UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

DEPARTAMENTO DE ANÁLISIS MATEMÁTICO Y MATEMÁTICA APLICADA

TRABAJO DE FIN DE GRADO

Criptografía con redes neuronales caóticas

Cryptography using chaotic Neural Networks

Tutora: Rosa María Pardo San Gil

Melisa Belmonte Jiménez

Grado en Matemáticas

Curso académico 2022 - 2023

Convocatoria Junio

Resumen

La criptografía está presente en prácticamente todas las comunicaciones que se realizan en Internet, y las Redes Neuronales Artificiales(RNA) son un campo de creciente interés desde hace años, por lo que no es sorprendente que hayan aparecido numerosas propuestas para su uso en criptografía. Dentro de estas propuestas, predominan las que le dan comportamiento caótico para dificultar que una red externa imite su comportamiento.

El objetivo de este trabajo es mostrar el estado del arte en el uso de RNA para criptografía y realizar una descripción del funcionamiento de las estructuras más relevantes.

Palabras Clave— Caos, Criptografía, Redes Neuronales, Árbol de Paridad, Sincronización

Abstract

Cryptography is present in almost every communication made through the Internet, and Artificial Neural Networks(ARN) have been a growing field of interest in the past years, therefore it is not surprising that there have been several ideas to use them in cryptography. Among these proposals, the most common are those that provide it with a chaotic behavior to prevent an external network from imitating.

The purpose of this work is to show the state of the art in the use of ARN for cryptography and offer a description of the behavior of the most relevant structures.

Key Words— Chaos, Criptography, Neural Networks, Tree Parity Machine, Synchronization

Índice

1. Introducción **1**

2. Criptografía **2**

2.1. Tipos más relevantes 2

2.1.1. Criptografía de Curva Elíptica 2

2.1.2. Criptografía Visual 3

2.1.3. Función *Hash* 4

2.2. Criptografía con caos 4

3. Criptografía con RNAs no sincronizadas **7**

3.1. Criptografía Visual 8

4. Sincronización de RNA **10**

4.1. Sincronización de redes de Alimentación hacia delante: Árbol de Paridad . . 10

4.1.1. Elección de la tasa de aprendizaje 11

4.1.2. Sincronización 12

4.1.3. Ejemplo 13

4.2. Sincronización de redes Recurrentes 14

4.2.1. Sincronización adaptativa de redes con retardo 14

5. Criptografía con RNAs sincronizadas **18**

5.1. Generar secuencias aleatorias 18

5.1.1. Red de Hopfield con retardo 18

5.2. Mejorar sistemas existentes 19

5.2.1. Curva elíptica 19

5.3. Árbol de paridad 19

5.3.1. Criptoanálisis del Árbol de Paridad 20

5.3.2. Mejoras a AP 22

6. Análisis de resultados del método AP **23**

6.1. Resultados AP original 23
6.1.1. Sincronización de 2 redes 23
6.1.2. Ataque por fuerza bruta 24
6.1.3. Ataque geométrico 25
6.1.4. Ataque genético 26
6.2. AP con mapa caótico 27
6.2.1. Sincronización de 2 redes 27
6.2.2. Ataque por fuerza bruta 28
6.2.3. Ataque geométrico 28
6.2.4. Ataque genético 28

7. Conclusión **30**

A. Anexo I: Redes Neuronales **31**
A.1. Clasificación de las RNA 31
A.1.1. Redes con Propagación hacia delante 32
A.1.2. Redes Recurrentes 33
A.2. Entrenamiento 37
A.2.1. Descenso gradiente 37
A.2.2. Propagación hacia atrás 38

B. Anexo II: Halftoning con Q'tron **40**

C. Anexo III: Librería RNA **41**

D. Anexo IV: Implementación AP **50**

1. Introducción

Las Redes Neuronales Artificiales (RNA) están adquiriendo cada vez mayor importancia en muchos nuevos ámbitos en los últimos años. Uno de estos ámbitos es la criptografía, necesaria para la transmisión segura de mensajes y, por tanto, presente en todas las comunicaciones electrónicas.

Actualmente, todos los métodos criptográficos usados se basan en la teoría de números, y la seguridad de los más populares depende de la incapacidad de los ordenadores de realizar ciertas operaciones en un tiempo razonable. Esto hace que la utilidad de dichos métodos se vea comprometida por la aparición de ordenadores para los que estas operaciones tienen una complejidad distinta, amenaza que ya se plantea con la computación cuántica. Su capacidad de procesamiento, considerablemente mayor que la de los paradigmas de computación actuales, lo hace capaz de romper los códigos con facilidad. Por este motivo, se busca en las redes neuronales, un sistema criptográfico cuya seguridad no se vea afectada por este aumento de potencia de proceso de los ordenadores.

Una de las debilidades que las redes neuronales pueden tener es la posibilidad de que un atacante simule su comportamiento con otra red. Para combatir esta debilidad, es habitual emplear redes con comportamiento caótico o añadir un mapa caótico que se aplica sobre la salida de la red para aumentar la seguridad de los métodos, este tipo de redes son a las que nos referimos como Redes Neuronales Caóticas.

En el primer apartado de este trabajo se aporta una breve introducción a los conceptos fundamentales de los métodos criptográficos empleados y al uso del caos en este campo.

En el segundo apartado se tratan los métodos criptográficos basados en RNA, que no utilizan la sincronización de redes.

En el tercer apartado se justifica el funcionamiento de formas de sincronización de RNA, que posteriormente se desarrollan en el cuarto apartado.

En el cuarto apartado se hace una descripción del estado del arte de la criptografía que utiliza RNA sincronizadas, destacando el Árbol de Paridad (AP).

Finalmente, en el último apartado se incluyen el análisis de la eficacia del método más relevante en este campo, el AP, frente a diversos ataques: fuerza bruta, geométrico y genético descritos en el apartado cuarto.

2. Criptografía

La criptografía se refiere al conjunto de técnicas usadas para transmitir mensajes privados por canales inseguros, de forma que solo a quienes está destinada la información puedan leerla.

La clasificación más común de los métodos es según si la clave que se usa es privada o no:

- Sistemas simétricos: tanto emisor como receptor usan la misma clave, por lo que esta debe de ser privada.
- Sistemas asimétricos: se utilizan distintas claves para el proceso de cifrado y descifrado. Estas se generan a través de datos compartidos por canales públicos, y es la propia estructura del sistema la que evita que un tercero consiga la clave para descifrar.

Actualmente, casi todos los sistemas que se usan son de clave pública o mixtos, que son aquellos que transmiten la clave privada cifrada asimétricamente. Los sistemas que se usan emplean algoritmos relacionados con teoría de números, curvas elípticas, teoría del caos, etc. para garantizar su seguridad. Dos de los protocolos de clave pública más comunes son el RSA para el cifrado y Diffie-Hellman para intercambio de claves.

2.1. Tipos más relevantes

Se pueden encontrar muchos tipos de criptografía, en los proximos 3 subapartados se introducen los que resultarán relevantes en el apartado sobre RNA en criptografía: criptografía de curva elíptica, criptografía visual y funciones hash.

2.1.1. Criptografía de Curva Elíptica

La seguridad de estos sistemas se basa en la dificultad de resolver un logaritmo discreto, encontrar el x que resuelve la ecuación $a^x = b$ para unos ciertos a y b. Resolver este logaritmo es un problema de complejidad exponencial, haciéndolo muy seguro para grupos finitos grandes. Sin embargo, la exponenciación discreta, su inverso, es una operación sencilla; con lo que aporta seguridad sin un coste computacional y temporal alto.

Aparte de servir como ejemplo de cómo los algoritmos actuales se basan en problemas que se resolverían con ordenadores de mayor potencia, este sistema aparecerá como uno de los métodos ya existentes que se han intentado mejorar con RNA.

Previo a la introducción del algoritmo, es necesario definir qué es un cuerpo y su característica:

Definición 2.1. ***Cuerpo****: Un cuerpo $\mathbb{K}$, es una estructura algebraica compuesta por un conjunto K no vacío y dos operaciones $(+ : K \times K \mapsto K, \cdot : K \times K \mapsto K)$ que cumplen las propiedades asociativa, conmutativa, distributiva y de existencia de inverso y elemento neutro.*

Definición 2.2. ***Característica****: La característica de un cuerpo $\mathbb{K}$, es menor entero positivo n tal que $\underbrace{i + \cdots + i}_{n \text{ veces}} = e$, donde i es el elemento neutro de $\cdot$ y e el de $+$.*

Para explicar su funcionamiento, primero definimos la curva elíptica que vamos a usar como $O(K)$ donde K es un cuerpo de característica distinta de 2 o 3 y $(x, y) \in O(K) \Leftrightarrow x, y \in K$ e $y^2 = x^3 + ax + b$. Para generar la clave se elige un número aleatorio k entre 0 y p (con p la característica de K) y otro número aleatorio $N \in O(K)$. De forma que la clave privada es k y la pública $Y = (kN, N)$.

El cifrado resultante del mensaje original M sería, eligiendo un r entero al azar, $(C_1, C_2) = (rN, M + rkN)$. Para descifrarlo, solo es necesario usar la siguiente equivalencia: $M \equiv C_2 - kC_1 = C_2 - k(rN) = M + rkN - k(rN) = M$.

2.1.2. Criptografía Visual

La criptografía visual se refiere a los métodos criptográficos para compartir imágenes. Las que queremos compartir se llaman imágenes objetivo, estas se separan en conjuntos de imágenes que llamamos transparencias. Cada una de ellas no revela ninguna de la información de la primera, pero al superponerlas todas se puede ver la imagen objetivo.

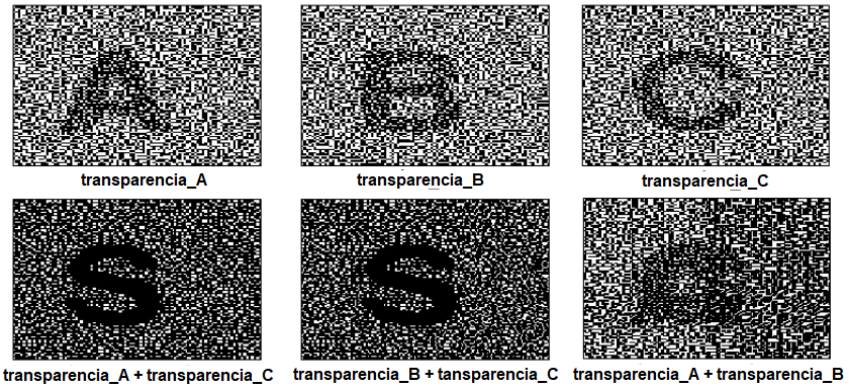


Figura 1: Ejemplo esquema de acceso [11]

Esta estructura se puede usar para crear un esquema de acceso, que hace que con un subconjunto de las transparencias se pueda ver otra imagen. De este modo, se pueden crear distintos niveles de acceso según las transparencias que se dan, siendo el máximo el que recibe todas. Por ejemplo, en figura superior, las transparencias A y C y las B y C juntas permiten ver la imagen objetivo, pero la A y la B, no.

2.1.3. Función *Hash*

Más que un tipo de criptografía, las funciones *hash* son una herramienta que se usa junto a la criptografía. Una función hash es un algoritmo que recibe un bloque de datos y devuelve un resumen de tamaño fijo. Estos resúmenes suelen consistir en cadenas de caracteres alfanuméricos. Idealmente, el resultado debería ser único para cada mensaje, pero la limitación en la longitud del resumen lo imposibilita. Hay 3 propiedades básicas que deben cumplir:

- Longitud fija del resumen
- Fácil de calcular
- Operación irreversible

Un ejemplo del empleo de estas funciones en criptografía son las contraseñas. Estos datos se almacenan mediante un hash, de forma que se evita que cualquiera que entre a la base de datos pueda leerlas. Luego, para verificar si la contraseña es correcta solo es necesario volver a aplicar el hash y comprobar si este coincide con el almacenado.

Más adelante, estas funciones aparecerán con un propósito similar al del ejemplo anterior, para la verificación de la sincronización de 2 redes.

2.2. Criptografía con caos

Los sistemas caóticos se han aplicado en la criptografía por su comportamiento aparentemente aleatorio dado por funciones relativamente simples. Sus tres usos principales han sido simular ruido en las comunicaciones, haciendo más difícil interceptarlas, crear claves aleatorias y definir funciones hash más seguras.

Todas las propuestas de métodos de clave pública se han basado en la posibilidad de sincronizar 2 sistemas. En concreto se suele utilizar la configuración Maestro-Esclavo, en la que dos circuitos con topologías idénticas, pero diferentes valores de componentes se sincronizan. El circuito del emisor, el Maestro, se acopla al Esclavo, el receptor, de forma que ambos se sincronizan, avanzando hasta un atractor caótico donde ambos coinciden. La posibilidad de sincronizar dos sistemas caóticos idénticos con este método fue demostrada por Pécora y Carroll [14].

Un buen resumen del por qué del interés en conectar criptografía con caos viene dado por la tabla de [1] donde resumen la conexión entre ambos por sus características principales:

Los sistemas caóticos más comunes en criptografía son los siguientes:

Característica caótica	Propiedad criptográfica	Descripción
Ergodicidad[1] Topológicamente transitivo[2]	Confusión	La salida de distintas entradas es diferente. (No parecen tener relación)
Sensibilidad a las condiciones iniciales y a los parámetros	Difusión	Una pequeña diferencia en la entrada da una salida muy distinta.
Determinístico	Pseudo-aleatoriedad determinística	Tiene resultados aparentemente aleatorios, que en verdad son determinísticos.
Complejidad	Complejidad algorítmica	Un algoritmo simple produce salidas complejas.

[1] Propiedad de un sistema que sugiere que un punto del mismo visitará todas las partes del espacio en el que se mueve.
[2] Una función $f : X \mapsto Y$ es topológicamente transitiva si para todo par de conjuntos abiertos no vacíos $U, V \subset X$ existe un $n \in \mathbb{N}$ tal que $f^{-n}(U) \cap V \neq \emptyset$

Cuadro 1: Comparación de las propiedades caóticas y criptográficas.

Circuito de Chua

El ingeniero Leon Ong Chua presenta en 1983 este circuito electrónico para proveer un sistema simple con comportamiento caótico que permitiera mostrar que el caos era un fenómeno físico robusto y no solo un resultado de un error de redondeo del ordenador (que era algo que no proporcionaban las ecuaciones de Lorentz). Su expresión como ecuaciones diferenciales es:

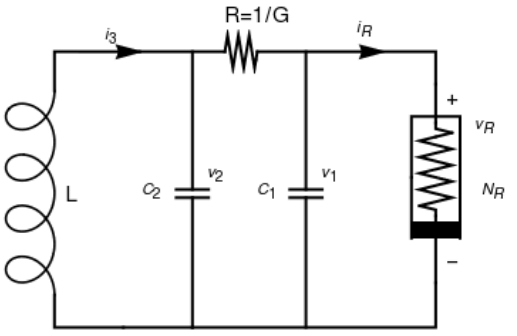


Figura 2: Circuito Chua

$$\begin{cases} \frac{\partial x(t)}{\partial t} = k\alpha(y - x - f(x)) \\ \frac{\partial y(t)}{\partial t} = k(x - y + z) \\ \frac{\partial z(t)}{\partial t} = -k(\beta y + \gamma z) \end{cases} \qquad \text{Donde } \alpha, \beta, \gamma \in \mathbb{R}.$$

Se ha usado a menudo en sistemas criptográficos para generar ruido o para crear un sistema asimétrico con el que generar números pseudoaleatorios para claves de cifrados simétricos.

Sistema de Chen

Presentado en 1999, este sistema es muy usado en criptografía. Inicialmente, esto se debía a que, al igual que el de Chua, este se podía simular fácilmente con un circuito. Las

ecuaciones que lo definen son:

$$\begin{cases} \frac{\partial x(t)}{\partial t} = \alpha(y - z) \\ \frac{\partial y(t)}{\partial t} = (\beta - \alpha)x - xz + \beta y) \\ \frac{\partial z(t)}{\partial t} = xy - \gamma z \end{cases} \qquad \text{Donde } \alpha, \beta, \gamma \in \mathbb{R}.$$

Mapa logístico

Presentado originalmente como modelo demográfico de tiempo discreto, esta ecuación aparece habitualmente en criptografía por su sencillez, ya que permite que se ejecuten con mayor velocidad los programas que la usan. La ecuación que lo define es $x_{n+1} = \alpha \cdot x_n(1 - x_n)$, con $\alpha \in \mathbb{R}$. Su uso en sistemas criptográficos tiende a ser como una capa extra de seguridad sobre métodos ya existentes, es una manera simple de añadir confusión al resultado final. Los cifrados que lo usan emplean como llave la condición inicial y el parámetro α. Estos cifrados empiezan en el primer uso en $n = 0$ y le suma 1 en cada interacción, de forma que se usa el x_n obtenido en la comunicación anterior.

3. Criptografía con RNAs no sincronizadas

Hacemos una primera división de los métodos de cifrado con RNA según si utilizan sincronización o no. En esta sección, tratamos con los que no la utilizan. Estos métodos son todos de clave privada, los datos que se pueden usar para ello suelen ser un conjunto de entrenamiento [8] o la propia topología de la RNA, es decir, una tupla de la forma (n.º neuronas de entrada, n.º capas ocultas, Pesos) [18]. Estos sistemas demuestran tener bastante fortaleza en las simulaciones, pero el tamaño de la clave es considerable y los propuestos hasta ahora no parecen mostrar una ventaja respecto a los métodos empleados actualmente.

Por otro lado, se ha utilizado la capacidad de las RNA con propagación hacia delante de aprender funciones para imitar sistemas de ecuaciones con comportamiento caótico. La ventaja del uso de RNA es que una vez ha aprendido la función, es más rápido que los métodos de aproximación de sistemas de ecuaciones diferenciales, aportando una mejora de coste tanto temporal como computacional a algoritmos ya existentes. Además, para recrear la dinámica, un atacante necesitaría todos los valores de la red y su estructura. Por ejemplo, en [13] usan una RNA para crear una dinámica caótica similar a la del sistema de ecuaciones de Chua, mencionado anteriormente como uno de los sistemas habituales para métodos de criptografía con caos. En este caso, se emplea como clave privada para cada interacción una de las coordenadas y el tiempo.

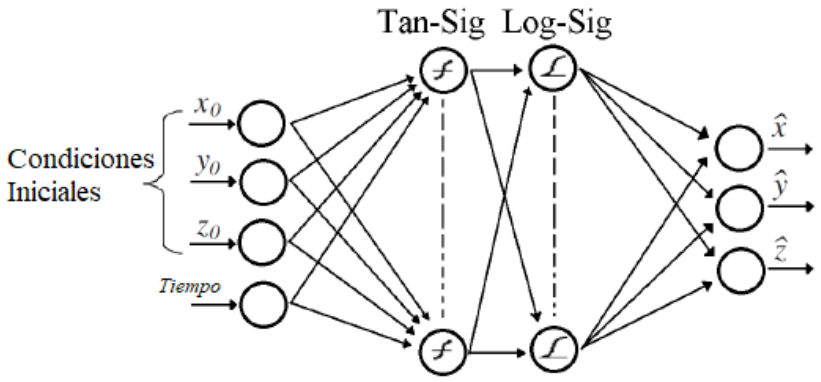


Figura 3: Red de [13].

Otro de sus usos ha sido el empleo de redes celulares a las que se les da un comportamiento caótico para crear secuencias pesudo-aleatorias que utilizar como clave; como en [15], donde una RNC con 3 neuronas con comportamiento caótico:

$$\begin{cases} \frac{\partial x_1}{\partial t} = -x_1 + A_1 y_1 + B_{11} x_1 + B_{12} x_2 \\ \frac{\partial x_2}{\partial t} = -x_2 + x_1 + x_3 \\ \frac{\partial x_3}{\partial t} = B_{32} x_2 \end{cases}$$

La clave privada permite calcular las condiciones iniciales del sistema, que se usan iterarlo

tantas veces como longitud tenga la clave. Llamando a las salidas de cada iteración i y_i, usando unos h y λ predefinidos, se calcula $v_i = h||y_i|| + \lambda$. Con su parte decimal expresada en binario con una precisión p: $u_i = 0.b_i^1...b_i^p$, se cifra el mensaje $M = m_1...m_l$ usando $c_i = (m_i + \oplus_{k=1}^{4} b_i^k) mod 256$.

3.1. Criptografía Visual

La criptografía con RNA no sincronizadas se ha aplicado frecuentemente a la criptografía visual, concretamente el modelo para separar una imagen en transparencias de [20], presentado originalmente en 2004, ha sido citado en muchos trabajos sobre criptografía por sus buenos resultados prácticos. Para este modelo, se emplea una RNA Q'tron, introducida en la sección de redes recurrentes.

Por simplicidad, se explica su funcionamiento para crear solo 2 transparencias, ya que, de querer crear más, el proceso es similar. Para lo que antes hemos llamado la preparación de los datos dentro de la fase de entrenamiento, previa a la aplicación del algoritmo es necesario convertir las imágenes a binario (*Halftoning*), para ellos dan un método cuyo funcionamiento se puede ver en el Anexo II.

Tomamos una imagen en binario O de tamaño $M \times N$ de la que queremos generar 2 transparencias T_1 y T_2. Tomamos una neurona Q'tron por pixel de cada imagen, llamamos $\mu_{ij}^{O} \in \{0,1\}$ a la correspondiente al pixel en las coordenadas $i \times j$ de la imagen y $\mu_{ij}^{T_1} \in \{0,1\}$ y $\mu_{ij}^{T_2} \in \{0,1\}$ a las de las transparencias. Como el valor de los píxeles de la imagen original viene dado, dejamos estas neuronas como fijas, mientras que el resto son libres. Llamamos Θ al conjunto de llas neuronas que componen nuestra red, de forma que está formado por ternas $(\mu_{ij}^{O}\, \mu_{ij}^{T_1}, \mu_{ij}^{T_2}) \in \Theta$ donde $i \in \{1...M\}$ y $j \in \{1...N\}$.

Nuestro objetivo es que la superposición de las imágenes transparencias permitan obtener la imagen original. Definiendo el blanco como 0 y el negro como 1, esta meta se puede expresar como una regla OR. Esto es porque la única manera de que una casillas se quede en blanco es que las casillas de ambas transparencias lo estén, ya que, con que uno de ellos sea negro, este es el que se ve.

t_1	t_2	o	E
0	0	0	0
		1	2,25
0	1	1	0,25
		0	1
1	0	1	0,25
		0	1
1	1	1	0,25
		0	4

Figura 4: Error de las transparencias y el objetivo

Para encontrar la función energía, nos centramos en 1 píxel y llamamos t_1 y t_2 a los de las transparencias y o al del original. Observamos que las combinaciones con OR cumplen que $1{,}5o - (t_1 + t_2) \leq 0{,}5$, mientras que para el resto de combinaciones $1 \leq |1{,}5o - (t_1 + t_2)|$. Por lo que se puede considerar que la forma correcta reduce el error $E(o, t_1, t_2) = (1{,}5o - (t_1 + t_2))^2$, donde elevamos el error al cuadrado para evitar valores negativos. Extendiendo a toda la imagen, la función energía queda expresada

como:

$$\mathbb{E}(\Theta) = \frac{1}{2} \sum_{(o,t_1,t_2)\in\Theta} E(y_o, y_{t_1}, y_{t_2})$$

donde las y_μ representa las salidas de las neurona μ.

4. Sincronización de RNA

Todos los sistemas criptográficos asimétricos presentados para redes neuronales se basan en la sincronización de RNA. En esta sección se pretenden dar los fundamentos necesarios para comprender los mecanismos de sincronización que se emplearán en el próximo apartado y justificar que, en efecto, los sistemas propuestos se sincronizarán.

Definición 4.1. ***Sincronización de RNA****: Si tenemos 2 redes neuronales A y B, cuyas salidas expresan las funciones* $f_A : X \mapsto Y$ *y* $f_B : X \mapsto Y$ *respectivamente, consideramos que se sincronizan existe un* $t_0 \in X$ *tal que para todo* $t > t_0$ $f_A(t) = f_B(t)$.

Como se comentó en el apartado 2, las redes neuronales se clasifican en función del flujo de datos, en redes con propagación hacia delante y redes neuronales recurrentes. Se presenta la estructura del árbol de paridad para las primeras y la sincronización adaptativa de redes con retardo para las recurrentes.

4.1. Sincronización de redes de Alimentación hacia delante: Árbol de Paridad

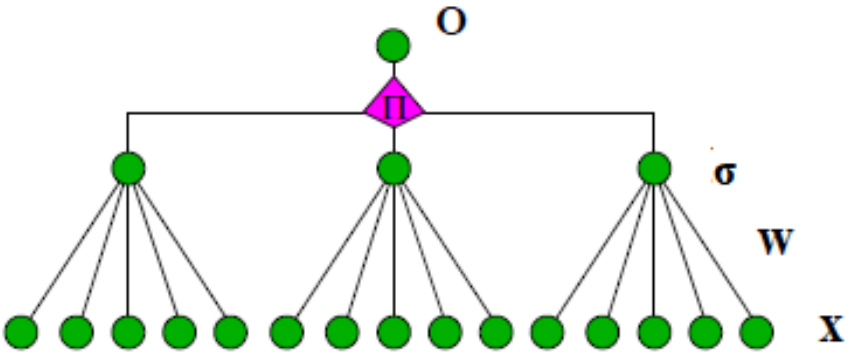


Figura 5: Esquema AP [7]

Las RNA basadas en el Árbol de Paridad (AP) se introducen en un artículo sobre la sincronización de RNA con esta estructura [12]. En él, consideran un anillo de Redes Neuronales en el que todas reciben una misma entrada y cada una se entrena con la salida de la anterior, hasta alcanzar un estado estacionario. La estructura AP consiste en 2 capas. La primera tiene K neuronas representadas por vectores N—dimensionales de peso, w_i, con salidas

$$o_i = \begin{cases} sign(x_i^T \cdot w_i) & si \quad x_i^T \cdot w_i \neq 0 \\ a & si \quad x_i^T \cdot w_i = 0 \end{cases}$$

con a previamente acordado como 1 o -1. La capa de salida consiste en una única neurona cuya salida es la multiplicación de las de la capa anterior ($o = \prod_{i=1}^{K} o_i$). Sus entradas

son K vectores N—dimensionales $\{x_1, ...x_N\}$, que al implementarlos se representan por matrices $K \times N$.

De cara a su aplicación para criptografía, introducen un cambio para facilitar la sincronización completa de los pesos, que es su acotación por un cierto valor L. De forma que la regla de aprendizaje de cada coordenada del peso de los perceptrones queda expresada, llamando $y_{ki} = w_{ki} + \frac{\eta_i}{N} \cdot f(\sigma_i,\ s) \cdot s \cdot x$, de la siguiente forma:

$$w_{ki}^{+} = \begin{cases} y_{ki} & si \quad o \cdot o_i > 0 \ y \ |y_{ki}| \leq L \\ L \cdot sign(y_{ki}) & si \quad o \cdot o_i > 0 \ y \ |y_{ki}| > L \\ \\ w_{ki} & si \qquad o \cdot o_i \leq 0 \end{cases} \qquad \forall i \in \{1, ..., K\} \ \forall k \in \{1, ..., N\}$$

Para el aprendizaje de la red se generan de forma aleatoria estas entradas. Si ambas tienen la misma salida, entonces se aplica la regla de aprendizaje sobre los perceptrones cuya salida coincide con la de la red. Esta regla consiste en modificar los pesos de la siguiente manera: $w_i^{+} = w_i + \frac{\eta}{N} \cdot \sigma(o_i) \cdot x \quad \forall i \in \{1...K\}$, donde η_i es la tasa de aprendizaje, N la dimensión del peso, f, la función de aprendizaje, o_i, la salida obtenida en el perceptrón i, o', la salida de la otra red y x, la entrada.

Para las funciones de aprendizaje se presentan 3 opciones que permiten llegar a una sincronización de las redes:

- **Hebbian:** $\sigma(o_i) = o_i$
- **Anti-Hebbian:** $\sigma(o_i) = -o_i$
- **Paseo Aleatorio (*Random Walk*):** $\sigma(o_i) = 1$

4.1.1. Elección de la tasa de aprendizaje

Para el caso más relacionado con este trabajo, que es la total sincronización de los pesos, habitualmente se usan con sus pesos normalizados, es decir, tras aplicar la regla de aprendizaje, se dividen los pesos por su norma. Esto se debe a que facilita su estudio, dando resultados más fiables de sincronización, y a que en simulaciones ha demostrado requerir menor tiempo para conseguir sincronizarse. Por ello se estudiará su sincronización para este caso concreto.

Para buscar los valores de la tasa de aprendizaje (η) para los que las redes se sincronizan, se emplea un modelo con 1 único perceptrón en cada red, de forma que cuando la regla se aplica, la suma de los vectores de los pesos se conserva, ya que dan el mismo paso, pero en sentido contrario. Llamamos w^A y w^B a los pesos de cada red, nos interesa estudiar $R = w^A \cdot w^B$, cuya dinámica en función de $\alpha = \frac{t}{N}$ se calcula en [12] como

$\frac{\partial R}{\partial \alpha} = \eta(R+1)(\sqrt{\frac{2}{\pi}}(1-R) - \eta\frac{\theta}{\pi})$ donde $\theta \in [0,\pi]$ es el ángulo entre w^A y w^B. Como los pesos están normalizados, R también se puede expresar como $cos(\theta)$.

Los puntos fijos de esta ecuación diferencial son $R = 1$ ($\theta = 0$), $R = -1$ ($\theta = \pi$) y $R = 1 - \frac{\eta\theta}{\sqrt{2\pi}}$. La sincronización se alcanza cuando $\theta = 0$.

Definimos la función $g_\eta(\theta) = \frac{1-cos(\theta)}{\theta\eta}$ de forma que se da $R = 1 - \frac{\eta\theta}{\sqrt{2\pi}}$ solo si $g_\eta(\theta) = 1$. Para ver los η para los que existe solución buscamos aquellos para los que el máximo de la función es mayor o igual que 1. Para ello, usamos su derivada $g'_\eta(\theta) = \frac{\sqrt{2\pi}(\theta sen(\theta)+1-cos(\theta)-1)}{\theta^2\eta}$.

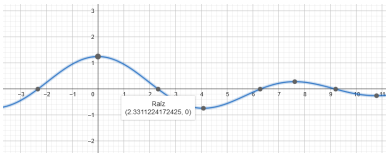


Figura 6: Gráfico de la función $g'_1(\theta)$

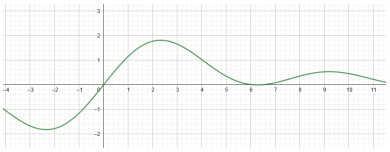


Figura 7: Gráfico de la función $g_1(\theta)$

Como se puede ver en las imágenes, el máximo de la función se encuentra en un $\theta_m \approx 2{,}33$, despejando el η que hace $g_\eta(\theta_m) = 1$ tenemos $\eta_m \approx 1{,}816$. Por lo tanto, existe solución para $\eta < \eta_m$. Esta elección de tasa de aprendizaje dentro de esta cota se confirma en las simulaciones, que muestran que las redes evolucionan a $R = 1$, se sincronizan, para $\eta < \eta_m$ y a $R = -1$ en otro caso.

4.1.2. Sincronización

Se puede interpretar el proceso de sincronización geométricamente: viendo cada entrada como K hiperplanos $X_1, ..., X_K$ de la forma $X_i : f(z_1, ..., z_N) = \sum_{j=1}^{N} x_{ij} z_j = 0$ en el espacio discreto N-dimensional $U = \{-L, ..., L\}^N$, los pesos como puntos en U $W_i = (w_{i1}, ..., w_{iK})$ y la salida oculta i-ésima como el lado del espacio que contiene W_i respecto X_i. Nuestro objetivo es reducir la distancia entre los pesos W_i de ambas redes. Si un paso disminuye esta distancia, lo llamaremos atractor, si las aleja, repulsor. Al desconocer el valor de las capas intermedias de la otra red, es inevitable que se den pasos repulsores durante la sincronización. Sin embargo, al estar alimentándose de las salidas de la otra red, la probabilidad de pasos atractores aumenta con el número de interacciones, llevando finalmente a la sincornización.

En cada paso hay 3 posibilidades para cada perceptrón i de la primera capa:

- $o \neq o'$**:** En este caso no se aplica ninguna regla, por lo que los pesos de las redes permanecen iguales.
- $o = o'$ **y** $o_i = o'_i$**:** Los pesos de ambos perceptrones se mueven en la misma dirección, por lo que sus distancias se mantienen. Excepto en el caso en el que una de ellas

supera el límite L o $-L$, en ese caso al dejar el peso de ese perceptrón en el límite esta distancia se reduce.

- $o = o'$ **y** $o_i \neq o'_i$**:** Solo el perceptrón de una de las redes se mueve. Según el sentido de su movimiento esto puede aumentar o disminuir la distancia con el otro. Por ejemplo, si uno de los elementos del vector pesos ya estaba sincronizado, al moverse se desincronizará.

La razón por la que, a pesar de la existencia de estos pasos repulsores, las redes pueden llegar a sincronizarse es que las probabilidades de los pasos dependen del valor de los pesos. Ya que tanto la salida de las redes están condicionadas por las de los perceptrones de la primera capa, y la de estos por el valor de los pesos, la probabilidad de que se de un paso u otro no es la misma. Cuanto más cerca de sincronizarse están ambas redes, más probable es que se den pasos atractores. Por ello, gracias a que ambas redes están aprendiendo la una de la otra, la tendencia de los pesos es sincronizarse, como confirman las simulaciones.

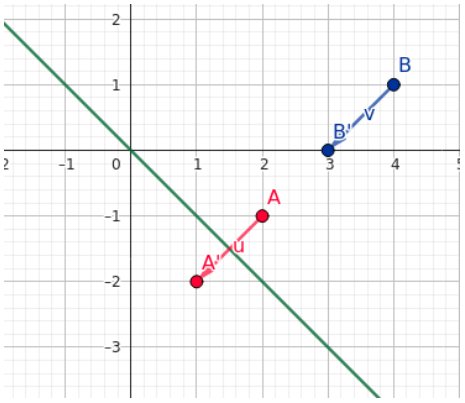
En el caso de la regla Hebbian y Anti-Hebbian, el sentido en el que se desplazan los pesos depende del signo de $x \cdot o_i$. Al depender el valor de o_i del de w_i y de x, las probabilidades de que $sign(x \cdot o_i) = 1$ y $sign(x \cdot o_i) = -1$ no son iguales, sino que, como se ha comentado antes, dependen del valor del peso. En [16] calcula que para Hebbian, la probabilidad de que sea positivo es mayor, mientras que en Anti-Hebbian, lo es que sea negativo. Esto quiere decir que en con la primera regla, los pesos tienden a empujarse hacia los límites. Mientras que en con la segunda, tienden a alejarse de estos. Por lo tanto, para la regla Hebbian, el valor absoluto de cada elemento de los pesos aumenta, hasta alcanzar un punto estable, mientras que con Anti-Hebbian, disminuyen.

4.1.3. Ejemplo

Por simplicidad suponemos una tasa de aprendizaje 1, y unas redes con la regla Anti-Hebiana. Aparte de dar siempre la misma entrada para la capa que observamos, $x = (1, 1)$. De forma que el cambio de peso en los pasos en los que corresponda, queda expresado como $w_i^+ = w_i - (o_i, o_i) \quad \forall i \in \{1, 2\}$.

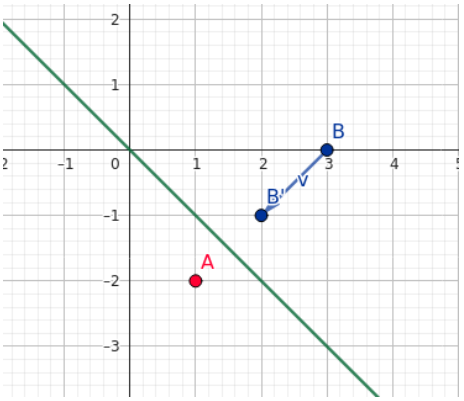
Paso 1:

Si empezamos con los pesos $w_A = (2, -1)$ y $w_B = (4, 1)$, la salida de las capas es 1, si la salida global acaba siendo 1 aplicamos la regla de aprendizaje. En este caso la distancia entre ambas se mantiene. (Si la salida global hubiera sido -1, no se aplica la regla por lo que la distancia se mantendría).



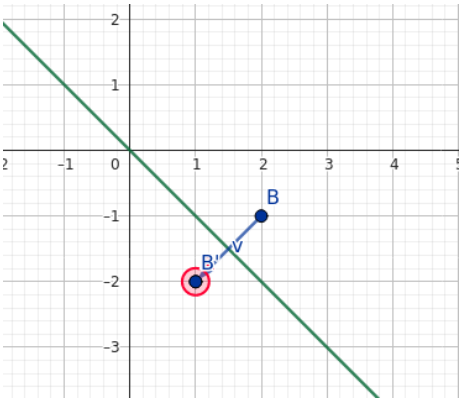
Paso 2:

Ahora los pesos son $w_A = (1, -2)$ y $w_B = (3, 0)$ en las capas ocultas 1 que recibe la entrada $x = (3, 3)$ La salida de las capas es -1 y 1 respectivamente, si la salida global acaba siendo 1 aplicamos la regla de aprendizaje solo sobre B. En este caso la distancia entre ambas se reduce.(Si la salida global hubiera sido -1, se aplicaría la regla sobre A y se daría lo que llaman un paso repulsivo, que es aquel en el que los pesos se separan)



Paso 3:

En este paso los pesos son $w_A = (1, -2)$ y $w_B = (2, -1)$ en las capas ocultas 1 que recibe la entrada $x = (3, 3)$ La salida de las capas es -1 y 1 respectivamente, si la salida global acaba siendo 1 aplicamos la regla de aprendizaje solo sobre B. En este caso la distancia entre ambas se reduce a 0.(Si la salida global hubiera sido -1, se aplicaría la regla sobre A y se habría dado un paso repulsivo)



4.2. Sincronización de redes Recurrentes

Para estas redes se suele mostrar especial interés en su sincronización en el caso continuo con retardos y, al ser análogas las demostraciones para las que no los tienen, en esta sección se trabajará con redes recurrentes continuas con retardos. Definiendo el retardo en la red como $\tau(t)$ podemos considerar los estados de una red recurrente con retardos determinados por la siguiente ecuación: $\frac{\partial x(t)}{\partial t} = -x(t) + A\sigma(x(t)) + B\sigma(x(t - \tau(t)) + I(t)$, donde $x(t) = (x_1(t), ...x_n(t))^T$ es el vector de estados de las n neuronas, A y B las matrices de pesos de conexión en t y t con retardo respectivamente, I es el vector de entrada (el voltaje que recibe de forma continua el circuito), τ es el retardo de transmisión y $\sigma = (\sigma_1, ..., \sigma_n)^T$ las funciones de activación de las neuronas.

4.2.1. Sincronización adaptativa de redes con retardo

Cao y Lu proponen un mecanismo de sincronización para redes neuronales acopladas idénticas con retardo por retroalimentación hacia atrás [2]. La red debe cumplir dos condiciones para que este mecanismo lleve a sincronización:

1. Cada función activación de las neuronas satisface la condición $0 \leq \frac{\sigma_i(x)-\sigma_i(y)}{x-y} \leq h_i$ y $0 \leq \frac{\tilde{\sigma}_i(x)-\tilde{\sigma}_i(y)}{x-y} \leq k_i$ para $x \neq y$ y ciertos h_i y k_i.
2. $\tau(t) \geq 0$ es una función derivable que cumple $0 \leq \tau(t) \leq \mu < 1 \ \forall t$

Para formular el problema de sincronización tomamos dos sistemas:

$$\frac{\partial x(t)}{\partial t} = -Cx(t) + A\sigma(x(t)) + B\tilde{\sigma}(x(t-\tau(t)) + I$$

$$\frac{\partial y(t)}{\partial t} = -Cy(t) + A\sigma(y(t)) + B\tilde{\sigma}(y(t-\tau(t)) + \varepsilon \odot (y(t) - x(t))$$

Donde $\odot$ se define como $\varepsilon \odot (x) = (\varepsilon_1 x_1, .., \varepsilon_n x_n)^T$ y $\varepsilon = (\varepsilon_1, ...\varepsilon_n)$ es la fortaleza de acoplamiento. Y medimos el error de sincronización como $e(t) = y(t) - x(t)$ que con las ecuaciones del sistema nos dejan con:

$$\frac{\partial e(t)}{\partial t} = -Ce(t) + Af(e(t)) + Bg(e(t-\tau(t)) + I + \varepsilon \odot (e(t))$$

donde $f = \sigma(e(t) + x(t)) - \sigma(x(t))$ y $f(0) = 0$ y $g = \tilde{\sigma}(e(t) + x(t)) - \tilde{\sigma}(x(t))$ y $g(0) = 0$.

Antes de presentar el teorema que define el aprendizaje de la red, para demostrarlo es necesario el siguiente lema:

Lema 4.1. *Para todo par de vectores $x, y \in \mathbb{R}_{1\times n}$ y matriz definida positiva $Q \in \mathbb{R}_{n\times n}$ se cumple la desigualdad $2x^T y \leq x^T Q x + y^T Q^{-1} y$*

Teorema 4.2. *Bajo las 2 condiciones iniciales, dos redes acopladas pueden ser sincronizadas cuando la fortaleza de acoplamiento se actualiza como $\varepsilon_i = -\alpha_i e_i^2(t)$ donde $\alpha_i > 0 \forall i \in \{1, ..., n\}$ son constantes positivas arbitrarias.*

Demostración. Empezamos introduciendo el funcional de Lyapunov:

$$V(t) \ = \ \frac{1}{2}e^T(t)e(t) + \frac{1}{2}\sum_{i=1}^{n}\frac{(\epsilon_i + l)^2}{\alpha_i} + \int_{t-\tau(t)}^{t}\frac{g^T[e(\theta)] \cdot g[e(\theta)]}{2(1-\mu)}d\theta$$

donde l es una constante positiva que se precisará más adelante y $e(t)$ es el sistema del error.

Calculando su derivada parcial en función de t tenemos:

$$\begin{aligned}\frac{\partial V}{\partial t} \ =& e^T(t)\frac{\partial e(t)}{\partial t} - \sum_{i=1}^{n}(\epsilon_i + l)e_i^2((t) + \frac{g^T(e(t)) \cdot g(e(t))}{2(1-\mu)} - \\ & (1 - \frac{\partial \tau}{\partial t}(t))\frac{g^T(e(t-\tau(t))) \cdot g(e(t-\tau(t)))}{2(1-\mu)}\end{aligned}$$

Nos centramos en la parte $e^T(t)\frac{\partial e(t)}{\partial t} - \sum_{i=1}^{n}(\epsilon_i + l)e_i^2((t)$ para reducirla:

$$e^T(t)\frac{\partial e(t)}{\partial t} - \sum_{i=1}^{n}(\epsilon_i + l)e_i^2(t) \ = e^T(t)(-Ce(t) + Af(e(t)) + Bg(e(t-\tau(t)) + \varepsilon \odot e(t)) - \sum_{i=1}^{n}\epsilon_i e_i^2(t) - l\sum_{i=1}^{n}e_i^2(t)$$

Observamos que $e^T(t)(\varepsilon \odot e(t)) = (e_1(t), ..., e_n(t)) \cdot (\epsilon_1 \cdot e_1(t), ..., \epsilon_n \cdot e_n(t))^T = \sum_{i=1}^{n}\epsilon_i e_i^2(t)$ y $e^T(t)e(t) = \sum_{i=1}^{n} e_i^2(t)$, que aplicado a la ecuación anterior la deja como:

$$e^T(t)\frac{\partial e(t)}{\partial t} - \sum_{i=1}^{n}(\epsilon_i + l)e_i^2(t) = e^T(t)(-Ce(t) + Af(e(t)) + Bg(e(t-\tau(t)))) - le^T(t)e(t)$$

Por el Lema 4.1 tomando Q como la identidad tenemos que $2e^T(t)Afe(t)) \leq e^T(t)A \cdot (e^T(t)A)^T + f^T(e(t)) \cdot f(e(t))$. Se puede hacer de forma análoga con B y g, con lo que nos queda la desigualdad:

$$e^T(t)\frac{\partial e(t)}{\partial t} - \sum_{i=1}^{n}(\epsilon_i + l)e_i^2((t) \leq -e^T(t)Ce(t) + \frac{1}{2}e^T(t)AA^Te(t) + \frac{1}{2}f^T(e(t))f(e(t)) + \frac{1}{2}e^T(t)BB^Te(t) + \frac{1}{2}g^T(e(t-\tau(t)))g(e(t-\tau(t))) - le^T(t)e(t)$$

A partir de la segunda condición del sistema ($0 \leq \frac{\partial \tau}{\partial t}(t) \leq \mu < 1$) podemos obtener $-\frac{1-\frac{\partial \tau}{\partial t}(t)}{1-\mu} \leq -1$.

Aplicando esta desigualdad y la anterior sobre $\frac{\partial V}{\partial t}$ obtenemos:

$$\frac{\partial V}{\partial t} \leq -e^T(t)Ce(t) + \frac{1}{2}e^T(t)AA^Te(t) + \frac{1}{2}f^T(e(t))f(e(t)) + \frac{1}{2}e^T(t)BB^Te(t) + \frac{1}{2}g^T(e(t-\tau(t)))g(e(t-\tau(t))) - le^T(t)e(t) + \frac{g^T[e(t)] \cdot g[e(t)]}{2(1-\mu)} - \frac{g^T[e(t-\tau(t))] \cdot g[e(t-\tau(t))]}{2} =$$

$$-e^T(t)Ce(t) + \frac{1}{2}e^T(t)AA^Te(t) + \frac{1}{2}f^T(e(t))f(e(t)) + \frac{1}{2}e^T(t)BB^Te(t) - le^T(t)e(t) + \frac{g^T[e(t)] \cdot g[e(t)]}{2(1-\mu)}$$

Por la primera condición que le pedimos al sistema ($0 \leq \frac{\sigma_i(x)-\sigma_i(y)}{x-y} \leq h_i$ y $0 \leq \frac{\tilde{\sigma}_i(x)-\tilde{\sigma}_i(y)}{x-y} \leq k_i$ para $x \neq y$ y ciertos h_i y k_i) tenemos las desigualdades: $|f_i(e_i(t))| \leq h_i|e_i(t)|$ y $|g_i(e_i(t))| \leq k_i|e_i(t)|$. A partir de ellas podemos deducir:

$$f^t(e(t))f(e(t)) = \sum_{i=1}^{n} f_i^2(e_i(t)) \leq \sum_{i=1}^{n} h_i^2 e_i^2(t) \leq he^T(t)e(t) \text{ donde } \ h = max\{h_i^2 : i \in \{1, ..., n\}\}$$

$$g^t(e(t))g(e(t)) = \sum_{i=1}^{n} g_i^2(e_i(t)) \leq \sum_{i=1}^{n} k_i^2 e_i^2(t) \leq k e^T(t)e(t) \text{ donde } k = max\{k_i^2 : i \in \{1, ..., n\}\}$$

Sustituyendo en la desigualdad anterior, esta queda:

$$\frac{\partial V}{\partial t} \leq -e^T(t)Ce(t) + \frac{1}{2}e^T(t)AA^Te(t) + \frac{1}{2}e^T(t)BB^Te(t) + (\frac{h}{2} + \frac{k}{2(1-\mu)} - l)e^T(t)e(t) \leq$$

$$e^T(t)(\lambda_{max}(-C) + \lambda_{max}(\frac{AA^T}{2}) + \lambda_{max}(\frac{BB^T}{2} + \frac{h}{2} + \frac{k}{2(1-\mu)} - l)e(t)$$

donde $\lambda_{max}(M)$ es el máximo de los autovalores de M.

Si elegimos $l = \lambda_{max}(-C) + \lambda_{max}(\frac{AA^T}{2}) + \lambda_{max}(\frac{BB^T}{2} + \frac{h}{2} + \frac{k}{2(1-\mu)} + 1$ podemos reducir la desigualdad anterior a $\frac{\partial V}{\partial t} \leq -e^T(t)e(t)$.

De aquí podemos sacar que $\frac{\partial V}{\partial t} = 0$ si y solo si $e(t) = 0$. Esto quiere decir que $E = \{[e(r), \epsilon]^T \in \mathbb{R}^{2n} : e(t) = 0,\ \epsilon = \epsilon_0 \in \mathbb{R}^n\}$ es el mayor conjunto invariante dentro de $M = \{\frac{\partial V}{\partial t} = 0\}$ para nuestros sistemas.

Por lo tanto, por el principio de invariabilidad funcional de ecuaciones diferenciales[5] la órbita converge asintóticamente al conjunto E, es decir, cuando $t \to \infty$ entonces $e(t) \to 0$ y $\epsilon \to \epsilon_0$. Luego la sincronización de las redes neuronales se consigue. □

La elección de las α influye en la velocidad de sincronización. A partir del teorema se puede derivar un corolario para redes sin retardos. Aparte, es posible adaptarlo para que, en vez de ser una acoplación unidireccional, sea bidireccional, necesitando solo la mitad de fortaleza de acoplamiento.

5. Criptografía con RNAs sincronizadas

Las RNA sincronizadas tienen 2 posibles usos en criptografía:

- Generar secuencias aleatorias. De entre estas destaca el árbol de paridad, ya que es sobre el que más se ha trabajado en los últimos 20 años y, por tanto, el que cuenta con más contenido.
- Mejorar sistemas existentes.

5.1. Generar secuencias aleatorias

Usando el mecanismo de la sección 5.2, se sincronizan 2 redes para producir códigos comunes. Las redes más habituales para ello son las redes de Hopfield (descritas en el apartado 3.1.2). El concepto detrás de su funcionamiento es muy similar en todas las propuestas, por lo que solo se presenta una de ellas como ejemplo.

5.1.1. Red de Hopfield con retardo

En la estructura que plantean Yu y Cao [19] se usa una Red de Hopfield con retardo para generar unas secuencias pseudoaleatorias con las que se crea otra secuencia binaria que se usa para cifrar el mensaje aplicando la función XOR y se emplean como clave los valores iniciales y la función retardo. Para que las secuencias sean pseudoaleatorias se usa una Red de Hopfield caótica cuyo comportamiento está descrito por la ecuación diferencial $\frac{\partial x(t)}{\partial t} = -C \cdot x(t) + A(t) \cdot \sigma(x(t)) + B \cdot \sigma(x(t - \tau(t)) + I$ donde $x(t)$ es el vector de estados de la red, τ es la función retardo, f la función de activación, C es una matriz diagonal y A es la matriz de conexión de los pesos en t y B en t con retardo.

Si $x_j(t) \in [d, e]$ entonces $y = \frac{x_j - d}{e - d} \in [0, 1]$ por lo que se puede representar como una secuencia binaria de la forma $y = b_1(x)b_2(x)...$ $(b_i(x) \in \{0, 1\}\ \forall i)$. Definiendo $\theta_k(x) = \begin{cases} 0 & si \quad x < k \\ & \\ 1 & si \quad x \geq k \end{cases}$,el bit i—ésimo de la secuencia se puede calcular como $b_i(x) = \sum_{r=1}^{2i-1} (-1)^{r-1} \Theta_{(e-d)(r/2i)+d}(x)$. A partir de las secuencias generadas por la red, se toma para cada elemento de x el bit i—ésimo (i acordado previamente) para generar una secuencia binaria de longitud predefinida m. Para cifrar el mensaje, se divide en partes de $m/8$ bytes que en su expresión binaria tienen tamaño m. Cada parte se cifra aplicando la función XOR sobre cada una de las partes (expresadas en binario) con la secuencia binaria.

5.2. Mejorar sistemas existentes

Esto principalmente consiste en la creación de ruido o el enmascaramiento de la señal usando la sincronización de 2 redes. Por ejemplo, en [22] emplean la sincronización de la sección 5.2 para crear un entorno con ruido en el que transmitir el mensaje que el sistema del receptor, sincronizado con el del emisor, puede imitar para obtener el mensaje. A continuación, se incluye un ejemplo de uso de redes para enmascarar las salidas de un sistema existente.

5.2.1. Curva elíptica

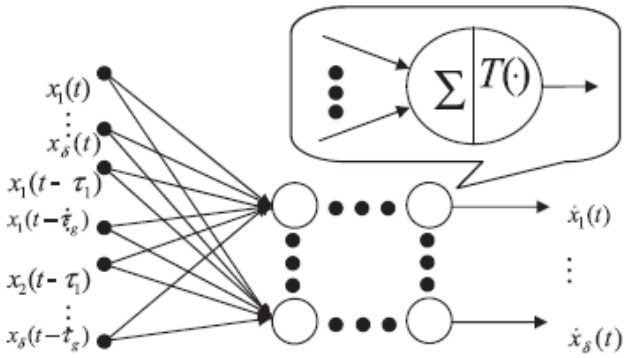


Figura 8: Red [6]

En [6] usan el cifrado de curva elíptica para evaluar la seguridad de enmascarar la señal con sistemas caóticos modelados por RNA. Se tienen 2 sistemas con configuración maestro—esclavo, descritas en el apartado 2.2, con dinámicas $N_m : \frac{\partial x(t)}{\partial t} = \sigma(x(t)) + \sum_{k=1}^{g} \sigma_k(x(t-\tau_k))$ y $N_e : \frac{\partial y(t)}{\partial t} = \tilde{\sigma}(y(t)) + \sum_{k=1}^{g} \tilde{\sigma}_k(y(t-\tau_k)) + D(t)$ donde D son las perturbaciones externas. Estos sistemas se aproximan con la red de la imagen de la derecha y su sincronización es la demostrada en [14].

Primero se comparte la clave pública y con ella se cifra el texto plano por el método de curva elíptica. El texto cifrado se pasa por el sistema maestro para transformarlo en una señal cifrada por enmascaramiento caótico. Por un canal público se transmite esta señal, que procesa el sistema esclavo para obtener el texto cifrado que se puede leer con la clave privada. En simulaciones este sistema mostró cierta mejora respecto al uso de la curva elíptica sola.

5.3. Árbol de paridad

Esta estructura, introducida en el apartado 5.1, es de la que más se ha escrito dentro de la criptografía con RNA; esto se debe a que es la única estructura de clave pública, que es

el tipo de criptografía más empleado actualmente. Al ser de clave pública, y, por tanto, no basar su seguridad en la protección de la clave, sino en su mecánica de funcionamiento, este método es el único que cuenta con criptoanálisis desarrollados para romperlo. Además, es el que cuenta con más líneas abiertas para su mejora, ya que a partir de estos criptoanálisis se han demostrado sus debilidades, que se pueden contrarrestar con cambios en partes del método.

La primera propuesta para criptografía con esta estructura aparece en 2002 [7], en ella los pesos sincronizados de dos RNA se usan como clave efímera para cifrar mensajes. La sincronización se lleva a cabo pasando por un canal público los resultados de entradas aleatorias y aplicando la regla de aprendizaje cuando las salidas coinciden en los perceptrones cuya salida (que se considera una salida oculta, ya que no se comparte) coincide con la de la red.

5.3.1. Criptoanálisis del Árbol de Paridad

Como se ha dicho anteriormente, al tratarse de un método de clave pública, se puede tratar de crear un ataque que permita obtener la clave final a partir de los datos públicos. Habiendo ya demostrado que tratar de sincronizar una red externa por fuerza bruta no resulta eficaz, estas propuestas aprovechan el conocimiento de la estructura del sistema o la posibilidad de utilizar muchas redes simultáneamente para obtener los pesos finales de las redes. Estos métodos se publicaron en respuesta a la propuesta de 2002 en ese mismo año [10] y son, todavía, los que se utilizan para probar el funcionamiento de las modificaciones que se proponen:

1. Probabilístico

Para este ataque, se usa la probabilidad de que el output en una de las neuronas sea 1 teniendo en cuenta la salida para cada neurona, para ver la probabilidad de que cada peso tenga un valor u otro. Llamando a la probabilidad de que $x_t \in \{-L, ..., 0, ..., L-1, L\}$ tenga un valor i $p_t(i) = P(x_t = i)$ entonces $p_0(i) = 1/2L + 1$ y la probabilidad de que para una salida o la salida oculta o_k sea 1 se podría expresar como $P(o_k = 1|o) = \frac{\sum_{(\alpha_1,...,\alpha_k),,} \prod_{i=1}^{k} \alpha_i = 0 y \alpha_k = 1 \prod_{i=1}^{k} p_i(\alpha_i)}{\sum_{(\alpha_1,...,\alpha_k),,} \prod_{i=1}^{k} \alpha_i = 0 \prod_{i=1}^{k} p_i(\alpha_i)}$

2. Genético

Imitan la forma en que sistemas biológicos evolucionan, de forma que los que tienen características útiles son potenciados. Se usan varias redes neuronales y, en cada paso, se desechan las que no coinciden con la salida de la red con la que se quiere coordinar y a las que coinciden se las multiplica. Para este ataque se empieza con una red con la estructura de las que se tratan de interceptar y pesos aleatorios y se autoimpone un límite de M redes en nuestra población. Este M dependerá de las capacidades de la computadora y el tamaño de la red, en cada paso cada red de la población evoluciona según 3 posibles casos:

1. $o^A \neq o^B$**:** En este caso las redes A y B no cambian por lo que las de la población tampoco.

2. $o^A = o^B$ **y hay menos de M redes:** Entonces, se sustituyen por 2^{k-1} variantes de sí mismas, tomando por k el número de salidas ocultas, simulando en cada una de ellas la combinación de salidas ocultas que puede haber tenido A para llegar a la salida final. Por ejemplo, si la red usada tiene 2 salidas ocultas y $o^A = 1$ hay 2 configuraciones de salidas ocultas de A: $o_0^A = o_1^A = 1$ ó $o_0^A = o_1^A = -1$.

3. $o^A = o^B$ **y hay M redes o más:** Igual que en el caso anterior eliminamos las redes con salidas distintas a o^A, pero en este caso las restantes las actualizamos con la regla habitual, con las salidas ocultas que hayan tenido.

Las ramificaciones hacen que tenga un coste computacional alto para modelos en los que hay un número alto de salidas ocultas, pero es eficaz para valores bajos.

3. Geométrico

Para este ataque se aprovecha la interpretación geométrica de la sincronización de la red. Recordamos que esta consite en ver cada entrada como K hiperplanos $X_1, ..., X_K$ de la forma $X_i : f(z_1, ..., z_N) = \sum_{j=1}^{N} x_{ij} z_j = 0$ en el espacio discreto N-dimensional $U = \{-L, ..., L\}^N$, los pesos como puntos en U $W_i = (w_{i1}, ..., w_{iK})$ y la salida oculta i-ésima como el lado del espacio que contiene W_i respecto X_i. Entonces cuando $o^A \neq o^C$ esto implica que hay pares de puntos W_i^A, W_i^C separados por el hiperplano X_i, es decir, que cumplen $d(W_i^A, W_i^C) > d(X_i, W_i^C)$. Si consideramos que A y C están cerca, esto implica las distancias entre sus pesos es pequeña, por lo que aplicando la desigualdad anterior, para un i en el que los pesos de A y C son distintos, la distancia entre el punto W_i^C y X_i es pequeña. Por esta razón, para este ataque, cuando las salidas de las redes que se quieren interceptar coinciden entre sí, pero difieren de la atacante, se cambia la salida oculta cuyo peso está más cerca del hiperplano de la entrada antes de aplicar el aprendizaje habitual con la salida de A.

Para aplicar este ataque, se comienza con una red, C, con pesos aleatorios y en cada paso se actualiza según 3 posibles escenarios:

1. $o^A \neq o^B$**:** En este caso las redes A y B no cambian por lo que la atacante tampoco.

2. $o^A = o^B = o^C$**:** Se actualiza C de la foma habitual.

3. $o^A = o^B \neq o^C$**:** En primer lugar busca $i_0, , i_0 = indmin|\sum_{j=0}^{N} w_{ij}^C \cdot x_{ij}|$ y actualiza la salida oculta de C $o_{i_0}^C = -o_{i_0}^C$. Entonces se actualiza con o^A.

Como el ataque cuenta con que C y A están cerca, mejora mucho cuando se utilizan varias redes con pesos aleatorios distintos, ya que así es más posible que en efecto una de ellas esté cerca y se sincronice con A antes de que lo haga B.

5.3.2. Mejoras a AP

Basándose en los ataques anteriores, se han buscado mejoras en el criptosistema que lo fortalezcan frente a estos. Estos cambios tratan de acelerar la sincronización de las redes para reducir los datos con los que trabajar en los ataques y, por tanto, dificultarlos o complicar más las salidas, ya sea con ruido o retardos para añadir una capa extra de seguridad.

La primera propuesta de mejora la dan los autores originales de este método, esta consiste en combinarla con un mapa caótico. Antes de tomar el signo para las salidas ocultas, se aplica un mapa caótico para crear ruido en la salida final. En concreto, proponen usar un mapa logístico $s(t+1) = \lambda(1-\beta)s(t)(1-s(t)) + \beta\tilde{h_k}(t)$ (dados $\lambda, \beta \in \mathbb{R}$), con $\tilde{h_k}(t)$ la normalización de $h(t) = \sum_{j=0}^{N} w_{kj}x_{kj}$, de forma que las salidas ocultas quedan expresadas como $o_k^A = sign(s^A(t)$. En este modelo λ, β y $s(0)$ serían públicos y λ y β se deben coger de forma que aseguren las convergencia de la redes.

Otro de los aspectos de la estructura que influye en su fortaleza es el uso de pesos discretos o continuos, ya que se ve en [9] que los discretos suponen una velocidad de sincronización mayor. De igual forma, como se comentó en el apartado sobre la sincronización del AP, también la normalización de los pesos acelera la sincronización, mejorando su seguridad.

Finalmente, el aspecto que más interés está generando en los últimos años es el método de comprobar la sincronización de las redes. La propuesta más reciente, es la dada en 2022 en [17], donde emplean una función hash calculada por emisor y receptor con los pesos de su red y las salidas de ambas para evaluar la probabilidad de que ambas redes estén sincronizadas. Este método muestra una buena mejora en su eficacia frente a los ataques anteriores; además, el autor plantea que, para hacerlos lo suficientemente robustos como para permitir su uso, se debe trabajar en seguir acelerando la sincronización de las redes.

6. Análisis de resultados del método AP

Como se ha descrito en el apartado anterior, el árbol de paridad es el método más destacado en este campo y se han analizado diversas formas de criptoanálisis para el mismo.

Para analizar los resultados de este método frente a diversos ataques, se ha implementado una librería RNA (Anexo II), basado en la programación orientada a objetos, que permita incorporar de forma estructurada nuevas capas y funciones y ser usadas de forma homogénea por los tests, facilitando el análisis de resultados.

La librería cuenta con una clase abstracta para las capas, a partir de la que crear los tipos de capas que pueden componer las redes. De la misma forma, dispone de otra clase abstracta para las funciones, obligando así a que todas las que se usen en esta librería tengan una función que devuelva su derivada. Finalmente, tiene una clase RNA con los algoritmos de propagación hacia delante y propagación hacia atrás ya implementados. Además de servir para crear objetos RNA de propagación hacia delante, esta clase sirve como base para extenderla en otro tipo de redes.

La propuesta de estructura de RNA para criptografía más relevante es el árbol de paridad (AP), por ello se incluye un ejemplo de su implementación en el Anexo III y se muestra una prueba de su fortaleza frente a un ataque por fuerza bruta, geométrico y genético en el siguiente apartado.

6.1. Resultados AP original

Para obtener un resultado compresivo y significativo, tomamos un AP pequeño. En concreto, uno que tiene 5 neuronas en la capa de entrada, representadas por pesos de dimensión 10 acotados por 10, es decir, que nuestras entradas serán matrices de 5×10 y los pesos oscilarán entre -10 y 10. En esta primera subsección vamos a comprobar el funcionamiento de la propuesta original del AP, para luego poder compararla con la mejora de incorporar un mapa logístico.

6.1.1. Sincronización de 2 redes

En primer lugar, ya que el funcionamiento de este algoritmo depende de su habilidad para sincronizar dos redes, comenzamos probando que efectivamente se sincronizan y en cuanto tiempo. Al inicializar los pesos de las redes de forma aleatoria, los resultados de cada ejecución son diferentes; sin embargo, todas las pruebas realizadas se han llegado a sincronizar en tiempos similares (una media de 0.24 segundos).

Aparte de conseguir que se sincronicen, resulta de interés ver la eficacia con la que lo hacen, es decir, cuánto más próximas se encuentran en cada paso. En la Figura 9, se puede ver el gráfico de una de las ejecuciones con el porcentaje de sincronización de las

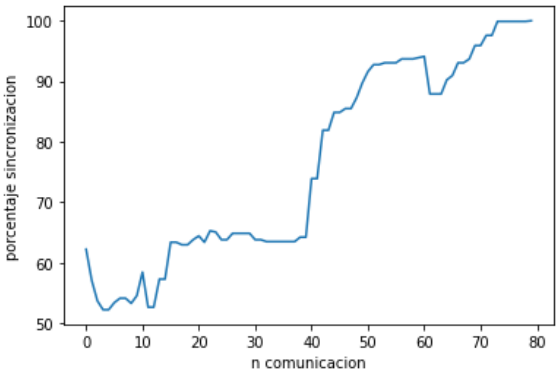
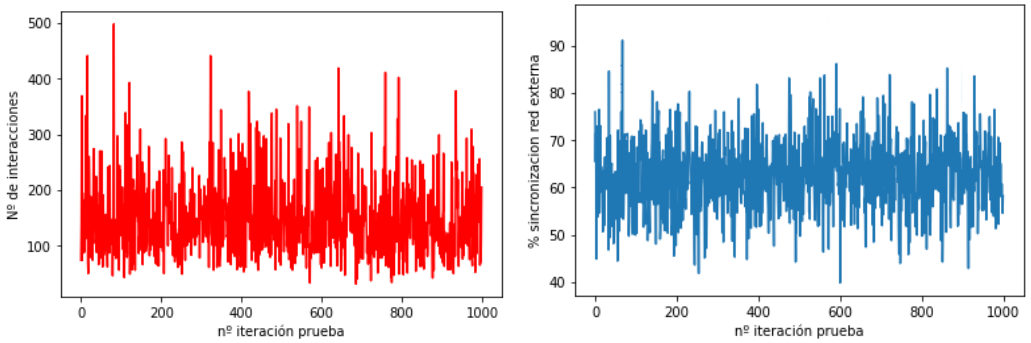


Figura 9: Sincronización de los árboles.

redes en cada iteración. Destacan los pasos en los que la sincronización se reduce, pasos repulsores, aunque la tendencia global sea en aumento. Como se había mencionado en el apartado sobre la sincronización de estas redes, es inevitable que se den este tipo de pasos. Sin embargo, según avanzan las interacciones, también lo hace la tendencia a tener pasos atractores; como podemos comprobar en la imagen, donde, desde la interacción número 15 apenas hay pasos repulsores.

6.1.2. Ataque por fuerza bruta

Una vez comprobada la sincronización, veamos que, hasta en su forma más básica, esta estructura es resistente frente a un ataque por fuerza bruta. En este caso, este ataque consiste en que una red externa intente sincronizarse con las salidas que las que se están comunicando comparten por un canal público. Igual que con la sincronización, la aleatoriedad en la inicialización de los pesos implica que tardan en sincronizarse un tiempo distinto en cada ejecución y también la red externa alcanza un grado distinto de sincronización. Respecto al porcentaje de sincronización de la red externa, si bien oscila considerablemente, en ninguna de las ejecuciones ha conseguido sincronizarse.



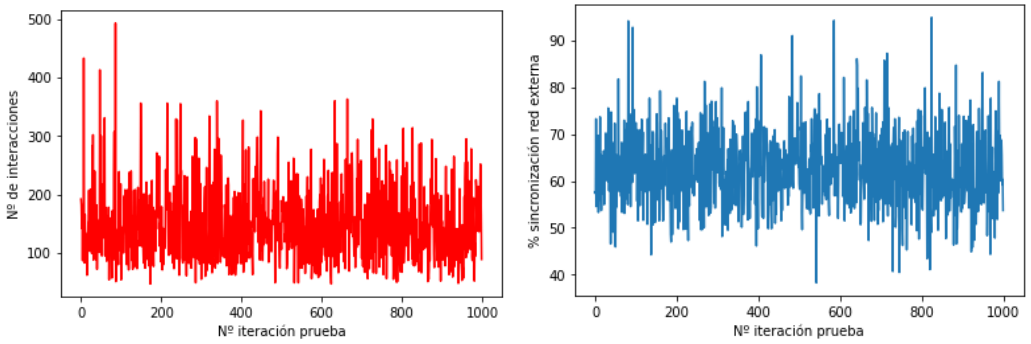
(a) Número de interacciones (b) % de sincronización red externa

Figura 10: Gráficos fuerza bruta 1000 ejecuciones

En el gráfico 10, a la derecha, se muestra el número de interacciones requeridas por las redes para sincronizarse en cada una de las 1000 ejecuciones realizadas para esta prueba, que han tenido una media de 148 interacciones. En el gráfico de la izquierda están los distintos porcentajes de sincronización obtenidos en las ejecuciones. Si bien en algunos casos se aproxima mucho, la media de estos porcentajes es un 62.45 % y se puede observar, al verlo junto con el gráfico del número de interacciones, que estos números altos se dan cuando las redes que se están comunicando tardan más en sincronizarse. Como se ha comentado anteriormente, el número de interacciones requeridas para sincronizarse tiene una gran relevancia en la fortaleza del método AP, y en este ejemplo podemos ver como un número mayor de las mismas, le afecta negativamente.

6.1.3. Ataque geométrico

Para el ataque geométrico, se ha creado una capa modificada para la primera de la red. El cambio introducido es que en el entrenamiento, la modificación de los pesos sigue la regla del ataque geométrico. Eso quiere decir que cuando la salida de la red no coincide con las que se intenta sincronizar busca la neurona cuyos pesos tienen menor distancia al hiperplano de la entrada y modifica solo estos. Para ver el efecto de la red, primero vamos a comprobar su ejecución con sólo una red intentando sincronizarse y luego vamos a hacerlo con 10, ya que se suele recomendar aplicar este algoritmo con más de una red.

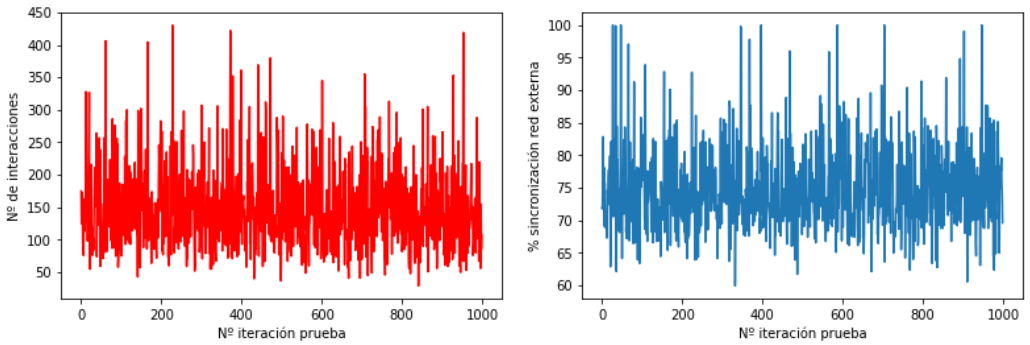


(a) Número de interacciones (b) % de sincronización red externa

Figura 11: Gráficos ataque geométrico, 1 red, 1000 ejecuciones

En el gráfico 11 de la izquierda están los distintos porcentajes de sincronización de la red en cada ejecución cuya media es un 63.5 %. Al depender de que los pesos estén cerca de los de la red con la que se intentan sincronizar, este ataque solo resulta realmente eficaz si se aplica con varias redes al mismo tiempo, aumentando así la probabilidad de que alguna esté cerca. Con una sola red, la media de sincronización no tiene un aumento relevante respecto al ataque por fuerza bruta.

En la figura 12 a la izquierda se, para cada ejecución, el porcentaje de sincronización de la red que mayor valor obtuvo, con media 74.5 %. En este caso, no solo el porcentaje



(a) Número de interacciones (b) % de sincronización red externa

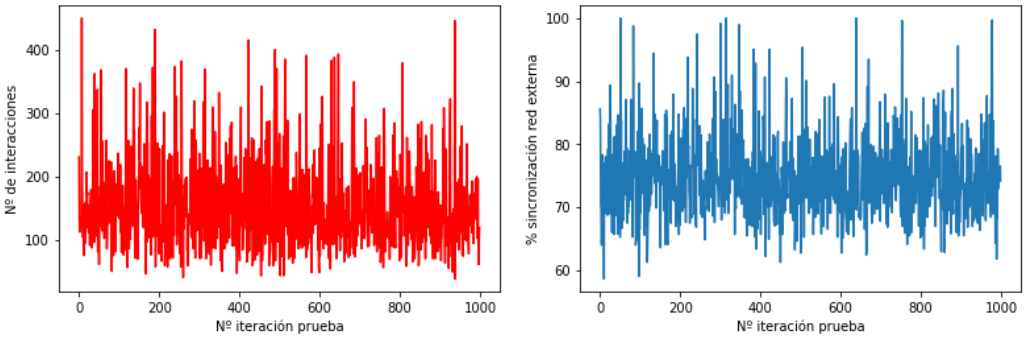
Figura 12: Gráficos ataque geométrico, 10 redes, 1000 ejecuciones

medio de sincronización es considerablemente mayor, sino que ha conseguido sincronizarse en 6 ocasiones. Aunque 6 de 1000 no es un valor excesivamente alto, si que es suficiente para hacer que la aplicación en sistemas reales de este sistema criptográfico no sea posible. Esta considerable mejora respecto a la aplicación con 1 sola red, conlleva un aumento en el coste computacional. En el caso de 1 sola red, el tiempo medio por sincronización ha sido de 0,15 segundos, mientras que con 10 redes ha sido de 0,75.

6.1.4. Ataque genético

Para el ataque genético, definimos una tamaño inicial de población y un máximo y en cada paso en el que las redes que se intentar interceptar han coincidido, y por tanto cambian sus pesos, aplica la regla de aprendizaje del ataque genético. Esta regla consiste en, si el número de redes aún no supera el máximo, multiplicar las redes según todas las posibles configuraciones y, si las supera, aplicar la regla habitual, eliminando las redes cuya salida no coincida con las redes con las que tratan de sincronizarse. Para la prueba hemos definido un tamaño inicial de 10 redes y un máximo de 200. Al tener que estar calculando la salida de tantas redes, el tiempo requerido por este algoritmo es mucho mayor, requiriendo 15 segundos en cada interacción mientras que el geométrico con 10 redes solo necesitaba 0.75. Aparte, este aumento no va ligado a una mejora de efectividad, ya que la media de sincronización de las redes ha sido muy similar a la del geométrico con 10 redes. Esto ha evidenciado lo ya comentado sobre la inviabilidad de este algoritmo para redes grandes.

En el gráfico de la izquierda de la figura 13, se ven, para cada ejecución, el porcentaje de sincronización de la red de la población cuyo valor era mayor, la media de estos porcentajes es un 74.8 %. Destaca, en primer lugar, que incluso los valores más bajos de estos porcentajes están por encima de la media de los ataques por fuerza bruta. En segundo lugar, destaca que ha habido 3 ocasiones en las que han sido capaces de sincronizarse con las redes. Es decir, en 3 de las 1000 interacciones ha roto el cifrado. Si bien este número no es muy alto, si que es suficientemente relevante como para no permitir el uso de este



(a) Número de interacciones (b) % de sincronización red externa

Figura 13: Gráficos ataque genético 1000 ejecuciones

cifrado en sistemas reales.

6.2. AP con mapa caótico

Con el fin de poder comparar los resultados con el AP original, volvemos a tomar la misma estructura para los árboles en estas pruebas. Recordamos que esto es, un árbol con 5 neuronas en la capa de entrada, con pesos de dimensión 10 acotados por 10.

6.2.1. Sincronización de 2 redes

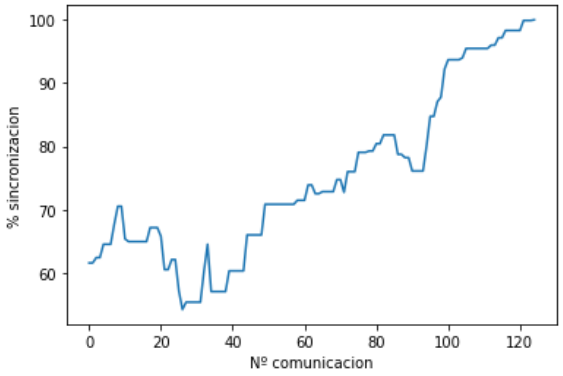
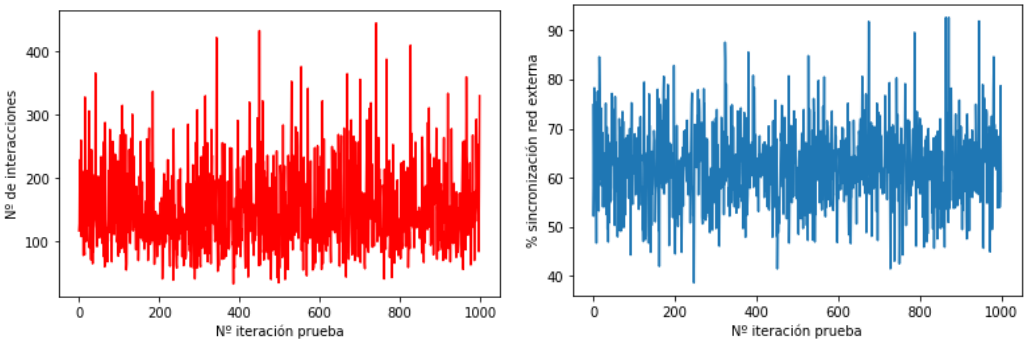


Figura 14: Sincronización de los árboles con el mapa logístico.

La introducción del mapa logístico antes del cálculo del signo hace aumentar los pasos repulsores, haciendo requerir más interacciones entre las redes para sincronizarse. Es por este aumento en el número de interacciones que la mejora frente a los ataques no va a ser excesiva y se requiere continuar trabajando en la velocidad de sincronización de las redes.

6.2.2. Ataque por fuerza bruta

Con una media de un 60.2 % de sincronización con la red, para este ataque no muestra una gran mejora. Este ataque ya no suponía una amenaza para la versión orignal de la red, por lo que mientras siga sin romperla, no es relevante que mejore o no su efectividad.



(a) Número de interacciones (b) % de sincronización red externa

Figura 15: Gráficos fuerza bruta 1000 ejecuciones, AP con mapa logístico.

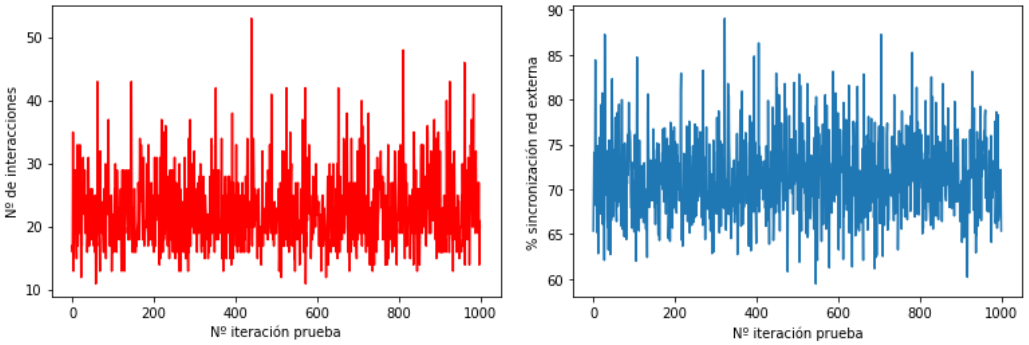
6.2.3. Ataque geométrico

Habiendo comprobado antes lo considerablemente mejor que funciona este ataque con más redes, lo aplicamos directamente con 10 redes atacantes. En la figura 16, en la izquierda se ve el porcentaje de sincronización con la red, con media 74.6 %. Destaca que en este caso no fue capaz de interceptar las redes en ningún caso, a pesar de estar considerablemente cerca en alguna ocasión. Por lo cerca que ha quedado en 2 de las iteraciones, parece razonable considerar que de continuar haciendo pruebas, alguna consiguiera romperlo. Vemos con esto que esta modificación mejora la fortaleza frente a este ataque, que el más eficaz frente a la original.

6.2.4. Ataque genético

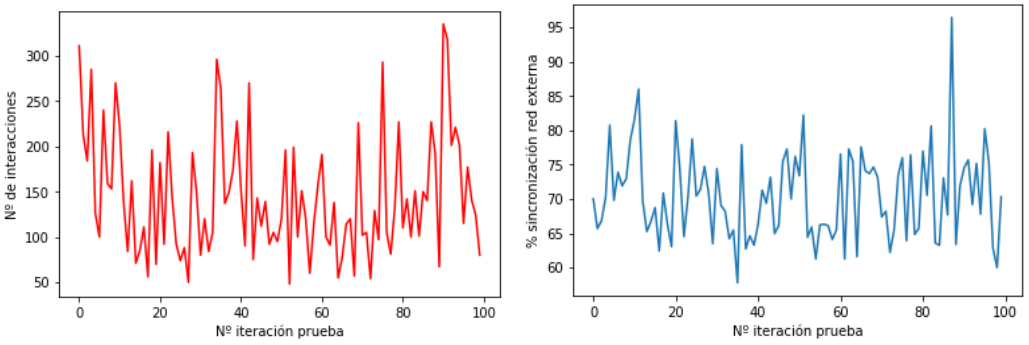
Debido al alto coste computacional, en este caso este ataque se comprobará con 100 ejecuciones, con los mismos parámetros de antes, 10 redes iniciales y 200 máximas. El gráfico de la izquierda de la figura 17, muestra el porcentaje de sincronización con la red, con media 70.4 %. En ninguna de las ocasiones ha llegado ha romperlo, aunque en una ha llegado al 90 %, por lo que igual que con el geométrico, cabe esperar que realizando suficientes pruebas, alguna llegara a sincronizarse.

Vistos todos los ataques, queda comprobada la mejora en fortaleza que da el mapa logístico al sistema. Además, ya que los ataques más eficaces siguen teniendo alguna posi-



(a) Número de interacciones (b) % de sincronización red externa

Figura 16: Gráficos ataque geométrico 1000 ejecuciones, AP con mapa logístico.



(a) Número de interacciones (b) % de sincronización red externa

Figura 17: Gráficos ataque genético 100 ejecuciones, AP con mapa logístico.

bilidad de romperlo, se reitera la necesidad de continuar trabajando en mejorar la velocidad de sincronización de las redes.

7. Conclusión

La criptografía con redes neuronales supone una prometedora alternativa a la criptografía tradicional, basada en teoría de números, que predomina actualmente. Sin embargo, se considera necesario seguir avanzando en este campo para hacer posible su uso extendido.

Se han presentado algoritmos basados en RNA usadas en criptografía para el cifrado de texto e imágenes, la generación de secuencias pseudoaleatorias para usar como claves y la mejora de otros criptosistemas. Todos tienen en común el uso de una red con comportamiento caótico para mejorar su seguridad y evitar que una red externa copie el comportamiento de la RNA utilizada para el descifrado.

Para los algoritmos de clave privada, se ha comprobado que la distribución de caracteres que hacen y su aleatoriedad son buenas. Sin embargo, no parece que la mejora que ofrecen sea suficientemente sustancial como para justificar mayor investigación en su fortaleza frente al criptoanálisis. Para posibilitar su uso en sistemas reales se estiman necesarias más pruebas que justifiquen su seguridad frente a ataques.

De entre los algoritmos propuestos destaca el árbol de paridad (AP); que se trata de un algoritmo asimétrico de clave pública con muchos trabajos publicados sobre el mismo, para el que, en este trabajo, se ha justificado la sincronización de redes y se han presentado ataques eficientes y áreas de mejora.

Al ser el algoritmo más relevante, para analizar su fortaleza frente a un ataque por fuerza bruta, geométrico y genético se ha hecho una implementación en *Python* del mismo. En las pruebas realizadas se ha podido observar que la estructura original es bastante fuerte frente a estos ataques. El ataque que más veces ha conseguido romperlo has sido el geométrico, con un 6 de 1000 para la estructura original. En la estructura con mapas caóticos, el más eficaz vuelve a ser el geométrico, aunque en este caso no se sincronizó con las redes en ninguna de las 1000 iteraciones. A pesar de no haberse sincronizado, estuvo cerca en varias ocasiones, por lo que se puede deducir, que incrementando el número iteraciones, podría llegar a romperlo.

Se concluye que entre las mejoras necesarias para fortalecer el AP frente a los ataques capaces de obtener su configuración con los datos públicos, destaca encontrar métodos para acelerar su sincronización. En este sentido, de las aportaciones más recientes, destaca la aportada en 2022 por Sarkar para descubrir la sincronización de las redes antes[17], donde concluye que la clave para que estos sistemas se puedan emplear de forma segura es reducir el número de interacciones necesarias para la sincronización.

A. Anexo I: Redes Neuronales

Las Redes de Neuronas Artificiales(RNA) son un modelo de aprendizaje automático a partir de ejemplos basados en la estructura y el funcionamiento del sistema nervioso. El aprendizaje, en vez de utilizar la descripción de un algoritmo, se realiza usando la información recibida, procesada según indique la estructura de la red, para averiguar los pesos de cada elemento en la función final. Los modelos varían según las reglas de aprendizaje, las funciones de activación y su diseño (el número de neuronas, sus conexiones, las capas, etc.).

En términos generales se puede describir una RNA como un conjunto de neuronas, que reciben unas entradas y dan unas salidas, conectados entre sí. Cada neurona consiste en una función, llamada función de activación, que se aplica sobre sus entradas ponderadas según unos pesos, que son los que se suelen modificar en el aprendizaje. También se les puede sumar a las entradas un valor, referido como sesgo. La expresión del efecto que tiene en una entrada x el paso por una neurona con función activación σ, vector de pesos w y sesgo b es $\sigma(x \cdot w + b)$.

La elección de la función de activación resulta muy relevante, ya que, junto con la propia estructura de la red, determina el alcance de los resultados y una mala elección podría hacer que la red no fuera capaz de funcionar correctamente. Algunas de las funciones más comunes son:

- Sigmoide: $\sigma_1(x) = \frac{1}{1+e^{-x}}$
- Tangente hiperbólica: $\sigma_2(x) = \frac{1-e^{-x\cdot\lambda}}{1+e^{-x\cdot\lambda}}$
- ReLu: $\sigma_3(x) = \begin{cases} x & si \quad x \geq 0 \\ 0 & si \quad x < 0 \end{cases}$
- Swish: $\sigma_4(x, \lambda) = \frac{x}{1+e^{-x\cdot\lambda}}$ y Logística: $\sigma_4(x, 1)$
- Paso Binario: $\sigma_5(x) = \begin{cases} x & si \quad x \geq \lambda \\ 0 & si \quad x < \lambda \end{cases}$
- Paridad: $\sigma_6(x) = sign(x)$

A.1. Clasificación de las RNA

Una de las clasificaciones que se puede hacer de las RNA es según si el flujo de datos por la red es lineal o no, llamadas redes con Propagación hacia delante (*Feedforward*) o redes Recurrentes respectivamente.

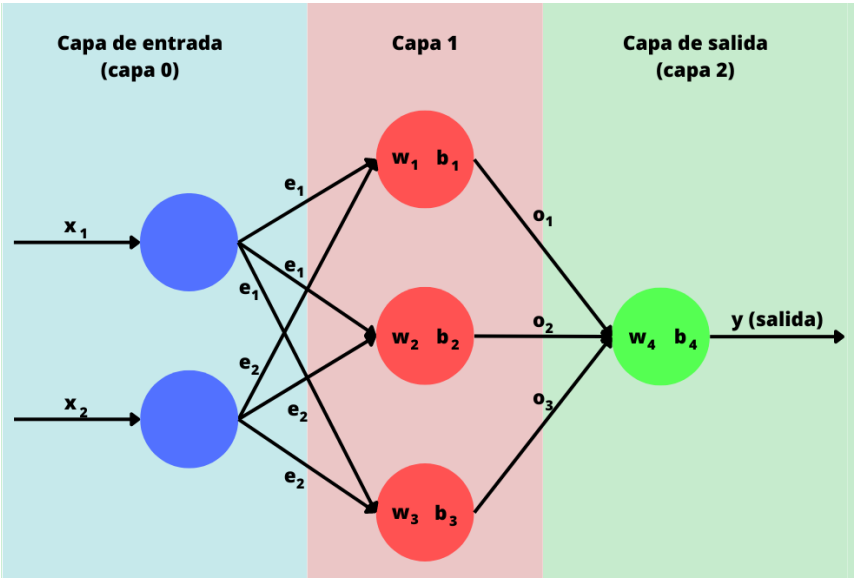


Figura 18: RNA con propagación hacia delante de 3 capas con 2 entradas y 1 salida

A.1.1. Redes con Propagación hacia delante

Las redes con propagación hacia delante, también llamadas perceptrones multicapa, se caracterizan por la transmisión lineal de sus datos. Esta característica nos permite agrupar las neuronas en capas según por cuantas han pasado las entradas antes de llegar a ellas, de forma que la capa 0 sería la capa que recibe los datos tal y como llegan a la red, la 1 a la que se envían los datos de la 0, etc. La capa 0 suele llamarse capa de entrada, la capa cuyas salidas son las de la red, capa de salida y al resto se les denomina capas ocultas. Al implementar las redes, como se suele utilizar la misma función activación en todas las neuronas de una capa, es habitual procesar las entradas por capas en vez de por neuronas, creando una matriz de pesos W y un vector sesgo B con los de las neuronas que componen la capa y aplicando la función sobre cada uno de los componentes del vector $x \cdot W + B$. Con esta convención, añadiendo subíndices para indicar la capa del peso y sesgo (capa 0: W_0, B_0, etc.), el resultado de aplicar la red sobre una entrada x se puede resumir con la función $F(x) = \sigma(W_N \cdot \sigma(...\sigma(W_1 \cdot x + b_1)...) + b_N)$.

Dentro de las redes con propagación hacia delante podemos encontrar más subtipos, de ellos destacan las **redes Convolucionales**, cuya característica distintiva es que las neuronas se especializan en cada paso. Esta especialización se consigue uniendo las neuronas con un subgrupo de las de la siguiente capa en vez de con todas ellas. El objetivo de esto es reducir el total de neuronas necesarias y la complejidad computacional de su ejecución. Su aplicación está principalmente orientada a tareas de clasificación, como puede ser los sistemas de recomendación, la clasificación y segmentación de imágenes...

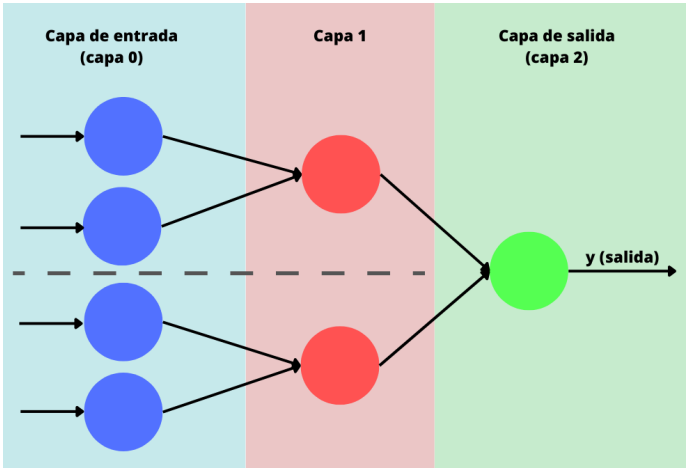


Figura 19: RNA convolucional con capa 1 especializada en 2 casos

A.1.2. Redes Recurrentes

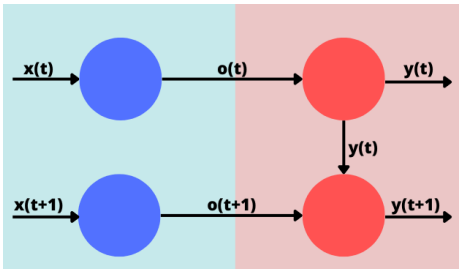


Figura 20: Ejemplo red recurrente

Unas de las limitaciones de las redes con propagación hacia delante es la presunción de independencia entre los ejemplares que procesa y la limitación en la longitud de los vectores a la definida en la capa de entrada. Para solucionar estos problemas aparecen las redes recurrentes(RNR), cuya principal característica es que cuentan con memoria, lo que les permite analizar series temporales. En estas redes las neuronas no solo transmiten la información a la siguiente, sino que pueden mandarla a una anterior o a ella misma. Es esta posibilidad de conectar las salidas de forma no lineal la que les da una memoria a corto plazo, ya que a la neurona no llegan solo los datos del paso anterior, sino que también recibe los de otras neuronas de pasos anteriores. Algunas de las aplicaciones de este tipo de redes son los traductores, reconocimiento de escritura, asistentes virtuales...

Las redes de este tipo que se tratan en el trabajo tienen la característica de que una vez reciben una entrada, sus neuronas se alimentan entre ellas hasta alcanzar la estabilidad. Para estas redes nos va a resultar de especial interés su dinámica, por lo que será habitual definirlas como el sistema discreto: $\partial x(k+1) = W\sigma(x(k))$, donde $x(t) = (x_1(t), ...x_n(t))^T$ es el vector de estados de las n neuronas de la red, W es la matriz de pesos de conexión en t y $\sigma = (\sigma_1, ..., \sigma_n)^T$ las funciones de activación de las neuronas.

Para hacer de estas redes sistemas continuos, se les pueden añadir componentes eléc-

tricos que mapeen el voltaje de entrada sobre el de salida. En este caso, se definen con una ecuación diferencial que en su forma más básica queda como: $\frac{\partial x(t)}{\partial t} = -x(t) + W\sigma(x(t)) + I(t)$, donde $x(t) = (x_1(t), ...x_n(t))^T$ es el vector de estados de las n neuronas de la red, W es la matriz de pesos de conexión en t,$\sigma = (\sigma_1, ..., \sigma_n)^T$ las funciones de activación de las neuronas e I es el voltaje de entrada.

Redes Neuronales de Hopfield

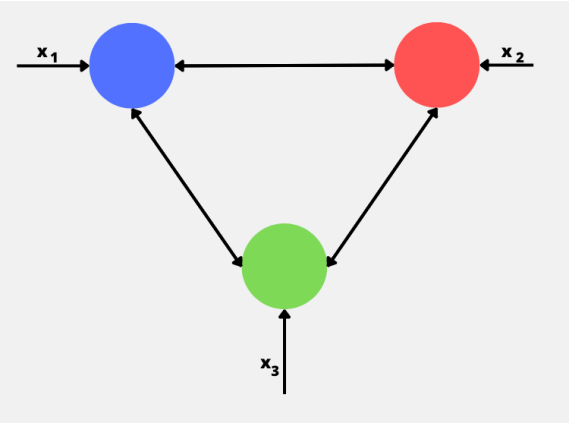


Figura 21: Ejemplo de red de Hopfield con 3 neuronas.

Un tipo muy habitual son las redes de Hopfield (RNH). Estas son redes recurrentes totalmente conectadas, es decir, que la salida de una neurona es entrada de todas las demás, cuya información se propaga hasta llegar a un estado estable. Tiene salidas binarias ({0,1}) o bipolares ({-1,1}), una matriz de pesos fija y el paso de una neurona a otra se decide según si la suma ponderada de la entrada es mayor o menor a un umbral predefinido. En este trabajo solo aparecen bipolares, por lo que en los casos en que difieran se describirán los procesos específicos de estas.

Para una red de Hopfield con n neuronas, utilizando la notación w_{ij} para referirnos al peso que conecta la salida de la neurona i con la entrada de la j, si quiere guardar los T patrones de entrada $P^t = [p_1^t, ..., p_n^t]$ (con $t \in \{1, ..., T\}$) los pesos se fijan en

$$w_{ij} = \begin{cases} \sum_{t=1}^{T} p_i^t \cdot p_j^t & si \quad i \neq j \\ 0 & si \quad i = j \end{cases}$$

Si se recibe una entrada $y = [y_1, ..., y_n]$, definiendo la salida de la neurona i en el paso t como $x_i(t)$, la evolución de las salidas sería:

$$x_i(t) = \begin{cases} y_i & si \quad t = 0 \\ 1 & si \quad t > 0 \ y \ \sum_{j=1}^{n} w_{ji} x_j(t-1) > \theta_i \quad i \in \{1, ..., N\} \\ -1 & si \quad t > 0 \ y \ \sum_{j=1}^{N} w_{ji} x_j(t-1) \leq \theta_i \end{cases}$$

La red continúa calculando salidas hasta que alcanza la estabilidad ($x_i(k) = x_i(k-1) \forall i \in \{1, ..., N\}$). En la práctica, no se tienen que actualizar los valores de todas las neuronas simultáneamente, y se pueden alterar secuencialmente.

Ejemplo A.1. *Una red de Hopfield con 4 neuronas con los patrones* $p_1 = [1, 1, 1, -1]$ *y* $p_2 = [-1, -1, -1, 1]$ *tendría como matriz de pesos* $W = \begin{pmatrix} 0 & 2 & 2 & -2 \\ 2 & 0 & 2 & -2 \\ 2 & 2 & 0 & -2 \\ -2 & -2 & -2 & 0 \end{pmatrix}$.

Para la entrada $[-1, -1, -1, -1]$ *con umbrales 0 el proceso que seguiría es:*

$t = 1 :\ (-1 \ -1 \ -1 \ -1) \quad W = (-2 \ -2 \ -2 \quad 6) \Rightarrow x(1) = (-1 \ -1 \ -1 \quad 1)$

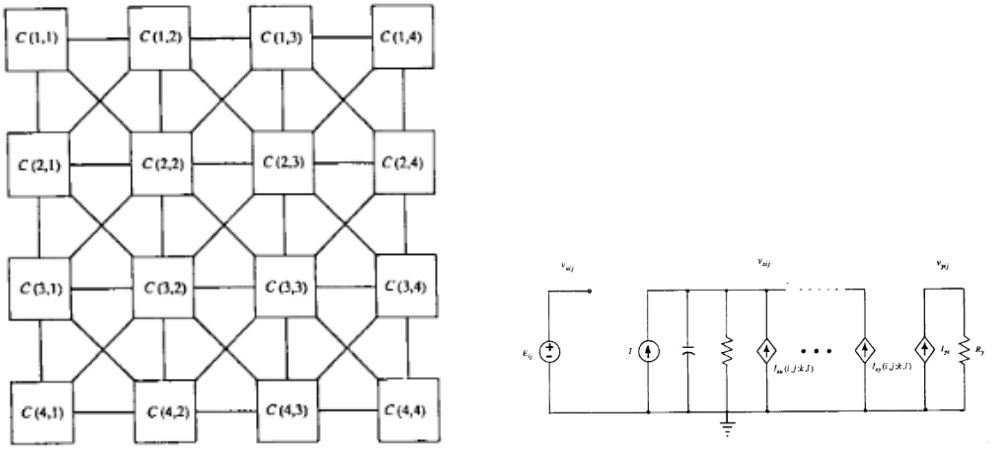
$t = 2 :\ (-1 \ -1 \ -1 \quad 1) \quad W = (-6 \ -6 \ -6 \quad 6) \Rightarrow x(2) = (-1 \ -1 \ -1 \quad 1)$

En el último paso coinciden por lo que pararía el proceso concluyendo que el patrón más cercano a la entrada es p_2.

Redes Neuronales Celulares

Propuestas por Chua y Yang en 1988 [3, 4], las redes neuronales celulares (RNC) son un modelo más general de las redes de Hopfield con procesamiento paralelo. Esto último se refiere a que las salidas de cada célula (nombre con el que se refieren en este modelo a las neuronas) se procesan simultáneamente, haciendo la velocidad de operación considerablemente más rápida. Su principal diferencia con las redes de Hopfield es que estas están localmente conectadas, de forma que la salida de cada célula está condicionada por las entradas y salidas de sus vecinas. Habitualmente el rango para determinar si las células son vecinas es 1, pero la definición genérica de las de C_{ij} para un rango r es $N_{ij} = \{C_{kl} : max(|k-i|, |l-j|) \leq r,\ 1 \leq k \leq M;\ 1 \leq l \leq N\}$. Estas redes se definen de forma continua, ya que la implementación de sus neuronas es siempre como circuitos. El funcionamiento de una célula C_{ij} fue descrito por el circuito en la imagen siguiente, cuya dinámica se puede representar como $\frac{\partial x_{ij}(t)}{\partial t} = -d_{ij}x_{ij}(t) + \sum_{C_{kl} \in N_{ij}} A(i, j; k, l)y_{kl}(t) + \sum_{C_{kl} \in N_{ij}} B(i, j; k, l)y_{kl} + I_{ij}$ donde:

- d_{ij} es un sesgo predefinido,
- I_{ij} es la entrada de la célula C_{ij},
- $A(i, j; k, l)$ son los pesos de la salida de C_{kl} en C_{ij},
- $B(i, j; k, l)$ son los pesos de la entrada de C_{kl} en C_{ij},
- x_{ij} es el estado de la célula C_{ij},



(a) RNC 4 × 4 [4]

(b) Circuito de una célula (RNC) [3]

Figura 22: Esquemas Redes Neuronales Celulares

- y_{ij} es la salida de la célula C_{ij}, dada por $y_{ij} = f(x_{ij}) = \frac{1}{2}(|x_{ij}(t)+1| - |x_{ij}(t)-1|)$

Redes Neuronales Q'tron

En las redes Q'tron cada neurona tiene 2 modos de funcionamiento, libre o fijo según si se pueden cambiar las salidas o no. En cada paso se selecciona una única neurona de entre las libres para cambiar su salida que, definiendo los estímulos de la neurona como $H_i = \sum_{j=1}^{n} w_{ij}(a_j y_j)$ (con n el número total de neuronas), se expresa como:

$$y_i(t+1) = \begin{cases} y_i(t)+1 & si \quad H_i > \frac{1}{2}|w_{ii}a_i| \ y \ y_i(t) < q_i - 1 \\ y_i(t)-1 & si \quad H_i < \frac{-1}{2}|w_{ii}a_i| \ y \ y_i(t) > 0 \\ y_i(t) & sino \end{cases}$$

donde $y_I \in \{0, 1, ..., q_i - 1\}$ es la salida de la neurona i, a_i es el peso activo de la neurona i y w_{ij} es el peso de conexión entre las neuronas i y j.

Los problemas de este tipo de redes se definen como minimización de su función energía:

$$\epsilon = \frac{-1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i y_i w_{ij} a_j y_j \ + K$$

con K constante elegida a conveniencia. Se definen como neuronas fijas aquellas cuyos datos ya conocemos y el resto se dejan libres, obteniendo la solución cuando las libres se estabilizan.

A.2. Entrenamiento

Podemos dividir el aprendizaje de cualquier modelo de aprendizaje automático en 3 fases:

1. **Preparación de los datos:** Asegurarse de que los datos están en el formato que requiere la red, es decir, el rango de los valores, la cantidad, etc.
2. **Entrenamiento o Aprendizaje:** Utilizar un conjunto de ejemplos para que la red obtenga los datos necesarios para generalizar y cumplir su función.
3. **Validación:** Comprobar si la red tiene el funcionamiento deseado. Para ello, normalmente se deja una parte del conjunto de ejemplos de entrenamiento sin usar, que luego se emplean en la validación.

La fase de entrenamiento es la más importante, ya que de ella depende el funcionamiento de la red. Cuando el conjunto de entrenamiento está etiquetado con las salidas esperadas, clasificamos la tarea como de aprendizaje supervisado y el caso en el que se pretende que sea la red la que defina las características útiles para definir las salidas, como aprendizaje no supervisado. Dentro del aprendizaje supervisado, el entrenamiento está orientado a reducir el índice de error, que consiste en una función de error y un término de regularización. Este último tiene el propósito de evitar que la red se ajuste en exceso a los ejemplos, haciéndola poco útil para ejemplares nuevos, a lo que se suele referir como *overfitting*.

Para manejar varios ejemplos de entrenamiento simultáneamente se suele hacer la media de su coste; sin embargo, cuando se manejan conjuntos muy grandes es habitual emplear métodos alternativos que permiten no tener que analizar todos los ejemplares. De estos métodos, el estocástico es el más usado; en él, se escoge en cada iteración un elemento de forma aleatoria y se computa el aprendizaje con este solo. Otra opción es crear grupos más pequeños (*mini-batch*) en cada iteración, también eligiendo aleatoriamente los elementos de cada grupo.

A.2.1. Descenso gradiente

En las redes neuronales, el método más habitual para aplicar el aprendizaje supervisado es con el descenso gradiente. Este permite calcular los valores de parámetros que minimizan el error en la red usando el gradiente de la función error.

Para la descripción del método, llamamos a los parámetros de la red (sus pesos y sesgos) $\alpha \in \mathbb{R}^N$, η a la tasa de aprendizaje y a la función de error $E : \mathbb{R}^N \to \mathbb{R}^N$. En primer lugar, aprovechamos la expresión como serie de Taylor de $E(\alpha + \Delta\alpha) \approx E(\alpha) + \sum_{k=1}^{N} \frac{\partial E(\alpha)}{\partial \alpha_k} \cdot \Delta\alpha_k$ para $\Delta\alpha$ suficientemente pequeños. Esta aproximación también se puede expresar con el gradiente como: $E(\alpha) + \nabla E(\alpha)^T \cdot \Delta\alpha$. Añadimos el subíndice i para distinguir el número

de iteración en α, de forma que $\alpha_{i+1} = \alpha_i + \Delta\alpha$, nuestro objetivo es reducir el error, es decir, $E(\alpha_{i+1}) < E(\alpha_i)$, que expresado en términos de la aproximación anterior queda como $E(\alpha_i)+\nabla E(\alpha_i)^T \cdot \Delta\alpha < E(\alpha_i)$. Por lo tanto, nuestro objetivo es maximizar $-\nabla E(\alpha)^T \cdot \Delta\alpha$, lo cual conseguimos a través de la desigualdad de Cauchy–Schwarz que afirma que para cualquier par de funciones f y g se cumple que $|f^T \cdot g| \leq ||f|| \cdot ||g||$; aplicada a nuestro caso ($-|\nabla E(\alpha)^T \cdot \Delta\alpha| > -||\nabla E(\alpha)^T|| \cdot ||\Delta\alpha||$) esto nos dice que el máximo se da cuando $\Delta\alpha = -\nabla E(\alpha)$. Dado que $\Delta\alpha$ debe ser pequeña, se le añade la variable η, a la que nos referiremos en adelante como tasa de aprendizaje, de forma que una iteración consiste en cambiar los parámetros de la siguiente forma: $\alpha_{i+1} = \alpha_i - \eta \cdot \Delta E(\alpha_i)$.

A.2.2. Propagación hacia atrás

Uno de los inconvenientes que presenta el método de descenso gradiente es que tener que calcular las derivadas parciales de todos los parámetros de la red resulta muy costoso computacionalmente. Para solucionar este problema, el método más popular es el de Propagación hacia atrás (*backpropagation*), que consiste en avanzar desde la última capa utilizando los valores de cada una para calcular los de la anterior.

Para explicar este mecanismo en más detalle, primero es necesario numerar las n_j neuronas de cada capa j ($j \in \{1, ..., C\}$), distinguir la salida antes de aplicar la función activación en la neurona i de la capa j como o_i^j e introducir la notación $\delta_i^j = \frac{\partial E}{\partial o_i^j}$. Con esta notación, la salida de la capa j sería $s^j = \sigma(o^j)$, donde $o^j = (o_1^j, ..., o_{n_j}^j)$. Este método solo resulta útil para redes con propagación hacia delante, ya que hace uso de su estructura de capas, por lo que podemos expresar o^j como$o^j = o^{j-1} \cdot W^j + B^j$ para $j \in \{2, .., C\}$.

La primera parte del funcionamiento del método es que, aplicando la regla de la cadena, podemos obtener una representación de las derivadas parciales que buscamos en función de δ_i^j y la salida de la neurona:

$$\frac{\partial E}{\partial w_{ik}^j} = \frac{\partial E}{\partial o_i^j} \cdot \frac{\partial o_i^j}{\partial w_{ik}^j} = \delta_i^j \cdot s_i^{j-1} \frac{\partial E}{\partial b_i^j} = \frac{\partial E}{\partial o_i^j} \cdot \frac{\partial o_i^j}{\partial b_i^j} = \delta_i^j \cdot 1$$

La segunda parte, y la que le da su nombre, es la expresión de cada δ_i^j en función del de la capa anterior. Para ello primero tratamos el caso de la última capa en la que aplicando la regla de la cadena obtenemos:

$$\delta_i^C = \frac{\partial E}{\partial o_i^C} = \frac{\partial E}{\partial s_i^C} \frac{\partial s_i^C}{\partial o_i^C} = \frac{\partial E}{\partial s_i^C} \sigma'(o_i^L)$$

Para el resto de capas j ($j \in \{1, .., C-1\}$) volviendo a aplicar la regla de la cadena,

tenemos

$$\delta_i^j = \sum_{k=1}^{n_j+1} \frac{\partial E}{\partial o_k^{j+1}} \frac{\partial o_k^{j+1}}{\partial o_i^j} = \sum_{k=1}^{n_j+1} \delta_k^{j+1} \frac{\partial o_k^{j+1}}{\partial o_i^j}$$

Centrándonos en $\frac{\partial o_k^{j+1}}{\partial o_i^j}$, si sustituimos o_k^{j+1} por su expresión en función de o_k^j se puede transformar en:

$$\frac{\partial(\sum_m w_{km}^{j+1}\sigma(o_m^j) + b_m^{j+1})}{\partial o_i^j} = \frac{\partial w_{kj}^{j+1}\sigma(o_i^j)}{\partial o_i^j} = w_{kj}^{j+1}\sigma'(o_i^j)$$

. Con lo que sustituyendo en la expresión anterior y reordenando los términos tenemos $\delta_i^j = \sigma'(o_i^j)((W^{j+1})^T\delta^{j+1})_i$.

Con estas fórmulas, el método permite calcular todas las derivadas parciales necesarias para el descenso gradiente en cada capa a partir de los datos de la anterior, reduciendo el tiempo y el coste computacional.

B. Anexo II: Halftoning con Q'tron

Figura 23: Ejemplo halftoning [21].

Halftoning se refiere a transformar una imagen en tonalidades grises a binario (normalmente 0: blanco, 1: negro), de forma que se siga pareciendo a la original.

El mecanismo propuesto en [20] para transformar la imagen en grises G de tamaño $M \times N$ en su *halftone* H usa un Q'tron por pixel de cada imagen. Llamamos $\mu_{ij}^G \in \{0, ..., 255\}$ al pixel en las coordenadas $i \times j$ de la imagen en grises y $\mu_{ij}^H \in \{0, ..., 255\}$ al pixel en las coordenadas $i \times j$ de la imagen en grises. Definimos las activaciones de G como 1 y las de H como 255.

La función energía para este proceso se define como una minimización de la diferencia de oscuridad en los píxeles de una zona de la imagen en grises y la que queremos generar:

$$\epsilon_1 = \frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N}(\sum_{(k,l)\in N_{ij}^r}(a^H y_{kl}^H - a^G y_{kl}^G))^2$$

donde el vecindario se define en función de un radio r como $N_{ij}^r = \{(k,l) : |i-k| \leq r, |j-l| \leq r\}$.

Para recuperar la imagen, es decir, el proceso inverso, la función energía sería:

$$\epsilon_1 = \frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N}(a^H y_{ij}^H - \frac{1}{(2r+1)^2}\sum_{(k,l)\in N_{ij}^r} a^G y_{kl}^G)^2$$

C. Anexo III: Librería RNA

Clase RNA

```
import funciones

class RNA:
    def __init__(self, funCoste = funciones.Errores.CuadraticoMedio(), funSalidaNorm = funciones.Funciones.Identidad().funcion):
        self.funCoste      = funCoste
        self.funSalidaNorm = funSalidaNorm
        self.capas         = []

    def anadeCapa(self, capa):
        self.capas.append(capa)

    def cambiaCoste(self, funCoste):
        self.funCoste = funCoste

    def propagacionHaciaDelante(self, entrada):
        salida = entrada
        for capa in self.capas:
            salida = capa.propagacionHaciaDelante(salida)
        self.salida = salida
        return salida

    def propagacionHaciaAtras(self, salidaEsperada, tasaAprendizaje = 0.3):
        error = self.funCoste.derivada(salidaEsperada, self.salida) #nabla A =(dC/da1, ... dC/daj)
        for capa in reversed(self.capas):
            error = capa.propagacionHaciaAtras(error, tasaAprendizaje)
        #return error

    def __call__(self, entrada):
        resultados = []
        for dato in entrada:
            resultados.append(self.propagacionHaciaDelante(dato))
        return list(map(self.funSalidaNorm, resultados))

    def entrena(self, entrada, salidaEsperada, iteraciones, tasaAprendizaje, imprime = False, imprimeCada = 100):
```

```
35          numEntrada = len(entrada)
36
37          for i in range(iteraciones):
38              error = 0
39              for j in range(numEntrada):
40                  salida = self.propagacionHaciaDelante(entrada[j]) #lo que
    obtenemos con los parametros actuales
41                  self.propagacionHaciaAtras(salidaEsperada[j],
    tasaAprendizaje) #Corrige parametros segun error
42                  error += self.funCoste.funcion(salidaEsperada[j], salida) #
    acumula errores para luego hacer la media
43              error /= numEntrada #media error
44
45              if ((i % imprimeCada == (imprimeCada-1) or i==0) and imprime):
46                  print('Iteracion n % d   error = % f' % (i+1, error))
47          return error
```

Clase capa

```
1  import numpy
2  from funciones import Funciones
3  from abc import ABCMeta, abstractmethod
4
5  class AbstractCapa(metaclass = ABCMeta):
6      def __init__(self):
7          self.entrada = None
8          self.salidaPreActivacion = None
9
10     @abstractmethod
11     def propagacionHaciaDelante(self, entrada):
12         raise NotImplementedError
13
14     @abstractmethod
15     def propagacionHaciaAtras(self, errorSalida, tasaAprendizaje):
16         raise NotImplementedError
17
18 #Capa para una red con propagacion hacia delante
19 class CapaConectada(AbstractCapa):
20     def __init__(self, numNeuronasEntrada, numNeuronasSalida,
    funcionActivacion = Funciones.Identidad()):
21         self.funcionActivacion = funcionActivacion
```

```
22          self.pesos = numpy.random.rand(numNeuronasEntrada,
    numNeuronasSalida) - 0.5
23          self.sesgos = numpy.random.rand(1, numNeuronasSalida) - 0.5
24
25      #devuelve sigma(xW+b) con x = entrada, W = pesos, b = sesgos
26      def propagacionHaciaDelante(self, entrada):
27          self.entrada = entrada
28          self.salidaPreActivacion = numpy.dot(self.entrada, self.pesos) +
    self.sesgos #xW+b
29          return self.funcionActivacion.funcion(self.salidaPreActivacion) #
    sigma(xW+b)
30
31      # error salida: nabla A =(dC/da1, ... dC/daj)
32      def propagacionHaciaAtras(self, errorSalida, tasaAprendizaje):
33          delta = self.funcionActivacion.derivada(self.salidaPreActivacion) *
     errorSalida # k = nabla A sigma'(Z)
34          errorEntrada =  numpy.dot(delta, self.pesos.T)# kW
35          self.pesos -= tasaAprendizaje * numpy.dot(self.entrada.T, delta) #
    la(xk)
36          self.sesgos -= tasaAprendizaje * delta # mk
37
38          return errorEntrada
39
40  #Capa en la que se conecta cada neurona con una unica neurona de la
    siguiente capa, cada una diferente. Usa como funcion activacion la de
    paridad (para TPM)
41  class CapaUnoUno(AbstractCapa):
42      def __init__(self, numNeuronasEntrada, n, l, caos = False, beta = 0.5):
43          self.l, self.caos = l, caos
44          self.pesos = numpy.random.randint(-l, l + 1, [numNeuronasEntrada, n
    ])
45          self.funCaos = Funciones.Logistica()
46          self.funCaos.x = 1
47          self.beta = beta
48
49      #devuelve sigma(xW) con x = entrada, W = pesos
50      def propagacionHaciaDelante(self, entrada):
51          self.entrada = entrada
52          self.salidaPreActivacion = numpy.sum(self.entrada * self.pesos,
    axis = 1) #xW
53          if self.caos:
```

```
54              self.salidaPreActivacion = [(1-self.beta)*self.funCaos.
    siguiente() + (self.beta*x)/2 for x in  self.salidaPreActivacion]
55          self.salida = [1 if i == 0 else i for i in numpy.sign(self.
    salidaPreActivacion)] #sigma(xW)
56          return self.salida
57
58      def propagacionHaciaAtras(self, errorSalida, tasaAprendizaje):
59          for i in range(len(self.pesos)):
60              for j in range(len(self.pesos[0])):
61                  self.pesos[i,j] = numpy.clip(self.pesos[i, j] + (
    errorSalida[i]*self.entrada[i,j]), -self.l, self.l)
62          return 0
63
64 #Multiplica todas las salidas entre si y devuelve el resultado (para TPM)
65 class CapaMult(AbstractCapa):
66      def __init__(self, reglaA):
67          self.reglaAprendizaje = reglaA
68
69      def cambiaReglaAprendizaje(self, reglaA):
70          self.reglaAprendizaje = reglaA
71
72      def propagacionHaciaDelante(self, entrada):
73          self.entrada = entrada
74          self.salida  = numpy.prod(entrada)
75          return self.salida
76
77      def propagacionHaciaAtras(self, errorSalida, tasaAprendizaje):
78          if errorSalida <= 0:
79              return [0 for _ in self.entrada]
80          else:
81              rA = self.reglaAprendizaje
82              if rA == "Random Walk":
83                  return [1 if self.salida == i else 0 for i in self.entrada]
84              else:
85                  aux = -1 if rA == "Hebbian" else 1
86                  return [(aux*self.salida) if self.salida*i>0 else 0 for i
    in self.entrada]
87
88
89 #Capa en la que se conecta cada neurona con una unica neurona de la
    siguiente capa, cada una diferente. Usa como funcion activacion la de
```

```
    paridad (para TPM)
class CapaUnoUnoAtGeom(CapaUnoUno):
    def propagacionHaciaAtras(self, errorSalida, tasaAprendizaje):
        noCoincide, i = True, 0
        while noCoincide and i< len(errorSalida):
             noCoincide = errorSalida[i] == 0
             i += 1
        if noCoincide:
            iMin, salMin = 0, self.salidaPreActivacion[0]
            for i in range(1, len(self.pesos)):
                if salMin > self.salidaPreActivacion[i]:
                    iMin, salMin = i, self.salidaPreActivacion[i]
            for j in range(len(self.pesos[0])):
                self.pesos[iMin,j] = numpy.clip(self.pesos[iMin, j] - (
    errorSalida[iMin]*self.entrada[iMin,j]), -self.l, self.l)
        else:
              for i in range(len(self.pesos)):
                  for j in range(len(self.pesos[0])):
                      self.pesos[i,j] = numpy.clip(self.pesos[i, j] + (
    errorSalida[i]*self.entrada[i,j]), -self.l, self.l)

        return 0
```

Clase y enumeración de funciones

Se tiene una clase abstracta para las funciones que requiere que todas las clases con esta interfaz tengan una función que devuelva su valor y otra que devuelva el de su derivada. En la enumeración de las funciones que se usa en la clase RNA se incluyen las funciones más relevantes, las mencionadas en el apartado de RNA de este trabajo. También se incluye una enumeración con las 3 posibles reglas de aprendizaje del TPM.

```
import numpy
from enum import Enum
from abc import ABCMeta, abstractmethod

class AbstractFuncion(metaclass = ABCMeta):
    @abstractmethod
    def funcion(self, x):
        pass
    @abstractmethod
    def derivada(self, x):
        pass
```

```
12
13 class Identidad(AbstractFuncion):
14     def funcion(self, x):
15         return x
16     def derivada(self, x):
17         return 0
18
19 class Sigmoide(AbstractFuncion):
20     def funcion(self, x):
21         return 1/(1 + numpy.exp(-x))
22     def derivada(self, x):
23         return self.funcion(x)*(1.0-self.funcion(x))
24
25 class TangenteH(AbstractFuncion):
26     def funcion(self, x):
27         return numpy.tanh(x)
28     def derivada(self, x):
29         return 1-self.funcion(x)**2
30
31 class ReLu(AbstractFuncion):
32     a = 1
33     def funcion(self, x):
34         return max(0,self.a*x)
35     def derivada(self, x):
36         return self.a
37     def setA(self, a):
38         self.a = a
39
40 class Swish(AbstractFuncion):
41     a = 1
42     s = Sigmoide()
43     def funcion(self, x):
44         return x*self.s.funcion(self.a*x)
45     def derivada(self, x):
46         y = self.funcion(x)
47         return y+self.s(x)*(1- y)
48     def setA(self, a):
49         self.a = a
50
51 class PasoBinario(AbstractFuncion):
52     paso = 0.5
```

```
53      def funcion(self, x):
54          return 0 if abs(x)<self.paso else 1
55      def derivada(self, x):
56          return 0
57      def setPaso(self, paso):
58          self.paso = paso
59
60  class Redondeo(AbstractFuncion):
61      digitos = 1
62      def funcion(self, x):
63          return numpy.around(x, self.digitos)
64      def derivada(self, x):
65          return 0
66      def setDigitos(self, digitos):
67          self.digitos = digitos
68
69  class Logistica(AbstractFuncion):
70      r = 3.95
71      def funcion(self, x):
72          self.x = self.r*x*(1-x)
73          return self.x
74      def derivada(self, x):
75          return self.r*(1-2*x)
76      def setR(self, r):
77          self.r = r
78      def siguiente(self):
79          return self.funcion(self.x)
80
81
82  class Paridad(AbstractFuncion):
83      def funcion(self, x):
84          cont = 0
85          for i in x:
86              if i == 1:
87                  cont+=1
88          if cont%2 == 0:
89              return -1
90          else:
91              return 1
92      def derivada(self, x):
93          return 0
```

```
94
95  #Funciones de activacion
96  class Funciones(Enum):
97      Identidad    = Identidad
98      Sigmoide     = Sigmoide
99      TangenteH    = TangenteH
100     ReLu         = ReLu
101     Swish        = Swish
102     PasoBinario  = PasoBinario
103     Redondeo     = Redondeo
104     Logistica    = Logistica
105     Paridad      = Paridad
106     def __call__(self):
107         return self.value()
108
109 class AbstractFuncionError(metaclass = ABCMeta):
110     @abstractmethod
111     def funcion(self, x, y):
112         pass
113     @abstractmethod
114     def derivada(self, x, y):
115         pass
116
117 class errorCuadraticoMedio(AbstractFuncionError):
118     def funcion(self, actual, correcto):
119         return numpy.mean(numpy.power(actual-correcto, 2))
120     def derivada(self, actual, correcto):
121         try:
122             return 2*(correcto-actual)/actual.size
123         except:
124             return 2*(correcto-actual)
125
126 class Multiplicacion(AbstractFuncionError):
127     def funcion(self, actual, correcto):
128         return actual*correcto
129     def derivada(self, actual, correcto):
130         return actual*correcto
131
132 #Funciones de error
133 class Errores(Enum):
134     CuadraticoMedio = errorCuadraticoMedio
```

```
135     Multiplicacion  = Multiplicacion
136     def __call__(self):
137         return self.value()
138 
139 #Reglas para la estructura TPM
140 class reglaAprendizaje(Enum):
141     Hebbian     = "Hebbian"
142     AntiHebbian = "Anti Hebbian"
143     RandomWalk  = "Random Walk"
144     def __call__(self):
145         return self.name
```

D. Anexo IV: Implementación AP

Clase AP

Se define como extensión de la clase RNA, añadiendo una función para verificar su sincronización con otra. Esta función sincronización, se usa comparando la matriz de peso directamente para poder ver en cada paso el nivel de sincronización; sin embargo, en sistemas reales, se consideraría sincronizados una vez se han dado resultados que indican sincronización, una cantidad preacordada de veces o por métodos más sofisticados.

Uno de estos métodos es el uso de Hash generado con la salida de la red y los pesos en el momento; por lo que la clase cuenta con una función que devuelve este hash para permitir un uso más realista de las redes, sin pasar los pesos para comprobar la sincronización.

```
from RNA import RNA
import numpy
from capa import CapaUnoUno, CapaMult, CapaUnoUnoAtGeom
from funciones import reglaAprendizaje, Funciones, Errores
import hashlib

class arbolParidad(RNA):
    def __init__(self, k, n , l, caos = False, reglaA = reglaAprendizaje.
    AntiHebbian()):
        self.inicializa(k, n , l, reglaA)
        self.anadeCapa(CapaUnoUno(k, n, l, caos))
        self.anadeCapa(CapaMult(reglaA))
        self.caos = caos

    def inicializa(self, k, n , l, reglaA):
        self.l, self.k, self.n = l, k, n
        self.funCoste      = Errores.Multiplicacion()
        self.funSalidaNorm = Funciones.Identidad().funcion
        self.capas = []
        self.tasaAprendizaje, self.salida = 1, 0

    def cambiaReglaAprendizaje(self, reglaA):
        self.capas[1].cambiaReglaAprendizaje(reglaA)

    def getPesos(self):
        return self.capas[0].pesos.copy()

    def __call__(self, entrada):
```

```
28              return self.propagacionHaciaDelante(entrada)
29
30      def sincronizacionCon(self, otra):
31          return 100 - numpy.average(100 * numpy.abs(self.getPesos() - otra.
        getPesos())/(2*self.l - 1))
32
33      def hash_pesos(self):
34          return hashlib.sha256((f"{self.salida}{self.getPesos()}").encode('
        UTF-8')).hexdigest()
35
36      def getDatosIntermedios(self):
37          return self.capas[0].salida
38
39      def cambiaDatosIntermedios(self, nuevosDatos):
40          self.capas[0].salida = nuevosDatos
41
42  class arbolParidadAtGeom(arbolParidad):
43      def __init__(self, k, n , l, caos = False, reglaA = reglaAprendizaje.
        AntiHebbian()):
44          self.inicializa(k, n , l, reglaA)
45          self.anadeCapa(CapaUnoUnoAtGeom(k, n, l, caos))
46          self.anadeCapa(CapaMult(reglaA))
```

Clase test

En la clase test se incluye una función coordinaSalidas que simula la comunicación entre dos redes hasta que se sincronizan, comprobando esta sincronización por medio de los hash de los pesos y salidas de las redes. También cuenta funciones para imitar ataques de redes externas:

- **Ataque por fuerza bruta:** la función fuerzaBruta incluye una tercera red que intenta sincronizarse con las redes aplicando el aprendizaje habitual con la salida de una de ellas en la simulación.
- **Ataque geométrico:** la función ataqueGeometrico crea tantas redes como se especifiquen que intentan sincronizarse con las redes aplicando el aprendizaje de este tipo de ataque, explicado en el apartado sobre criptografía con AP.
- **Ataque genético:** la función ataqueGenetico crea tantas redes como se especifiquen que intentan sincronizarse con las redes aplicando el aprendizaje de este tipo de ataque, explicado en el apartado sobre criptografía con AP. Limitando el número de redes totales por el segundo parámetro.

```
import time
import matplotlib.pyplot as plt
import numpy
from statistics import mean
from arbolParidad import arbolParidad, arbolParidadAtGeom
import copy
from enum import Enum

class Test():
    def __init__(self, k, n, l, caos = False):
        self.l, self.k, self.n = l, k, n
        self.caos = caos
        self.A, self.B = arbolParidad(k, n, l, caos), arbolParidad(k, n, l,
    caos)
        self.sincronizacion()

    def sincronizacion(self):
        self.porcentajeSincro = self.A.sincronizacionCon(self.B)

    def entradaAleatoria(self):
      return numpy.random.randint(-self.l, self.l + 1, [self.k, self.n])

    def sincroMaxLista(self, Es):
        sincro = 0
        for E in Es:
            if E.sincronizacionCon(self.A) > sincro:
                sincro = E.sincronizacionCon(self.A)
        return sincro

    def coordinaSalidas(self):
        X = self.entradaAleatoria()

        xA, xB = self.A(X), self.B(X)

        self.A.propagacionHaciaAtras(xB)
        self.B.propagacionHaciaAtras(xA)

        self.sincronizacion()
        self.historial.append(self.porcentajeSincro)
        return X, xA, xB

```

```
41
42      def testBasico(self):
43          self.historial, cont, t_inicial = [], 0, time.time()
44
45          while(self.A.hash_pesos() != self.B.hash_pesos()):
46              self.coordinaSalidas()
47              cont += 1
48
49          return (0, time.time() - t_inicial, cont)
50
51      def fuerzaBruta(self):
52          self.historial, cont, t_inicial, E = [], 0, time.time(),
    arbolParidad(self.k, self.n, self.l, self.caos)
53
54          while(self.porcentajeSincro != 100):
55              X, xA, xB = self.coordinaSalidas()
56              if xA == xB == E(X):
57                  E.propagacionHaciaAtras(xA)
58              cont += 1
59
60          return (E.sincronizacionCon(self.A), time.time() - t_inicial, cont)
61
62      def ataqueGeometrico(self, N = 10):
63          self.historial, cont, t_inicial, Es = [], 0, time.time(), []
64          for i in range(N):
65              Es.append(arbolParidadAtGeom(self.k, self.n, self.l, self.caos)
    )
66
67          while(self.porcentajeSincro != 100):
68              X, xA, xB = self.coordinaSalidas()
69              if xA == xB:
70                  for E in Es:
71                      E(X)
72                      E.propagacionHaciaAtras(xA)
73              cont += 1
74
75          return (self.sincroMaxLista(Es), time.time() - t_inicial, cont)
76
77      def ataqueGenetico(self, N = 10, M = 200):
78          self.historial, cont, t_inicial, Es = [], 0, time.time(), []
79          for i in range(N):
```

```
            Es.append(arbolParidad(self.k, self.n, self.l, self.caos))

        while(self.porcentajeSincro != 100):
            X, xA, xB = self.coordinaSalidas()
            if xA == xB:
                no1, EsCopia = int(xA == -1), list(Es)
                for E in EsCopia:
                    res = E(X)
                    if len(Es) + 2**(self.k-1) < M:
                        Es.remove(E)
                        for i in range(2**self.k):
                            aux, lista = i, []
                            for _ in range(self.k):
                                lista.append((-1)**int(auxporciento2))
                                aux = aux//2
                            if (lista.count(-1)%2 == no1):
                                Eaux  = copy.deepcopy(E)
                                Eaux.cambiaDatosIntermedios(lista)
                                Eaux.propagacionHaciaAtras(xA)
                                Es.append(Eaux)
                    elif xA == res:
                        E.propagacionHaciaAtras(xA)
                    elif len(Es)>1:
                        Es.remove(E)
            cont += 1

        return (self.sincroMaxLista(Es), time.time() - t_inicial, cont)

    def graficoSincronizacion(self):
        plt.plot(range(len(self.historial)), self.historial)

        plt.xlabel('N comunicacion')
        plt.ylabel('porcentaje sincronizacion')

        plt.show()

    class Ataques(Enum):
        FuerzaBruta = 'Ataque por Fuerza Bruta'
        Geometrico  = 'Ataque Geometrico'
        Genetico    = 'Ataque Genetico'

```

```
121 def prueba(n: int, ataque: Test.Ataques, caos: bool)-> int:
122     resS, resT, resC = [], [], []
123     for i in range(n):
124         t = Test(5, 10, 10, caos)
125         if ataque == Test.Ataques.FuerzaBruta:
126             (sincro, tiempo, cont) = t.fuerzaBruta()
127         elif ataque == Test.Ataques.Geometrico:
128             (sincro, tiempo, cont) = t.ataqueGeometrico()
129         elif ataque == Test.Ataques.Genetico:
130             (sincro, tiempo, cont) = t.ataqueGenetico()
131         else:
132             print('b')
133             (sincro, tiempo, cont) = t.testBasico()
134         resS.append(sincro)
135         resT.append(tiempo)
136         resC.append(cont)
137         if sincro == 100:
138             print('Roto por el atacante')
139 
140     plt.plot(range(len(resS)), resS)
141     plt.xlabel('N iteracion prueba')
142     plt.ylabel('porcentaje sincronizacion red externa')
143     plt.show()
144 
145     plt.plot(range(len(resC)), resC, color="red")
146     plt.xlabel('N iteracion prueba')
147     plt.ylabel('N de interacciones')
148     plt.show()
149 
150     print("La media del porcentaje de sincronizacion es: ", str(mean(resS))
    , " porciento")
151     print("La media del numero de interacciones es: ", str(mean(resC)))
152     print("La media del tiempo para sincronizarse es: ", str(mean(resT)), "
     segundos")
153     return resS
```

Referencias

[1] Gonzalo Alvarez and Shujun Li. Some basic cryptographic requirements for chaos-based cryptosystems. *I. J. Bifurcation and Chaos*, 16:2129–2151, 08 2006.

[2] Jinde Cao and Jianquan Lu. Adaptive synchronization of neural networks with or without time-varying delay. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 16(1):013133, 2006.

[3] L.O. Chua and L. Yang. Cellular neural networks: applications. *IEEE Transactions on Circuits and Systems*, 35(10):1273–1290, 1988.

[4] L.O. Chua and L. Yang. Cellular neural networks: theory. *IEEE Transactions on Circuits and Systems*, 35(10):1257–1272, 1988.

[5] Jack Hale. *Theory of functional differential equations*. Springer-Verlag, 1971.

[6] Feng-Hsiag Hsiao. Applying elliptic curve cryptography to a chaotic synchronisation system: neural-network-based approach. *International Journal of Systems Science*, 48(14):3044–3059, 2017.

[7] Ido Kanter, Wolfgang Kinzel, and Eran Kanter. Secure exchange of information by synchronization of neural networks. *EPL (Europhysics Letters)*, 57, 02 2002.

[8] Harpreet Kaur and Tripatjot Singh Panag. Cryptography using chaotic neural networks. *International Journal of Information Technology and Knowledge Management*, pages 417–422, 2011.

[9] W. Kinzel and Ido Kanter. Neural cryptography. *9th International Conference on Neural Information Processing*, 3:1351 – 1354 vol.3, 12 2002.

[10] Alexander Klimov, Anton Mityagin, and Adi Shamir. Analysis of neural cryptography. In *Lecture Notes in Computer Science*, pages 288–298, 12 2002.

[11] Fausto Montoya Vitini y Jaime Muñoz Masqué Luis Hernández Encinas. Esquemas criptográficos visuales. *CSIC*, 2000.

[12] R Metzler, W Kinzel, and Ido Kanter. Interacting neural networks. *Phys. Rev. E*, 62:2555–2565, 08 2000.

[13] A. Mhetras and N. Charniya. Cryptography based on artificial neural networks and chaos theory. *International Journal of Computer Applications*, 133, 2016.

[14] Louis Pecora and T. Carroll. Synchronization in chaotic system. *Physical Review Letters*, 64:821, 03 1990.

[15] Jun Peng, Du Zhang, and Xiaofeng Liao. A digital image encryption algorithm based on hyper-chaotic cellular neural network. *Fundam. Inform.*, 90:269–282, 01 2009.

[16] Andreas Ruttor. *Neural Synchronization and Cryptography.* PhD thesis, Julius Maximilians Universitat de Wurzburgo, 2007.

[17] Arindam Sarkar. Mutual learning-based efficient synchronization of neural networks to exchange the neural key. *Complex and Intelligent Systems*, 2022.

[18] Eva Volna, Martin Kotyrba, Vaclav Kocian, and Michal Janosek. Cryptography based on neural network. *European Conference on Modelling and Simulation*, 05 2012.

[19] Wenwu Yu and Jinde Cao. Cryptography based on delayed chaotic neural networks. *Physics Letters A*, 356(4):333–338, 2006.

[20] Tai-Wen Yue and Suchen Chiang. A neural-network approach for visual cryptography and authorization. *International journal of neural systems*, 14:175–87, 07 2004.

[21] Fan Zhang and Xinhong Zhang. Image inverse halftoning and descreening: a review. *Multimed Toodls Appl*, 78:21021–21039, 2019.

[22] Jin Zhou, Tianping Chen, and Lan Xiang. Chaotic lag synchronization of coupled delayed neural networks and its applications in secure communication. *Circuits Systems and Signal Processing*, 24:599–613, 10 2005.