los años 60-70 sería todo un logro técnico

Imagen que contiene Escala de tiempo

Descripción generada automáticamente

**Figura 1 Charla del grupo de usuarios de R: “Introducción a las Redes Neuronales”**

# **1.- DESARROLLO HISTÓRICO DE LA IA, PRINCIPALES HITOS: 1950 A 1980**

En (Bonifacio, M; Sanz Molina, A. 2001) se atribuye el inicio de la Historia de las Redes Neuronales, al científico aragonés Santiago Ramón y Cajal (1852 – 1934), descubriendo en 1888 en la cátedra de Histología de la Facultad de Medicina de la Universidad de Barcelona como se transmitían los impulos entre las células nerviosas que componían la matera gris del sistema nervioso, transmitiéndose no a través de una red continua como se creía hasta el momento sino a través de *neuronas* que se encontraban separadas las unas de las otras funcionando como unidades independientes, esto le permitió entre 1899 y 1904 publicar su obra magna “Histología del sistema nervioso del hombre y de los vertebrados” y que le valió en 1906 para conseguir el Premio Nobel en Medicina que tuvo que compartir con Camillo Golgi.

Así pues se establece que el sistema nervioso estaba compuesto por células individuales que se denominan neuronas, ampliamente interconectadas entre sí, estableciendo que la información podía fluir de unas a otras neuronas desde las dendritas hasta el axón atravesando el soma.

Diagrama

Descripción generada automáticamente

**Figura 2 Estructura de una neurona biológica típica**

**Source: Wikipedia**

Estas señales se transmitirían de forma eléctrica o química (esta última la más habitual) a través de las conexiones presinápticas y postsináptica entre ellas, fluyendo por tanto en una única dirección. En distancias cortas la señal no se degradaba, pero a distancias largas, la naturaleza “entendió” que debía codificar la señal en forma de frecuencia de pulsos digitales.

Por lo que el aprendizaje en los modelos biológicos se tiene por variaciones de la intensidad de las sinapsis que puede ser modulada a lo largo del tiempo mediante:

-El establecimiento de nuevas conexiones

-Ruptura de otras

-Muerte o eliminación de neuronas

-etc

## 1.1.- Un primer modelo técnico de redes neuronales: Maculloch - Pitts

Aunque Ramon y Cajal sin duda descubre las neuronas, la idea moderna de su aplicabilidad práctica a un elemento mecánico capaz de ser inteligente que por emulación al cerebro humano cabe fijarla como punto de partida en los estudios que en durante la década de 1940 realizaron Warren Sturgis Maculloch (1898 – 1969) y Walter Harry Pitts (1923 – 1969) cuando publican en 1943 el artículo denominado “Cálculo lógico de ideas inherentes en la actividad nerviosa”.

El paper procede de una invitación por parte de McCulloch cuando éste invita en 1942 a Pitt a su casa en Chicago y discutieron sobre si el sistema nervioso podría ser considerado una clase de dispositivo de informática universal como ya fuera descrito por el pensador racionalista Gottfried Leibniz (1646 – 1716) inventor del sistema binario y desarrollador independiente junto con Newton del Cálculo Infinitesimal pero que además consideraba la naturaleza compuesta por *mónadas* indivisibles y armoniosas.

Así pues planean el primer modelo matemático de *red neuronal artificial,* donde la unidad standard del modelo es una *neurona* sencilla que aún a día de hoy sigue siendo el estándar de referencia tanto en el campo de las *redes neuronales* como incluso de el del *deep learning* que se usa en la actualidad:

Diagrama

Descripción generada automáticamente

**Figura 3 Modelo matemático de la *neurona* McCulloch - Pitt**

**Source:** [**https://dcain.etsin.upm.es/~carlos/bookAA/05.1\_RedesNeuronalesIntroduccion.html**](https://dcain.etsin.upm.es/~carlos/bookAA/05.1_RedesNeuronalesIntroduccion.html)

Este primer modelo hace uso de la función Heaviside, que aseguraba una salida de tipo 0 – 1. Como características adicionales cabe enumerar las siguientes:

* *Umbral de* activación: La *neurona de McCulloch-Pitts* se activa cuando la suma ponderada de las *entradas* supera un cierto *umbral* y produce una salida
* *Entradas binarias*: Las *entradas* a la neurona son binarias, es decir, toman valores de 0 o 1. Esto refleja la idea de que una *neurona* biológica está o no está activada
* *Pesos sinápticos*: Cada *entrada* tiene asociado un *peso sináptico* que indica su importancia relativa para la activación de la neurona. Los *pesos* son valores numéricos que multiplican las entradas antes de sumarse para determinar si la neurona se activa o no
* *Función de activación*: La neurona utiliza una función de activación simple, que simplemente compara la suma ponderada de las entradas con el umbral. Si la suma supera el umbral, la neurona emite una salida de 1; de lo contrario, emite una salida de 0
* *Salida binaria*: La salida de la *neurona* es binaria de modo que será 1 si está activada y 0 en caso contrario (se muestra abajo ejemplo con 2 entradas):

Por tanto con estas *neuronas* ya se podía construir toda la lógica booleana al estar incluidas las funciones *AND, OR* y *NOT* del siguiente modo:

* *AND*: (conjunción lógica) se configura por medio de una neurona con dos sinapsis excitadoras y un umbral de 2. Sólo cuando se activan ambas dendritas se alcanza el umbral y se produce la excitación de la neurona
* *OR*: (disyunción lógica) se construye del mismo modo, pero estableciendo un umbral de 1
* *NOT*: (negación) se configura como una neurona con una sinapsis inhibidora y un umbral de 0. Esta neurona se dispara siempre y cuando no reciba una señal inhibidora

Ejemplo 01: Implementación de la conjunción lógica *AND*

Se consideran 2 *entradas* binarias y se trata de reproducir la siguiente *tabla de verdad* que debe ser reproducidad por el modelo:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

La pregunta es ¿Cuánto deben valer los pesos *pesos* y el *bias* para que la siguiente función reproduzca la tabla de verdad?

En tiempos de McCulloc – Pitts aún no se tenía un algortimo de entrenamiento válido y el ajuste se hacía directamente, en este caso la función quedaría tal como sigue:

Y esta neurona debidamente entrenada con la tecnología de la época, podría conectarse a otra.

Nótese que el principio básico para entrenar una red va a ser tanto en esa época como en la actualidad encontrar los *pesos* y el *bias* bien heurísticamente como se hacía en aquella época, bien mediante algún *algoritmo* adecuado tal y como se comentará más adelante

\_\_\_\_\_

## 1.2.- La computación moderna. Hacia el congreso Dartmouth

En la época de desarrollo del *modelo McCulloch – Pitts* coinciden varios hechos que no sólo cimientan la *Inteligencia Artificial* sino que fundamentaron toda la computación moderna, en este sentido cabe destacar la figura de Alan Turing (1912 – 1954) que antes de tratar de responder en 1950 a la pregunta de “¿Puede pensar una máquina?” y describir en consecuencia su famoso *Test de Turing*, ya en 1936 define lo que se conoce como una *máquina de Turing* que es un modelo matemático‌ que representa una computadora idealizada con capacidad para ejecutar cualquier *algoritmo*. Dicha *maquina de Turing* es la base de los ordenadores actuales y básicamente se compone de una cinta infinita (memoria de la computadora rellena de 0 – 1 o Missing), un cabezal lector/escritor (que puede o bien editar un símbolo, o bien moverse a izquierda o derecha, o bien avanzar al siguiente estado), un registro de estado (que informa cuándo debe detenerse la máquina) y una tabla de instrucciones tal y como podría muestra en el siguiente ejemplo.

Ejemplo 02: Creando una *máquina de Turing*

En la siguiente web: <https://morphett.info/turing/turing.html#LoadMenu> se observa que si se transcribe el código *ejemplos/programaTuring.txt* se reproduce el *algoritmo* 2x + 1 donde puede verse una versión extendida y bien explicada en el siguiente muy recomendable vídeo:

<https://www.youtube.com/watch?v=NS-NQ5mCSs8>

\_\_\_\_\_

Así pues gracias al desarrollo del anterior concepto que desemboca en la creación de la *maquina de Turing universal* y a la arquitectura que otro de los grandes padres de la computación como era John von Neuman (1903 – 1957) diseña, ya en 1946 se lanza el *ENIAC*, el primer computador moderno que por la época ocupaba toda una habitación a diferencia de los portátiles actuales y que incorporaba la *arquitectura de John von Neuman* que a su vez incorporaba el concepto de *máquina de Turing universal*.

Por tanto fue una época eufórica que, a pesar de las limitaciones teóricas dadas por el famoso *teorema de incompletitud* de Kurt Gödel (1906 – 1978) enunciado en la década anterior, todo hacía pensar que se podría llegar a crear máquinas pensantes, de hecho a finales de la década de los 40, aparece la figura de *Donald* *Hebb* que en 1949 publica el libro “Teoría de la conducta” donde define la conocida *regla de aprendizaje Hebb*:

“Las conexiones sinápticas se fortalecen cuando 2 o más neuronas se activan de forma contigua en el tiempo y en el espacio”

donde: es la salida de una neurona de la red

es la entrada a una neurona de la red

es la tasa de aprendizaje

Nótese que en la iteración *t* + 1, si la entrada y la salida del estado anterior están activadas, el resultado del peso final será el del estado *t* más un refuerzo que es la *tasa de aprendizaje*, en caso contrario se queda igual e incluso si la salida binaria en vez de suponerse 0 – 1 se supone (-1) y (+1) podría salir debilitada.

Así pues con todos estos mimbres en 1956 se celebra el primer congreso de *inteligencia artificial* denominada en inglés como la *Dartmouth Summer Research Project on Artificial Intelligence* donde se centra las bases de los actuales supercomputadores y se empieza a organizar y a estructurar el conocimiento en la materia existente hasta la época. Organizado por John McCarthy (primero en acuñar el término *inteligencia artificial*) junto con Marvin Minsky, Nathaniel Rochester y Claude Shannon (este último padre de la *teoría de la comunicación*)

Un periódico con texto e imágenes

Descripción generada automáticamente

**Figura 4 Padres de la AI**

**Source:** <https://indiaai.gov.in>

## 1.3.- La aparición del perceptrón

A finales de los 50 un licenciado en letras y doctor en filosofía, Frank Rosenblatt (1928-1971) construye el primer hardware del perceptrón denominado Mark I en 1957, Frank Rosenblatt estaba muy interesado por la psicología cognitiva e incluso llegó a ser jefe de la sección de sistemas cognitivo del Laboratorio Cornell Aeronaútico en Búfalo (Nueva York)

En 1959 escribe el libro “Principios de Neurodinámica” enuncia lo que se conoce como el:

Teorema Fundamental del Perceptrón:

Si los patrones usados para entrenar el perceptrón son sacados de 2 clases linealmente separables, entonces el algoritmo del perceptrón converge y toma como superficie de decisión la de un hiperplano entre estas 2 clases

\_\_\_\_\_

El perceptrón estaba destinado a ser realmente una máquina en vez de un algoritmo y por eso finalmente se construyó el Mark I Perceptron, basado en las ideas del perceptrón *de*Frank Rosenblatt. El Mark I Perceptron se dedicaba exclusivamente a la clasificación de imágenes. Todo el hardware estaba construido a medida y utilizaba potenciómetros para determinar los pesos de cada entrada por Perceptrón así como una cámara capaz de producir imágenes de 400 pixeles de resolución (20x20). Para ello se entrenó introduciendo cientos de fotografías de hombres y mujeres con diferentes estilos de pelo y maquillaje. Una vez terminado el proceso de entrenamiento, se introdujeron esta vez modelos de caras que no había visto nunca antes durante dicho proceso de entrenamiento. Finalmente, la máquina decidía si la foto era hombre o mujer con una elevada tasa de éxito, aunque como era monocapa aún dicha tasa de acierto no llegaba a ser como las actuales.

Pero ¿En qué consistía este modelo y se diferenciaba de la *neurona* de *McCulloch – Pitts*?

* Aunque ambos modelos trabajan adecuadamente en datos linealmente separables, el modelo *McCulloch – Pitts* sólo trabaja con *entradas* booleanas, mientras que el *perceptrón* puede trabajar con *entradas* reales
* En el modelo *McCulloch – Pitts* las *entradas* no son ponderadas mientras que en el *perceptrón* el modelo puede tomar pesos con respecto a las *entradas* proporcionadas

Así pues, se tiene ya un modelo entrenable bajo los conceptos desarrollados por *Hebb*:

Diagrama

Descripción generada automáticamente

**Figura 5 Esquema de una *neurona perceptrónica***

**Source: wikipedia**

De modo que la neurona de Rosemblatt se matematiza bajo la siguiente formulación:

Donde algunos ejemplos de funciones son las de tipo “positivo” o “negativo”, “todo” o “nada”:

Finalmente hay que tener en cuenta, que ahora sí, el teorema mencionado anteriormente hace mención al modo en como el *perceptrón* puede “aprender” cómo separar una población linealmente separable siguiendo el siguiente proceso iterativo:

1. Si el enésimo elemento del vector de entrenamiento *x*(n) es clasificado correctamente por el vector de pesos *w*(n), no se lleva a cabo ninguna corrección:
2. En caso contrario se toma un valor η(n) que puede ser una constante mayor que 0 y se lleva a cabo las siguiente actualización:

Ejemplo 02: Entrenando mi primer *perceptrón*

En el código *ejemplos/programa01.py* se le presenta a un único *perceptrón*  una tabla de componentes binarias y se desea que la aprenda a partir de una inicialización de pesos nula:

Texto

Descripción generada automáticamente

En este caso los valores a tomar por el *perceptrón* serán 1 si el producto escalar (en este caso sin término *bias*) supera el valor de 0.5 y 0 en caso contrario. El programa genera paso a paso la transformación que tiene lugar en los pesos por la mera aplicación de la *regla de Hebb* y ofrece la siguiente estructura final, en la que fácilmente puede comprobarse que este *perceptrón* ha aprendido la tarea encomendada:

Texto

Descripción generada automáticamente

\_\_\_\_\_

Ejercicio 01: Utilizando el código anterior, definal una tabla de verdad para la función *AND* y muestre que el *perceptrón* puede aprenderla. ¿Qué ocurre cuando se diseña la tabla *XOR*?

\_\_\_\_\_

Aunque el

## 1.4.- El primer invierno de la IA

La euforia de los primeros años de la Inteligencia Artificial se vería truncada a finales de los 60 a raíz del siguiente libro donde interviene uno de los “padres” de la computación en concreto Marvin Minsky que publica “Perceptrons: An Introduction to Computational Geometry” (Minsky, M., and Part, S.; 1969)

El libro establece las bases del conocimiento de los perceptrones hasta la época e impone algunas restricciones a la teoría de Rosenblatt. El problema principal se refería a la posibilidad de tener algoritmos que permitiesen entrenar a redes perceptrónicas más avanzadas ya que hasta el momento se tenía claro como entrenar un perceptrón pero no la conexión de estos conjunta entre sí, así pues proponen un ejemplo como el anterior, donde una función lógica apartemente simple como la función *XOR* no había modo de ser aprendida con los alogritmos disponibles para la época.

Claramente con una *capa* oculta lo anterior se conseguiría, pero el problema radicaba en ¿Cómo se podía aprender los pesos de esta estructura más compleja?

Tabla

Descripción generada automáticamente



Lo anterior hizo que se paralizasen una gran parte de las inversiones e investigación en la IA, no obstante, a pesar de lo desesperanzador del trabajo del Minsky y Papert, algunos irreductibles siguieron durante los 70s obteniendo resultados una década después en los maravillosos años 80. Entre ellos cabe destacar a Bernard Widrow (1929 - ) que mantuvo viva la esperanza a la vez que desarrolla junto a su estudiante Ted Hoff la ADALINE y MADALINE

Conceptualmente la Adalina (en español), es lo mismo que un Perceptrón sin embargo, cambia el algoritmo de aproximación que en este caso es similar a la regla del gradiente descendente estocástico:

- La función respuesta es de tipo lineal

- Las entradas pueden ser de tipo continuo

- Existe un término adicional, denominado *bias* que no actúa como umbral

por tanto de arquitectura similar al perceptrón, la Adalina se expresa por la ecuación:

La adalina, se utiliza para la supresión de ecos telefónicos en las comunicaciones telefónicas o por satélite, tiene el inconveniente de funcionar bien si los patrones donde se aplican cumplen la hipótesis de independencia lineal, pudiendo por tanto fallar ante patrones linealmente separables (donde el *perceptrón* o los *svm* funcionan perfectamente) pero funcionando en general mejor que el *perceptrón* cuando dichos datos no son linealmente separables.

Su entrenamiento hace uso de la regla de Widrow-Hoff donde en realidad se trata de minimizar la función de error siguiente:

Llegándose finalmente a una expresión de actualización de pesos análoga al caso anterior del perceptrón:

Aslí pues en estas nuevas redes, se tienen salidas reales frente a las binarias del *perceptrón*, y además el *algoritmo de aprendizaje* resulta más cómodo y está conectado directamente a la minimización de una *función de error* general, asentando los cimientos del gran resugimiento de la IA en los 80 con la aparición del algoritmo de *backpropagation* que permitiría la extensión de las *redes* de un único *layer* a más de uno.

# **2.- SUPUESTOS BÁSICOS DE LOS MODELOS DE REDES NEURONALES. EL ALGORITMO DE BACKPROPAGATION**

Las *redes neuronales*, que se van a tratar van a ser las de tipo multicapa y en este capítulo se destacan los desarrollos de los años 80 que aún perduran en la actualidad, sobre todo el anteriormente mencionado *algoritmo de backprogation*.

En suma, las *redes neuronales* son unos sistemas que por lo general hacen uso de muy pocos supuestos sobre las observaciones en sí y al igual que en el caso de los *SVM*, dichos supuestos están más enfocados a la forma funcional en sí que toman sus especificaciones funcionales de tipo matemático.

Una hipótesis que se supone habitualmente es que, sobre todo en las redes basadas en aprendizajes de tipo supervisado, que las variables explicativas deberían estar *normalizadas* aunque, como se indica en (Bonifacio, M; Sanz Molina, A. 2001), en general no es un requisito.

Otra hipótesis tiene que ver con la *inicialización* de los *pesos* con el que comienza el proceso de entrenamiento. Las redes a considerar podrán usar como función de activación neuronales cualquer función, por tanto, al ser tanto estas como las entradas de tipo continuo, lo que va a ocurrir es que se generan superficies multidimensionales donde el riesgo de caer en mínimos locales y no alcanzar el mínimo absoluto o al menos mínimos suficientemente aceptables y estables va a ser elevado, por tanto, se huye de hacer *inicializaciones de pesos* nulas y se van a suponer de carácter aleatorio por lo que en estos entrenamientos va ser necesario fijar una *semilla de aleatorización* y también va ser conveniente en los modelos que se construya, probar con varias *semillas* para ver si la red es estable y se converge a los mismos o parecidos resultados o por el contrario se está ante inestabilidades e imposibilidad de llegar a conclusiones claras. Aunque existe eso sí algunas pautas para elegir los mencionados *pesos* que se resumen a continuación

- Conviene elegirlos de modo aleatorio (excepto que se tenga alguna información o idea a priori sobre los valores de las soluciones finales)

- En general se prefieren que sean pequeños y próximos a cero pero no nulos

Ejemplo 03: superficies *neuronales*

En los códigos que se encuentra en *ejemplos/programa02\_01.R* y *ejemplos/programa02\_02.R* se traza la superficie a la que da lugar las funciones cuadráticas cuando se trata de ajustar un *perceptrón* o en sentido más amplio una *adaline* (o una función en sentido totalmente general), mientras en el caso *lineal* las superficies derivadas de buscar el mínimo de una función lineal resultan ser “lisas” y por tanto la convergencia se obtiene de modo rápido y único, cuando la función ya se cambia y por ejemplo se toma una de tipo logístico, la superficie se torna mucho más compleja y con formas extrañas que pueden dar lugar a los problemas comentados anteriormente.

Imagen que contiene Gráfico de superficie

Descripción generada automáticamente Gráfico, Gráfico de superficie

Descripción generada automáticamente

\_\_\_\_\_

## 2.1. El teorema de aproximación de funciones y las redes neuronales multicapas

Tras el primer invierno de la IA, los años 80 van al ser el momento en el que se establecen las bases de los desarrollos actuales. El gran reto era poder al menos entrenar una *red* de 3 capas: *entradas – neuronas – output* ya que lo que se tenía era *monocapa* y como se sabe no permitía llegar ni al aprendizaje de la función *XOR*.

Ejemplo 04 La implementación de la función *XOR*

En el código que se encuentra en *ejemplos/programa03.R* se encuentra cómo con ayuda de la librería *neuralnet* en R es posible dar solución al problema planteado añadiendo una estructura de 3 capas en la que una capa es la *entrada*, otra es la capa de *neuronas oculta* y posteriormente hay una *neurona* de salida de tipo lineal.

Texto

Descripción generada automáticamente

Claramente si se pone el modelo simple con una única neurona (la de salida), no se llega a entrenar adecuadamente la anterior tabla de verdad, mientras que cuando se añade la capa se puede comprobar que se logra nuestro objetivo de modo sencillo,

Imagen que contiene Diagrama

Descripción generada automáticamente Gráfico

Descripción generada automáticamente con confianza media

¿Se podría con la neurona única entrenar la función *AND* y la *OR*?

\_\_\_\_\_

Así pues las *redes* que más se van a utilizar, al menos en una primera fase son las denominadas *Feedforward networks* o *FFN* que son redes que pueden tener más de una *capa* oculta denominadas *layers* y los pesos están dirigidos de las capas anteriores a las posteriores, de ahí el nombre en inglés de “hacia adelante” sin posibilidad de *retro-alimentación* (que se desarrollaron posteriormente y que a día de hoy toman su forma en los conocidos modelos *LSTM*).

Así pue, bajo las *FFN* uno de los resultados más notables que se obtuvieron a finales de la década de los 80, en concreto en 1989 por George Cybenko dio una versión específica para *funciones de activación sigmoideas* que se enuncia a continuación:

Teorema de aproximación de funciones de las FFN

Dada una *función de activación sigmoidad* que verifique:

Entonces la siguiente función, formada por sumas finitas de las anteriores funciones según la siguiente expresión:

Es densa en

\_\_\_\_\_

Es decir, el anterior teorema básicamente nos dice que las funciones *sigmoideas* pueden aproximar cualquier función continua en un conjunto compacto y además el desarrollo matemático no sólo se quedó aquí, sino que además lo anterior es cierto para las funciones límites de las *sigmoideas* es decir, para las funciones tipo *RELU* que no son más que las funciones básicas de los *perceptrones*, por tanto la década de los 80 no finalizará sólo con la posibilidad (al menos teórica) de entrenar cualquier *FFN*, sino que además acaba con este resultado que hacía soñar sobre las posibilidades a obtener con la IA del momento, si con una capa oculta se puede entrenar cualquier función ¡Qué no se podrá conseguir con múltiples capas ocultas!

Sin embargo, el anterior teorema tiene un problema y es que no se indica el número de *neuronas* necesario para conseguir tal fin y por tanto podría darse casos que se necesitase tal número de neuronas que en la práctica el entrenamiento no finalizase en un tiempo prudencial, aunque claramente las limitaciones a la IA de los 70 habían sido claramente superadas.

## 2.2. El algoritmo de backpropagation

El desarrollo del *algoritmo de backpropagtion* en su versión moderna se lo debemos a David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams en 1986 en el libro titulado “Parallel Distributed Processing: Explorations in the Microstructure of Cognition”, donde se explora en profundida la teoría y las aplicaciones de las *redes neuronales*, dedicando una parte significativa a explicar dicho *algoritmo*.

A continuación se va a explicar haciendo uso para ello notas basadas en parte en el curso de Andrew Ng. referencia académica en el mundo del Machine & Deep Learning. Se trata de analizar, usando los códigos python de *ejemplos/backpropagation* y en concreto se comienza por el denominado *rrnn\_basic1.py* donde se irá paso por paso resaltando las partes del código más importantes:

0.- Librerías básicas a considerar

Para que el código se ejecute correctamente, es necesario que se copie la carpeta *ejemplos/backpropagation* con todo su contenido, ya que como se comenta aquí se crea adicionalmente unas funciones que están en una librería definida por el usuario denominada *dnn\_utils\_v2* junto con otra que permite importar los datos y que se denomina *testCases\_v4a*

Pantalla de computadora con letras

Descripción generada automáticamente con confianza baja

En concreto la *librería* *dnn\_utils\_v2* contiene:

* Contiene unas funciones para usarse en Python como son la *sigmoid*(Z) y la *relu*(Z)
* La función *relu\_backward* que es el equivalente a la derivada parcial de *relu*
* La función *sigmoid\_backward* es el equivalente a la derivada parcial de la *sigmoid*

Nótese que un elemento esencial en este primer paso es la fijación de una semilla de aleatorización para que todo sea reproducible y se generen los resultados exactamente iguales

1.- Inicialización de parámetros de la red neuronal

Se considera una red de 3 *capas* (aunque realmente es de 2, una intermedia y otra de salida) con la siguiente estructura: LINEAR inpt > RELU outp > LINEAR inpt > SIGMOID outp

Texto

Descripción generada automáticamente

Ejercicio 02: Dibujar la arquitectura de la red neuronal indicada por 1.- sabiendo que las matrices de *pesos* resultan ser las siguientes:

Texto

Descripción generada automáticamente

\_\_\_\_\_

Aunque aquí se analiza el caso sencillo, cuando se consideran L *layers* lo que hay que tener muy en cuenta es que los tamaños de las *entradas* y *salida*s cuadren en términos de dimensiones matriciales

Pantalla negra con letras blancas

Descripción generada automáticamente con confianza media

2.- Implementación del módulo *forward propagation*

Como se observa, el *algoritmo* tiene 2 partes fundamentales en una los resultados se propagan hacia adelante (es la que se describe a continuación) y más adelante en el punto 4.- se calculan los errores yendo hacia atrás (de ahí su nombre). En esta parte se van a dar 3 pasos:

* Paso 1: Dinámica lineal hacia adelante. Sencillamente consiste en una función que permite aplicar adecuadamente los productos escalares (hay que fijarse en las dimensiones de entrada – salida) y se trata de emular la siguiente expresión que no es más que un producto escalar de vectores y una suma vecotrial:

Para ello se hace uso de la siguiente función donde es clave la función *numpy* denominada *dot* que en este caso dada una matriz *n* x *m* y un vector *m* x 1 daría lugar a un vector *n* x 1:

Texto

Descripción generada automáticamente

Nota: obsérvese que aunque aquí *b* es un número, en python existe el concepto de *broadcasting*, por tanto cuando se encuentra ese número ante un vector *n* x 1, automáticamente se conviere en un vector *n* x 1 con valores iguales en todas sus posiciones.

* Paso 2: Activación neuronal. Aquí es donde se usan las *neuronas*. Nótese que se van a considerar *sigomideas* y *RELU*s, pero podrían usarse también otros tipos de *neuronas*, el *algoritmo* es “agnóstico” al tipo de funciones a considerar aunque para ello habría que enriquecer el fichero *dnn\_utils\_v2,py* con las funciones deseadas.

Texto

Descripción generada automáticamente con confianza media

* Paso 3: Dinámica para las L *capas*. Este paso no se analiza en profundidad, ya que se hace el análisis sólo para una única capa oculta, pero en caso de considerarse debe tenerse en cuenta que se deben echar los valores hacia adelante a lo largo de las *L* capas

3.- Calcular una función de pérdida

Tras llegar a la *neurona output* o de *salida* el paso aquí es asociar lo que se conoce como una *función de pérdida* o *de coste* o en algunos casos, cuando se está ante un *output* continuo, se suele usar una *función cuadrática* o de *energía* (esto se comenta más adelante)

Para el caso que ocupa aquí, y dado que se va aplicar la red para un caso de clasificación, la *función de coste* a considerar es la siguiente:

Donde se observa que se compara la *entrada* observada binaria frente a al resultado predicho por la *red* en la última *capa* (que en este caso será la capa *L* = 1)

Texto

Descripción generada automáticamente

4.- Implementación del módulo *backward propagation*

Una vez calculada la *función de coste*, debe calcularse sus derivadas parciales respecto a los parámetros y modificar mediante pequeños incrementos (variaciones) los *pesos*, por el momento se ha seguido la primera parte del proceso mostrado en el siguiente esquema del profesor Andrew Ng:

Imagen de la pantalla de un celular con texto e imágenes

Descripción generada automáticamente con confianza media

**Figura 6 Esquema resumen del *algoritmo de backpropagation***

**Source: Andrew Ng Deep Learning course**

Para ver cómo se calculan los “gradientes hacia atrás”, se consideran los siguientes pasos:

* Paso 1: Dinámica lineal hacia atrás.

Aplicando ahora la idea del conocido *algoritmo del gradiente descendente* el primer paso es calcular las siguientes derivadas parciales:

Donde se supone conocido el dato de la derivada de la *función de coste* respecto a los parámetros:

Lo que se consigue con el siguiente código:

Texto

Descripción generada automáticamente

* Paso 2: Cálculo de las derivadas parciales de las funciones *simoideas* y *RELU*s mediante la aplicación de la regla de la cadena

*Texto

Descripción generada automáticamente*

* Paso 3: Caso de *L* *capas*. Este caso no se describe aquí, aunque análogo requerirá más código al necesitar repetir los pasos anteriroes unas *L* -1 veces adicionales

5.- Actualizar parámetros

El resultado final del paso anterior son los diferenciales (numéricos) tanto de los *pesos* como de los *bias*, así pues se aplica, vía el *coeficiente de aprendizaje* , la actualización de dichos parámetros de un modo análogo a como se hacen en el *gradiente descendente* que por ende es imitado bajo la *regla de Hebb* tal como sigue:

Texto

Descripción generada automáticamente

Ejemplo 05: Una predicción cuasi perfecta

El siguiente ejemplo puede resultar muy engañoso en cuanto a lo que se espera de las *redes neuronales*. En este caso, se ha utilizado un *dataset* denominado *banknote\_authentication.csv* donde con el código de *ejemplos/backpropagation/rrnn\_basic2.py* se crea un modelo siguiendo los pasos y las sintaxis anteriormente comentada para observar que, efectivamente funciona y donde se llega obtener unos resultados sorprendentes tal como sigue:

Gráfico, Histograma

Descripción generada automáticamente 

Estando con un *dataset* con unas proporciones de 0 y de 1 sobre el 60% - 40% el anterior resultado es cuanto menos sorprendente. No obstante, si se aplica la *regresión logística* que viene por defecto en *sklearn*, el resultado obtenido en el test resulta ser de:



Lo que significa que aunque el modelo de *redes* mejora a la *regresión logística*, dicha mejora es porque ya los datos tienen una alta capacidad predictiva y obviamente un modelo más avanzado que una *regresión logística* puede, como es este caso, fácilmente alcanzar la predicción perfecta.

En la realidad la anterior circunstancia no suele ocurrir y es conveniente estar muy seguro de que no existe ningún problema ni en las variables (sobre-predictividad) o en los datos (sesgos de selección)

\_\_\_\_\_

## 2.3. Usando *redes neuronales supervisadas* en la actualidad, algunas librerías útiles en R y python

Actualmente y como se vió en el ejemplo 04, existen varios paquetes especializados para *redes neuronales* tanto en *R* como en *python*.

Claramente uno de ellos es el ya denominado *neuralnet*, que viene bien para casos de modelos sencillos pero si se intenta con muchos datos, muchas *entradas* y una cantida elevada de *neuronas* o de *layers* intermedios es fácil ver cómo, los tiempos de cómputo se elevan, si los datos son muy pesados no se llega a formar el modelo por problemas tipo *out of memory* y el equema que se crea va a resultar ser poco entendible por el lío de conexiones que se crearía, por lo que no suele ser una opción para la resolución de problemas de tamaño medio.

En general tan en *R* como en *python* antes de dar el paso al *deep learning*, cabe encontrar 2 librerías adecuada para este tipo de problema que resultan ser respectivamente la *nnet* y la ya mencionada *sklearn* donde en este último caso se encuentra un módulo de *redes neuronales* bastante sencillo de usar tal y como se muestra en el siguiente ejemplo.

Ejemplo 06: Modelo de *regresión lineal* vs *red neuronal* para el precio del metro cuadrado

En el código que puede encontrarse en *ejemplos/programa05.py* se plantea una *red neuronal* con 2 capas ocultas 10 x 10 que se plantea fácilmente con la siguiente estructura y obteniendo un *R*2 de 0.78:

Texto

Descripción generada automáticamente

Mientras que con una regresión lineal, se habría obtenido lo siguiente:

Texto

Descripción generada automáticamente

Aquí habría que plantearse ¿A cambio de qué se ha pasado de un 0.75 a un 0.78 en el *test*? ¿Cuántos parámetros tiene la *regresión lineal*? ¿ Y la *red neuronal*? ¿Qué modelo resulta más *interpretable*? ¿Cuál es más fácil para *subirlo a producción*?

\_\_\_\_\_

# **3.- DISTINTOS TIPOS DE REDES NEURONALES**

Hasta ahora se ha visto sólo una sub-tipología de *redes neuronales* dentro de las que son de tipo *supevisado* que son las denominadas *hacia adelante* o *feed forward* y dentro de éstas ha sido explicado el *algoritmo back-propagation* aplicado a las que son de *clasificación* que lógicamente tiene una *función de coste* característica a optimizar, mientras que en el ejemplo 06 se ha usado una de tipo *regresión* que en este caso, se entrena de modo análogo al anterior caso sólo que la *función de coste* es de tipo “energía” o “cuadrática” que también hay que minimizar:

Siendo el procedimiento para encontrar los pesos en este caso totalmente análogo.

Sin embargo dentro de las de tipo *supervisado* y de los modelos previos al cambio de milenio, existen muchos más tipos de *redes neuronales* y además están las de tipo *no supervisado* y alguna tipología más tal y como se puede distinguir en el siguiente esquema:



**Figura 7 Clasificación en grandes líneas de las *redes neuronales***

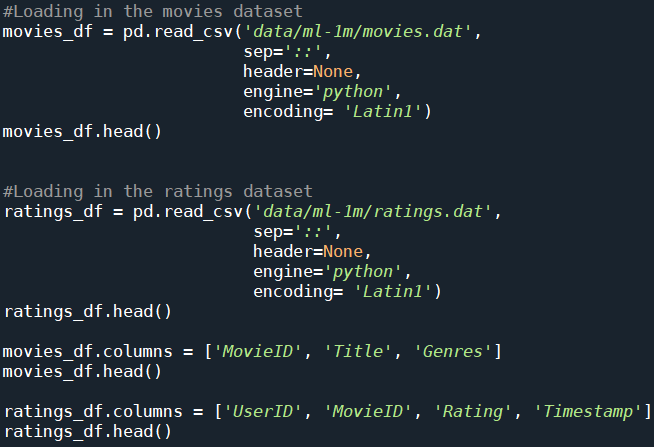
**Source: Elaboración propia & Chat-GPT**

Finalmente cabe comentar que aunque predomine el grupo de *redes neuronales supervisadas* el *algoritmo de backprogation*, este no es el único existente pudiéndose encontrar otros como por ejemplo el denominado *aprendizaje estocástico* donde se realizan alteraciones aleatorias en los *pesos* y se observa si la función de energía aumenta o disminuye. Si la energía de la red decrece, entonces se acepta el cambio en función de una determinada, función de distribución de probabilidades, los principales tipos de *redes* que utilizan este tipo de aprendizaje son los denominados: *Boltzman Machine* o *Cauchy Machine* que no se analizan en detalle aquí. Claramente en las redes de corte *no supervisado* y en las de tipo por *refuerzo*, la riqueza de *algoritmos* diversos resulta ser bastante mayor.

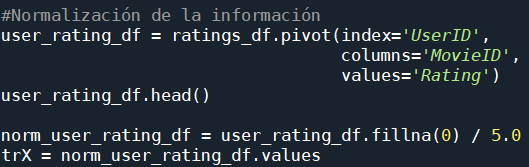
Ejemplo 07: Minimizando energía en una modelo de Máquina de Boltzmann Restringuida

Aunque en el siguiente punto se analizan algunas de las *redes no supervisadas*, se muestra el siguiente caso de uso en el programa de *ejemplos/programa05\_boltzmann.py* en el que construye un recomendador que estaría a medio camino, entre lo que sería un *entrenamiento de corte supervidado* y otro de corte *no supervisado*, ya que como se observan existen puntuaciones que se usan en el entrenamiento (guía supervisora), cuya asociación se pretende lograr aprendiendo la distribución de los datos más que ajustando el dato al valor (auto - aprendizaje por distribución del comportamiento de los datos).

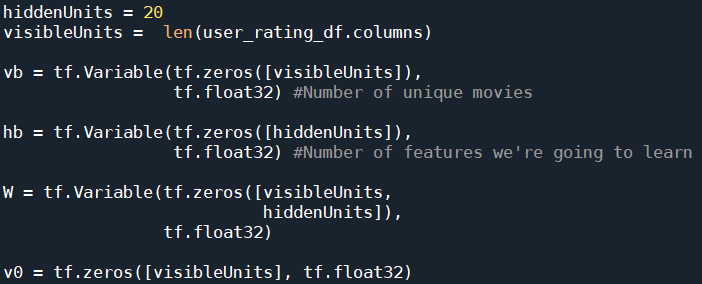
Aquí se leen los siguientes datos de evaluaciones de películas:



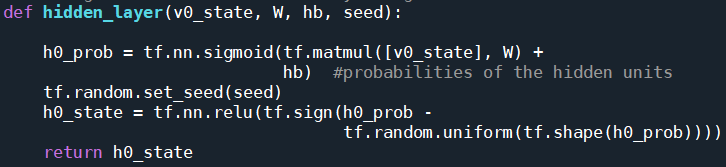
Se preparan los datos de modo que por fila se tienen los usuarios, por columnas las películas y las celdas intermedias son los ratings que además se normalizan a un valor entre 0 y 1 dividiendo por 5:



A continuación se consideran los parámetros del modelo. En este caso se van a considerar 20 estados ocultos, recuérdese que en esta tipología de modelo, todas las *neuronas* están totalmente interconectadas:



Se define la *capa oculta* con un parámetro *seed* para poder reproducir el experimento, si no se introduce este parámetro, el código es reproducible pero los resultados pueden variar. Obsérvese que se hace uso de funciones tipo *sigmoideas* para la modelización de la probabilidad y después de tipo *relu* para aplicar puntos de corte y decidir conexiones de activación – no activación:



Texto

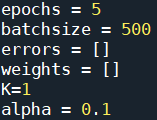
Descripción generada automáticamente

Se define la *función de error cuadrática* o *de energía* a minimizar:

Texto

Descripción generada automáticamente

Finalmente se aplica el algoritmo de entrenamiento que como puede observarse a día de hoy aún no está suficientemente “encapsulado” como una función de python. Se inicializa pues con los siguientes parámetros de inicialización:



Obteniéndose, tras la ejecución del subsiguiente bucle, el siguiente *error de reconstrucción*:

Gráfico, Histograma

Descripción generada automáticamente

Que viene a indicar que con este reducido número de *epoch*, los *batch* tomados en ciclos de 12 por cada *epoch*, mantienen un error máximo en torno a 0.10 en términos absolutos, y el sistema se mantiene estable (habría que ver con un número más amplio de *epoch*s, si el anterior *error* disminuye o no).

Finalmente, con el siguiente fragmento de código se indica cómo se puede hacer la predicción para para un caso concreto (se toma el usuario 215):

Texto

Descripción generada automáticamente

Nótese que se tiene un *tensor* fila de 3706 posiciones tal y como se observa a continuación:



Donde se aplica el modelo:

Texto

Descripción generada automáticamente

Y al final se obtienen las recomendaciones asociadas al usuario basadas en sus puntuaciones:

Pantalla de computadora con letras

Descripción generada automáticamente

\_\_\_\_\_

# **4.- REDES NEURONALES NO SUPERVISADAS**

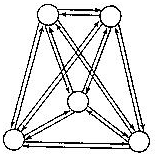
Aunque el anterior ejemplo mostraba en cierto modo cómo podía aplicarse un *algoritmo* para entrenar un modelo donde se tratase de ajustar distribuciones de la *target* más que la *target* en sí, lo cierto es que en *redes neuronales* existen varias familias de modelos que pueden aplicarse para cuestiones de *clusterización* sin la necesisdad de tener que ajustar e incluso de tener constancia de una variable *target*, por tanto la red es entrenada sin el beneficio de un “maestro” que guíe a lo largo del aprendizaje. Algunos modelos usuales para este tipo de aprendizaje son las siguientes:

* El Modelo de red de Hopfield:

Este modelo fue propuesto por el físico John Hopfield durante los años 80. En este caso se tiene una única capa de neuronas donde todas están conectadas entre sí donde cada neurona recibe unas entradas y emite una salida, en este sentido, si se centra la atención en una única neurona, el funcionamiento es muy similar al de un perceptrón de hecho, este modelo puede ser visto como perceptrones todos conectados entre sí excepto consigo mismo.

Ejemplo 08: Un esquema *neuronal de Hopfield* sencillo

¿Cuántas neuronas formaría la representación esquemática del siguiente modelo de Hopfield?



¿Qué parte del anterior esquema representa las conexiones? ¿Cuántos pesos habrían? ¿Cómo se representaría esquemáticamente una de estas estructura cuando ?

\_\_\_\_\_

Así pues elegida una de las neuronas, su entrada en la iteración está determinada por la suma ponderada de los valores de las demás neuronas existentes en la iteración anterior tal como sigue:

Posteriormente, el valor se obtiene de aplicar la función de salida a la anterior cantidad donde las funciones generadoras de la salida resultan ser neuronas de tipo binario siendo una opción habitual la siguiente:

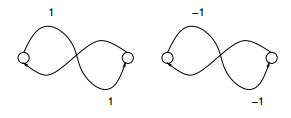
Por lo que:

Como hipótesis simplificadora del modelo está la exigencia de simetría de los pesos o lo que es lo mismo que: En este sentido cabe notar que los pesos no se actualizan, sino que lo que se actualiza en sí son los contenidos de las neuronas, así pues, dos son los modos de que se actualicen las neuronas:

- *Modo asíncrono* (*operación de Glauber*) donde en un instante sólo una neurona de la red actualiza su estado

- *Modo síncrono* (*operación Little*) donde en un instante varias o todas las neuronas actualizan su estado a la vez a partir del anterior

Ejemplo 09: *Modo asíncrono* vs *síncrono*



Bajo un *modo asíncrono*, la red de la izquierda alcanzará un estado estable con el contenido final en las neuronas igual a {1; 1} o {-1; -1} según el contenido inicial de las neuronas, mientras que la de la derecha alcanzará el estado estable igual a {1; -1} o {-1; 1}

Sin embargo, la izquierda bajo un *modo* *síncrono* no existirá convergencia a un estado estable.

\_\_\_\_\_

En estas redes sólo se tiene un único contacto con el exterior y es la configuración inicial de todo el sistema, es decir asociar un valor 1 o -1 que en un momento inicial se introduce en las neuronas, posteriormente, toda la dinámica de la red se comporta de modo determinista interesando la convergencia final de la estructura, el proceso finaliza cuando se llegue, bien al límite máximo de iteraciones impuestos, bien a un estado que cumpla que para toda neurona:

Siendo el objetivo del modelo almacenar un determinado conjunto de patrones en forma de estados estables de la red, así ante la entrada de un cierto patrón, se espera que la salida sea uno de los estados estables memorizados, actuando la red de este modo como una *memoria asociativa*.

Además se tiene la propiedad que si un patrón es estable y converge, si se introduce en el momento inicial, la convergencia sería hacia el mismo patrón en sí, por tanto la red es *autoasociativa*.

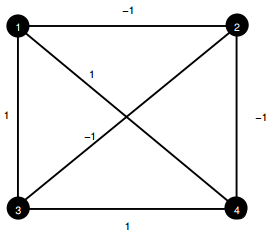
En este tipo de redes se define una función de energía tal como sigue (caso de umbrales nulos que es el más sencillo)

Se demuestra que un estado estable de la red es un mínimo local de la anterior función de energía, por lo que el aprendizaje consiste en conseguir que la red almacene como estados estables un conjunto de recuerdos previamente dados, por tanto la regla de aprendizaje a definir debe encontrar el conjunto de pesos que haga que la anterior función de energía tenga a esos patrones como mínimos locales, así pues, si se tiene un número de patrones a memorizar con ,se tendría que una regla de aprendizaje, la de Hebb se expresaría del siguiente modo:

Donde los pesos se proporcionan de forma directa y no de modo incremental como en el aprendizaje de Hebb en general

Ejemplo 10: Apredizaje del patrón (1; -1; 1; 1)

Estímese los pesos a partir de la fórmula anterior y demuéstrese que se corresponden a los del siguiente *grafo*



Si se ejecuta el código *ejemplos/06\_Hopfield\_sencillo.R* se demuestra que la *matriz de pesos* es la siguiente:

Imagen que contiene reloj, objeto, foto, naranja

Descripción generada automáticamente

Y la función *hopfield* implementa el *algoritmo* anteriormente comentado

\_\_\_\_\_

Finalmente se deja el siguiente link a un proyecto donde se explica claramente el objetivo del reconomiento de patrones cuando se aplica a la escritura humana:

<https://github.com/angellotrivino/RNA-Hopfield-Reconocimiento/tree/master>

En este caso, tras dibujar algo parecido a una I, el algoritmo converge hacia la letra “I”:

 Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Figura 8 Reconocimiento de patrones con una *red Hopfield***

**Source: Link anterior**

Finalmente estas redes se han aplicado con éxito en:

- Reconocimiento de imágenes y caracteres

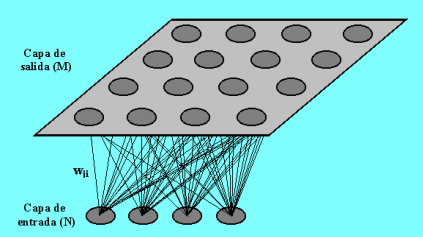
- Reconocimiento de voz

- Problemas complejos como el Problema del Viajero

* La *red neuronal* de *Kohonen*:

Este modelo fue propuesto por el físico de origen finlandés Teuvo Kohonen, también sobre la misma época que el anterior modelo, aunque caben ser clasificadas, como se verá a continuación como una red más de tipo competitiva. El modelo se organiza como una arquitectura unidireccional de dos capas:

* + Una primera capa de entrada que comporta como buffers de información que la distribuye a la siguiente capa dichas entradas son muestras estadísticas:
  + Una segunda capa de neuronas, existiendo un total de: neuronas de modo que cada una de estas neuronas está conectada a cada una de las entradas mediante las ponderaciones



**Figura 9 Esquema de la arquitectura de una *red de Kohonen***

**Source: Link anterior**

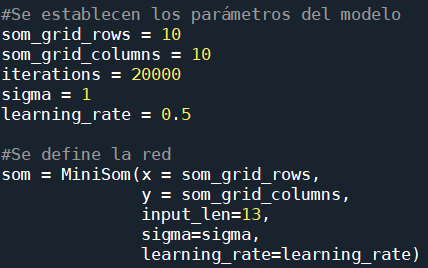
El entrenamiento de la red se realiza en base a una medida de similitud entre los pesos que llegan a cada una de las neuronas (que es un vector de dimensión *N*) y la entrada en sí, seleccionándose al vector de pesos ganador del siguiente modo:

En el modelo de Kohonen la novedad principal en el aprendizaje no sólo implica una modificación de los pesos correspondientes a la neurona ganadora, sino que además se hace uso de una *Función de Vecindad*, que implica que simultáneamente se modifican también los peso de la neurona del alrededor, así pues se considera la siguiente regla de aprendizaje:

Así por ejemplo en la siguiente expresión ¿Cuál es el parámetro que marca el ritmo de aprendizaje y que suele aparecer en los modelos supervisados? La función de vecindad informa de cómo es el entorno en sí de las neuronas que se afectan tomando como referencia la neurona ganadora que en principio tendría una posición en la matriz, pero aparte de sus *N pesos*, se modificarán también los *N pesos* de las neuronas del alrededor.

Ejemplo 11: Detectando fraude con una *red de Kohonen*

En el código *Ejemplos/programa07.py* se trata de realizar una clusterización *no supervisada* para ver si existen relaciones profundas entre los datos más allá que la relación directa *supervisada* de las explicativas con la *target*. Para tal fin se plantea la siguiente *red de kohonen*:



Posteriormente se observa *mapa autoorganizado* que se genera:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

En este caso se ve una región la del cuadrante (4; 0), (4; 4), (10;4) y (10; 10) donde se observa una particular concentración del fraude frente al resto donde hay zonas de menor concentración de fraude (o mayor concentración de no fraudes)

\_\_\_\_\_

Para finalizar, se comentan algunas aplicaciones reales que tienen este tipo de redes son las siguientes:

- Reducción de dimensiones, encontrando los rasgos más importantes del espacio sensorial

- Preprocesamiento de datos para otros sistemas

- Análisis cluster (como el aquí realizado)

-Etc

# **5.- INTRODUCCIÓN AL DEEP LEARNING**

Volviendo de nuevo al campo del *entrenamiento supervisado*, se tiene que una de las técnicas que se han popularizado en los últimos tiempos tiene que ver con el concepto de *visión artificial* o más bien de *reconocimiento de imágenes*. Aunque como se observará, son un caso particular del concepto de las *redes neuronales*, sin embargo, el pleno desarrollo no se alcanzó hasta alrededor del año 2010. De hecho aunque a finales de los 80, Yann LeCun y otros desarrollaron y usaron *redes convolucionales CNN*, tecnológicamente sus desarrollos y los de otros como Hinton, Osindero y Teh no pudieron fluctificar adecuadamente por falta de potencia de los ordenadores del momento, además de ser técnicas que aún no estaban debidamente optimizadas.

Sin embargo, a partir del año 2010 se suman muchos hechos donde aparte de la persistencia del propio *Hinton* y *Bengio* que continuaban con sus desarrollos desde finales de los 80, tiene lugar entre otros 2 hechos clave como que se dispusiera ya de *GPU*s fundamentales para acelerar los entrenamientos de la época y por otro lado se contaba con una base de datos de imágenes debidamente etiquetadas denominada *ImageNet* por tanto, los investigadores ya podían comenzar y mejorar los algoritmos existentes comenzando una evolución muy rápida que se puede contrastar por los subsiguientes desarrollos como por ejemplo:

* *2012 AlexNet*: red desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton, gana la competencia ImageNet, marcando un hito importante en el rendimiento de las redes convolucionales profundas
* *2014 GANs*: son un nuevo tipo de red cuya arquitectura deriva del *Deep Learning* y aunque no se desarrollan aquí, permiten la generación de datos sintéticos (incluidas imágenes)
* *2014 VGG* y *GoogleNet*: que presenta mejoras significativas en profundidad y estructura con respecto a otras redes neuronales
* *2015 ResNet*: de Microsoft Research, con su arquitectura de red residual, permite entrenar redes mucho más profundas, superando el problema de la degradación en redes muy profundas

Por tanto hacia el 2015 – 2020 se consigue tener auténticas *redes* tipo *deep learning* cuyas características fundamentales son:

* Básicamente se propone una estructura *multicapa de neuronas* y que aunque aparentemente parece que complica el sistema, lo que se consigue realmente es simplificarlo, ya que se consigue mejores representaciones de imágenes con neuronas distribuidas horizontal que verticalmente

Gráfico

Descripción generada automáticamente

**Figura 10 Esquema general de una *deep learning***

**Source: Deep Learning with python**

* Lo que existe en las capas intermedias son filtros de imágenes de tipo matricial, cuyo contenido numérico van a ser parámetros de la red que se tendrán que entrenar:

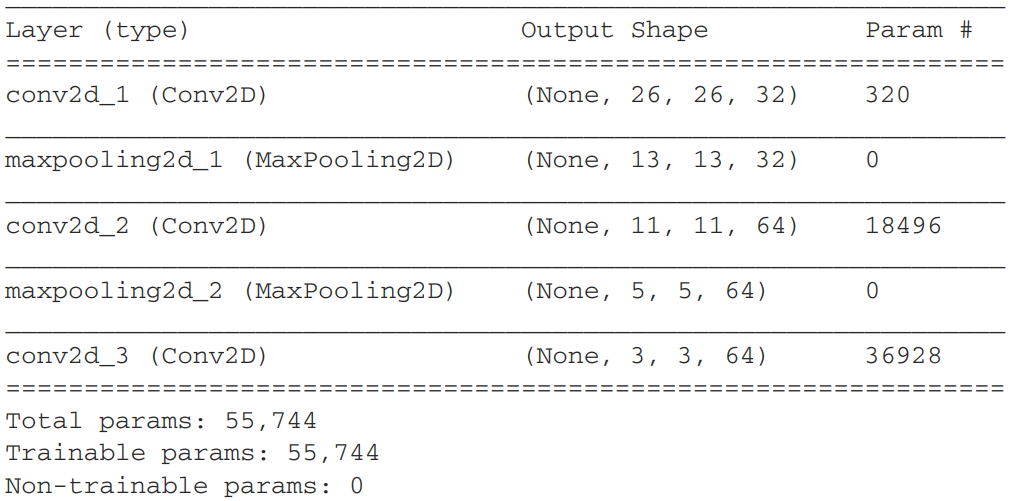
Diagrama

Descripción generada automáticamente

**Figura 11 Esquema de una capa convolucional**

**Source: Deep Learning with python**

* A lo anterior se le denominan *capas convolucionales* y se definen fácilmente en python como se verá en el ejemplo haciéndose uso de matrices, generalmetne tridimensionales denominadas *tensores* tal como sigue donde las dimensiones de los objetos *conv2d* y *maxpool* suelen ser (*altura*; *amplitud*; *canales*):



**Figura 12 Resumen paramétrico de un modelo “sencillo” de *Deep Learning***

**Source: Deep Learning with python**

* También cabe destacar que por lo general en estos modelos suele haber una *capa final* que suele estar totalmente conectada tal y como ya se han estudiado
* Finalmente si se observa tanto en la figura anterior como en los ejemplos que se desarrollarán, a poco que se trabajen estos modelos, el número de parámetros crece rápidamente, por lo que hay que tener en cuenta determinados problemas como el *over-fitting* provocado por un exceso de parámetros, solucionables con técnicas como el *drop-out* y aquí también ha que considerar el *over-training* provocando por un entrenamiento excesivo del modelo y que puede solucionarse deteniendo el entrenamiento mientras este se sitúe en fases de mejoras significativas antes de que comience a empeorar las predicciones en el *conjunto de validación*

Ejemplo 12: Reconocimiento de escritura mediante una red densa

En este primer caso que se puede encontrar en *Ejemplos/programa08.py* se hace el equivalente a un “hola mundo” para el caso de deep learning, proponiendo una red con 512 neuronas en una única capa densa usando el framework de *keras* que se comenta más adelante y que como se observa, a día de hoy tras instalar la correspondiente librería, su uso resulta muy sencillo y viene con varios dataset de prueba como el conocido *mnist*

Texto

Descripción generada automáticamente

Un aspecto que resulta clave en estos problemas es adecuar las dimensiones de los datos, para ello el método *reshape* suele ser muy utilizado y se debe prestar atención en la forma que deben adquirir los *input* que es bastante similar al caso *backpropagation* ya comentado, de hecho, el entrenamiento se base en variaciones del anterior algoritmo.

Texto

Descripción generada automáticamente

Si todo está adecuado, la *red* se entrenará con la línea:



Dando lugar a los siguientes resultados en training:

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Finalmente en este caso si se le pasan los datos de test, el acierto que se obtiene sería:

Texto

Descripción generada automáticamente



Muy en línea con el entrenamiento realizado y donde el número de parámetros no es nada despreciable, algo más de 400.000 tal y como se observa:

Imagen que contiene Texto

Descripción generada automáticamente

\_\_\_\_\_

Como se ha observado, la programación a día de hoy con *keras* es bastante sencilla e incluso con otro framework como *pytorch*, la diferencia a día de hoy ya no resulta tan alta e incluso, por cuestiones de velocidad, resulta más recomendable a día de hoy que hace algunos años (antes de la pandemia del 2020 por ejemplo), donde tanto la instalación, como el manejo en sí del framework eran más complejos.

Ejemplo 13: Usando *pytorch* y distinguiendo entre pájaros y aviones

Este ejemplo que se puede encontrar en *data/programa09.py* se hace uso de las imágenes que se pueden descargar fácilmente mediante la siguiente función:

cifar10 = datasets.CIFAR10(data\_path, train=True, download=False)

Claramente si es la primera vez que se llama conviene poner el parámetro *download* como *True*, obteniéndose imágenes como las que siguen:

|  |  |
| --- | --- |
|  |  |

Se están ante imágenes que son matrices tridimencionales de cuadrícula 32 x 32, por tanto cuando se transforman a tensores, quedan como una matriz de dimensión (3; 32; 32), aplicado a todo el dataset, en *pytorch* se haría del siguiente modo mediante la función *ToTensor()*:

Texto

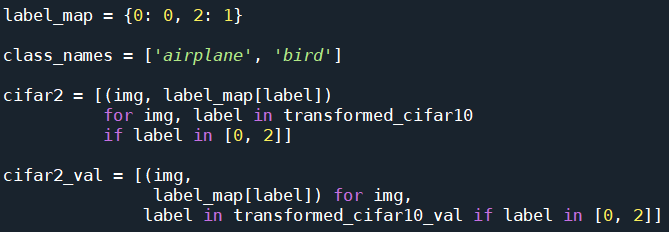
Descripción generada automáticamente

Dataset que queda ahora “tensorizado” como:

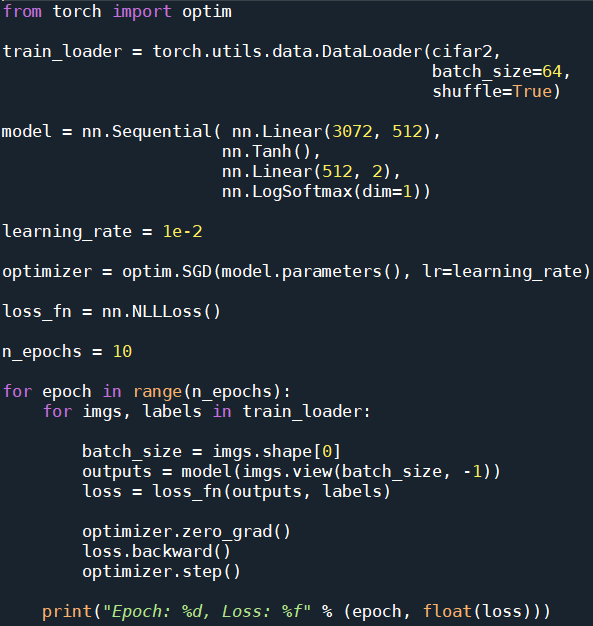
Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

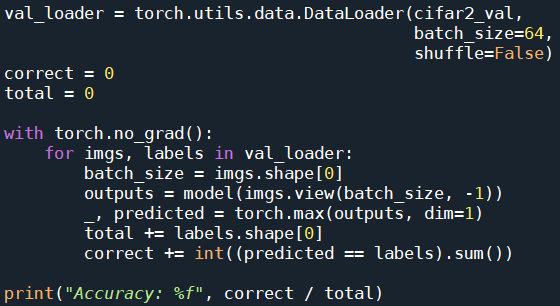
Para comenzar el entrenamiento de la red, se selecciona en primer lugar los datos que corresponden a aviones y pájaros asignando 0 y 1 respectivamente:



Las modelizaciones con redes se “llaman” con *import torch.nn* as nn creándose el siguiente bloque de entrenamiento:

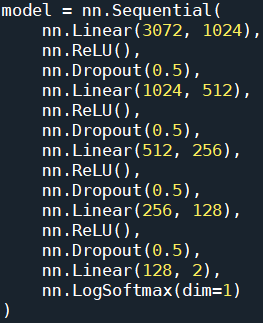


Llegándose a un acierto del 77% tal y como se muestra con:



Donde se ha usado 1.574.402 parámetros.

Finalmente si se pregunta a Chat GPT por una mejora de lo anterior propone la siguiente arquitectura que va a contener unos 3.836.034 parámetros:



Donde básicamente existe varias capas densas de ReLU y funcioines lineales dónde se les hace un Dropout al 50% siendo dichas capas densas cada vez menores. Los resultados, bajo (estas semillas) alcanzan ahora el 84%, por lo que cabe decir que el acierto que se produce mejora un 5% más, no obstante, una mejora que alcance al 90% requeriría además otros tipos de capas como las convolucioneales y no sólo aplicar “densidad paralela”

\_\_\_\_\_

# **6.- NOCIONES BÁSICAS SOBRE REINFORCEMENT LEARNING**

Para finalizar la parte de *inteligencia artificial* no se puede pasar por alto hacer al menos una mera introducción sobre quizás una de las técnicas que aunque está más separada de la parte de clasificación y de *análisis supervisad – no supervisado*, sin embargo cabe considerarla auténtica *Inteligencia Artificial* porque en cierto modo está en la base de *artilugios* capaces de aprender por sí mismos de modo autónomo.

## 6.1.- Un problema clásico: El viajero

El TSP es uno de los más típicos y mejor estudiados problemas de la optimización combinatoria, pero a pesar de ello aún no se conocen algoritmos que lo resuelvan de forma exacta. Los algoritmos utilizados para su resolución sólo ofrecen aproximaciones, y la solución óptima sólo es alcanzable para casos concretos del problema.

A este problema se le cataloga como un problema NP-completo, lo que significa que el esfuerzo computacional que se debe llevar a cabo para encontrar una solución óptima crece de forma exponencial con la entrada del problema, que en el caso concreto de TSP sería el número de nodos o vértices de la red.

Por lo tanto, el número de nodos de la red va a ser fundamental en determinar la complejidad del problema. Cuanto mayor sea el número de nodos, mayor va a ser el número de rutas posibles, y por lo tanto mayor será el esfuerzo requerido para calcular todas ellas. Así pues, el número de rutas posibles entre N nodos va a ser igual a N! lo que hace que la resolución del TSP mediante la obtención de todas las rutas posibles y comparación entre ellas sea poco factible incluso para un número de nodos no elevado. Sólo con tener una red simple de 7 nodos, ya sería necesario calcular más de 5000 combinaciones, (7!=5040), y elevando simplemente hasta 10 el número de vértices de la red, las posibles rutas se disparan hasta más de tres millones (10!=3.628.800).

En cuanto a las soluciones óptimas, no existen métodos que resuelvan de manera exacta cualquier TSP, pero sí que existen soluciones exactas para algunos casos concretos. A lo largo de los últimos cincuenta años se han producido continuos avances, lo que ha permitido obtener soluciones óptimas para el TSP cada vez con mayor número de nodos, hasta alcanzar una cifra de casi 86.000. Los principales logros obtenidos en los últimos 70 años vienen resumidos en la siguiente tabla:

Tabla

Descripción generada automáticamente

**Figura 13: Tabla resumen de los principales logros en el problema TSP hasta 2006**

**Fuente:** [**http://idelab.uva.es/el-problema-del-viajante**](http://idelab.uva.es/el-problema-del-viajante)

La primera formulación matemática fue realizada en el siglo XIX por W.R. Hamilton y Thomas Kirkman. El juego icosiano de Hamilton era un rompecabezas recreativo basado en encontrar un ciclo hamiltoniano, que en realidad es una solución al problema de TSP en un grafo (nota: el TSP es el ciclo hamiltoniano de menor longitud en un grafo conexo).

El TSP se puede formular como un modelo de Programación Lineal Entera. Aunque se conocen varias formulaciones para resolver este problema, dos de ellas son las más destacadas: la propuesta por Miller, Tucker y Zemlin, y la propuesta por Dantzig, Fulkerson y Johnson. En el siguiente gráfico podemos ver la solución óptima para un problema de 100 nodos.

Diagrama

Descripción generada automáticamente

**Figura 14: Resolución del TSP para 100 nodos**

**Fuente:** [**https://baobabsoluciones.es/blog/2020/10/01/problema-del-viajante/**](https://baobabsoluciones.es/blog/2020/10/01/problema-del-viajante/)

Como se verá más adelante, el Reinforcement Learning debería ser un entorno natural en el que resolver este tipo de problemas tal y como se propone en *André L. C. Ottoni; Erivelton G. Nepomuceno; Marcos S. de Oliveira; Daniela C. R. de Oliveira 2020* sin embargo, las soluciones han ido por otras líneas distintas y como se verá, el marco que define el RL es perfectamente adaptable también para resolver este tipo de problemas. En este caso el agente, es decir, el vehículo aprenderá las rutas más cortas a través de la combinación de recompensa-castigo.

Ejemplo 14: Ejemplo de resolución de un problema tipo TSP sencillo en R con 5 nodos

Una metodología que puede ayudar a algunos casos pequeños es el que se muestra el código *data/programa10.R* que muestra cómo se resolvería el problema TSP para el sencillo caso de los nodos con la siguiente matriz de distancias

Texto

Descripción generada automáticamente con confianza media

Sin embargo, para resolver problemas tipo TSP de mayor dificultad se requiere o bien software especializado o programación específica avanzada para el problema.

\_\_\_\_\_

## 6.2.- Breves pinceladas de programación dinámica e introducción histórica al Reinforcement Learning

Sin ánimo de ser exhaustivo y según *Sutton, R. S.; Barto, A. G. 2018* la historia del *Reinforcement Learning* discurre por 3 ramas independientes antes de que se formase el concepto moderno en sí:

* La primera de esas ramas está relacionada con la psicología del aprendizaje animal por mediante *fallo – recompensa* se introduce desde los primeros momentos del desarrollo de la I. A. en 1948 cuando Alan Turing comienza a trabaja en la denominada *Ley del Efecto* y empiezan a surgir muchos ingenios mecánicos (precursores de los robores de Boston Dinamics). En paralelo como se ha visto surge también la rama *supervisada* de la *IA* aunque hay algunas variantes que cabe destacar como la de 1977 que Nueva Zelanda, John Andrea desarrolla el sistema *STeLLA* que aprende totalmente por *fallo – recompensa* y que llega a evolucionarlo hasta que en 1998 es capaz de implementar un sistema de asignación de crédito bancario completo. No obstante los trabajos más conocidos en esta línea se sitúan a principios de los 60 con sistemas que aprenden por sí solos, desde jugar un simple “piedra-paper-tijera”, hasta que a finales del siglo XX llegan a ser campeones mundiales de Ajedrez e incluso de Go. También durante los años 60 aparecen los problemas de *aprendizaje de autómatas* desarrollados por el matemático – físico ruso M. L Tsetlin y otros que desarrollan la teoría *stochastic learning automata* introduciendo elementos de la teoría de la estadística y la probabilidad en los algoritmos de aprendizaje
* La segunda rama la relacinada con los *problemas de control óptimo* y su solución mediante el uso de *funciones-valor* y de la *programación dinámica* cuyos principios se enuncian a continuación. Esta rama apareció a finales de los 50 para describir el problema del diseño de controles donde hay que minimizar o maximizar la medida del comportamiento de un sistema dinámico, en este área destaca una figura central como la de de Richard Bellman, el cual, mediante sus famosas ecuaciones, extiende la teoría ya esbozada a finales del siglo XIX por Hamilton y Jacobi. Con Bellman se introduce la base del *reinforcement learning* moderno a finales de los 50 con el concepto clave de *Markov Decision Processes MDPs* que son extendidos posteriormente por Ronald Howard en 1960 mediante el concepto de *Policy Iteration*, siendo todos éstos, elementos esenciales de lo que se conoce como *Programación Dinámica*. En estos problemas se comienza a lidiar con el conocido concepto de la *maldición de la dimensionalidad* o lo que es lo mismo, la computación en esta área crece exponencialmente con el número de *variables – estados*. La total integración por tanto de los métodos con los de aprendizaje de la anterior rama no ocurriría hasta 1989 con los trabajos de Chris Watkins que desarrolla el actual formalismo y tratamiento de estos problemas bajo el marco *MDP* siendo éste, el adoptado en la actualidad y que se desarrolla aquí
* Finalmente, la tercera rama es la de los métodos de *diferencia temporal*, que también con unos orígenes similares a los anteriores, llegan a imponerse en 1989 también con Chris Watkins cuando desarrolla el *Q – algoritmo* integrando por tanto estas ramas y demostrado su potencia en 1992 con el trabajo de Gerry Tesauro en el juego del backgamon.

En estas fases juega un papel central la *programación dinámica* que cabe ser vista como un principio general aplicable a la resolución de una serie de problemas que gocen de la *propiedad de descomponibilidad* y que se entiende fácilmente si se considera el siguiente problema clásica conocido como *el* *problema de la mochila* que matemáticamente ofrece el siguiente aspecto:

En todo problema de estas característcas 2 son los pasos que cabe considerar:

* Encajar el problema bajo una familia de problemas de naturaleza semejante: así pues el problema *KP* anterior no es más que un caso particular de la siguiente familia dependiente del parámetro *E* (que resulta ser un número entero entre 0 y *b*):

Al parámetro *E* se le denomina *variable de estado*, contenida en el *espacio de estados* Se tiene además que dentro de la familia el problema a resolver es interesando la función que valdrá si resulta que *E* es mayor estricto a *b* o menor estricto a 0

* El siguiente paso consiste en buscar una relación recurrente que conecte los valores óptimos de los diferentes problemas de la familia. Esta parte no se desarrolla aquí, aunque la idea es la de construir un programa con recurrencia partiendo de que si el excursionista ha ocupado una parte *E* de la mochila con los objetos desde 1 hasta *i* - 1, entonces queda un espacio *E* – *b* para ser o no relleno con los objetos existentes desde *i* hasta *n*. Por tanto, se crean 3 condiciones para las siguientes etapas en el relleno de la mochila que se dejan aquí enunciadas:
  + Etapa *n*:
  + Etapa *n* – 1:
  + Etapa genérica *i* deducida de la anterior:

Se observa por tanto que el objetivo de la *programación dinámica* consiste en crear una red de problemas de fácil resolución a partir de uno dado, de modo que finalmente se llegue a una solución óptima que permitiría derivar la correspondiente asignación. En este sentido, los estados son fijos y se accede a ellos paso tras paso, no existe aleatoriedad en estos sucesos, además se observa que en este caso en particular, sólo se está considerando un problema muy sencillo con un único espacio de sucesos que se puede complicar haciendo muy intratable la conversión de un problema de *optimización lineal* como el anterior en otro de *programación dinámica* con *recursividad* como se ha observado anteriormente.

No obstante el *reinforcement learning* ofrecerá un marco más general, englobando la anterior metodología bajo una teoría más compacta.

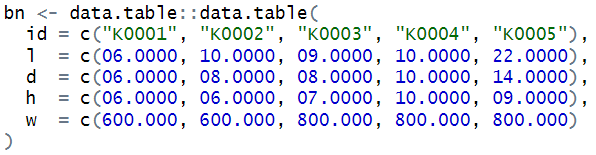
Ejemplo 15: Optimización de stocks en un pallet

En *data/programa11.R* se muestra cómo se puede resolver un problema tipo “mochila” con R donde se trata de asignar cajas de un determinado tamaño – peso:

Texto

Descripción generada automáticamente

A contenedores que están limitados también por tamaño – peso:



En este caso se usan las funciones *gbp4d\_solver\_dpp\_prep\_create\_p()* y *gbp4d\_solver\_dpp()* aplicado a un único pallet dando lugar al siguiente resultado que visualmente sería:

Diagrama

Descripción generada automáticamente

\_\_\_\_\_

## 6.3.- Las bases del Reinforcement Learning

El principal objetivo de los *algoritmos* basados en *RL* es aprender y adaptarse a los cambios del entorno. Para llegar a estos se hace uso de señales o de algún tipo de feedback externo (recompensas) que dependen además de las elecciones que realiza dicho *algoritmo*. En su versión más básica un *reinforcement learning* responde al siguiente esquema:

Diagrama

Descripción generada automáticamente

**Figura 15: Building Blocks básicos de un RL**

**Fuente: Ciaburro, G (2019)**

Bajo este esquema, el elemento clave que se requiere “adiestrar” es el *agente* que interactúa y con el entorno mediante *acciones* y recibe feedback en modo de *recompensas* positivas o negativas, siendo capaz de cambiar incluso el *estado* del *entorno* en función de las *acciones* realizadas.

En *RL* se trata de maximizar mediante los comportamientos a realizar mediante la recolección de recompensas, así pues, un *agente*:

* Puede realizar una *acción* continua o discreta (aquí se analizarán éstas últimas principalmente)
* La *acción* dependerá de la situación que se resume mediante el concepto de *estado* del sistema
* Continuamente monitoriza el *entorno* y cambia el *estado*
* La elección de la *acción* no es trivial y requiere cierto grado de *inteligencia*
* El *agente* estará dotado de *memoria*

### El agente

El *agente* por tanto aprende por *fallo – recompensa*, aplicando acciones hasta que obtiene la *recompensa* o hasta que paran los intentos. Aunque aquí se tratarán problemas con un horizonte temporal finito, éstos pueden tener horizontes temporales tales que se pueden considerar prácticamente infinitos. Para que el *agente* pueda interaccionar por tanto debe ser dotados de *sensores* capaces de recibir las *sensaciones* del *entorno* para que éstas sean evaluadas como un *fallo* o una *recompensa*.

Con respecto al *entorno*, es necesario tener una descripción formal de éste, no es importante tanto saber de qué está constituido pero sí es clave hacer algunos supuestos hacerca de las propiedades que tiene, en *RL* se asume que el *entorno* puede ser descrito mediante un *Markov Decision Process* o *MDP*.

### Características de un entorno tipo MDP

En general el tipo de problemas que se van a enfrentar van a ser de tipo estocástico de modo que en cada instante *t* se producirá un estado de entre una multiplicidad de posibilidades.

En un *proceso estocástico markoviano*, la ocurrencia del estado va a depender de lo que haya acontecido en *t* y no de lo que haya sucedido con anterioridad (ya se vió esto en los modelos de *redes bayesianas*), por lo que el instante del momento *t* determina el comportamiento a futuro. En un *proceso markoviano* como el anterior, si el proceso está en un estado *s*, el agente puede escoger una *acción a* de entre un conjunto de posibles acciones en dicho estado. En ese caso, el agente se mueve hacia un nuevo estado *s’* y por tanto recibe una recompensa .

Los *procesos de Markov* tienen la propiedad de que pueden ser representados por lo que se denomina *matrices de transición* que son aquellas cuyas filas suma la unidad y guardan una serie de propiedades matemáticas muy interesantes junto a una representación en forma de grafo. Así pues, resultan equivalentes la siguiente matriz y el diagrama de interrelación de estado que se representa más claramente con el grafo asociado:

|  |  |
| --- | --- |
|  |  |

Nótese que en una matriz de transición lo que se expresa, para cada elemento *i*; *j* es la probabilidad de pasar del *estado i* al *estado j* entre los instantes *t* y *t* + 1.

Ejemplo 16: Matrices de transición y previsión meteorológica sencilla

En *data/programa13.R* se observa mediante el uso de la librería *markovchain* como se resuelve un problema de transición de estados markovianos de un modo sencillo y bastante directo:

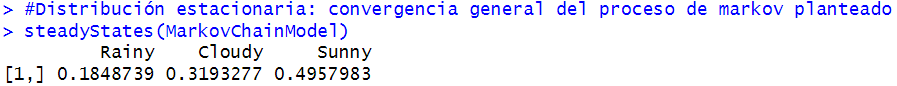
|  |  |
| --- | --- |
|  |  |

Se puede tener la previsión a 1 semana, partiendo de un estado inicial como que hoy es soleado:

Texto

Descripción generada automáticamente con confianza media

Aunque la convergencia general, independiente del estado inicial que se tenga a largo plazo sería la siguiente:



\_\_\_\_\_

### Funciones – valor de un entorno tipo MDP

Bajo el anterior marco de actuación, el estado *t* + 1 se define por tanto como:

Donde la función se denomina *función – valor* del entorno *MDP*. Por tanto se tiene que en cada iteración, el *entorno* responde del siguiente modo:

* Provee una *recompensa*:
* Se produce un cambio de *estado*:

### 

### Políticas

Define el comportamiento del *agente* en un instante temporal dado *t*. Las políticas hacen uso del (los) *estado* (s) detectados en el *entorno* y de las *acciones* a tomar cuando se tienen dichos *estados,* por tanto, corresponde al conjunto de reglas o de asociaciones *estímulo – respuesta*. La *política* es un concepto fundamental en *RL* en el sentido de que ella por sí sola es suficiente para determinar el comportamiento del *agente*.

En los *MDP*, el objetivo del *RL* es por tanto aprender una *política* en la que para cada estado *s* en el que se encuentre el *agente*, se pueda, para una *acción* específica de dicho agente maximizar el *reinforcement* que recibe durante la secuencia completa de *acciones*, por tanto el total de *reinforcement* que se deriva de una política se calcula como sigue:

En el caso de que el número de estados pueda llegar a ser infinito se introduce un factor de descuento, de modo que la anterior expresión quedaría como:

El factor de descuento puede ser modificado a lo largo del tiempo (aunque en la anterior expresión se ha puesto como fijo). Así pues, si dicho factor es menor que 1, la anterior suma convergería a un valor finito, mientras que si el factor es 0, el *agente* no tendrá interés en *recompensas futuras* y se limitará a *maximizar* la del estado *actual* y en caso de que sea igual a 1 o se aproxime a este número, el *agente* tratará de maximizar *recompensas futuras* a expensas de las inmediatamente actuales. En ningún caso, por tanto, el factor de descuento será mayor que la unidad.

Las políticas pueden ser determinísticas, cuando la misma *acción* es tomada para un *estado* dado, o probabilísticas cuando la *acción* es escogida basada en algún cálculo de distribuciones entre las opciones y los *estados* dados.

## 6.4.- Desarrollo de un caso típico de reinforcement learning: El K-armed bandit

### El dilema exploración – explotación

Según este dilema típico de *RL*, el *agente* explora todas las posibles *acciones* para cada posible *estado*, para que de este modo pueda encontrar aquella por la que obtendrá mayores *recompensas*, por tanto, hay 2 posibles decisiones a tomar:

* *Explotación*: que consiste en ejecutar la mejor decisión dada la información que se tiene hasta el momento
* *Exploración*: que consiste en obtener mayor información para poder encontrar mejores decisiones que las que se tiene hasta el momento

Por tanto, será clave bajo este dilema encontrar un equilibrio entre las anteriores opciones. En este sentido las *políticas* resultan esenciales porque dirigen la forma en la que escoger las distintas *acciones* de actuación. Así pues, hasta un determinado tiempo *T* el *agente* ha realizado y “observado” un determinado conjunto de *acciones* y puede optar por tomar la mejor en base (y esto es importante) a la información recogida hasta el momento o sin embargo, seguir explorando alguna otra por si pudiera llegar a mejores resultados, básicamente en esto consiste el caso que se desarrolla en este apartado.

### Planteamiento y breve análisis del *K-armed bandit*

Este problema es clásico en la literatura de *RL* y fue introducido por H. Robbins en 1952, donde se observa claramente el anteriormente mencionado *dilema exploración – explotación*. Básicamente lo que se propone delante de nosotros es el típico juego de las tragaperras con distintas palancas (k – armed bandits) que seleccionar y con una distribución de los premios asociada a cada una de las máquinas. Cuando llega por tanto un jugador a una sala de un salón de juego, se encuentra por tanto con una fila amplia de tragaperras y el jugador elige una de estas y pulsa “el manguito” correspondiente, posteriormente tiene la opción de continuar en esa máquina (explotación) donde ha podido ganar y aumenta su conocimiento sobre su funcionamiento y la *función probabilidad subyacente* de ocurrencia de premios, o bien puede cambiar a otra máquina de la cual puede no tener ningún tipo de información adicional y por tanto debe invertir en obtenerla.

En términos matemáticos el problema se plantea del siguiente modo:

* Cada máquina es modelada como una *distribución de probabilidad* que tiene asociado un determinado *valor medio* con una *desviación estándar* en concreto, que pueden variar a lo largo del tiempo de forma dependiente o independiente a éste
* La *distribución de probabilidades* no es conocida a priori por ninguno de los jugadores (*agentes*) pero puede ser aprendida por éstos
* En cada instante temporal, sólo uno de los *k* “manguitos” es accionado y una *recompensa* estocática sería observada
* El objetivo por tanto consiste en *maximizar la suma de recompensas* durante un determinado intervalo temporal

En este sentido se representan las recompensas que van a ser independientes e idénticamente distribuidas según una distribución de probabilidad con media desconocida

Como cabe intuir, cada *algoritmo* especifica qué nivel (o manguito) debe ser *accionado* dada su historia pasada y las recompensas obtenidas hasta el momento, lo que constituye la *política* a aplicar.

Por tanto, en el momento de diseñar una *política* hay que balancear el tiempo (o la inversión) que se va a emplear en conocer las distribuciones de las *k* máquinas y la *recompensa* a obtener en cada una de ellas.

Como puede intuirse, este caso tiene muchas aplicaciones como las siguientes:

* Anuncios online: en estos casos cabe usar algoritmos de tipo ML para dinámicamente localizar anuncios a páginas web que esté actuando bien, evitando aquellas de actuación más lenta
* Sistemas de localización de noticias: cuando un usuario visita un site, debe elegir una noticia de entre una serie de artículos, el sitio recibe una recompensa por cada selección y por tanto, dado que se desea maximizar el número de clicks, se trata de ofrecer en primer lugar aquellas noticias con mayor propensión a tener un click
* Pruebas de tratamientos de salud: elección de entre varios tratamientos disponible, donde el responsable trata de decidir que tratamiento utilizar mientras se minimizan la pérdida de pacientes (en general no humanos)
* Selección de personal
* Selección de portfolios financieros

### Implementación y solución del *MAB*. Métodos greedy

Hay varios modos de resolver un problema tipo MAB, uno de los más populares son los denominados métodos *greedy* donde se considera el siguiente comportamiento trivial por parte de un *agente*:

* Inicialmente, durante un tiempo se “juega” cada uno de las *k* máquinas consideradas equitativamente
* La máquina que haya ofrecido mayor recompensa, es la que se juega sistemáticamente después

A la mejor *acción* se la denomina *acción greedy* y por tanto a este método se denominaría *método greedy* donde se siguen formalmente los siguientes pasos:

* En un momento temporal *t* se calcula por cada *acción a* la siguiente cantidad:
* Se selecciona la acción de máximo valor:

Como se observa, bajo este método no se hace más que una somera *exploración* al principio y después se procede a la *explotación*.

Sin embargo, se puede alterar el proceso anterior del siguiente modo obteniéndose lo que se conoce como *método -greedy* donde se realizan las siguientes acciones:

* En un momento temporal *t* se calcula por cada *acción a* la siguiente cantidad:
* Se selecciona la acción de máximo valor con probabilidad :
* Se selecciona una con probabilidad cualquiera de las *acciones* disponibles para el agente

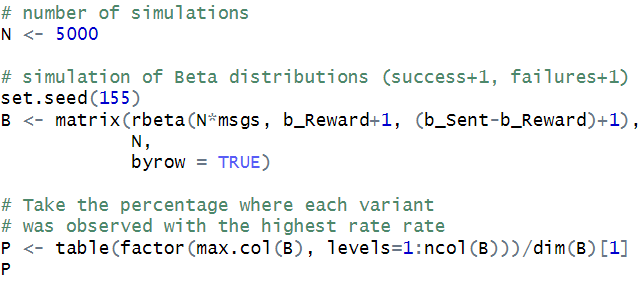
Además se puede hacer que el factor decaiga en valor conforme avanza el tiempo, por lo que se estaría ante un *método -greedy con decaimiento progresivo*. Parte de lo anteriormente expuesto se observa en el siguiente ejemplo.

Ejemplo 17: Implementación del *MBA* con R y alternativas *greedy*

En el programa data/*ejemplo14.R* se supone 3 tipos de anuncios on-line que siguen una distribución de respuestas tipo *binomial* con la siguiente tasas de respuestas una vez contrastadas empíricamente (véase <https://www.r-bloggers.com/2020/09/multi-armed-bandit-with-thompson-sampling/> para más información):

* Variante 1: *N* = 1000, respuestas = 100 luego *RR* = 10% como los parámetros no tienen que seguir una distribución fija mediante el uso del *Teorema de Bayes*, se plantea su *distribución conjugada* de éxitos que tiene en cuenta la variabilidad de ese parámetro *p* y en consecuencia, cabe asociar la siguiente distribución *beta* (*conjugada*) de parámetros *B*(101; 901)
* Variante 2: *N* = 1000, respuestas = 110 luego *RR* = 11% como los parámetros no tienen que seguir una distribución fija mediante el uso del *Teorema de Bayes*, se plantea su *distribución conjugada* de éxitos que tiene en cuenta la variabilidad de ese parámetro *p* y en consecuencia, cabe asociar la siguiente distribución *beta* (*conjugada*) de parámetros *B*(111; 891)
* Variante 3: *N* = 100, respuestas = 10 luego *RR* = 10% como los parámetros no tienen que seguir una distribución fija mediante el uso del *Teorema de Bayes*, se plantea su *distribución conjugada* de éxitos que tiene en cuenta la variabilidad de ese parámetro *p* y en consecuencia, cabe asociar la siguiente distribución *beta* (*conjugada*) de parámetros *B*(11; 91)

Se realizan una 5000 simulaciones y se toman las proporciones de respuestas en cada variante:



Resultando el siguiente vector:



Se observa que se converge a la situación con más observaciones y mejor tasa. ¿Cómo sería el resultado en la siguiente distribución de *recompensas* y *observaciones* (110; 120; 130) / (1000; 800; 700)



Explíquese este resultado tan extremo.

\_\_\_\_\_

### Las Ecuaciones de Bellman

Para finalizar esta sesión, se comenta esta último resultado que proviene de los años 50 cuando Richard Bellman establece los principios de la *programación dinámica* mediante el uso de sus famosas ecuaciones.

La base teórica que sustenta estas ecuaciones se base en el principio de *divide y vencerás* , considerando la notación anterior con estas ecuaciones se pretende dotar de un método de valoración de los estados, en función de las acciones a tomar así pues, si se pretende valorar el estado inicial, el valor a tomar vendría regido por:

Bajo el *principio de optimalidad de Bellman*, se establece que *una política óptima tiene la propiedad de que cualquiera que sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una política óptima en relación con el estado resultante de la primera decisión*. Por tanto, se trata de buscar sub-problemas sencillos de resolver que lleven finalmente a la resolución del problema total, algo parecido al *problema de la mochila* que se analizó anteriormente. Esto implica que, bajo el planteamiento anterior, las decisiones futuras pueden plantearse del siguiente modo:

Como se observa, lo que está dentro del corchete es , por tanto cabe plantear la siguiente expresión:

Expresión que permite deducir directamente las *Ecuaciónes de Bellman* que se derivan de la siguiente dinámica temporal:

Como se observa, resulta clave la descripción de la *función – valor V*, y es necesario el conocimiento de todas las *acciones* posibles dado un determinado *estado*, junto a su valoración, esto implica su resolución permite entender cómo es la metodología que se utiliza.

# **BIBLIOGRAFÍA**

**Bonifacio, M; Sanz Molina, A. (2001)** *Redes Neuronales y Sistemas Borrosos* Ed. RAMA ISBN 84-7897-466-0

**Bonifacio, M; Sanz Molina; A. Serrano-Cinca C.(1995)** *Fundamentos de las redes neuronales artificiales: hardware y software* JO - Scire: Representación y organización del conocimiento, ISSN 1135-3716, Vol. 1, Nº 1, 1995, pags. 103-125

**Gareth, J.; Witten, D.; Hastie, T. y Tibshirani R. (2013)** *An Introduction to Statistical Learning with Applications in R* Springer Science + Business Media New York ISBN 978-1-4614-7137-0

**Hastie, T.; Tibshirani, R.; Friedman, J. (2008)** *The Elements of Statistical Learning. Data Mining, Inference, and Prediction* Springer

**Terence, L. F. (1999)** *Feedforward Neural Network Methodology* Springer-Verlag New York, Berlín, Heidelberg ISBN 0-387-98745-2

**Wehrens, R.; M. C. Buydens, L. (2007)** *Self and Super-organizing Maps in R: The Kohonen Package. Journal of Statistical Software* Oct 2007, Vol 21, Issue 5