Treball final de grau

GRAU DE MATEMÀTIQUES

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Machine Learning: Modelos Ocultos de Markov (HMM) y Redes Neuronales Artificiales (ANN)

Autor: Juan Tornero Lucas

Director: Dr. Josep Vives
Realitzat a: Departament de Matemàtiques i Informàtica.
Barcelona, June 29, 2017

Introducción

El comienzo del siglo XXI ha visto el nacimiento de una nueva revolución, la de la información. Sólo entre 2013 y 2015 la humanidad generó más datos que en toda su historia previa. Esta avalancha de bytes ha servido como catalizador para el desarrollo de herramientas capaces de clasificar grandes cantidades de información. Es en este marco donde los denominados algoritmos de Machine Learning han encontrado un terreno fértil donde prosperar.

Se define Machine Learning como la disciplina del ámbito de la Inteligencia Artificial que crea mecanismos capaces de aprender de forma autónoma. En general, la característica principal de todos los algoritmos de este tipo es que su comportamiento está determinado por una serie de parámetros, y mediante el empleo de grandes muestras de entrenamiento, se busca modificar estos parámetros de manera que el algoritmo sea capaz de llevar a cabo una tarea clasificatoria de manera eficaz.

Dos de los mecanismos de Machine Learning más famosos son los Modelos Ocultos de Markov (HMM) y las Redes Neuronales Artificiales (ANN).

Los HMM se enmarcan dentro de la teoría de las Cadenas de Markov, y aunque estos procesos estocásticos son conocidos desde su introducción en 1907 por el matemático ruso Andréi Markov, la gran cantidad de aplicaciones prácticas que tienen los ha convertido en uno de los modelos probabilísticos más empleados en la actualidad.

Por otro lado, las Redes Neuronales Artificiales son un modelo computacional basado en el comportamiento de las neuronas en los cerebros biológicos. Ya en los años 50 se crearon los primero algoritmos de este tipo, pero en 1969 Minsky y Papert demostrarón que estos primeros ANN's eran incapaces de procesar el operador lógico XOR (disyuntor lógico de dos operandos que solo es cierto cuando los dos valores de entrada son diferentes), lo cual fue un duro golpe para este campo. Sin embargo, en 1975 este obstáculo fue superado por Paul Werbos con la creación del algoritmo de Back-Propagation, y hoy en día esta técnica se encuentra en pleno renacimiento, con compañías como Google o Facebook usando ANN's de gran complejidad en muchos de sus algoritmos estrella.

En este trabajo estudiamos primero procesos en espacios de secuencias, marco teórico de las cadenas Markov, y daremos unos cuantos resultados importantes como la existencia y unicidad de estos procesos empleando una versión del Teorema de Extensión de Kolmogorov.

Continuaremos definiendo qué es una cadena de Markov y comentaremos sus principales características y propiedades

En el tercer capítulo analizamos los Modelos Ocultos de Markov y daremos algunos ejemplos que hemos utilizado en la realidad.

El cuarto capítulo estará dedicado a las Redes Neuronales Artificiales: definiciones básicas, componentes y principales algoritmos que se emplean, así como un par de ejemplos donde se aplican.

En el último capítulo terminamos dando las nociones de cómo sería un modelo que combinara ANN's y HMM's para poder explotar las principales ventajas de ambos.

1. Marco Teórico de las Cadenas de Markov

En este primer capítulo desarrollamos toda la teoría en la que se asienta las Cadenas de Markov. Empezaremos viendo qué son las secuencias y, a continuación, el tipo de espacio de probabilidad conocido como el de *Procesos*. Además, mediante una versión adaptada del *Teorema de Extensión de Kolmogorov* demostramos que estos procesos están definidos de forma única.

Secuencias

Comenzamos definiendo el conjunto canónico $X = \{0, 1, \ldots, m-1\}$ con $m \in \mathbb{N}$ llamado *alfabeto*, cuyos elementos se denominan *símbolos*, los cuales no tienen por qué ser necesariamente números naturales. Si $x \in X^{\mathbb{Z}}$, entonces $x = \cdots x_{-1}x_0x_1\cdots$ es una secuencia numerable bi-infinita, donde los índices $i < 0$ denotan el pasado de la secuencia, y los $i \geq 0$ el futuro, particularmente el índice $i = 0$ es el primer símbolo desconocido de la secuencia.

En estos términos definimos una palabra $w \in X^l$ de longitud l, como una l-tupla de X. $\emptyset$ denotará la palabra vacía de longitud 0. Una subsecuencia s es una estructura $s = (w, a, b)$, donde w es una palabra y $a, b \in \mathbb{Z}$ tq $|w| = b - a + 1$. Así, s puede también escribirse $s = s_a \cdots s_b$, y a representaría el tiempo inicial, y b el tiempo final. Una secuencia x contendrá a la subsecuencia s si $\forall t \in [a, b], s_t = x_t$. En muchas ocasiones identificaremos s y w indistintamente, ya que lo único que las diferencia es el contexto temporal que le da a w la subsecuencia s.

El conjunto $A_s = \{x \in X^{\mathbb{Z}} | x_i = s_i \forall i \in [a, b]\}$ es el conjunto de las secuencias de $X^{\mathbb{Z}}$ que contienen a s. Si por ejemplo $s = (\emptyset, (a, a-1))$, entonces $A_s = X^{\mathbb{Z}}$.

Por último, el conjunto X^* denotará el de todas las palabras.

Procesos

Un proceso Q es un espacio de probabilidad estacionario en un espacio de secuencias.

Una medida de probabilidad es una función que asigna probabilidades a conjuntos del espacio medible de probabilidades $(X^{\mathbb{Z}}, \mathbb{X})$, donde $\mathbb{X}$ es el conjunto definido como la menor colección de subconjuntos de $X^{\mathbb{Z}}$ tal que:

1. Para toda secuencia $s, A_s \in \mathbb{X}$.
2. $\mathbb{X}$ es cerrado bajo complementos y uniones contables. Es decir, $\mathbb{X}$ es una σ-álgebra.

En definitiva, $\mathbb{X}$ es la menor σ-álgebra que contiene los conjuntos A_s fijados por s.

Definiremos P la medida de probabilidad sobre $X^{\mathbb{Z}}$ tal que $P(s) = P(A_s)$, tenemos en particular:

$$P(\emptyset) = P(X^{\mathbb{Z}})$$

En nuestra definición de proceso nos referíamos también a ellos como estacionarios. Esto quiere decir que si D es la función desplazamiento $D : X^{\mathbb{Z}} \to X^{\mathbb{Z}}$, que actúa sobre todo $x \in X^{\mathbb{Z}}$ de manera que $D(x_t) = x_{t+1}$, es decir D desplaza el tiempo de origen, entonces se cumple $P(D^{-1}(A)) = P(D^{-1}(D(A))) = P(A)$. Esto en particular nos permitirá tener bien definido $P(w)$.

Finalmente podemos definir de forma más formal un proceso Q como el espacio de probabilidades estacionario $(X^{\mathbb{Z}}, \mathbb{X}, P)$.

Sea w una palabra y $s = (w, a, b)$, entonces si P estacionario $P(w) = P(s)$. De manera trivial podemos obtener que, si W_l es el conjunto de las palabras de longitud $l > 0$:

$$\sum_{z \in W_l} P(wz) = \sum_{z \in W_l} P(zw) = P(w) \tag{1}$$

Además, para $w = \emptyset$:

$$P(\emptyset) = 1 \tag{2}$$

El recíproco también es cierto, y cualquier función de X^* que satisfaga (1) y (2) define un proceso.

Unicidad de los Procesos

Este apartado estará dedicado a demostrar la unicidad de los procesos mediante el siguiente Teorema. Nos basamos en la demostración dada por Daniel Ray Upper[1].

Teorema. Dado $f : X^* \to [0, 1]$ que satisface:

1. $f(\emptyset) = 1$
2. $\forall w \in X^*, f(w) = \sum_{z \in X^*} f(zw) = \sum_{z \in X^*} f(wz)$

existe un único proceso $Q = (X^{\mathbb{Z}}, \mathbb{X}, P)$ tq $\forall w \in X^*, P(w) = f(w)$.

Este resultado deriva del Teorema de Extensión de Kolmogorov (TEK). Usamos las notaciones $\mathcal{B}(\mathbb{R}^n)$ y $\mathcal{B}(\mathbb{R}^{\mathbb{N}})$ para referirnos a las σ-álgebras en $\mathbb{R}^n$ y $\mathbb{R}^{\mathbb{N}}$ respectivamente.

Teorema de Extensión de Kolmogorov. Suponemos que nos dan un conjunto de medidas de probabilidad μ_n consistentes en $(\mathbb{R}^n, \mathcal{B}(\mathbb{R}^n))$, es decir que cumplen:

$$\mu_{n+1}(\prod_{i=1}^{n}(a_i, b_i] \times \mathbb{R}) = \mu_n(\prod_{i=1}^{n}(a_i, b_i]).$$

Entonces existe una única medida de probabilidad P' en $(\mathbb{R}^{\mathbb{N}}, R^{\mathbb{N}})$, con

$$P'(x \in \prod_{i=1}^{n}(a_i, b_i]) = \mu_n(\prod_{i=1}^{n}(a_i, b_i]).$$

No demostraremos el TEK directamente, sino que lo usaremos para demostrar una variante del mismo que se adapte a nuestras necesidades, teniendo en cuenta las diferencias que existen entre el espacio de probabilidad $(\mathbb{R}^{\mathbb{N}}, \mathcal{B}(\mathbb{R}^{\mathbb{N}}), P')$ y el de un proceso estacionario $(X^{\mathbb{Z}}, \mathbb{X}, P)$

1. $\mathbb{R}^{\mathbb{N}}$ es producto de copias de $\mathbb{R}$, mientras que $X^{\mathbb{Z}}$ es producto de copias de un conjunto finito X. Para resolver esta diferencia utilizaremos una aplicación inyectiva $g : X \to \mathbb{R}$.
2. Los elementos de $\mathbb{R}^{\mathbb{N}}$ son secuencias infinitas mientras que los de $X^{\mathbb{Z}}$ son, de hecho, secuencias bi-infinitas. Utilizaremos de nuevo una aplicación, esta vez biyectiva $h : \mathbb{N} \to \mathbb{Z}$. Consideraremos los dígitos en el orden $0, 1, -1, 2, -2, \ldots$.
3. Por último, P' no necesita ser estacionario, así que usaremos el TEK para provar que P existe, y luego veremos que es estacionario.

Introducimos primero las aplicaciones g y h.

$g : X \to \mathbb{R}$ sólo ha de cumplir que sea inyectivo, sin importar la imagen de los símbolos $x \in X$.

La aplicación biyectiva $h : \mathbb{N} \to \mathbb{Z}$ será

$$h(n) = \begin{cases} n/2 & n \text{ par} \\ (1-n)/2 & n \text{ impar} \end{cases}$$

Su inversa, por lo tanto:

$$h^{-1}(y) = \begin{cases} 2y & y > 0 \\ 1 - 2y & y \leq 0. \end{cases}$$

Vemos como h envía $1, 2, 3, 4, 5$ a $0, 1, -1, 2, -2$ respectivamente, y viceversa con h^{-1}. Definiremos el conjunto $J(n) = \{y \in \mathbb{Z} | h^{-1}(y) \leq n\}$, y los valores $d(n)$ y $c(n)$ como los mayores y menores de $J(n)$ respectivamente, caracterizadas de la siguiente manera:

$$c(n) = min(h(n), h(n-1)) = \begin{cases} \frac{2-n}{2} & n \text{ par} \\ \frac{1-n}{2} & n \text{ impar} \end{cases}$$

$$d(n) = max(h(n), h(n-1)) = \begin{cases} \frac{n}{2} & n \text{ par} \\ \frac{n-1}{2} & n \text{ impar} \end{cases}$$

$$J(n) = \{c(n), \ldots, d(n)\}$$

Observamos que c y d cumplen las propiedades:

P1. Si n par, entonces $h(n+1) < 0, c(n+1) = c(n) - 1, d(n+1) = d(n)$.

P2. Si n impar, entonces $h(n+1) > 0, c(n+1) = c(n), d(n+1) = d(n) + 1$.

Demostración

P1. Si n par, entonces $n = 2i$, para algún $i \in \mathbb{N} \backslash 0$, y

$$h(n+1) \overbrace{=}^{n+1 \text{ impar}} \frac{1-(n+1)}{2} = \frac{-2i}{2} = -i < 0$$

y además

$$\begin{cases} c(n+1) \overbrace{=}^{n+1 \text{ impar}} \frac{1-(2i+1)}{2} = -i \\ c(n) \overbrace{=}^{n \text{ par}} \frac{2-2i}{2} = -i+1 \end{cases}$$

Luego $c(n+1) = c(n) - 1$ y

$$\begin{cases} d(n) \overbrace{=}^{n \text{ par}} \frac{2i}{2} = i \\ d(n+1) \overbrace{=}^{n+1 \text{ impar}} \frac{(2i+1)-1}{2} = i \end{cases}$$

Por lo tanto $d(n+1) = d(n)$.

P2. Si n impar, entonces $n = 2i+1$, para algún $i \in \mathbb{N}\backslash\{0\}$, y

$$h(n+1) \overbrace{=}^{n+1 \text{ par}} \frac{2i+2}{2} = i+1 > 0$$

y también

$$\begin{cases} c(n+1) \overbrace{=}^{n+1 \text{ par}} \frac{2-(2i+2)}{2} = -i \\ c(n) \overbrace{=}^{n \text{ impar}} \frac{1-(2i+1)}{2} = -i \end{cases}$$

Luego $c(n+1) = c(n)$, y

$$\begin{cases} d(n) \overbrace{=}^{n \text{ impar}} \frac{(2i+1)-1}{2} = i \\ d(n+1) \overbrace{=}^{n+1 \text{ par}} \frac{2i+2}{2} = i+1 \end{cases}$$

Finalmente $d(n+1) = d(n) + 1$.

Necesitaremos también definir una serie de conjuntos de subsecuencias, una función que nos aplique esos conjuntos y, por último, un conjunto producto.

Empezamos definiendo estos conjuntos de subsecuencias:

1. $X^{[a,b]} \subset X^*$ el conjunto de todas las subsecuencias $s = (w, a, b)$ que empiezan en el tiempo a y terminan en el tiempo b. Vemos cómo incluimos a s en el conjunto X^* de las palabras, por la identificación que hacemos de s y w.
2. De forma similar $X^{J(n)} \subset X^*$ denotará el conjunto de todas las subsecuencias $s = (w, (c(n), d(n)))$ que empiezan en $c(n)$ y terminan en $d(n)$. Como $|J(n)| = n$, $X^{J(n)}$ es un producto de n copias de X. La diferencia entre X^n y $X^{J(n)}$ es cómo están etiquetadas las coordenadas, en el primer caso el orden es $1, 2, \ldots, n$ mientras que el segundo lo hace como $c(n), \ldots, d(n)$. Por las propiedades P1 y P2 demostradas anteriormente, tenemos que:

Si n es impar:

$$X^{J(n+1)} = X^{J(n)} \times X$$

Si n es par:

$$X^{J(n+1)} = X \times X^{J(n)}$$

Ahora definimos la función H que envía subsecuencias en $X^{J(n)}$ a subsecuencias en $\mathbb{R}^n$. En el caso que $n = \infty$, H enviará secuencias bi-infinitas de $X^{\mathbb{Z}}$ a secuencias infinitas de $\mathbb{R}^{\mathbb{N}}$. El objetivo de esta función es reordenar los argumentos de las coordenadas y enviarlas a $\mathbb{R}$. Si $x \in X^{J(n)}$, entonces hay una subsecuencia $v \in \mathbb{R}^n$ que satisface $v_i = g(x_{h(i)}), \forall i \in \{1, \ldots, n\}$. Recordamos que g es una función a la que solo imponemos que sea inyectiva, por lo tanto no es invertible. Definimos $H : X^{J(n)} \to \mathbb{R}^n$ como $H(x) = v$. Si $x = x_{c(n)}, \ldots, x_{d(n)} \in X^{J(n)}$, entonces:

$$H(x_{c(n)}x_{c_n+1} \ldots x_{d(n)}) = g(x_0)g(x_1) \ldots g(x_{|c(n)|+d(n)+1})$$

Esta función puede ser igualmente definida para el caso $x \in X^{\mathbb{Z}}$. Y aunque H no es invertible, porque g no es invertible, sí que tiene conjuntos inversos. Por ejemplo, si $A \subset \mathbb{R}^n$, entonces:

$$H^{-1}(A) = \{x \in X^{J(n)} | H(x) \in A\}.$$

Finalmente, definimos el conjunto producto S como un subconjunto de $X^{[a,b]}$ de la siguiente forma $S = S_a \times \ldots \times S_b$, donde $S_i \subset X$. Si $S' \subset X$, entonces $S \times S'$ es un producto contenido en $X^{[a,b+1]}$. De forma natural definimos el cilindro $C(S)$ de la siguiente manera:

$$C(S) = \{x \in X^{\mathbb{Z}} | x_i \in S_i, i \in [a, b]\}$$

De forma parecida a como definíamos A_s, tenemos que $C(S)$ es el conjunto de todas las secuencias de $X^{\mathbb{Z}}$ que coinciden con la subsecuencia en S.

A partir de $C(S)$ definimos también el cilindro desplazado $D(S)$ como:

$$D(S) = \{x \in X^{[a-1,b-1]} | x_i \in S_{i+1}, i + 1 \in [a, b]\}.$$

Lema. Los cilindros C y D verifican:

1. $C(S) = C(S \times X) = C(X \times S)$
2. $D(S \times X) = D(S) \times X = X \times D(S)$
3. $D(C(S)) = C(D(S))$

Demostración.

Sea $S = S_a \times \ldots \times S_b$.

1. Por definición, $\forall i \in \mathbb{Z}, x_i \in X$:

   $C(S) = \{x \in X^{\mathbb{Z}} | x_i \in S_i, i \in [a,b]\} = \{x \in X^{\mathbb{Z}} | x_{a-1} \in X \text{ y } x_i \in S_i, i \in [a,b]\} = C(X \times S)$

   De la misma manera:

   $C(S) = \{x \in X^{\mathbb{Z}} | x_i \in S_i, i \in [a,b]\} = \{x \in X^{\mathbb{Z}} | x_{b+1} \in X \text{ y } x_i \in S_i, i \in [a,b]\} = C(S \times X)$

2. Teniendo en cuenta $S \times X \subset X^{[a,b+1]}$:

$$\begin{aligned} D(S \times X) &= \{x \in X^{[a-1,b]} | x_{b+1} \in X \text{ y } x_i \in S_{i+1}, i+1 \in [a,b]\} \\ &= \{x \in X^{[a-1,b-1]} | x_i \in S_{i+1}, i+1 \in [a,b]\} \times X \end{aligned}$$

   De forma semejante a lo visto anteriormente también tenemos:

$$\begin{aligned} D(S \times X) &= \{x \in X^{[a-2,b-1]} | x_{a-2} \in X \text{ y } x_i \in S_{i+1}, i+1 \in [a,b]\} \\ &= X \times \{x \in X^{[a-1,b-1]} | x_i \in S_{i+1}, i+1 \in [a,b]\} \end{aligned}$$

3.
$$\begin{aligned} D(C(S)) &= \{x \in X^{\mathbb{Z}-1} | x_i \in S_{i+1}, \forall i+1 \in [a,b] \text{ y } x_j \in X, \forall j \notin [a,b]\} \\ &= \{x \in X^{\mathbb{Z}} | x_i \in S_{i+1}, \forall i+1 \in [a,b]\} \\ &= C(D(S)) \end{aligned}$$

   cqd

Corolario. Sea $S = S_{c(n)} \times \ldots \times S_{d(n)} \subset X^{J(n)}$ un conjunto producto, y sea

$$A = C(S) = \{x \in X^{\mathbb{Z}} | x_i \in S_i, i \in [c(n), d(n)]\}$$

Entonces tenemos

$$D(S) = \{x \in X^{[c(n)-1, d(n)-1]} | x_i \in S_{i+1}, i+1 \in [c(n), d(n)]\}$$

y

$$D(A) = \{x \in X^{\mathbb{Z}} | x_i \in S_{i+1}, i+1 \in [c(n), d(n)]\}$$

Además $D(A) = C(D(A))$ y, por el lema $D(A) = C(D(S) \times X)$.

Finalmente, observamos que para cada $S \in X^{[a,b]}$ existe un conjunto $S' \in X^{J(n)}$, donde $n = min(-2a, 2b+1)$, tal que $C(S) = C(S')$. Como cada conjunto cilindrico puede ser escrito como $C(S)$ para algún $S \in X^{[a,b]}$, esto significa que cada conjunto cilíndrico puede ser escrito como $C(S')$ para algún $S' \in X^{J(n)}$.

Finalmente, ya estamos en disposición de demostrar nuestra variante del TEK.

Teorema. Suponemos que tenemos una secuencia de medidas v_n en $X^{J(n)}$ que satisfacen que $\forall n$ y para todo $S \subset X^{J_n}$, si n es impar,

$$v_n(S) = v_{n+1}(S \times X) \tag{T.1}$$

y si n es par,

$$v_n(S) = v_{n+1}(X \times S) \tag{T.2}$$

$$v_n(S) = v_{n+1}(D(S) \times X), \tag{T.3}$$

Entonce existe un único proceso estacionario $Q = (X^{\mathbb{Z}}, \mathbb{X}, P)$ tal que, $\forall n, S \subset X^{J_n}$,

$$P(C(S)) = v_n(S)$$

La demostración la haremos en dos partes: primero demostraremos que P existe y, posteriormente comprobaremos que P es estacionario.

Demostración. Definimos una secuencia de medidas μ_n en $\mathbb{R}^{\mathbb{N}}$ de la siguiente manera: si $A \subset \mathbb{R}^{\mathbb{N}}$, entonces definimos $\mu_n(A) = v_n(H^{-1}(A)) = v_n\{x \in X^{J(n)} | H(x) \in A\}$, por la definición de anticonjuntos que establece H.

La consistencia de estas μ's se puede comprobar:

$$\mu_{n+1}(A \times \mathbb{R}) = v_{n+1}(H^{-1}(A \times \mathbb{R})).$$

Dado que

$$H^{-1}(A \times \mathbb{R}) = \begin{cases} H^{-1}(A) \times X & n \text{ impar} \\ X \times H^{-1}(A) & n \text{ par,} \end{cases}$$

obtenemos para μ_{n+1}

$$\mu_{n+1}(A \times \mathbb{R}) = \begin{cases} v_{n+1}(H^{-1}(A) \times X) & n \text{ impar} \\ v_{n+1}(X \times H^{-1}(A)) & n \text{ par.} \end{cases}$$

Ahora, aplicando T.1 y T.2 a la derecha de las igualdades

$$\mu_{n+1}(A \times \mathbb{R}) = v_n(H^{-1}(A)) = \mu_n(A).$$

Aplicando TEK sobre las μ_n's, obtenemos una única P' en $(\mathbb{R}^{\mathbb{N}}, \mathcal{B}(\mathbb{R}^{\mathbb{N}}))$ que concuerda con μ_n. Por lo tanto, si $A = A_1 \times \ldots \times A_n, \quad \forall i \in \{1, \ldots, n\}$, entonces existe una única probabilidad P' que concuerda con μ_n

$$P'(x | x_i \in A_i, i \in \{1, \ldots, n\}) = \mu_n(A)$$

Ahora pasamos a demostrar la segunda parte del Teorema. Definimos P tq $\forall S \in \mathbb{X}$,

$$P(S) = P'(H(x) | x \in S)$$

Probamos primero que P existe. Sea $S \in X^{J(n)}$, tenemos

$$v_n(S) = \mu_n(H(S)) = P'\Big(a \in \mathbb{R}^{\mathbb{N}} | a_i \dots a_n \in H(S)\Big) = P\Big(x \in X^{\mathbb{Z}} | x_{c(n)} \dots x_{d(n)} \in S\Big) = P(C(S))$$

Por lo tanto, esta P es en realidad una extensión de las μ_n's y existe.

Para ver que P es estacionaria, necesitamos demostrar que $P(D(A)) = P(A)$ para cualquier $A \in \mathbb{X}$ que contenga al cilindro $C(S)$. Llamaremos a esta colección Θ

$$\Theta = \{C(S) | S \subset X^{J(n)} \text{ para algún } n\}$$

Aunque cada $A \in \Theta$ pueden tener asociado más de un par n, S siempre escogeremos el de menor n -que tiene asociado una única S-.

Si n es par, T.3 y el lema demostrado anteriormente nos da

$$\begin{aligned} P(A) = v_n(S) &= v_{n+1}(D(S) \times X) = P(C(D(S)) \times X) \\ &= P(C(D(S))) = P(D(A)) \end{aligned}$$

Y por último, vemos que ocurre lo mismo con n impar

$$\begin{aligned} P(A) = v_n(S) &= v_{n+1}(S \times X) = v_{n+1}(D(S \times X) \times X) \\ &= P(C(D(S \times X) \times X)) = P(C(D(S \times X) \times X)) \\ &= P(C(D(S) \times X \times X)) = P(C(D(S) \times X) \times X) \\ &= P(D(A) \times X) = P(D(A)) \end{aligned}$$

Por lo tanto, $P(A) = P(D(A))$. *cqd*

Finalmente estamos en disposición de demostrar nuestro nuestro teorema de partida.

Teorema Dado $f : X^* \to [0, 1]$ que satisface:

1. $f(\emptyset) = 1$
2. $\forall w \in X^*, f(w) = \sum_{z \in X^*} f(zw) = \sum_{z \in X^*} f(wz)$

existe un único proceso $Q = (X^{\mathbb{Z}}, \mathbb{X}, P)$ tq $\forall w \in X^*, P(w) = f(w)$.

Demostración. Primero construiremos v_n's que satisfagan T.1, T.2 y T.3 para a continuación aplicar el Teorema que acabamos de demostrar.

$\forall w \in X^*$ y $|w| = n$, sea $S = \{w\}$; entonces definimos

$$v_n(S) = f(w).$$

Para un $S \subset X^{J(n)}$, S es una unión disjunta de conjuntos de la forma $S_w = \{w\}$, para cada $w \in S$. Podemos entonces definir v_n en todos los subconjuntos de $X^{J(n)}$ como

$$v_n(S) = \sum_{w \in S} v_n(S_w) = \sum_{w \in S} f(w).$$

Para que las v_n's sean realmente una medida de probabilidad han de cumplir $v_n\Big(X^{J(n)}\Big) = 1$. Por inducción, si $n = 0$ entonces $X^{J(n)} = \emptyset$ y, por lo tanto $f(\emptyset) = 1$. Para los próximos pasos de la inducción debemos usar la condición 2 del Teorema, separando los casos par e impar.

Suponemos que (1) se cumple en n impar, entonces

$$\begin{aligned} v_{n+1}\Big(X^{J(n+2)}\Big) &= \sum_{w \in X^{J(n)}} f(w) = \sum_{w \in X^{J(n+1)}} \sum_{x \in X} f(wx) \\ &= \sum_{w \in X^{J(n+1)}} f(w) = v_n\Big(X^{J(n)}\Big) = 1. \end{aligned}$$

En el caso de n par se hace de forma semejante

$$\begin{aligned} v_{n+1}\Big(X^{J(n+2)}\Big) &= \sum_{w \in X^{J(n)}} f(w) = \sum_{w \in X^{J(n+1)}} \sum_{x \in X} f(xw) \\ &= \sum_{w \in X^{J(n+1)}} f(w) = v_n\Big(X^{J(n)}\Big) = 1. \end{aligned}$$

Vemos que se satisface T.1 para n impar

$$\begin{aligned} v_n(S) &= \sum_{w \in S} f(w) = \sum_{w \in S} \sum_{x \in X} f(wx) \\ &= \sum_{z \in (S \times X)} f(z) = v_{n+1}(S \times X). \end{aligned}$$

Para n par, comprobamos T.2

$$\begin{aligned} v_n(S) &= \sum_{w \in S} f(w) = \sum_{w \in S} \sum_{x \in X} f(xw) \\ &= \sum_{z \in (X \times X)} f(z) = v_{n+1}(X \times S). \end{aligned}$$

Por último, asumiendo que n es par vemos T.3

$$v_n(S) = \sum_{w \in S} f(w) = \sum_{w \in S} \sum_{x \in X} f(wx) = \sum_{z \in (S \times X)} f(z).$$

Recordamos que, mientras que $X \times S \subset X^{J(n)}$, no ocurre lo mismo con $S \times X \nsubseteq X^{J(n)}$, sin embargo sí sabemos $D(S \times X) = D(S) \times X \subset X^{J(n)}$, y f no depende del índice temporal, así que

$$v_n(S) = \sum_{z \in (S \times X)} f(z) = \sum_{z \in (T(S) \times X)} f(z) = v_{n+1}(D(S) \times X).$$

Ya podemos aplicar el Teorema anterior para ver que $\exists!$ proceso $Q = (X^{\mathbb{Z}}, \mathbb{X}, P)$ tal que, $\forall n, S \in X^{J(n)}$, tenemos que $P(C(S)) = v_n(S)$. Por lo tanto, si definimos $S = \{w\}$ para cualquier longitud n y palabra w, se cumple

$$f(w) = v_n(S) = P(C(S)) = P(w)$$

cqd

Ejemplo de Proceso

Para ilustrar el concepto de proceso, pondremos un pequeño ejemplo basado en un dado 'ideal' de seis caras, $X = \{1, 2, 3, 4, 5, 6\}$ y cualquier palabra de longitud l tendrá asociada una probabilidad de $P(w) = \left(\frac{1}{6}\right)^l$. En particular, si la palabra es de longitud $l = 0$, entonces $P(w) = \left(\frac{1}{6}\right)^0 = 1$. Si además tenemos otra palabra $z \in X$ entonces, a partir de (1) vemos que $f(wz) = f(zw) = \left(\frac{1}{6}\right)^l \sum_{i=1}^{6} \frac{1}{6} = \left(\frac{1}{6}\right) = f(w)$. Por lo tanto cumple tanto (1) como (2) y es un proceso. *cqd*

2. Cadenas de Markov

En este capítulo veremos qué es una Cadena de Markov, probaremos que se tratan de procesos en un espacios de secuencias y, por lo tanto, se les puede aplicar el Teorema de Unicidad que acabamos de ver. Veremos los diferentes comportamientos que los estados asociados a una cadena son capaces de tener y demostraremos varias propiedades básicas.

Definición y Propiedades de una Cadena de Markov

Una *Cadena de Markov (MC)* de n-estados es un triplete (S, A, π), donde S es un conjunto de orden n, A es una matriz cuadrada de dimensión n, y $\pi = \{\pi_1, \ldots, \pi_n\}$ es un vector de longitud n, que cumplen las siguientes propiedades:

1. Cada columna de A tiene suma 1.
2. $\sum_i \pi_i = 1$.
3. $\pi A = \pi$.

Los elementos de S son los estados, A es la matriz de transición y π es la distribución estacionaria sobre los estados S.

Si $\mathbb{S}$ es la σ-álgebra definida por el conjunto de los subconjuntos de $S^{\mathbb{Z}}$, entonces $(S^{\mathbb{Z}}, \mathbb{S})$ será nuestro espacio medible, y definimos nuestra medida de probabilidades P de manera que si $v = v_0 v_1 \ldots v_{l-1}$, con $v_i \in S$,

$$P(v) = \pi_{v_0} a_{v_0 v_1} \ldots a_{v_{l-2} v_{l-1}},$$

Definiendo $P(\emptyset) = 1$, vemos que una MC es un proceso estacionario.

Recordamos que un proceso es estacionario si cumple:

$$\sum_{z \in S} P(wz) = \sum_{z \in S} P(zw) = P(w) \tag{1}$$

$$P(\emptyset) = 1, \tag{2}$$

La condición (2) se cumple trivialmente. Para ver la (1) sea $z \in S$, entonces

$$P(\emptyset z) = P(z\emptyset) = P(z) = \pi_z.$$

Por tanto, aplicando la propiedad 2 de las MC

$$\sum_{z \in S} P(z\emptyset) = \sum_{z \in S} P(\emptyset z) = \sum_{z \in S} \pi_z \overset{P2MC}{=} 1 = P(\emptyset).$$

Ahora sea la probabilidad de la unión de v y z, y teniendo en cuenta la primera propiedad de las MC

$$P(vz) = P(v)a_{v_{l-1}z} \to \sum_{z\in S} P(vz) = P(v)\sum_{z\in S} a_{v_{l-1}z} \overbrace{=}^{P1MC} P(v)\cdot 1 = P(v)$$

En el otro sentido, la unión de z con v nos da

$$P(zv) = \pi_z a_{zv_0} \dots a_{v_{l-2}v_{l-1}}$$

Por lo tanto

$$\sum_{z\in S} P(zv) = \left(\sum_{z\in S} \pi_z a_{zv_0}\right) a_{zv_0} \dots a_{v_{l-2}v_{l-1}}$$

Finalmente, aplicando la tercera propiedad de las MC, si tenemos en cuenta que $\sum_{z\in S} \pi_z a_{zv_0}$ es el elemento v_0 del vector πA, tenemos

$$\sum_{z\in S} \pi_z a_{zv_0} \overbrace{=}^{P3MC} \pi_{v_0}.$$

Y terminamos viendo que

$$\sum_{z\in S} P(zv) = \pi_{v_0} a_{v_0v_1} \dots a_{v_{l-2}v_{l-1}} = P(v).$$

Concluimos que (S, A, π) define un proceso en $(S^{\mathbb{Z}}, \mathbb{S}, P)$, *cqd*

Las MC's pueden considerarse procesos "sin memoria", es decir, que la distribución de probabilidad futura para $X = (X_1, \dots, X_n)$ secuencia de variables aleatorias, depende únicamente del valor actual, más formalmente:

$$P(X_{n+1} = x_{n+1}|X_n = x_n, \dots, X_1 = x_1) = P(X_{n+1} = x_{n+1}|X_n = x_n)$$

Esta identidad es la denominada *propiedad de Márkov*.

Si además $\forall i, j$ estados de la MC se cumple $\forall n > 1, p(i, j) = P(X_{n+1} = j|X_n = i)$, entonces se dice que la cadena es temporalmente homogénea. Además, los términos de la matriz de transición son constantes $A = a_{ij} = p(i, j)$.

Vemos un ejemplo para ilustrar mejor cómo funciona una MC[2]. La *Ruina del Jugador* es un juego de apuestas en el que ganas 1€ con probabilidad $p = 0.4$ y pierdes 1€ con probabilidad $1 - p = 0.6$. Suponemos que dejas de jugar si llegas a tener N€ o 0€.

Tenemos entonces:

$$\begin{aligned} &a_{i,i+1} = 0.4, \quad a_{i,i-1} = 0.6, \quad \text{si } 0 < i < N \\ &a_{0,0} = 1, \quad a_{N,N} = 1 \end{aligned}$$

Para $N = 4$

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 \\ 0.6 & 0 & 0.4 & 0 & 0 \\ 0 & 0.6 & 0 & 0.4 & 0 \\ 0 & 0 & 0.6 & 0 & 0.4 \\ 0 & 0 & 0 & 0 & 1.0 \end{pmatrix} \end{array}$$

En este modelo los estados 0 y N corresponden a estados fijos, ya que son los únicos estados alcanzables, es decir, que tienen probabilidad mayor que 0, son ellos mismos: $p(0,0) = p(N,N) = 1$. A este tipo de estados se les denomina absorbentes.

Ya hemos definido $p(i,j)$ como la probabilidad de ir del estado i al j en un paso. Ahora nos planteamos cómo computar la probabilidad de ir de i a j en $m > 1$ pasos

$$p^m(i,j) = P(X_{n+m} = j | X_n = i) \tag{1}$$

Por ejemplo, consideramos la MC de la Ruina del Jugador para el caso N=4 ¿cuál es la probabilidad de pasar de tener 2€ a tener 0€ en dos jugadas?

Simplemente consideramos los diferentes estados posibles y la propiedad

$$P(B|A) = \frac{P(B \cap A)}{P(A)}. \tag{2}$$

Entonces tenemos:

$$\begin{aligned} P(X_2 = 0 | X_0 = 2) &= \sum_{k=0}^{4} P(X_2 = 0, X_1 = k | X_0 = 2) = \sum_{k=0}^{4} \frac{P(X_2 = 0, X_1 = k, X_0 = 2)}{P(X_0 = 2)} \\ &= \sum_{k=0}^{4} \frac{P(X_2 = 0, X_1 = k, X_0 = 2)}{P(X_1 = k | X_0 = 2)} \frac{P(X_1 = k | X_0 = 2)}{P(X_0 = 2)} \\ &= \sum_{k=0}^{4} P(X_2 = 0 | X_1 = k, X_0 = 2) P(X_1 = k | X_0 = 2) \end{aligned}$$

Aplicando la Propiedad de Markov vemos que

$$P(X_2 = 0 | X_1 = k, X_0 = 2) P(X_1 = k | X_0 = 2) = p(2,k)p(k,0), \quad \forall\ 0 \leq k \leq 4$$

y por lo tanto

$$P(X_2 = 0|X_0 = 2) = \sum_{k=0}^{4} p(2,k)p(k,0)$$
$$= 0 \cdot 0 + 0.6 \cdot 0.6 + 0 \cdot 0 + \ldots = 0.6 \cdot 0.6 = 0.36$$

Una generalización de esta fórmula viene dada por el siguiente teorema.

Teorema. La probabilidad de transición en el paso m, $P(X_{n+m} = j|X_n = i)$ es la m-ésima potencia de la matriz de transición $a^m(i,j) = p^m(i,j)$.

Primero probaremos la ecuación de Chapman-Kolmogorov

$$p^{n+m}(i,j) = \sum_k p^m(i,k)p^n(k,j)$$

Tras esto, solo será necesario sustituir $n = 1$ para para demostrar

$$p^{1+m}(i,j) = \sum_k p^m(i,k)p^1(k,j)$$

Utilizamos lo deducido en el ejemplo anterior

$$P(X_{m+n} = j|X_0 = i) = \sum_k P(X_{n+m} = j, X_m = k|X_0 = i),$$

y usando la propiedad (2) de nuevo

$$P(X_{m+n} = j|X_0 = i) = \sum_k P(X_{n+m} = j, X_m = k|X_0 = i) = \frac{P(X_{n+m} = j, X_m = k, X_0 = i)}{P(X_0 = i)}$$
$$= \frac{P(X_{n+m} = j, X_m = k, X_0 = i)}{P(X_m = k, X_0 = i)} \frac{P(X_m = k, X_0 = i)}{P(X_0 = i)}$$
$$= P(X_{m+n} = j|X_m = k, X_0 = i)P(X_m = k|X_0 = i)$$

Aplicamos a la última expresión la Propiedad de Markov y (1)

$$P(X_{n+m} = j|X_m = k)P(X_m = k|X_0 = i) = p^m(i,k)p^n(k,j)$$

cqd

Retomando la MC de la Ruina del Jugador podemos ahora calcular de forma completa A^2, aplicando este último teorema.

$a_{00}^2 = a_{44}^2 = 1$ al ser estados absorbentes
$a_{13}^2 = a_{12}a_{23} = 0.4^2 = 0.16$ al tener que subir dos veces
$a_{11}^2 = a_{12}a_{21} = 0.4 \cdot 0.6 = 0.24$ la cadena tiene que subir primero y luego bajar
...

Finalmente, tendríamos la matriz de transición

$$A^2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 \\ 0.6 & 0.24 & 0 & 0.16 & 0 \\ 0.36 & 0 & 0.48 & 0 & 0.16 \\ 0 & 0.36 & 0 & 0.24 & 0.4 \\ 0 & 0 & 0 & 0 & 1.0 \end{pmatrix} \end{array}$$

Clasificación de Estados

Para terminar el apartado sobre Cadenas de Markov veremos la clasificación que de sus estados puede hacerse. Normalmente nos interesará saber el comportamiento de la cadena dado un estado inicial

$$P_x(A) = P(A|X_0 = x).$$

Primero introduciremos el concepto de $T_y = min(n \geq 1 : X_n = y)$ como el tiempo del primer retorno a y, suponiendo que partimos de este estado (obviamente no tenemos en cuenta este tiempo inicial). Y derivado de esto

$$\tau_{yy} = P_y(T_y < \infty)$$

la probabiliad de que de que X_n vuelve a y tras empezar en y. De forma intuitiva, la probabilidad de que X_n vuelva a y al menos dos veces es τ_{yy}^2, dado que tras el primer retorno, la cadena está en y, y la probabilidad de retornar una segunda vez es otra vez τ_{yy}.

De esta manera, por inducción

$$T_y^k = min(n > T_y^{k-1} : X_n = y)$$

es el tiempo del k-ésimo retorno a y. Y también

$$P_y(T_y^k < \infty) = \tau_{yy}^k$$

la probabiliad de que de que X_n vuelve a y k veces tras empezar en y.

Por lo tanto, hay dos posibilidades:

1. $\tau_{yy} < 1$: por lo tanto la probabilidad de retornar k veces es $\tau_{yy}^k \to 0$ cuando $k \to \infty$. Y por lo tanto eventualmente la MC no retorna a y. En este caso el estado y se dice que es transitivo.

2. $\tau_{yy} = 1$: La probabilidad de retornar k veces es $\tau_{yy}^k = 1$, así que la MC retorna infinitamente al estado y, el cual se dice en este caso que es recurrente.

Retomando el ejemplo de la Ruina del Jugador, vemos que eventulmente la cadena se queda atascada en 0€ o N€. Para el caso N=4 es fácil comprobar que estos dos estados son recurrentes, ya que $p(0,0) = 1$, y por lo tanto

$$P_0(T_0 = 1) = 1,$$

es decir, $\tau_{00} = 1$. De forma similar se ve que N es recurrente. En general si y es un estado absorbente, entonces y es también recurrente.

Para el caso $N = 4$ vemos que 1,2,3 son estados transitivos. Empezando por 1, la probabilidad de que la cadena nunca vuelva a este estado

$$P_1(T_1 = \infty) \geq p(1,0) = 0.6 > 0$$

es mayor que 0, y por lo tanto $\tau_{11} < 1$. De forma similar para 2

$$P_2(T_2 = \infty) \geq p(2,1)p(1,0) = 0.36 > 0.$$

Por último, para el caso 3 también se puede comprobar viendo que en este caso se puede ir directamente al estado 4, también absorbente

$$P_3(T_3 = \infty) \geq p(3,4) = 0.4 > 0$$

De hecho, se puede demostrar que el límite de la k-exponencial de la matriz de transción A en este caso es

$$\lim_{k\to\infty} A^k = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 \\ 57/65 & 0 & 0 & 0 & 8/65 \\ 45/65 & 0 & 0 & 0 & 20/65 \\ 27/65 & 0 & 0 & 0 & 38/65 \\ 0 & 0 & 0 & 0 & 1.0 \end{pmatrix} \end{array}$$

Lo que confirma que la cadena eventualmente se queda atascada bien en 0€ o en 4€ (para el caso N=4).

3. Modelos Ocultos de Markov (HMM)

En los Modelos ocultos de Markov (HMM), la MC subyace tras las observaciones. Los estados del HMM solo pueden ser inferidos a partir de los símbolos registrados. Correlacionando las observaciones y las transiciones de estado lo que se busca es encontrar el estado de secuencias más probable.

Un HMM puede ser entendido como una especie de doble proceso estocástico:

- El primer proceso estocástico es un conjunto finito de estados, cada uno de ellos generalmente asociado a una distribución de probabilidad multidimensional. Mediante la denominada matriz de transición se controla las transiciones entre estados.
- El segundo proceso estocástico es aquel en que cualquier estado puede ser observado, es decir, analizaremos lo observado sin ver en qué estado está ocurriendo, de aquí el epíteto *oculto* que define a este modelo.

Componentes de un HMM

Para definir completamente un HMM, se necesitan cinco elementos:

1. Los N estados del modelo $S = \{S_1, \ldots, S_N\}$
2. Los M diferentes símbolos $V = \{V_1, \ldots, V_M\}$ que pueden observarse. Si las observaciones son contínuas, obviamente M es infinito.
3. La matriz de transición $A = \{a_{ij}\}$, donde $a_{ij} = P(q_{t+1} = j | q_t = i)$, siendo q_t el estado actual. Esta matriz es equivalente a la vista en la definición de las MC. Hay que observar que si uno de los a_{ij} es definido cero, permanecerá cero durante todo el proceso de entrenamiento.
4. La probabilidad de distribución de los símbolos en cada estado, $B = \{b_j(k)\}$, donde $b_j(k)$ es la probabilidad de observar el símbolo v_k en el estado S_j, viene dada por

$$b_j(k) = P(o_t = v_k | q_t = j), \forall j \in \{1, \ldots, N\}, \forall k \in \{1, \ldots, M\},$$

donde v_k denota el k-ésimo símbolo del alfabeto, y o_t el actual vector de observaciones.

Se deben satisfacer ciertas restricciones:

$$b_j(k) \geq 0, \forall j \in \{1, \ldots, N\}, \forall k \in \{1, \ldots, M\} \text{ y además } \sum_{k=1}^{M} b_j(k) = 1, \forall j \in \{1, \ldots, N\}$$

En el caso contínuo tendremos una función de densidad contínua, en vez de un conjunto de probabilidades discretas. En este caso lo que especificamos es el conjunto de parámetros de la

función de densidad, aproximada como una suma ponderada de M distribuciones Gaussianas (GHMM),

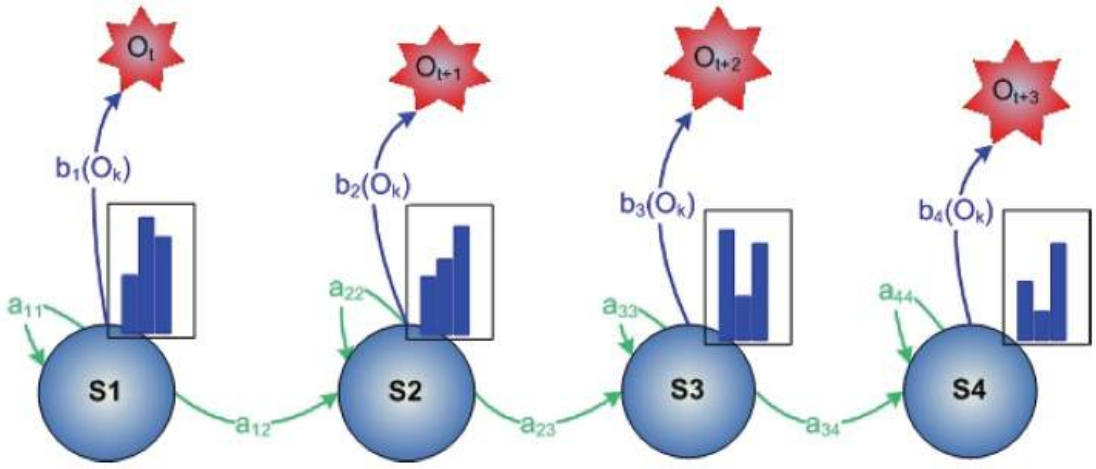
$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} N(\mu_{jm}, \Sigma_{jm}, o_t)$$

donde c_{jm} son los pesos, μ_{jm} los vectores de medias, y Σ_{jm} las matrices de covarianzas. Observar que c_{jm} debe satisfacer las condiciones estocásticas $c_{jm} \geq 0, \forall m \in \{1, \ldots, M\}$ y

$$\sum_{m=1}^{M} c_{jm} = 1, \forall j \in \{1, \ldots, N\}$$

5. La distribución inicial de estados $\pi = \{\pi_i\}$, donde π_i es la probabilidad de que el modelo está en el estado S_i en el tiempo inicial $t = 0$, con

$$\pi_i = P(q_1 = i), \forall i \in \{1, \ldots, N\}$$



Ejemplo de un HMM

Para denotar los parámetros de un HMM con frecuencia se usa

$$\lambda = (A, B, \pi)$$

para denotar distribuciones discretas, o bien

$$\lambda = (A, c_{jm}, \mu_{jm}, \Sigma_{jm}, \pi)$$

Cuando se trata de distribuciones contínuas asociadas a funciones de densidad.

Arquitectura de los HMM

El siguiente diagrama muestra la arquitectura general de un HMM de parámetros λ con los dos procesos estocásticos asociados a este modelo.

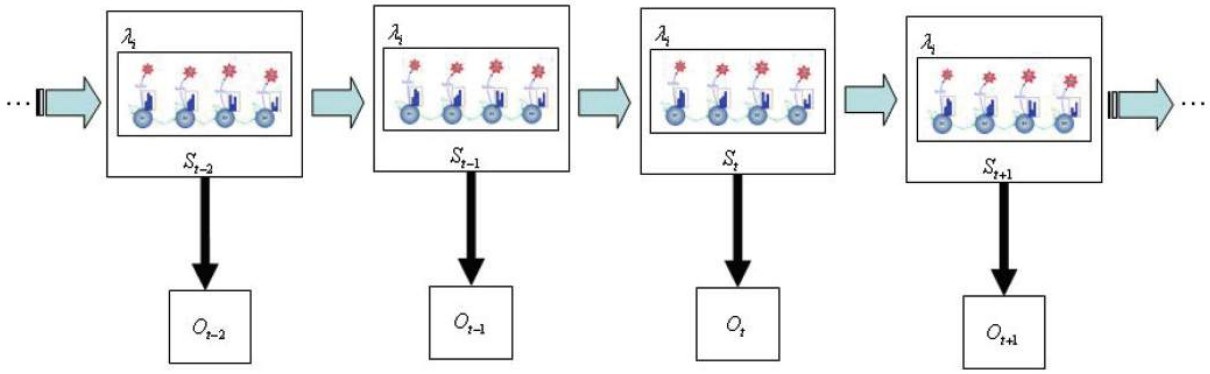


Figure 1: *Esquema del funcionamiento de un HMM.*

Para cada tiempo t tenemos la variable aleatoria $S(t)$ que representa el estado oculto (no observable directamente). La variable O es la de las observaciones.

El primer proceso estocástico resulta de que el valor de la variable oculta $S(t)$ en el tiempo t, conocidos todos los valores de S anteriores solo depende de $S(t-1)$. Por lo tanto la propiedad de Markov es satisfecha.

El segundo procesos estocástico viene de que el valor de $O(t)$ depende de la variable oculta asociada $S(t)$.

Algoritmos Relacionados con los HMM's

El objetivo de un HMM, a grosso modo, es aprender a clasificar los datos proporcionados. Es decir, un HMM tiene que ser capaz de discernir las diferencias de comportamiento que posean los diferentes estados a partir del *stream* de observaciones con el que lo alimentamos.

En la historia de los HMM han destacado los estudios de tres problemas:

1. Problema de Evaluación

Dada la secuencia de observaciones $O = \{o_1, \ldots, o_m\}$, ¿Cuál es la probabilidad $P(O|\lambda)$ de que O haya sido generada por el modelo λ, Dado un λ.

2. Problema de Decodificación

Dada la secuencia de observaciones $O = \{o_1, \ldots, o_m\}$, ¿Cuál es la secuencia de estados más probable dado el modelo λ?

3. Problema de Aprendizaje

Dada la secuencia de observaciones $O = \{o_1, \ldots, o_m\}$, ¿Cómo debemos ajustar los parámetros de λ para maximizar $P(O|\lambda)$?

El problema de evaluación es la piedra angular en muchos estudios de reconocimiento de voz. El problema de decodificación resulta útil a la hora de segmentar, y el problema de aprendizaje debe ser resuelto si queremos entrenar un HMM para su uso en labores de reconocimiento.

Los algoritmos para resolver estos problemas son muy conocidos, en nuestro caso usaremos la versión dada por Przemyslaw Dymarski[3].

Problema de Evaluación: Forward Algorithm

Dada una secuencia de observaciones $O = \{o_1, \ldots, o_T\}$ y un modelo $\lambda = (A, B, \pi)$, se trata de averiguar $P(O|\lambda)$. Lo primero que uno se plantea es que este cálculo se puede realizar directamente mediante la fórmula de Bayes:

$$P(O|\lambda) = \frac{P(\lambda|O)P(O)}{P(\lambda)}.$$

Sin embargo, el número de operaciones requeridas es del orden de N^T, lo que no resula efectivo computacionalmente. Existe sin embargo un método de menos complejidad que involucra la utilización de una variable auxiliar *forward* (α) de la forma

$$\alpha_t(i) = P(o_1, o_2, \ldots, o_t, q_t = i|\lambda)$$

Para el cálculo de esta variable podemos definir una relación recursiva de la forma

$$\begin{cases} \alpha_1(j) = \pi_j b_j(o_1), \forall j \in \{1, \ldots, N\} \\ \alpha_{t+1}(j) = b_j(o_{t+1}) \sum_{i=1}^{N} \alpha_t(i) a_{ij}; \forall j \in \{1, \ldots, N\}, \forall t \in \{1, \ldots, T-1\} \end{cases}$$

Las α_T's se pueden calcular utilizando la recursión. Así que la probabilidad requerida es dada por

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

Este método es conocido comúnmente como *forward algorithm*.

De forma similar podemos definir una variable *backward* $\beta_t(i)$

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, \ldots, o_T|q_t = i, \lambda)$$

Como el estado actual es i, $\beta_t(i)$ es la probabilidad de tener en t el estado i sabiendo la historia futura parcial $o_{t+1}, o_{t+2}, \ldots, o_T$.

Igual que antes, planteamos la fórmula recursiva que nos dará la variable *backward*

$$\begin{cases} \beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij} b_j(o_{t+1}); \forall i \in \{1, \ldots, N\}, \forall t \in \{1, \ldots, T-1\} \\ \beta_T(i) = 1, \forall i \in \{1, \ldots, N\}. \end{cases}$$

Combinando ambas obtenemos

$$\alpha_t(i)\beta_t(i) = P(O, q_t = i|\lambda) \quad \forall i \in \{1, \ldots, N\}, \forall t \in \{1, \ldots, T\}.$$

Finalmente, obtenemos que

$$P(O|\lambda) = \sum_{i=1}^{N} P(O, q_t = i|\lambda) = \sum_{i=1}^{N} \alpha_t(i)\beta_t(i)$$

Más adelante desarrollaremos de forma detallada esta combinación *forward-backward* para resolver el *Problema de Aprendizaje* mediante el algoritmo *Baum-Welch*.

Nota: en cada caso, tanto en las variables backward como forward hay que realizar una normalización en cada tiempo t.

Problema de Decodificación: Algoritmo de Viterbi

El problema de decodificación gira en torno a saber cuál es la secuencia de estados más probable dada una secuencia de observaciones $O = o_1, o_2, \ldots, o_T$ y un modelo $\lambda = (A, B, \pi)$.

El problema de buscar esta secuencia más probable es que, muchas veces, quedarnos sólo con las secuencias de q_t's más probables resulta en una secuencia de estados poco significativa.

Por ello introducimos el denominado *Algoritmo de Viterbi*, que nos permite encontrar el estado de secuencias que tiene mayor verosimilitud.

Utilizaremos una variable auxiliar

$$\delta_t(i) = \max_{q_1, q_2, \ldots, q_{t-1}} P(q_1, q_2, \ldots, q_{t-1}, q_t = i, o_1, o_2, \ldots, o_t|\lambda),$$

Es decir, la probabilidad más alta de la secuencia parcial de estados y observaciones hasta t, dado el estado actual i. Para $t = 1$ se define como $\delta_1(j) = \pi_j b_j(o_1), \forall j \in \{1, \ldots, N\}$, para el resto de términos se usa la fórmula de recursividad

$$\delta_{t+1}(j) = b_j(o_{t+1}) \Big[\max_{1 \leq i \leq N} \delta_t(i) a_{ij} \Big] \quad \forall i \in \{1, \ldots, N\}, \forall t \in \{1, \ldots, T-1\}$$

Esto significa que empezamos nuestro cálculo desde $\delta_T(j), \forall j \in \{1, \ldots, N\}$, mantenemos un puntero al estado "ganador", que será $j* = arg\max_{1 \leq i \leq N} \delta_T(j)$, y a partir de aquí realizamos un *back-track* en la secuencia redefiniendo $j*$ en cada paso, obteniendo de esta manera el conjunto de estados que se pide.

Problema de Aprendizaje: Algoritmo Baum-Welch

El problema de aprendizaje se centra en cómo podemos ajustar los parámetros del HMM para que se ajusten de la mejor forma a las observaciones.

Uno de los criterios más utilizados es el de máxima verosimilitud. Dado el HMM de parámetros λ y las observaciones O, la verosimilitud puede ser expresada como

$$L = P(O|\lambda).$$

Conocer el modelo que maximiza $\lambda = (A, B, \pi)$ es una operación que no puede ser resuelta analíticamente. Existen métodos iterativos, como *Baum-Welch*, o métodos basados en gradientes -como veremos en el capítulo de Redes Neuronales- para encontrar localmente máximos apropiados para ser parámetros del modelo.

Algoritmo Baum-Welch

El algoritmo Baum-Welch, también conocido como *Forward-Backward*, ya que utiliza estas variables que definimos previamente al tratar el problema de evaluación, utiliza una cantidad auxiliar $Q(\lambda, \lambda')$ para comparar dos modelos λ y λ'.

$$Q(\lambda, \lambda') = \sum_q P(q|O, \lambda) log\Big[P(O, q, \lambda')\Big]$$

También definimos unas nuevas variable a partir de las *backward* y *forward*.

La primera, referida habitualmente en la literatura como probabilidades *smoothed*, es la probabilidad de estar en el estado i en el instante t dadas las observaciones O y el modelo λ. Se obtiene combinando α y β de acuerdo al Teorema de Bayes.

$$\gamma_t(i) = P(q_t(i)|O, \gamma) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}$$

Observamos que, si sumamos cada $\gamma_t(i)$ en todos los instantes de tiempo, obtenemos el número de transiciones que se realizan desde i: $\sum_{t=1}^{T-1} \gamma_t(i)$.

La segunda variable nueva, también combina las variables *backward* y *forward* para definir la probabilidad de estar en el estado i en el instante t, y en el estado j en el instante $t+1$; dadas las observaciones O y el modelo γ.

$$\xi_t(i,j) = \frac{P(q_t = i, q_{t+1} = j, O|\lambda)}{P(O|\lambda)} = \frac{\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\sum_{k=1}^{N}\sum_{l=1}^{N}\alpha_k(t)a_{kl}\beta_l(t+1)b_l(y_{t+1})}$$

Si hacemos igual que antes y sumamos cada $\xi_t(i,j)$ para cada instante de tiempo, obtenemos el número esperado de transiciones desde el estado i al estado j en las observaciones O:

$$\sum_{t=1}^{T-1}\xi_t(i,j).$$

Por lo tanto la relación entre $\gamma_t(i)$ y $\xi_t(i)$ viene dada por

$$\gamma_t(i) = \sum_{j=1}^{N}\xi_t(i,j) \quad \forall i \in \{1,\ldots,N\}, \forall t \in \{1,\ldots,M\}.$$

Una vez calculadas estas variables, para resolver el problema de aprendizaje, se definen los nuevos parámetros $\lambda' = (A', B', \pi')$ del HMM utilizando las *smoothed* de acuerdo a las siguiente fórmulas

Probabilidades iniciales: probabilidades de estar en el estado i en $t = 1$:

$$\pi' = \gamma_1(i) \quad \forall i \in \{1,\ldots,N\}.$$

Matriz de transición: cada a_{ij} viene representado por el cociente entre el número esperado de transiciones entre i y j, entre el número total de transiciones desde i:

$$a'_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)} \quad \forall i \in \{1,\ldots,N\}, \forall j \in \{1,\ldots,N\}.$$

Probabilidades de emisión: cociente entre el número de veces que se tiene el estado j y se oberva o_k, y el número esperado de veces que se pasa por el estado j:

$$b'_j(o_k)\frac{\sum_{t=1;o_t=o_k}^{T}\gamma_t(i)}{\sum_{t=1}^{T}\gamma_t(j)} \quad \forall i \in \{1,\ldots,N\}, \forall j \in \{1,\ldots,N\}.$$

En resumen, el algoritmo Baum-Welch se divide en tres fases:

1. Se determina un modelo inicial, que aunque en teoría se puede seleccionar aleatoriamente, es conveniente hacerlo con una cierta aproximación realista para facilitar la convergencia del algoritmo.
2. Se realiza el cálculo de transiciones y símbolos de emisión más probables del modelo inicial, determinando las $\alpha's, \beta's, \gamma's$ y $\xi's$.
3. Se redefinen los parámetros a partir de lo calculado en el punto 2 para crear un nuevo modelo que mejore en verosimilitud al modelo inicial.

Modelo Gaussiano Oculto de Markov: GHMM

En este apartado describiremos un HMM cuyas observaciones tienen una distribución Gaussiana multivariante de dimensión L, diferente para cada estado. Esto quiere decir que, si tenemos K estados diferentes, cada uno de estos estará asociado a una distribución de parámetros $(\mu_i, \Sigma_i) \quad \forall i \in \{1, \ldots, K\}$, con μ_i el vector de medias y Σ_i la matriz de covariancias del estado i.

Al ser un modelo contínuo, existen infinitos símbolos (de hecho, $V = \mathbb{R}$). Usaremos la notación para los modelos continuos de $b_j(o_t) = P(o_t|q_t = i), \quad \forall j \in \{1, \ldots, N\} \quad \forall t \in \{1, \ldots, M\}$; que nos indicará la verosimilitud de la observación o_t respecto i.

Tenemos en cuenta que la función de densidad asociada al estado i será:

$$N(x, \mu_i, \Sigma_i) = \frac{1}{\left(2\pi\right)^{L/2} |\Sigma_i|^{1/2}} e^{-0.5(x-\mu_i)\Sigma_i^{-1}(x-\mu_i)^t},$$

y por lo tanto

$$b_j(o(t)) = \sum_{k=1}^{K} c_{jk} N(x, \mu_j, \Sigma_j).$$

Aplicación de un GHMM

Vemos a continuación cómo podemos emplear un GHMM para modelizar un sistema estocástico como es el de la Bolsa [4] [5].

Nuestro punto de partida será suponer que los retornos pueden pertenecer a dos estados $X = \{1, 2\}$, uno cuando el mercado tiende al alza (1), y otro cuando tiende a la baja (2), en este caso como las observaciones son escalares nuestros estados tendrán parámetros (μ_i, σ_i), con $\mu_1 > \mu_2$.

Nota: nosotros suponemos que las observaciones que el HMM clasifique como estado 1 serán las correspondientes al estado alcista porque corresponden a una μ positiva, sin embargo, por lo general las σ''s de este estado son mayores que las del estado 2, y cabe la posibilidad de que en observaciones muy negativas, aunque sean las menos probables, encajen mejor con la distribución del estado 1 que con la del estado 2 en contra de lo intuitivo.

Nuestro caso de estudio será el correspondiente a los valores diarios de apertura del IBEX35 entre el 19/10/1990 y 12/08/1991.

	Fecha	Valor	Retornos
1	19/10/1990	2208.03	0
2	20/10/1990	2268.76	2.8%
3	21/10/1990	2309.00	1.8%
4	24/10/1990	2331.88	1%
5	25/10/1990	2398.47	2.9%

Table 1: IBEX 35: Valores de Apertura

Trabajaremos con los retornos (R), y estableceremos como condiciones iniciales μ_1 y σ_1 la media y desviación estandard, respectivamente, de los retornos positivos y μ_2 y σ_2 la media y desviación estandard de los retornos negativos. En cuanto a la matriz de transición y a la probabilidad inicial, los suponemos de la siguiente manera:

$$A = \{a_{ij}\} = \{P(X_t = j | X_{t-1} = i)\} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix}$$

$$\begin{cases} \pi_1 = 0.5 \\ \pi_2 = 1 - \pi_1 = 0.5 \end{cases}$$

Tras aplicar el algoritmo de BaumWelch, obtenemos la estimación de los parámetros:

$$\mu = \begin{pmatrix} 5.8 \cdot 10^{-3} \\ -3.6 \cdot 10^{-3} \end{pmatrix}$$

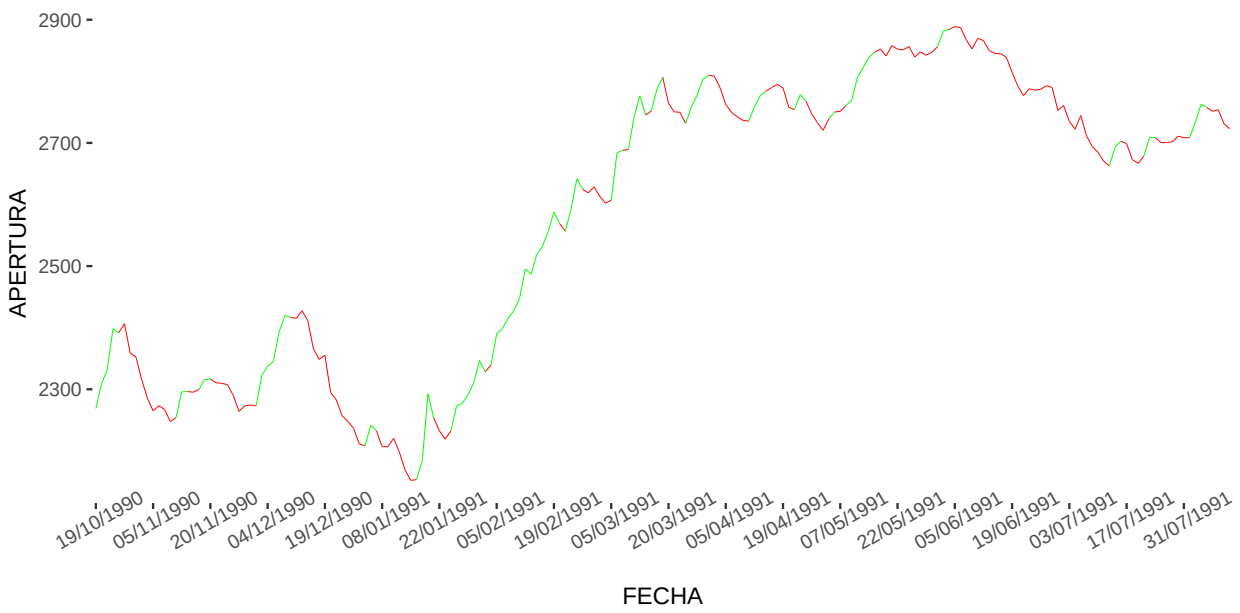
$$\sigma = \begin{pmatrix} 9.43 \cdot 10^{-5} \\ 4.17 \cdot 10^{-5} \end{pmatrix}$$

La nueva matriz de transición y probabilidad inicial son:

$$A = \begin{pmatrix} 0.802 & 0.198 \\ 0.196 & 0.804 \end{pmatrix}$$

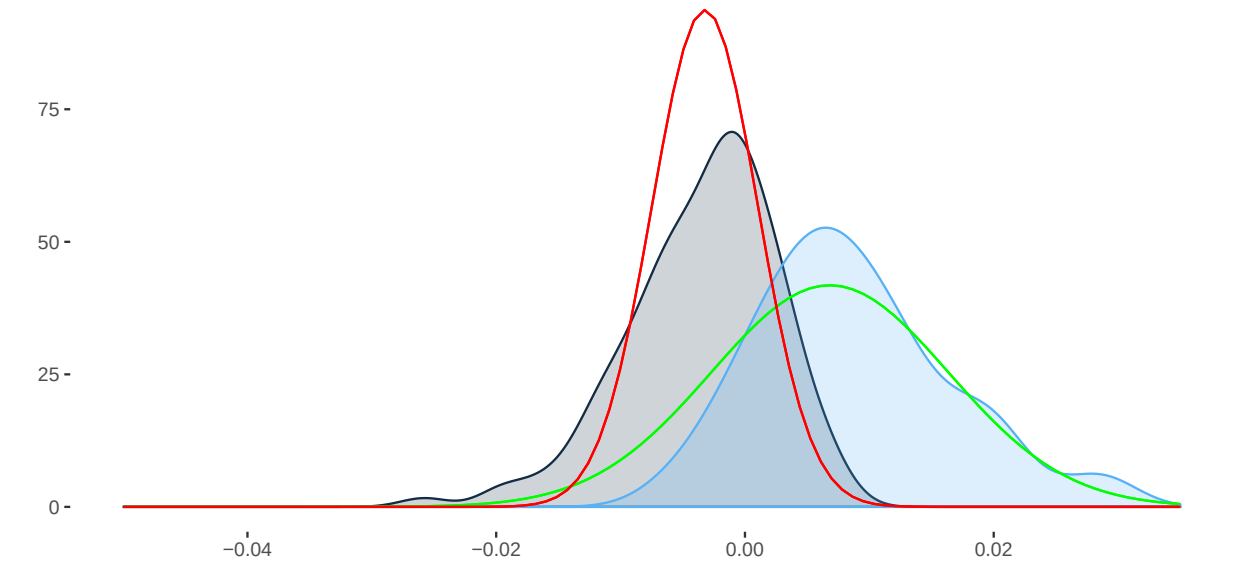
$$\begin{cases} \pi_1 = 1 \\ \pi_2 = 0 \end{cases}$$

Para ilustrar la capacidad clasificadora del HMM, mostramos un gráfico de la evolución de los valores del IBEX35 en el periodo estudiado, junto a la clasificación que ha hecho el algoritmo de Viterbi de la secuencia de estados más probables.



Gráfica 1. *Gráfico del IBEX 35 entre el 19/10/1990 y 12/08/1991, en verde se han indicado los retornos clasificados como alcistas, y en rojo los bajistas.*

Vemos cómo nuestra clasificación de los retornos se aproxima de manera muy efectiva al comportamiento que muestra la gráfica. Aunque hay retornos positivos y negativos en ambos estados, la distribución a la que se atribuye cada retorno se hace en función del régimen alcista/bajista en que se encuentre la serie.



Gráfica 2. *Distribución de los retornos pertenecientes a cada estado, junto con el gráfico de la Gaussiana asociada. La línea roja (resp. verde) representa la función de densidad asociada al estado bajista (resp. alcista). Las formas sombredas respresentan la distribución de los retornos clasificados como bajistas (resp. bajistas). Vemos cómo las funciones de densidad y las distribuciones de los retornos son bastante semejantes.*

4. Redes Neuronales Artificiales

Introducción

En este capítulo estudiaremos otro de los métodos de Machine Learning más utilizados: redes neuronales artificiales (ANN por sus siglas en inglés).

Comentaremos brevemente cuál es la base biológica que incorporan, y a continuación explicaremos los elementos básicos que emplean estas implementaciones.

Luego nos meteremos de lleno a analizar el principal algoritmo que se utiliza a la hora de entrenar redes neuronales artificiales para que sean capaces de realizar tareas concretas de manera efectiva: el algoritmo de *Back-Propagation*. Destacaremos el principal problema que implica su implementación: la dificultad de su convergencia; y daremos dos variantes de este algoritmo que tienen como objetivo minimizar este efecto: el Método de α Adaptable y el Algoritmo de *Resilient Back-Propagation*.

Base Biológica

El cerebro humano es un complicado mecanismo capaz de resolver problemas de gran complejidad. Estamos aún lejos de entender completamente cómo funciona, mucho menos de poder replicar uno artificialmente; sin embargo, sí que tenemos un buen conocimiento para explicar las operaciones básicas que realiza un cerebro.

Para entender cuál es el fundamento de una Red Neuronal Artificial (ANN), primero tenemos que tener un conocimiento básico de los mecanismos internos del cerebro. El cerebro es la parte central del sistema nervioso y está formado por una extensa red neuronal, consistente en aproximadamente 10^{11} neuronas interconectadas.

El centro de cada neurona se llama núcleo. El núcleo está conectado a otros núcleos mediante las dendritas y los axones. Esta conexión es la denominada conexión sináptica.

Las neuronas pueden disparar pulsos eléctricos a través de las conexiones sinápticas, los cuales son recibidos a través de las dendritas de otras neuronas. Cuando una neurona recibe suficiente impulso eléctrico, envía asimismo un pulso eléctrico a través de sus axones, y así sucesivamente. Este proceso permite propagar información a través de la red neuronal.

La cuestión es que esta conexión sináptica cambia durante la vida de las neuronas, fortaleciéndose o debilitándose dependiendo del uso. Esta plasticidad de la sinapsis es lo que permite al cerebro aprender, y es el principio que se aplica en las ANN's para poder entrenar una red neuronal artificial para tareas específicas a través de grandes muestras de ejemplos.

Definición de Red Neuronal Artificial (ANN)

Una neurona artificial puede ser implementada de muchas maneras. Comenzamos con la red neuronal de una capa, es decir, la que tiene entre el input y el output únicamente una capa de neuronas 'intermediarias' que se denomina capa oculta. La definición formal que relaciona en una ANN el input y el output es[6]:

$$y(x) = g\Big(\sum_{i=0}^{n} w_i x_i - \theta_i\Big)$$

donde $x = (x_1, \ldots, x_n)$ son los inputs e $y(x)$ es el output. Los parámetros $w = w_0, \ldots, w_n$ son los pesos, y determinan cómo deben ser de intensificados los inputs. θ_i es el umbral o sesgo del input i.

La función g es una función de activación que balancea cómo de potente debe ser el output, si existe, de la neurona tomando como argumento la suma de los input. En el caso de neuronas reales, g es una función δ de Kronecker que determina una relación binaria de todo o nada para el output. Sin embargo, no se suele implementar de esa manera una neurona artificial, ya que como veremos luego, los algoritmos para entrenarlas suelen venir determinados por funciones diferenciables.

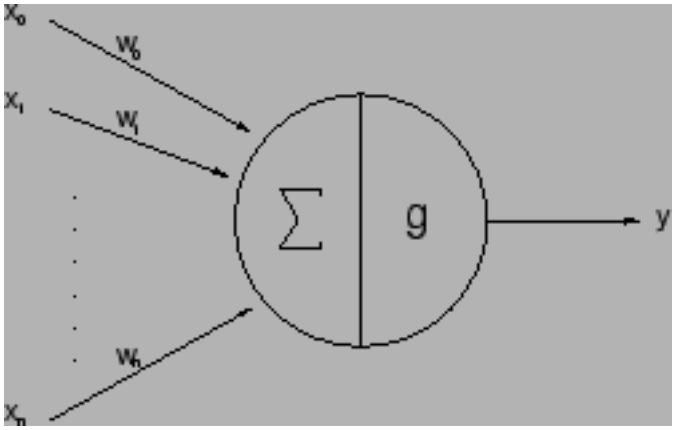


Figure 2: *Esquema de una ANN de una sola capa oculta.*

Entre los tipos de funciones de activación más habituales están:

$$f_1(x) = \alpha x, \alpha \in \mathbb{R}$$

$$f_2(x) = \begin{cases} S, & \text{si } x > S; \\ x, & \text{si } |x| \leq S; \\ -S, & \text{en cualquier otro caso.} \end{cases}$$

Donde S es el output saturado (valor máximo que puede tener el output).

Por último están las funciones sigmoidales, que son de las más utilizadas, particularmente la logística (y que será la que nosotros usaremos por defecto).

$$f_3(x) = \frac{m}{1 + e^{-x}}$$

con m el coeficiente de magnitud. Esta función es diferenciable y con un dominio $D(f_3(x)) = (0, m)$.

La función sigmoidal es una función real de variable real diferenciable. Si $y(x) = \frac{1}{1+e^{-x}}$,

$$\frac{dy(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = y(x)(1 - y(x))$$

por lo que $y(x)$ tiene una derivada simple. En adelante, la función de activación que utilizaremos será la sigmoidal logística para $m = 1$.

Backpropagation

Entre las ANN's existentes, una de las más empleadas por su alta eficiencia es el Perceptrón Multicapa (MLP). Estudiaremos el MLP en su implementación habitual, mediante el uso del algoritmo del *Back-Propagation*. Nuestra versión de este algoritmo es la descrita por Jian-Rong Chen[6] . A diferencia del Perceptrón de una única capa, el MLP puede implementar gran variedad de funciones complejas, incluída la función XOR, que no puede ser realizada por el de una única capa como demostraron Minsky y Papert[7].

En un MLP una unidad solo puede conectarse a la capa adyacente siguiente, no permitiéndose conexiones recurrentes ni en la misma capa. Sea K el número de capas, M el número de inputs y N el número de outputs. El input en una unidad (siempre que no sea en la capa de entrada) es la suma de los outputs de unidades conectadas en la capa anterior. Sea x_i^j el input de la unidad i en la capa j, w_{ij}^k el peso de la conexión entre la unidad i en la capa k y la unidad j de la capa $k+1$, y_i^j el output de la unidad i en la capa j y, por último, θ_i^j el umbral (o sesgo) de la unidad i en la capa j. Entonces tenemos que los input de la capa $k+1$ son:

$$x_j^{k+1} = \sum_i w_{ij}^k y_i^k$$

donde

$$y_i^j = f(x_i^j) \tag{1}$$

siendo f la función de activación. Nosotros utilizaremos como f la función sigmoide

$$f(x_i^j) = \frac{1}{1 + e^{-\frac{x_i^j - \theta_i^j}{T}}}.$$

Dependiendo de las aplicaciones un output que tome valores negativos es necesario, y podemos utilizar entonces la función

$$f(x_i^j) = \frac{1 - e^{-\frac{x_i^j - \theta_i^j}{T}}}{1 + e^{-\frac{x_i^j - \theta_i^j}{T}}}.$$

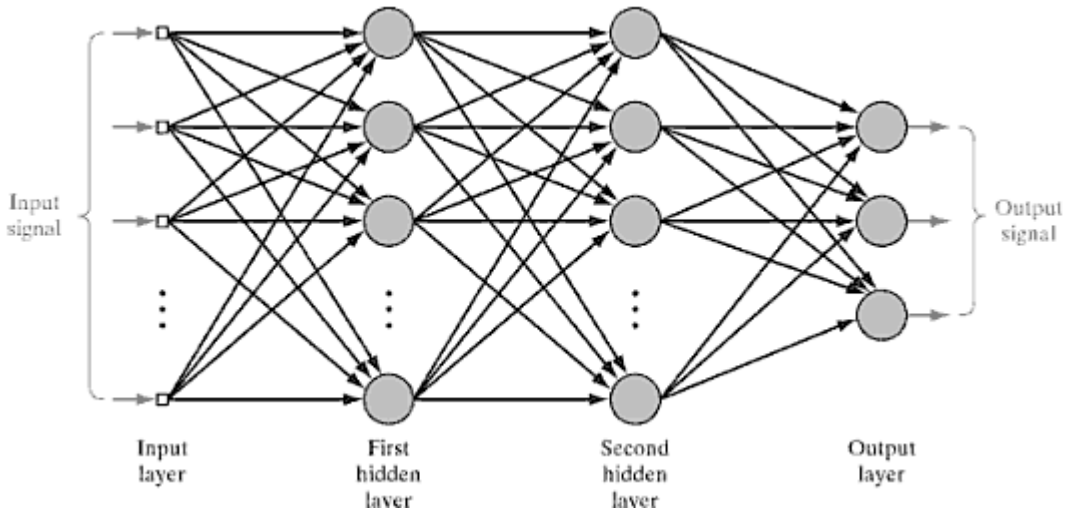


Figure 3: *Esquema de un Perceptrón Multicapa.*

Hay dos fases en el algoritmo de *Back-Propagation*, la primera es computar el cálculo a través de las capas utilizando (1). La segunda fase es actualizar los pesos, operación que se realiza computando el error entre el valor esperado y el valor real calculado en la primera fase. Este proceso clasifica el algoritmo de *Back-Propagation* dentro de la categoría de los algoritmos de aprendizaje supervisado. Básicamente el algoritmo de *Back-Propagation* es un algoritmo de gradiente descendente.

A continuación explicaremos cómo se realiza la actualización de los pesos, una vez obtenido el output a través de (1). Primero definiremos una función de error $\mathcal{E}$ que nos dará cuenta de la discrepancia entre el valor calculado y el real. Sea N el número de outputs, d_i el output deseado y y_i el obtenido, con $i \in \{1, \ldots, N\}$. Entonces

$$\mathcal{E} = \frac{1}{2} \sum_j (d_j - y_j)^2, \tag{2}$$

El error total será simplemente $\mathcal{E}_{total} = \sum_{k=1}^{N} \mathcal{E}_k$. Nuestro objetivo será minimizar este error total, para ello utilizamos un algoritmo de gradiente descendente. Este tipo de algoritmo busca un mínimo en la función (la función error en nuestro caso) dando pasos proporcionales al negativo del gradiente. Es por ello que es tan importante que la función que usamos como función de activación sea derivable.

Así, el ajuste de los pesos es proporcional a la derivada

$$\Delta w_{ij}^k = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^k} \tag{3}$$

donde η es el tamaño del paso, importante para asegurar la convergencia. También se puede añadir un término de inercia, mediante una dependencia de recursividad, que nos ayudará a mejorar la convergencia evitando rápidos cambios en Δw_{ij}^k:

$$\Delta w_{ij}^k(t) = -\eta \frac{\partial \mathcal{E}}{\partial w_{ij}^k(t)} + \alpha \Delta w_{ij}^k(t-1), \tag{4}$$

donde α es un positivo real pequeño.

Estrictamente hablando, la $\mathcal{E}$ que se utiliza en (3) debería ser $\mathcal{E}_{total}$. Sin embargo es mucho más práctico actualizar pesos con cada input de una muestra de entrenamiento, en vez de usar toda la muestra al completo. Estos dos casos se denominan batch y online, respectivamente. En este caso puede utilizarse el $\mathcal{E}$ definido en (2).

Así, para cada capa de un perceptrón simple, tendríamos el output dado por

$$y_j = f(\sum_{i=1}^{n-1} w_i I_i + \theta_j) = f(s).$$

El sesgo θ_j puede añadirse como input siempre activo ($I_n = 1$) con peso $w_n = \theta_j$

$$y_j = f(\sum_{i=1}^{n} w_i I_i) = f(s).$$

Si el ouput deseado es d_j, entonces de (2) tenemos

$$\frac{\partial \mathcal{E}}{\partial w_i} = \frac{\partial \mathcal{E}}{\partial y_j} \frac{\partial y_j}{\partial w_i} = -(d_j - y_j) f'(s) I_i \ .$$

Sea

$$\delta_j = (d_j - y_j).$$

Entonces el ajuste de pesos viene dado por

$$\Delta w_i = -\eta \frac{\partial \mathcal{E}}{\partial w_i} = \eta \delta_j f'(s) I_i \ .$$

Este esquema de ajuste de pesos es denominado la regla Delta. El algoritmo de Back-Propagation en realidad es una generalización de la regla Delta.

Sea una red MLP, con N capas y suponemos que nuestra función de activación f es la sigmoide logística, entonces

$$\frac{\partial \mathcal{E}}{\partial y_j^N} = -(d_j - y_j^N),$$

donde el superíndice N indica que el output es de la capa N y de (1) obtenemos

$$\frac{\partial y_j^N}{\partial x_j^N} = f'(x_j^N) = y_j^N(1 - y_j^N).$$

Como $x_j^{k+1} = \sum_i w_{ij}^k y_i^k$, entonces

$$\frac{\partial x_j^N}{\partial w_{ij}^{N-1}} = y_i^{N-1}.$$

Y por lo tanto, podemos escribir la parcial del error respecto de los pesos como

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{ij}^{N-1}} &= \frac{\partial \mathcal{E}}{\partial y_j^N}\frac{\partial y_j^N}{\partial x_j^N}\frac{\partial x_j^N}{\partial w_{ij}^N} \\ &= -(d_j - y_j^N)y_j^N(1 - y_j^N)y_i^{N-1} \ .\end{aligned}$$

Sea

$$\delta_j^{N-1} = (d_j - y_j^N)y_j^N(1 - y_j^N). \tag{5}$$

Luego

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^{N-1}} = -\delta_j^N y_i^{N-1}. \tag{6}$$

Entonces, teniendo en cuenta (4) y (5) tenemos que en la iteración $t+1$ los pesos serán

$$w_{ij}^{N-1}(t+1) = w_{ij}^{N-1}(t) + \eta\delta_j^{N-1}y_i^{N-1} + \alpha[w_{ij}^{N-1}(t) - w_{ij}^{N-1}(t-1)]. \tag{7}$$

Obviamente, esta es la actualización de los pesos para la capa del output. Pero como la delta la podemos definir así

$$\delta_j^{N-1} = -\frac{\partial \mathcal{E}}{\partial y_j^N}\frac{y_j^N}{\partial x_j^N} = -\frac{\partial \mathcal{E}}{\partial x_j^N}.$$

existe una fórmula general para $w_{ij}^k, \forall k \in \{1, \ldots, N-1\}$ de la siguiente manera

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_{ij}^k} &= \frac{\partial \mathcal{E}}{\partial y_j^{k+1}} \frac{\partial y_j^{k+1}}{\partial x_j^{k+1}} \frac{\partial x_j^{k+1}}{\partial w_{ij}^k} \\ &= \frac{\partial \mathcal{E}}{\partial y_j^{k+1}} y_j^{k+1}(1 - y_j^{k+1}) y_i^k \\ &= y_j^{k+1}(1 - y_j^{k+1}) y_i^k \sum_l \frac{\partial \mathcal{E}}{\partial x_l^{k+2}} \frac{\partial x_l^{k+2}}{\partial y_j^{k+1}} \\ &= y_i^k y_j^{k+1}(1 - y_j^{k+1}) \sum_l \frac{\partial \mathcal{E}}{\partial x_l^{k+2}} w_{jl}^{k+1} = -y_i^k y_j^{k+1}(1 - y_j^{k+1}) \sum_l w_{jl}^{k+1} \delta_l^{k+1}.\end{aligned}$$

Sea

$$\delta_j^k = y_j^{k+1}(1 - y_j^{k+1}) \sum_l w_{jl}^{k+1} \delta_l^{k+1}. \tag{8}$$

Entonces tenemos que

$$\frac{\partial \mathcal{E}}{\partial w_{ij}^k} = -\delta_i^k y_i^k.$$

Esta ecuación es similar a (6). Finalmente obtenemos el algoritmo de Back-Propagation combinando (7) y (8), llamado así porque las δ's se propagan hacia atrás, comenzando por (5).

En general, la manera de ejecutar este algoritmo es:

1. Asignar valores pequeños y aleatorios a los pesos y sesgos del MLP.
2. Dependiendo de si el algoritmo se ha programado según el método batch u online, se le pasa un input de la muestra de entrenamiento, o la muestra entera y determinar las δ's.
3. Actualizar pesos según el algoritmo de Back-propagation.
4. Iterar hasta alcanzar la tolerancia de error deseado.

El problema del algoritmo de Back-propagation, como ocurre en todos los que poseen ciertas características geométricas, es que resulta lento en su convergencia, ya que η depende mucho de la topología local y curvatura de las superfícies del error $\mathcal{E}$. Así, si introducimos una η pequeña el algoritmo convergerá de forma extremadamente lenta, mientras que si η es muy grande, provocaremos que el algoritmo vaya saltando de un lado a otro de los valles de $\mathcal{E}$, en vez de seguir la curva que marca el camino a su mínimo.

Por esta razón, daremos dos variantes del algoritmo de Back-propagation que se usan frecuentemente para acelerar su convergencia, la *Técnica de α Adaptable* y el *Algoritmo de Resilient Back-propagation* de Martin Riedmiller[9].

Técnica de α Adaptable

En esta variante del algoritmo de Back-propagation se añade un término de inercia, siguiendo el modelo de (4). En esta técnica introducimos una dependencia de la α (tamaño de salto en el gradiente) que aparece en (4) respecto al número de iteración siguiendo este modelo:

$$\begin{aligned}
&\alpha(t) = \alpha(t-1)(1 - h(t)\sqrt{\mathcal{E}(t)}), \quad t \geq 2,\\
&h(t) = a_1 h(t-1) + a_2 \Delta\mathcal{E}(t) \quad t \geq 2,\\
&\Delta\mathcal{E}(t) = \mathcal{E}(t) - \mathcal{E}(t-1) \quad t \geq 2,
\end{aligned}$$

donde $\alpha(t)$ es el tamaño de salto en el momento t. $\mathcal{E}(t)$ es la suma cuadrática de errores entre el output deseado y el obtenido en la iteración t

$$\mathcal{E} = \frac{1}{2}\sum_{k=1}^{N}\sum_{i=1}^{P}(d_i^k - o_i^k)^2,$$

con N el número de capas del MLP y P el número de outputs. Así, $\Delta\mathcal{E}(t), t \geq 2$ sería el decremento de $\mathcal{E}(t)$, y $h(t)$ será un filtro recursivo de $\Delta\mathcal{E}(t)$: un filtro recursivo reutiliza uno o más de sus outputs como inputs, y permite controlar de esta manera que haya cambios bruscos en $\Delta\mathcal{E}(t)$, creando un versión del algoritmo más estable.

Los parámetros a_1 y a_2 del filtro recursivo $h(t)$ son los encargados de controlar esta adaptación. Para una gran muestra de entrenamiento, un valor pequeño de a_1 y grande en a_2, combinado con una η de gran tamaño también es lo que más favorece la convergencia[8].

Un análisis del algoritmo nos permite ver que si $h(t)$ es positivo, entonces la tendencia de $\mathcal{E}(t)$ en el pasado inmediato es incrementar, por lo que $1 - h(t)\mathcal{E}(t) < 1$, y por lo tanto α decrecerá en este paso. De forma similar concluimos que si la tendencia de $\mathcal{E}(t)$ es decrecer, entonces α crecerá. Y finalmente, si $\mathcal{E}(t)$ es muy pequeño, significará que el MLP ya casi ha terminado de aprender, y la adaptación de muy baja magnitud, estabilizando el algoritmo.

Resilient Back-propagation

Este algoritmo, desarrollado por Martin Riedmiller[9] para MLP con método de aprendizaje batch, tiene como objetivo principal eliminar la influencia del tamaño de la derivada parcial en la actualización de los pesos. Por ello, se considera únicamente el signo de la derivada para indicar la dirección en que tiene que ir esta actualización. Consideramos la conexión de la unidad $i,\ i \in \{1,\ldots,I\}$ de una capa con la unidad $j, j \in \{1,\ldots,J\}$ de la siguiente capa en la iteración t:

$$\Delta w_{ij}(t) := \begin{cases} -\Delta_{ij}(t), & \text{si } \frac{\partial\mathcal{E}(t)}{\partial w_{ij}(t)} > 0,\\ +\Delta_{ij}(t), & \text{si } \frac{\partial\mathcal{E}(t)}{\partial w_{ij}(t)} < 0,\\ 0, & \text{si } \frac{\partial\mathcal{E}(t)}{\partial w_{ij}(t)} = 0, \end{cases}$$

donde $\frac{\partial \mathcal{E}(t)}{\partial w_{ij}(t)}$ es la suma para cada componente de la muestra (batch learning).

Si reemplazamos $\Delta_{ij}(t)$ por una constante se obtiene la regla de actualización de pesos de Manhattan.

Para determinar los valores de $\Delta_{ij}(t)$, utilizamos también un filtro de recursivo de la forma

$$\Delta_{ij}(t) := \begin{cases} \eta^+ \Delta_{ij}(t-1), & \text{si } \frac{\partial \mathcal{E}(t-1)}{\partial w_{ij}} \frac{\partial \mathcal{E}(t)}{\partial w_{ij}} > 0, \\ \eta^- \Delta_{ij}(t-1), & \text{si } \frac{\partial \mathcal{E}(t-1)}{\partial w_{ij}} \frac{\partial \mathcal{E}(t)}{\partial w_{ij}} < 0, \\ \Delta_{ij}(t-1), & \text{si } \frac{\partial \mathcal{E}(t-1)}{\partial w_{ij}} \frac{\partial \mathcal{E}(t)}{\partial w_{ij}} = 0, \end{cases}$$

$$\text{donde } 0 < \eta^- < 1 < \eta^+.$$

Como ya comentamos antes, puede ocurrir que la actualización de pesos haya hecho que nos saltemos un mínimo, lo cual implica que el signo de la derivada parcial del correspondiente w_{ij} cambie, así que disminuimos el valor de actualización $\Delta_{ij}(t)$ un factor η^-. En el caso contrario, para acelerar la convergencia, este valor de actualización es incrementado en $\eta^+(> 1)$. En caso de que hayamos alcanzado un mínimo, no hay ninguna actualización.

En general, para evitar tener un gran número de parámetros libres, se escogen valores constantes de η^+ y η^-. Dos valores que, de acuerdo a las simulaciones realizadas por Riedmiller, parecen funcionar bien independientemente de la muestra, son $\eta^+ = 1.2$ y $\eta^- = 0.5$.

Hay tres parámetros a considerar: Δ_0, valor inicial de actualización de pesos, Δ_{max}, que es el límite del tamaño del salto y Δ_{min}, mínimo peso de actualización.

Cuando inicializamos el algoritmo, todos los valores de actualización de pesos son Δ_0, que debe ser escogido teniendo en cuenta también los valores iniciales de los pesos. Si por ejemplo estos pesos están en el rango $[-1, 1]$,un buen valor será $\Delta_0 = 0.1$.

El valor de Δ_{max} no es de gran importancia en la mayoría de casos, pero conviene fijar uno para evitar posibles problemas de rebote alrededor de un mínimo. Por último, Δ_{min} sí que es un valor de importancia, ya que fijar un valor adecuado del mismo evitará que el algoritmo se quede atascado si se encuentra con algún mínimo local subóptimo. En su estudio Riedmiller fija un valor de $\Delta_{min} = e^{-6}$.

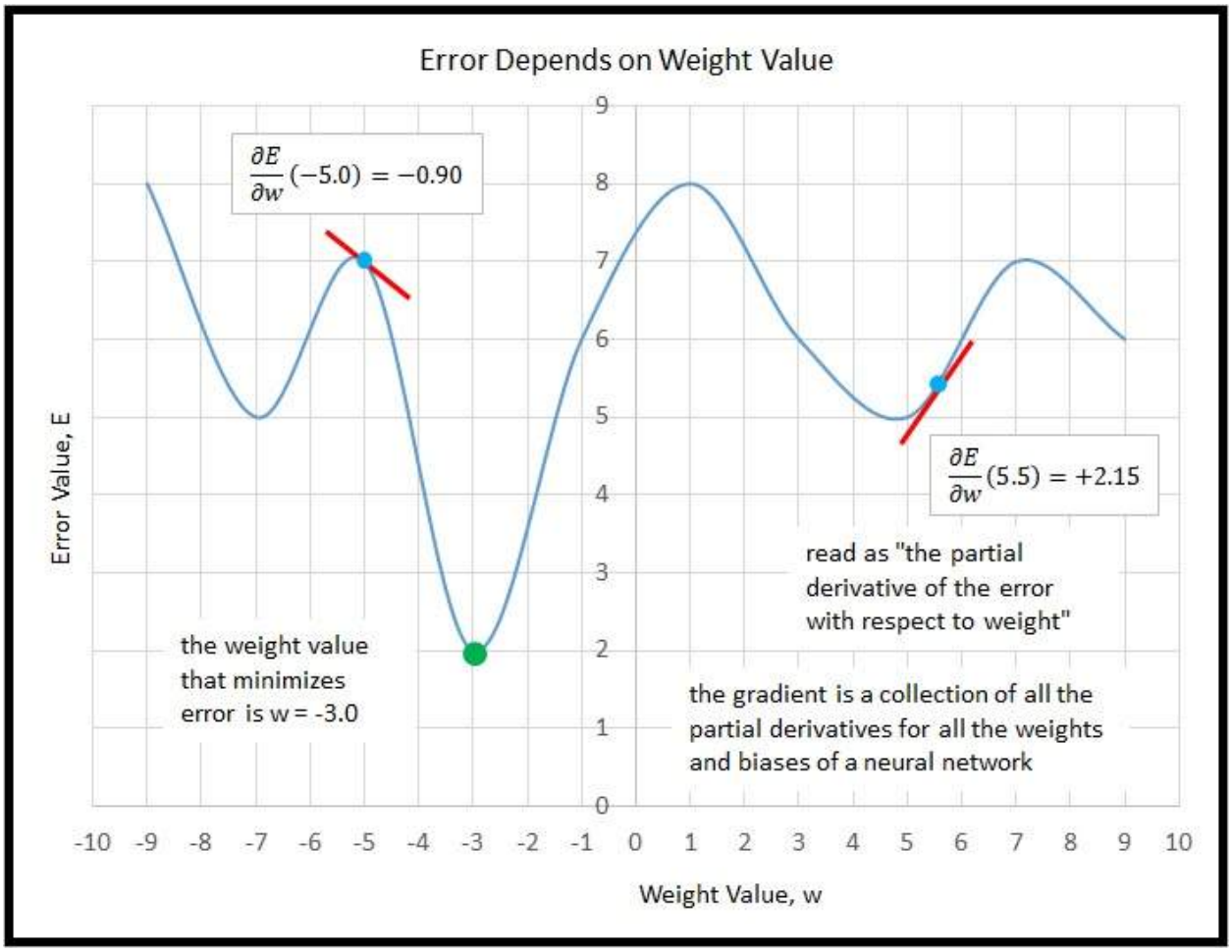


Figure 4: *Esquema de cómo funciona el algoritmo de Resilient Back-Propagation, buscando el mínimo local de la función de error.*

En pseudo-código, el algoritmo de Resilient Back-propagation sería

$$\forall i,j : \Delta_{ij}(t) = \Delta_0$$
$$\forall i,j : \frac{\partial \mathcal{E}}{\partial w_{ij}(t-1)} = 0$$

---

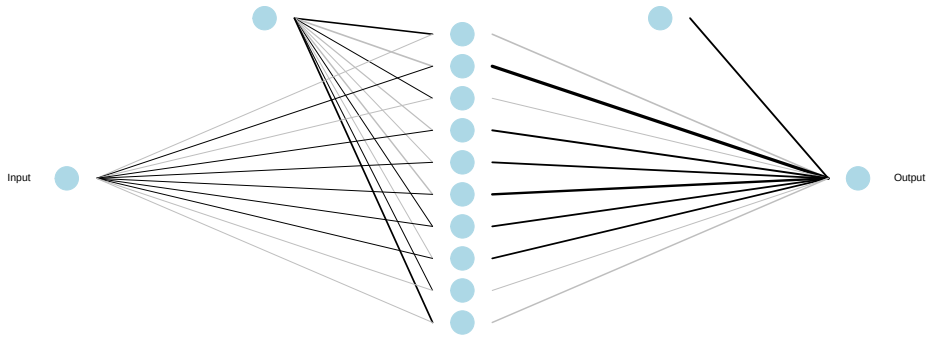
repetir
  computar gradiente $\frac{\partial \mathcal{E}}{\partial w}(t)$
  para todos los pesos y sesgos **hacer**
    si $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1)\frac{\partial \mathcal{E}}{\partial w_{ij}}(t) > 0$ **entonces**
      $\Delta_{ij}(t) = \mathbf{min}(\Delta_{ij}(t-1)\eta^+, \Delta_{max})$
      $\Delta w_{ij}(t) = -\ \mathbf{sign}\left(\frac{\partial \mathcal{E}}{\partial w_{ij}}(t)\right)\Delta_{ij}(t)$
      $w_{ij}(t+1) = w_{ij}(t) + \Delta_{ij}(t)$
      $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1) = \frac{\partial \mathcal{E}}{\partial w_{ij}}(t)$
    si no, **si** $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1)\frac{\partial \mathcal{E}}{\partial w_{ij}}(t) < 0$ **entonces**
      $\Delta_{ij}(t) = \mathbf{maximum}(\Delta_{ij}(t-1)\eta^-, \Delta_{min})$
      $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1) = 0$
    si no, **si** $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1)\frac{\partial \mathcal{E}}{\partial w_{ij}}(t) = 0$ **entonces**
      $\Delta w_{ij}(t) = -\ \mathbf{sign}\left(\frac{\partial \mathcal{E}}{\partial w_{ij}}(t)\right)\Delta_{ij}(t)$
      $w_{ij}(t+1) = w_{ij}(t) + \Delta_{ij}(t)$
      $\frac{\partial \mathcal{E}}{\partial w_{ij}}(t-1) = \frac{\partial \mathcal{E}}{\partial w_{ij}}(t)$
    fin si
  fin para
hasta que converja

Ejemplo Red Neuronal Artificial

Veremos a continuación un par de aplicaciones de una ANN, usando el algoritmo de *Resilient Backpropagation*.

En el primer ejemplo entrenaremos una red neuronal para que sea capaz de calcular el cuadrado de un número.

Usaremos una muestra aleatoria de 50 números entre el 0 y el 10 como input, y el cuadrado de estos números como output de una ANN con 10 neuronas ocultas y una tolerancia del error $\mathcal{E} = 0.01$.



En el gráfico se muestra un esquema de esta ANN, cuanto mayor es el peso de la conexión más oscuro es el color de la representación de esta conexión.

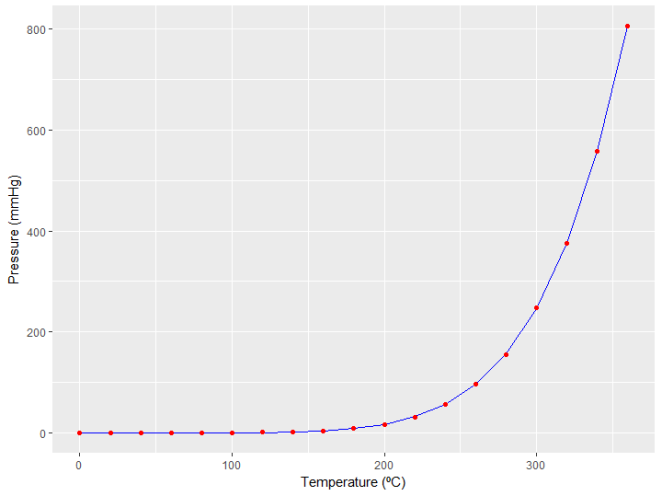
Veamos la precisión de esta ANN calculando el cuadrado de los naturales del 1 al 10.

Input	Output	Error
1.00	0.62	0.38
2.00	4.06	0.06
3.00	9.00	0.00
4.00	15.96	0.04
5.00	25.03	0.03
6.00	35.99	0.01
7.00	48.98	0.02
8.00	64.03	0.03
9.00	80.98	0.02
10.00	99.80	0.20

Table 2: Resultados ANN

El otro ejemplo que veremos será una ANN entrenada para deducir la relación entre la temperatura (ºC) y la presión de vapor de mercurio en milímetros (de mercurio), a partir de 19 observaciones.

Temperature (ºC)	Pressure (mmHg)
0	0.0002
20	0.0012
40	0.0060
60	0.0300
80	0.0900
100	0.2700
120	0.7500
140	1.8500
160	4.2000
180	8.8000
200	17.3000
220	32.1000
240	57.0000
260	96.0000
280	157.0000
300	247.0000
320	376.0000
340	558.0000
360	806.0000



En azul la curva original, y en rojo los puntos determinados por la ANN

5. Modelo Mixto HMM-ANN

La característica de los HMM es su capacidad de discernir los diferentes estados aún cambiando el paradigma del modelo, es decir, es capaz de adaptarse a lo largo del tiempo. Sin embargo, los HMM están limitados en el número de diferentes estados que son capaces de distinguir, ya que si no están bien diferenciados, sus distribuciones de probabilidad acaban solapándose. En este sentido las ANN resultan mucho más eficaces, con una muestra de entrenamiento lo suficientemente grande estos mecanismos son capaces de distinguir una gran cantidad de outputs diferentes. Sin embargo, las ANN no disponen de esa habilidad de adaptación temporal.

Uno de los ejemplos más interesantes hoy en día es el reconocimiento de voz. Podemos decir que los HMM son mejores a la hora de seguir un discurso y las ANN en lo que sería la distinción de las diferentes sílabas que lo componen.

Por esta razón han surgido estudios que tratan de crear modelos mixtos para explotar las ventajas de ambos métodos. En este capítulo explicamos cómo podría ser un modelo de este tipo.

Redes Neuronales con Plasticidad Temporal

Un fenómeno que ocurre en el cerebro es la plasticidad temporal (STDP), donde la conexión de la sinapsis modula la probabilidad de que un evento pre-sináptico (acción sobre las dendritas) cause un evento post-sináptico (acción sobre los axones). En otras palabra, la conexión sináptica influye a la hora de que un input sea o no procesado. De la misma forma en una ANN los pesos pueden ser modificados a partir de reglas que incorporen información de la sinapsis mediantes reglas STDP.

Básicamente, una regla STDP dicta que si un evento presináptico se lanza justo antes que uno postsináptico, entonces el peso de la sinapsis es reforzado. Si es al revés, esta conexión se debilita. En Nessler et al.[10] desarrollaron una versión de STDP para computar el algoritmo de Maximización de la Esperanza en un circuito neuronal.

En este modelo, se usa una red neuronal de N inputs de una sola capa (totalmente conectada) con K unidades en la capa oculta, y se utiliza de función de activación exponencial. El output que genera entonces es $y_k(t) = e^{g_k(t)}$, con

$$g_k(t) = w_{k0} + \sum_{i=1}^{N} \delta_i^k(t) w_{ki}(t).$$

Si nos fijamos, respecto a la definición habitual, lo que cambia es el término de la δ_i^k, que depende de si el input estaba activo o no.

$$\delta_i^k(t) = \begin{cases} 1 & \text{si el input } i \text{ se activó en el intervalo } [t-\sigma, t] \\ 0 & \text{en caso contrario} \end{cases}$$

El valor de σ dependerá del caso.

Planteamiento Modelo HMM-ANN

La idea de este modelo mixto, que proviene de un paper de Amirhossein Tavenaei y Anthony S. Maida[10] es utilizar ANN's entrenadas específicamente para cada uno de los estados del HMM, que será de tipo Gaussiano (GHMM), de manera que la verosimilitud de cada estado para las observaciones vendrá determinado por los pesos de las redes neuronales[11].

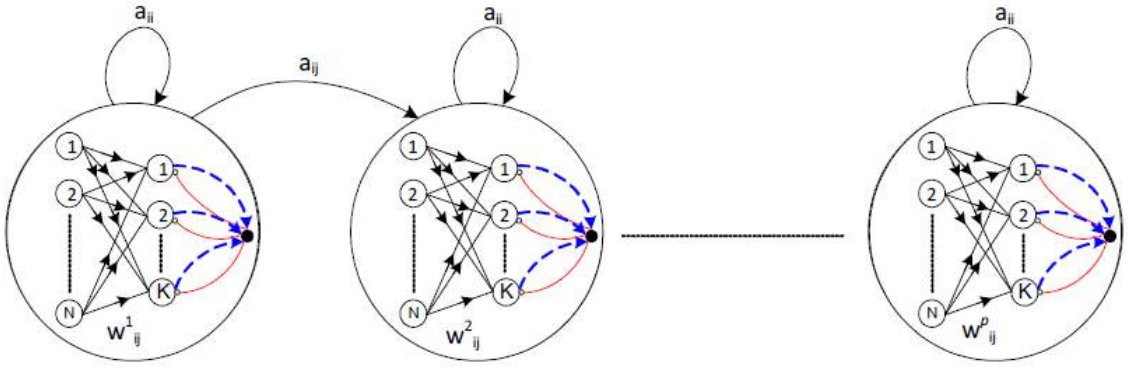


Figure 5: *Vemos cómo cada uno de los estados del GHMM viene representado por una red neuronal, específicamente entrenada para reconocer inputs de este estado. Esto nos permite entrenar de manera independiente cada ANN.*

Para entrenar independientemente cada ANN, al introducir un input perteneciente a un estado k, lo que hacemos es estimular en mayor grado la ANN asociada a este estado k, mientras que el resto apenas se estimulan.

Para entrenar las ANN's usaremos una versión de las reglas STDP vistas anteriormente. Pero primero recordamos que sobre una muestra y de dimensión N, definimos la función de densidad de una Gaussiana como:

$$N(x, \mu_i, \Sigma_i) = \frac{1}{\left(2\pi\right)^{N/2} |\Sigma_i|^{1/2}} e^{-0.5(x-\mu_i)\Sigma_i^{-1}(x-\mu_i)^t},$$

Por lo tanto, en un modelo mixto tendríamos K mezclas $z_1, z_2, \ldots, z_K$ con $\sum z_k = 1$ y

$$P(y|z_k = 1) = N(y|\mu_k, \Sigma_k),$$

$$P(y|z) = \prod_{k=1}^{K} N(y|\mu_k, \Sigma_k)_k^z,$$

y la probabilidad de $P(y)$ será

$$P(y) = \sum_z P(z)P(y|z) = \sum_{k=1}^{K} \pi_k N(y|\mu_k, \Sigma_k) \text{ donde } \sum_k \pi_k = 1, 0 \leq \pi_k \leq 1.$$

Sea una muestra y_r de y, definimos ahora $R(z_{kr})$ como

$$P(z_k = 1|y) = R(z_{kr}) = \frac{\pi_k N(y_r|\mu_k, \Sigma_k)}{\sum\limits_{j=1}^{K} \pi_j N(y_r|\mu_j, \Sigma_j)}.$$

Para simplificar, supongamos que las muestras son independientes ($\Sigma = I$).

$$R(z_{kr}) = \frac{\pi_k e^{-0.5(x-\mu_k)(x-\mu_k)^t}}{\sum\limits_{j=1}^{K} \pi_j e^{-0.5(x-\mu_j)(x-\mu_j)^t}}. \tag{1}$$

Tendremos que $R(z_{kr})$ alcanza el máximo cuando $y_r = \mu_k$. Como hemos comentado antes, lo que hacemos es sustituir las distribuciones de los estados por Redes Neuronales, así que suponemos que para la neurona k, su peso $w_{ki}, i \in \{1, \ldots, N\}$ aproxima μ_k y, por lo tanto, podemos sustituir el componente $-0.5(x - \mu_k)(x - \mu_k)^t$ por el producto escalar entre w e y, $w^t y$ de manera que, con esta reformulación, obtenemos

$$R(z_{kr}) = C\frac{\pi_k e^{w_k^t y_r}}{\sum\limits_{j=1}^{K} \pi_j e^{w_j^t y_r}}, \tag{2}$$

con C una constante de normalización. En este punto usamos las reglas del Algoritmo de Baum Welch para recalcular la μ_k, o la w_k bajo la suposición con la que trabajamos, y también las π_k

$$\mu_k = w_k = \frac{\sum\limits_{s=1}^{M} R(z_{ks} y_s)}{\sum_{s=1}^{M} R(z_{ks})}, \tag{3}$$

$$\pi_k = \frac{\sum_{s=1}^{M} R(z_{ks})}{M} \tag{4}$$

donde M es el número de muestras de entrenamiento.

En esta ANN, k es la neurona de output que acaba de disparar y la cual se supone corresponde al μ_k del GHMM, y se actualiza con cada nueva muestra. Para calcular el w_k final, lo que

haremos será, al valor μ_k que obtengamos, añadirle un término proveniente de aplicar una adaptación de la regla STDP de Nessler, teniendo en cuenta si ha habido señal presináptica en el input i en el intervalo $[t-\sigma, t]$ de la siguiente manera:

$$\Delta w_{ki} = \begin{cases} e^{-w_{ki}+1} - 1 & \text{si } \delta_i^k(t) = 1 \\ -1 & \text{si } \delta_i^k(t) = 0 \end{cases}$$

con $i \in \{1, \ldots, N\}$. En el primer caso se refuerza la conexión, mientras que en el segundo se debilita.

Otro parámetro del GHMM es el coeficiente de mezcla π_k obtenido en (4). El peso del sesgo w_{k0}, que asemejaría π_k y representaría la media de veces que se activa la neurona k, también está sujeto a las reglas STDP de la siguiente manera:

$$\Delta w_{k0} = z_k e^{-w_{k0}+1} - 1.$$

Y tendríamos $w_{k0}^{new} = w_{k0} + \Delta w_{k0}$. Para el resto de w_{ki} la regla varía un poco, añadiéndose un término η_k proporcional al número de veces que ha disparado la neurona k. Y tenemos

$$w_{ki}^{new} = w_{ki} + \eta_k \Delta w_{ki}.$$

Redefiniendo las π_k como hemos visto en función del peso del sesgo tendríamos

$$\pi_k^{ann} = \frac{e^{w_{k0}}}{\sum\limits_j e^{w_{j0}}}.$$

Introducimos este término en (2) para terminar de definir $R(z_{kr})$ en función únicamente de los pesos de la ANN.

$$R(z_{kr}) = C \frac{\frac{e^{w_{k0}}}{\sum\limits_j e^{w_{j0}}} e^{w_k^t y_r}}{\sum\limits_{l=1}^{K} \frac{e^{w_{l0}}}{\sum\limits_j e^{w_{j0}}} e^{w_l^t y_r}},$$

$$P_(z \text{ dispara}|y_r) = R(z_{kr}) = C \frac{e^{w_{j0}+w_k^t y_r}}{\sum\limits_{l=1}^{K} e^{w_{l0}+w_l^t y_r}}.$$

Finalmente, extraemos la probabilidad de que el input sea del tipo que la ANN ha reconocido utilizando $g_k(t)$, que hemos visto en el apartado anterior, definida como

$$g_k(t) = w_{k0} + \sum_{i=1}^{N} \delta_i^k(t) w_{ki}(t).$$

Así, la probabilidad en el tiempo t según este modelo sería

$$P_{snn}(t) = \underset{k \in K}{max} \frac{e^{g_k(t)}}{Z},$$

donde Z es una constante de normalización.

Conclusiones

En este trabajo hemos estudiado las Cadenas de Markov/Modelos Ocultos de Markov y las Redes Neuronales Artificiales desde un punto de vista teórico, sobre todo en el primer caso, para entender mejor la base que sustenta ambas herramientas.

Hemos visto cómo las MC's/HMM's están bien definidas y de manera única en un espacio de probabilidad estacionario. Se han analizado los diferentes comportamientos que pueden tener los estados y, en el caso de las HMM's, cómo se justifica mediante teoría Bayesiana los principales algoritmos que se emplean.

En el caso de las ANN's se ha comentado el principio biológico en el que se basa su implementación y hemos repasado los principales algoritmos que suelen utilizarse para entrenarlas, que actúan bajo el principio de encontrar mínimos locales en la función de error.

Finalmente, hemos visto que bajo determinadas circunstancias se puede definir una metodología que aúne HMM's y ANN's para aprovechar las principales ventajas de ambos métodos.

Bibliografía

1. Theory and Algorithms for Hidden Markov Models and Generalized Hidden Markov Models. Daniel Ray Upper (1989). Rice University.
2. Essentials of Stochastic Processes. Richard Durrett (2012). Duke University.
3. Hidden Markov Models,Theory and Aplications. Edited by Przemyslaw Dymarski and published by InTech (2011).
4. Regime Shifts: Implications for Dynamic Strategies. Mark Kritzman, Sébastien Page and David Turkington (2012).
5. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. Lawrence R. Rabiner (1989).
6. Theory and Applications of Artificial Neural Networks. Jian-Rong Chen (1991). Durham theses, Durham University. http://etheses.dur.ac.uk/6240/ \

7. Perceptrons: an Introduction to Computational Geometry. Marvin Minsky and Seymour Papert (1969).

8. Scaling Relationships in Back Propagation LearningComplex System vol-2 No.l pp38-44Tesauro, G . and Janssens, B. (1988).

9. Rprop - Description and Implementation Details. Martin Riedmiller (1994). University of Karlsruhe.

10. Bayesian Computation Emerges in Generic Cortical Microcircuits through Spike-Timing-Dependent Plasticity. Bernhard Nessler, Michael Pfeiffer, Lars Buesing and Wolfgang Maass (2013).

11. Training a Hidden Markov Model with a Bayesian Spiking Neural Network. Amirhossein Tavanaei y Anthony S. Maida (2016). University of Louisiana at Lafayette.

Adjunto

Algoritmo de Baum-Welch

Este código es la implementación del algoritmo Baum-Welch (Capítulo 2):

```
#ALGORITMO BAUM-WELCH
BaumWelch = function(returns, mu, sigma,
                     n_states=3, Tolerance=7*10^{-2}, maxstep=1000){

  change_likelihood=c(rep(Inf, n_states))
  likelihood=data.frame()

  returns=as.data.frame(returns)
  mu = as.data.frame(mu)
  A=data.frame(rep(1/n_states,n_states))
  A[1:n_states]=rep(1/n_states,n_states)
  p=rep(1/n_states,n_states)

  # Nuestra Matriz de transción incial
  # A=data.frame(c(0.9,0.1),c(0.1,0.9))


  # Nuestras probabilidades iniciales
  # p = c(0.6, 0.4)

  k=ncol(returns)
  if(k==1){sigma=as.data.frame(sigma)}
  L=nrow(returns)

  B=data.frame(c(rep(0,L)))
  B[1:n_states]=c(rep(0,L))
  forward=B
  backward=B
  smoothed=B
  xi=vector("list", L-1)
  xi[1:L-1]=list(data.frame(c(rep(0,n_states)),c(rep(0,n_states))))
  iteration=1
  while(change_likelihood[1] > Tolerance & change_likelihood[2] >
        Tolerance & iteration<=maxstep){

  #SECCION A
  for(i in 1:n_states){
    if(k!=1){
```

```
  R=returns
  for(j in 1:nrow(returns)){
    R[j,]=returns[j,]-mu[,i]
  }
  #Calculamos los valores de la proyección sobre las Gaussianas de los valores.
      B[,i] = exp(-.5*apply((as.matrix(R)%*%
                               solve(as.matrix(sigma[[i]])))*
                               as.matrix(R), 1,
                               function(x)sum(x)))/((2*pi)^(k/2)*
                               sqrt(abs(det(as.matrix(sigma[[i]]))))))
    }else{
      B[,i] = exp(-.5*((returns-mu[i,])*sigma[i,]^(-1)*
                      (returns-mu[i,])))/(sqrt(2*pi*sigma[i,]))
    }
  }


# SECCION B
forward[1,]=p*B[1,] # Valores Forward
forward[1,] = forward[1,]/sum(forward[1,])

for (t in 2:L){
 aux=c(rep(0,n_states))
 for(i in 1:n_states){
  aux[i] =  aux[i] + sum(forward[t-1,]*A[,i])
 }
 forward[t,]=aux*B[t,]
 forward[t,] = forward[t,]/sum(forward[t,])
}

#SECCION C
backward[L,]=B[L,] #Valores Backward
backward[L,]=backward[L,]/sum(backward[L,])

t=L-1
while (t>=1){
    aux=c(rep(0,n_states))
    for(i in 1:n_states){
      aux[i] =  aux[i] + sum(A[,i]*backward[t+1,])
    }
backward[t,]=aux*B[t+1,]
backward[t,]=backward[t,]/sum(backward[t,])
t=t-1
}
```

```
#SECCION D
for (t in 1:L){ #Probabilidades Smoothed
  smoothed[t,]= forward[t,]*backward[t,]
  smoothed[t,]= smoothed[t,]/sum(smoothed[t,])
}

t=1
while(t<=L-1){
aux = diag(0,n_states,n_states)

for(j in 1:n_states){
  aux[j,]=as.matrix(forward[t,j]*backward[t+1,]*B[t+1,])
}

xi[[t]] = A*aux
xi[[t]]=xi[[t]]/sum(xi[[t]])
t=t+1
}

#SECCION E
p=smoothed[1,]

exp_num_transitions=data.frame(c(rep(0,n_states)),c(rep(0,n_states)))
for(i in 1:n_states){
  for(j in 1:n_states){
   exp_num_transitions[i,j]= #Número total de transiciones
     sum(sapply(lapply(xi,
              function(x)sum(x[i,j])), function(x)(sum(x))))
  }
}

for(i in 1:n_states){
  if(sum(sapply(lapply(xi, function(x)sum(x[i,])), function(x)(sum(x))))!=0){
  #Redefinimos matriz de transición
  A[i,] = exp_num_transitions[i,]/sum(sapply(lapply(xi,
                                  function(x)sum(x[i,])), function(x)(sum(x))))
  }else{A[i,]=c(rep(0,n_states))}
if(k!=1){
  for(j in 1:k){
    if(sum(smoothed[,i])!=0){mu[j,i]=sum(smoothed[,i]*returns[,j])/
                                        sum(smoothed[,i])}else{mu[i,j]=0}
  }
}else{
    if(sum(smoothed[,i])!=0){
```

```
        #Redefinimos mu y sigma
        mu[i,]=sum(smoothed[,i]*returns)/sum(smoothed[,i])
        sigma[i,]=apply((smoothed[,i]*(returns-mu[i,])^2),2,
                        function(x)sum(x)/sum(smoothed[,i]))
      }
  }

    #TOLERANCE
    if(k!=1){
      R=returns
      for(j in 1:nrow(returns)){
        R[j,]=returns[j,]-mu[,i]
      }
    # Verosimilitud para determinar convergencia del algoritmo
    likelihood[iteration, i] = log(sum(smoothed[,i]/sum(smoothed[,i])*
                                         exp(-.5*apply((as.matrix(R)%*%
                                         solve(as.matrix(sigma[[i]])))*
                                         as.matrix(R), 1,
                                         function(x)sum(x)))/((2*pi)^(k/2)*
                                         sqrt(abs(det(as.matrix(sigma[[i]]))))
                                         )
                                       )
                                     )
    }else{
    likelihood[iteration, i] = log(sum(smoothed[,i]/sum(smoothed[,i])*
                                         exp(-.5*((returns-mu[i,])*sigma^(-1)*
                                         (returns-mu[i,])))/
                                         ((2*pi)^(k/2)*
                                         sqrt(cov(as.data.frame(returns)))
                                         )
                                       )
                                     )
    }
  }
  if(iteration>1){change_likelihood= abs(likelihood[iteration,]
                                         -likelihood[iteration-1,])}
  iteration=iteration+1
  }

return(list(mu=mu, sigma=sigma, p = p, A=A, forward=forward, backward=backward,
            smoothed=smoothed, exp_num_transitions)
       )
}
```

Algoritmo Modelo Mixto GHMM-ANN

```
GHMM-ANN = function(returns, w, p, n_states=2, n_samples=10, Tolerance=7*10^{-2}){

  sample=returns
  n_samples=10
  TL = length(sample)

  #STDP Rule
  w_change = function(w,L, n_neurons, n_states, stdp){
      aux=list(data.frame())
      for(i_1 in 1:L){
        aux[[i_1]]=w
        if(i_1 %in% stdp[[1]]){poisson_train=m[,i_1]}else{poisson_train=rev(m[,i_1])}
          for(i_2 in 1:n_trains){
            if(poisson_train[i_2]==1){aux[[i_1]][i_2,]=
            exp(-aux[[i_1]][i_2,]+1)-1}else{aux[[i_1]][i_2,]=rep(-1,n_inputs)}
        }
    }
  return(aux)
  }

  p_new=p
  w_new=w
  for (s in 1:n_samples){
  p=p_new
  w=w_new
  #Sample
  returns=sample[seq((s-1)*trunc(TL/n_samples)+1,s*trunc(TL/n_samples)),]


  returns=as.data.frame(returns)
  L=length(returns)
  for(i in 1:n_samples){samples[[i]]= returns[((i-1)*round(L/n_samples)+1):(i*round(L/n_
  w = as.data.frame(w)
  pi=rep(1/n_states, n_states)
  #pi=c(0.6,0.4)


  m=matrix(nrow = n_neurons, ncol = L)
  R=list(as.data.frame(m), as.data.frame(m))
  R[1:n_states]=list(data.frame(c(rep(0,L))))
  iteration=1
```

```
  #Generate Poisson Train
  n_trains=n_neurons*n_states
  high_freq=n_trains*.75
  low_freq=n_trains/10
  df=1/n_trains
  m = matrix(ncol = L, nrow = n_trains)
  for(i in 1:L){
    m[,i] = as.double(runif(n_trains)<ifelse(i %in% stdp[[1]], high_freq*df, low_freq*df
  }
  eta=apply(m, 1, function(x)sum(x))

  #Baum-Welch Adaptation
  for(i in 1:n_states){
    for(j in 1:L){
      aux=pi[i]*exp(p[seq(((i-1)*n_inputs+1),i*n_inputs),1])*
                exp(apply(w_new[[j]][seq(((i-1)*n_inputs+1),i*n_inputs),]*
                as.double(returns[j,]),1,function(x)sum(x)))
      R[[i]][j,] = aux/sum(aux)
    }
  }
  aux=returns
  mu=list()
  for(i in 1:n_states){
    for(j in 1:L){
      aux[j,] = R[[i]][j,]*returns[j,]
    }
    mu[[i]]=sum(aux[j,])/sum(R[[i]])
    pi[[i]]=sum(R[[i]])/nrow(R[[i]])
    pi=pi/sum(pi)
  }


    w_new=w_change(w,L,n_neurons, n_states, stdp)
    for(i in 1:L){w_new[[i]]= mu[[i]] + 0.01*eta*w_new[[i]]}
    p_new=pi*exp(-p+1)-1
  }

return(list(mu=mu, pi=pi, p = p))
}
```