Trabajo práctico 1
Redes Neuronales y Aprendizaje Profundo

Ignacio Ezequiel Cavicchioli
Padrón 109428
icavicchioli@fi.uba.ar

10/9/2025

Índice

1 Introducción 2

2 Ejercicio 1 3
2.1 Consignas . . . 3
2.2 Desarrollo . . . 3
2.3 Análisis . . . 3

3 Ejercicio 2 7
3.1 Consignas . . . 7
3.2 Desarrollo . . . 7
3.3 Análisis . . . 8

4 Ejercicio 3 9
4.1 Consignas . . . 9
4.2 Desarrollo . . . 9
4.2.1 2 Neuronas en capa oculta . . . 10
4.2.2 4 Neuronas en capa oculta . . . 13
4.2.3 10 Neuronas en capa oculta . . . 16
4.3 Análisis . . . 18

5 Ejercicio 4 19
5.1 Consignas . . . 19
5.2 Desarrollo . . . 19
5.3 Inciso a . . . 20
5.3.1 *minibatch* de 40 muestras . . . 20
5.3.2 *minibatch* de 100 muestras . . . 21
5.3.3 *minibatch* de 1000 muestras . . . 22
5.3.4 Análisis . . . 23
5.4 Inciso b . . . 24
5.4.1 *minibatch* de 40 muestras . . . 24
5.4.2 *minibatch* de 1 muestras . . . 26
5.4.3 Análisis . . . 27
5.5 Análisis global . . . 27

6 Ejercicio 5 28
6.1 Consignas . . . 28

6.2 Desarrollo 28
6.3 Análisis 35

7 Ejercicio 6 36
7.1 Consignas 36
7.2 Desarrollo 36
7.2.1 CNNs 36
7.2.2 MLPs 39
7.3 Análisis 41

8 Ejercicio 7 42
8.1 Consignas 42
8.2 Desarrollo 42
8.3 Análisis 45

9 Ejercicio 8 47
9.1 Consignas 47
9.2 Desarrollo 47
9.2.1 XOR de 2 entradas - 2 neuronas en capa oculta 47
9.2.2 XOR de 2 entradas - 4 neuronas en capa oculta 48
9.2.3 XOR de 2 entradas - 10 neuronas en capa oculta 49
9.2.4 XOR de 2 entradas - 2 neuronas en capa oculta 49
9.2.5 XOR de 2 entradas - 2 neuronas en capa oculta 50
9.2.6 XOR de 4 entradas - 4 neuronas en capa oculta 51
9.2.7 XOR de 4 entradas - 5 neuronas en capa oculta 52
9.2.8 XOR de 4 entradas - 6 neuronas en capa oculta 52
9.3 Análisis 52

10 Conclusiones 53

1. Introducción

Este documento presenta el desarrollo de las consignas del trabajo práctico N°2 de la materia de **Redes Neuronales y Aprendizaje Profundo**. El código correspondiente fue realizado en *Jupyter notebooks*, *Python*, adjuntados a la entrega en formato PDF. Toda imagen o implementación requeridas para el análisis se explicitarán en el presente archivo, por lo que la lectura del código en sí queda a discreción del lector. La teoría relevante será presentada y discutida en la sección pertinente.

2. Ejercicio 1

2.1. Consignas

Implemente un perceptrón simple que aprenda la función lógica AND y la función lógica OR, de 2 y de 4 entradas. Muestre la evolución del error durante el entrenamiento. Para el caso de 2 dimensiones, grafique la recta discriminadora y todos los vectores de entrada de la red.

2.2. Desarrollo

Lo primero que se hizo fue generar el dataset para entrenar los perceptrones, que no es más que la tabla de verdad de la función lógica que se quiere aprender.

Las figuras 1, 2,3 y 4 muestran el error en función de la iteración (de entrenamiento) para los perceptrones que emulan las funciones AND y OR de 2 y 4 entradas. Como se discutió en la teórica, el perceptrón es capaz de aprender estas funciones lógicas simples, por lo que el error final del entrenamiento es cero.

La figura 6 muestra la frontera de decisión del perceptrón que aprendió la función OR, y la 5 muestra la frontera para la compuerta AND. Los puntos del mismo color son de la misma clase (0 o 1), y son adecuadamente segregados por la frontera. Contrario a SVM, el perceptrón simple no tiene la distancia desde la frontera hasta las muestras en su función de costo, por lo que la recta que separa las clases no es única.

Para encontrar la fórmula de la recta que hace de frontera de decisión se puede partir de la expresión de la sumatoria de los X con sus respectivos pesos y el bias, y suponer que $Y = 0$, lo que significa que estás parado sobre la frontera, tu muestra no es ni de una clase ni de la otra.

$$X1 \cdot w1 + X2 \cdot w2 + b = 0$$

Si suponemos que $X1$ es nuestra abscisa que vamos a barrer y $X2$ es la ordenada, la recta toma la forma de:

$$X2 = \frac{w1 \cdot X1 + b}{-w2}$$

Y entonces $m = -w1/w2$ y b $= -b/w2$

2.3. Análisis

Los perceptrones lograron aprender sus funciones lógicas designadas sin error y en una cantidad finita de iteraciones del algoritmo. Esto es el resultado esperado, se sabe que puede aprender este tipo particular de problemas.

Una parte interesante del ejercicio fue el armado del algoritmo de gradiente descendiente, regla por la cual el perceptrón va moviendo su frontera de decisión. Los gráficos muestran un error decreciente en el tiempo, que a veces aumenta espuriamente para luego ser corregido, porque el gradiente descendiente va actualizando los pesos en función del error, intentando moverse hacia fronteras que predigan correctamente.

La obtención de la recta que hace de frontera en 2D solo requirió del desarrollo de fórmulas hecho en la sección previa.

En líneas generales, el ejercicio sirvió como una buena introducción a los perceptrones y el concepto de gradiente descendiente.

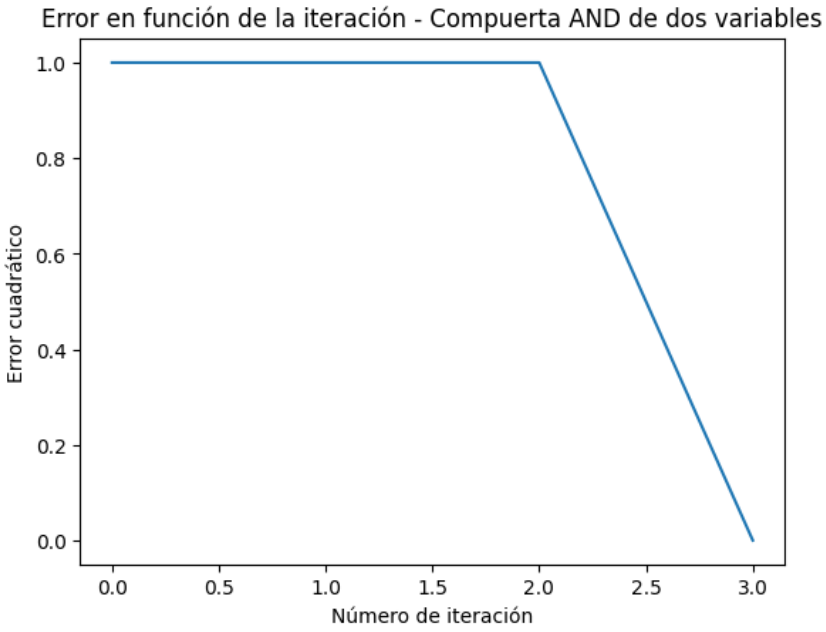

Figura 1: Error de entrenamiento en el tiempo para compuerta AND de 2 entradas

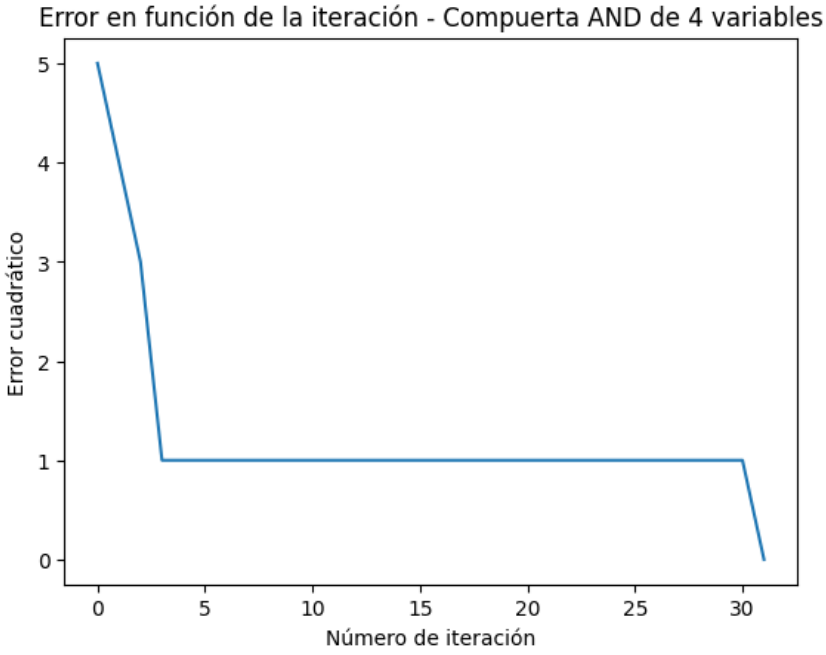

Figura 2: Error de entrenamiento en el tiempo para compuerta AND de 4 entradas

Figura 3: Error de entrenamiento en el tiempo para compuerta OR de 2 entradas

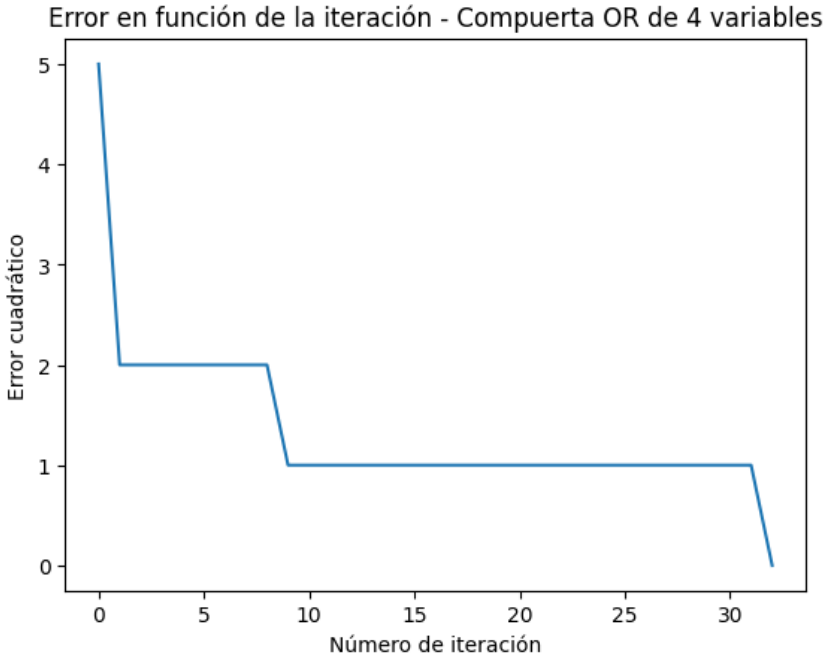

Figura 4: Error de entrenamiento en el tiempo para compuerta OR de 4 entradas

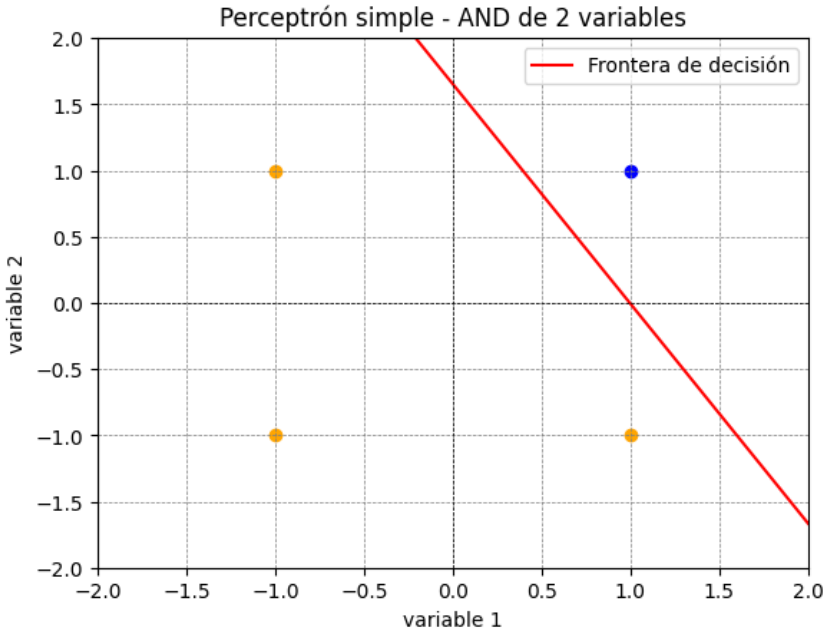

Figura 5: Frontera encontrada para compuerta AND de 2 entradas

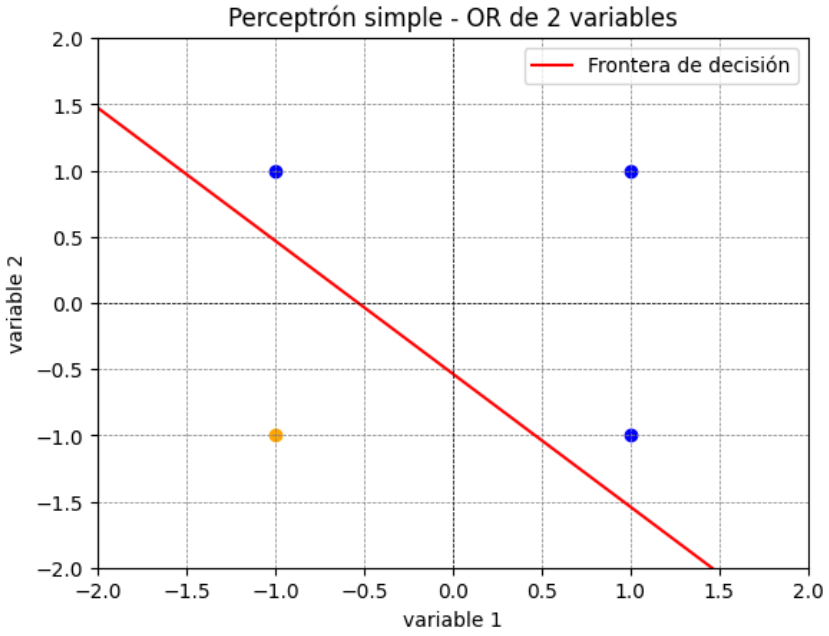

Figura 6: Frontera encontrada para compuerta OR de 2 entradas

3. Ejercicio 2

3.1. Consignas

Determine numéricamente cómo varía la capacidad del perceptrón simple en función del número de patrones enseñados.

3.2. Desarrollo

La capacidad de un perceptrón está definida en la p.111 (expresión 5.63) del libro "Introduction to the theory of neural computation". La idea es básicamente determinar la cantidad de patrones aleatorios (con etiqueta aleatoria) para el cual existe una frontera. En el libro se demuestra que la capacidad de almacenamiento máxima del perceptrón converge a $p_{max} = 2N$, con p la cantidad de patrones y N, el número de pesos sinápticos que tiene el perceptrón (o cantidad de unidades de entrada), con $N \to$ ínf.

Para corroborar experimentalmente esta aserción, se entrenó un perceptrón de un N particular con cada vez más patrones y se registró el resultado, para luego graficarlo.

En pocas palabras, el código genera conjuntos de patrones aleatorios y entrena un perceptrón simple para clasificarlos. Repite el experimento múltiples veces[1] y calcula con qué frecuencia el perceptrón logra aprender sin errores (es decir, converge). Luego grafica la probabilidad de éxito en función de $\alpha = \frac{p}{N}$, donde p es la cantidad de patrones y N la dimensión de entrada, mostrando así la capacidad del perceptrón. Se espera que la recta vertical $p/N = 2$ interseque la gráfica en $p = 1/2$ (como en la literatura citada).

los resultados para $N = 30$ se pueden ver en la figura 7, y para $N = 10$, en la 8. No se hizo con más unidades de entrada porque condiciona mucho el tiempo de ejecución y la cantidad de epochs necesarias para lograr un buen resultado.

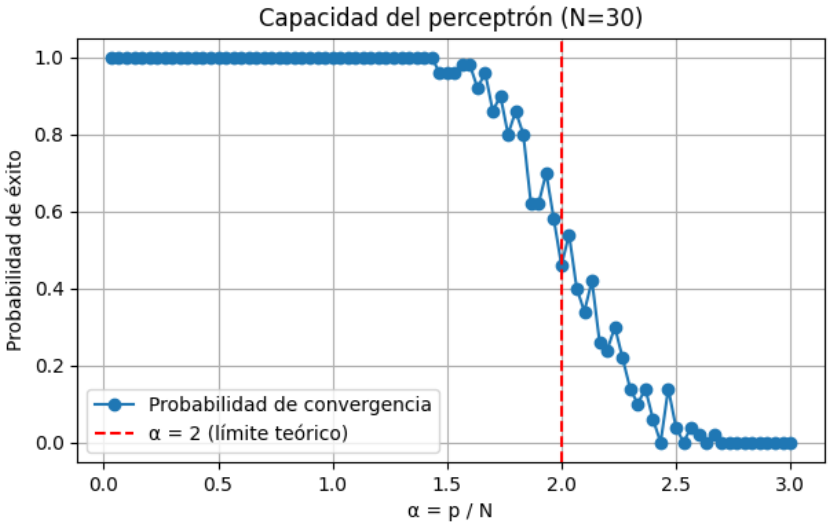

Figura 7: Ensayo de capacidad $N = 30$

[1] Para un cierto p se hacen varios experimentos y se promedia, por eso hay una cantidad de unidades de entrada y de repeticiones

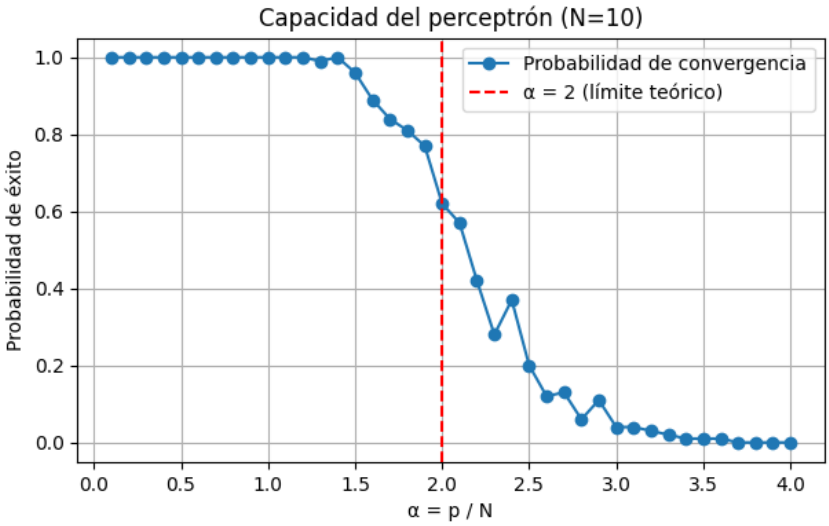

Figura 8: Ensayo de capacidad $N = 10$

3.3. Análisis

La figura 7 muestra que la capacidad de memorizar (el eje Y) del perceptrón es perfecta (vale 1) y, a medida que aumenta la cantidad de patrones, decrece hasta hacerse cero. El cruce por el punto $p/N = 2$ se da aproximadamente en $p \simeq 0{,}5$, resultado coincidente con la teoría del libro mencionado.

La transición, o flanco, no es perfectamente vertical, sino que decrece casi como una función sigmoidea. Esto es debido al uso de valores de N relativamente pequeños y el promediado empleado. A forma comparativa se incluye la imagen 8, de $N = 10$. En esta se ve que la capacidad tarda más en converge a cero (en $N = 30$ se hace cero en 2,5, y en $N = 10$, en 3,5), y la intersección del límite teórico es más cercana a 0,6 y no 0,5.

Resumiendo, aún con las limitaciones en la cantidad de unidades de entrada, se empieza a divisar una tendencia que parece coincidir con la teoría.

4. Ejercicio 3

4.1. Consignas

Implemente un perceptrón multicapa que aprenda la función lógica XOR de 2 y de 4 entradas (utilizando el algoritmo Backpropagation y actualizando en batch). Muestre cómo evoluciona el error durante el entrenamiento.

4.2. Desarrollo

En este inciso se entrenó un MLP para que aprendan la XOR, pero se hizo con 2, 4 y 10 neuronas en la capa oculta. La idea general fue crear las tablas de verdad de las funciones XOR de 2 y 4 entradas y entrenar MLP´s, registrando el error en función de las epochs.

No se usa 1 solo perceptrón porque su funcionamiento es incompatible con la función XOR. El problema fundamental es que no existen pesos que permitan aprender perfectamente el problema. Voy a intentar una pequeña demostración basándome en las fórmulas básicas del perceptrón y suponiendo una toma de decisión por alguna función que categorice 0 y 1.

La explicación de a continuación debería ayudar a entender que la limitación se puede hasta pensar de forma geométrica, no es dependiente del entrenamiento o función de activación[2].

$$y = \begin{cases} 1 \text{ si } h > 1/2 \\ 0 \text{ si } h \leq 1/2 \end{cases}$$

donde h es la salida del perceptrón simple:

$$h = \alpha_1 x_1 + \alpha_2 x_2$$

x1	x2	h	y
0	0	$\leq 1/2$	0
0	1	$> 1/2$	1
1	0	$> 1/2$	1
1	1	$\leq 1/2$	0

Resolviendo el sistema de ecuaciones para $(0, 0)$:

$$0 \cdot \alpha_1 + 0 \cdot \alpha_2 = h \leq 1/2$$

Para $(0, 1)$:

$$0 \cdot \alpha_1 + 1 \cdot \alpha_2 = h \geq 1/2 \Rightarrow \alpha_2 \geq 1/2$$

Para $(1, 0)$:

$$1 \cdot \alpha_1 + 0 \cdot \alpha_2 = h \geq 1/2 \Rightarrow \alpha_1 \geq 1/2$$

Para $(1, 1)$:

$$1 \cdot \alpha_1 + 1 \cdot \alpha_2 = h \leq 1/2 \Rightarrow \alpha_1 + \alpha_2 \leq 1/2$$

Sumando las dos desigualdades obtenemos

$$\alpha_1 + \alpha_2 \geq 1,$$

[2]Excepto en el caso de usar una función de activación no monótona. Se me ocurrió definir, por elegir valores simples, una función que vale 0 para todo valor fuera del intervalo $(\frac{1}{10}, \frac{3}{8})$ y 1 dentro de él. Con esta nueva función, el perceptrón podría aprender la función XOR, ya que existen, por ejemplo, $\alpha = \frac{1}{2}$ que cumplen: para $(0, 0) \to h = 0 \to y = 0$, para $(1, 0)$ y $(0, 1) \to h = \frac{1}{4} \to y = 1$, y para $(1, 1) \to h = \frac{1}{2} \to y = 0$. Esta función no permite resolver la AND o la OR, fue revisado en un visor de geometría 3D.

Lo cual contradice $\alpha_1 + \alpha_2 \leq \frac{1}{2}$. Por lo tanto, no existen α_1, α_2 que satisfagan todas las condiciones: el perceptrón simple no puede aprender la función XOR. Ahora, cuando se agrega otro perceptrón en paralelo y se le da uno que decide en base a esta capa oculta, la XOR se vuelve "aprendible".

4.2.1. 2 Neuronas en capa oculta

En este primer caso se utilizó un perceptrón multicapa con una capa oculta de dos neuronas, con el objetivo de que aprenda la función XOR. Se trabajó tanto con la XOR de dos entradas como con la de cuatro entradas, manteniendo la misma arquitectura a fin de observar las diferencias en la capacidad de representación del modelo.

Para la XOR de dos entradas, la red logró converger correctamente, alcanzando un error cercano a cero y reproduciendo exactamente la tabla de verdad de la función. En las Figuras 9 y 10 se muestra la evolución del error por época y la frontera de decisión obtenida, respectivamente, donde puede verse una separación clara entre las clases.

En contraste, al aplicar la misma arquitectura a la XOR de cuatro entradas, la red no consiguió reducir el error a cero, incluso habiendo hasta triplicado la cantidad de epochs. (Figura 11).

Esto apunta a que dos neuronas ocultas no son suficientes para modelar la complejidad de la XOR de cuatro entradas, y se supone que se debe principalmente al incremento exponencial en cantidad de combinaciones posibles. La red ya no dispone de capacidad suficiente como para memorizar los patrones, siendo esto una manifestación práctica de la capacidad teórica analizada en el ejercicio anterior.

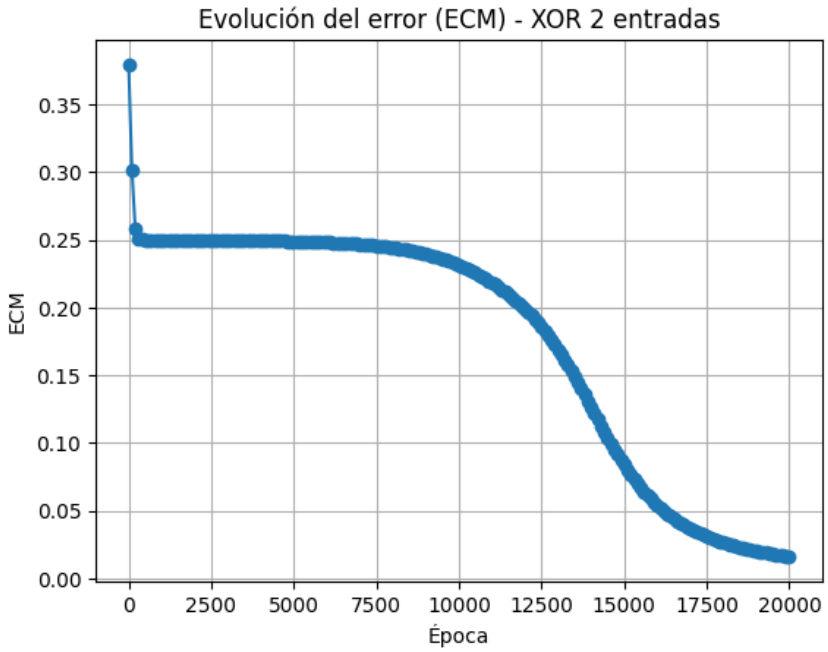

Figura 9: Error por época - XOR de 2 entradas - MLP 2 neuronas en capa oculta

Entradas	**Target**	**Salida sigmoide**	**Predicción (umbral 0.5)**
[0 0]	0	0.082	0
[0 1]	1	0.927	1
[1 0]	1	0.927	1
[1 1]	0	0.077	0

Cuadro 1: Resultados del perceptrón con dos entradas - MLP 2 neuronas en capa oculta

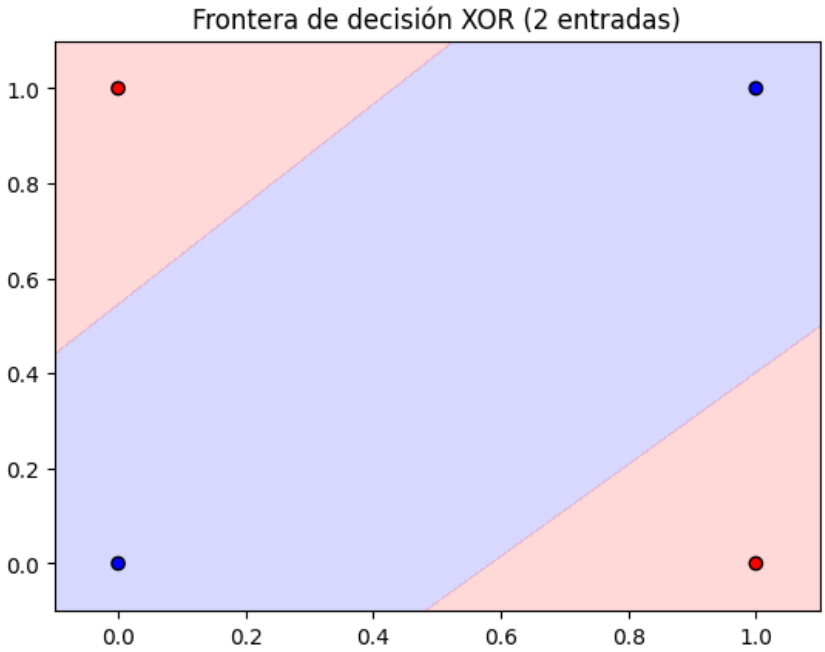

Figura 10: Frontera de decisión - XOR de 2 entradas - MLP 2 neuronas en capa oculta

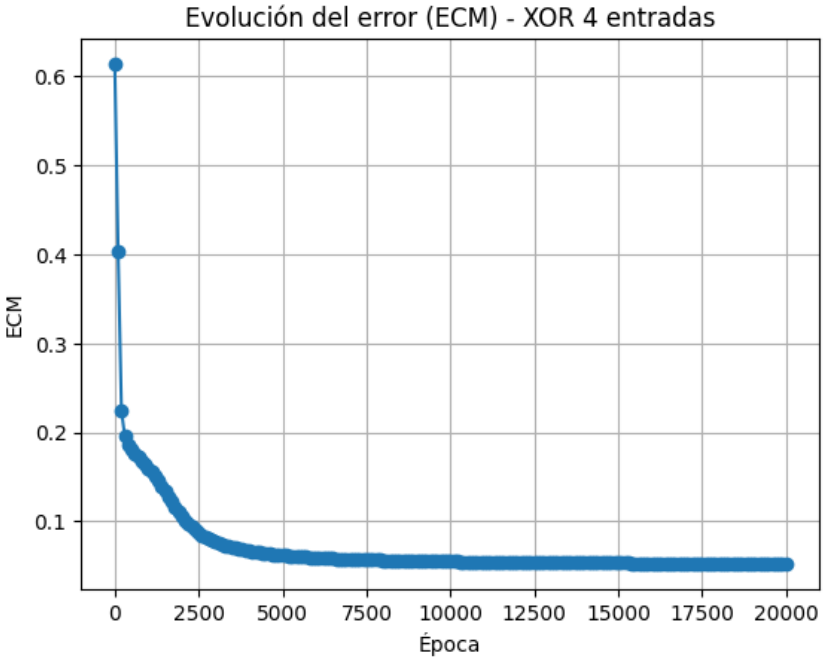

Figura 11: Error por época - XOR de 4 entradas - MLP 2 neuronas en capa oculta

Entradas	**Target**	**Salida sigmoide**	**Predicción (umbral 0.5)**
$[-1\ -1\ -1\ -1]$	0	0.725	1
$[1\ -1\ -1\ -1]$	1	0.837	1
$[-1\ 1\ -1\ -1]$	1	0.701	1
$[1\ 1\ -1\ -1]$	0	0.036	0
$[-1\ -1\ 1\ -1]$	1	0.708	1
$[1\ -1\ 1\ -1]$	0	0.048	0
$[-1\ 1\ 1\ -1]$	0	0.045	0
$[1\ 1\ 1\ -1]$	0	0.009	0
$[-1\ -1\ -1\ 1]$	1	0.879	1
$[1\ -1\ -1\ 1]$	0	0.063	0
$[-1\ 1\ -1\ 1]$	0	0.030	0
$[1\ 1\ -1\ 1]$	0	0.029	0
$[-1\ -1\ 1\ 1]$	0	0.055	0
$[1\ -1\ 1\ 1]$	0	0.036	0
$[-1\ 1\ 1\ 1]$	0	0.011	0
$[1\ 1\ 1\ 1]$	0	0.027	0

Cuadro 2: Resultados del perceptrón con cuatro entradas - MLP 2 neuronas en capa oculta

4.2.2. 4 Neuronas en capa oculta

En este caso se incrementó el número de neuronas en la capa oculta a cuatro, con el objetivo de mejorar la capacidad de representación del MLP. Para la XOR de dos entradas, la red continúa convergiendo correctamente, alcanzando un error nulo y generando fronteras de decisión curvas (Figuras 12 y 13). Esta curvatura parece ser causada por la complejidad de la red; mientras más unidades tiene la capa oculta, más grados de libertad tienen las fronteras. Esto se refleja también en la tabla de resultados, donde todas las predicciones coinciden con los valores esperados.

Para la XOR de cuatro entradas, la red aún no logra converger a un error nulo (Figura 14). En la tabla correspondiente se puede ver que varias predicciones todavía difieren del target esperado. Se sigue confundiendo en la primer entrada.

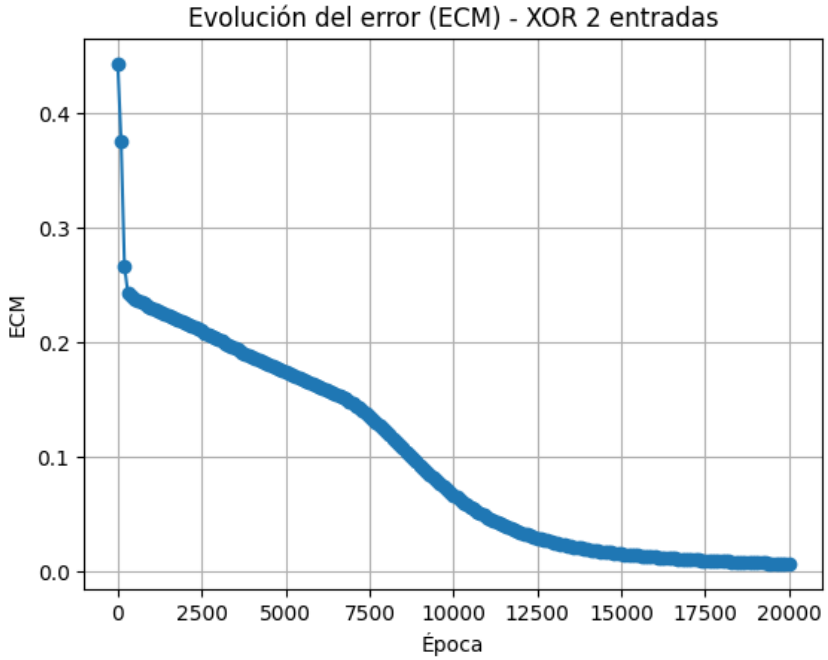

Figura 12: Error por época - XOR de 2 entradas - MLP 4 neuronas en capa oculta

Entradas	**Target**	**Salida sigmoide**	**Predicción (umbral 0.5)**
[0 0]	0	0.059	0
[0 1]	1	0.902	1
[1 0]	1	0.946	1
[1 1]	0	0.102	0

Cuadro 3: Resultados del perceptrón con dos entradas - MLP 4 neuronas en capa oculta

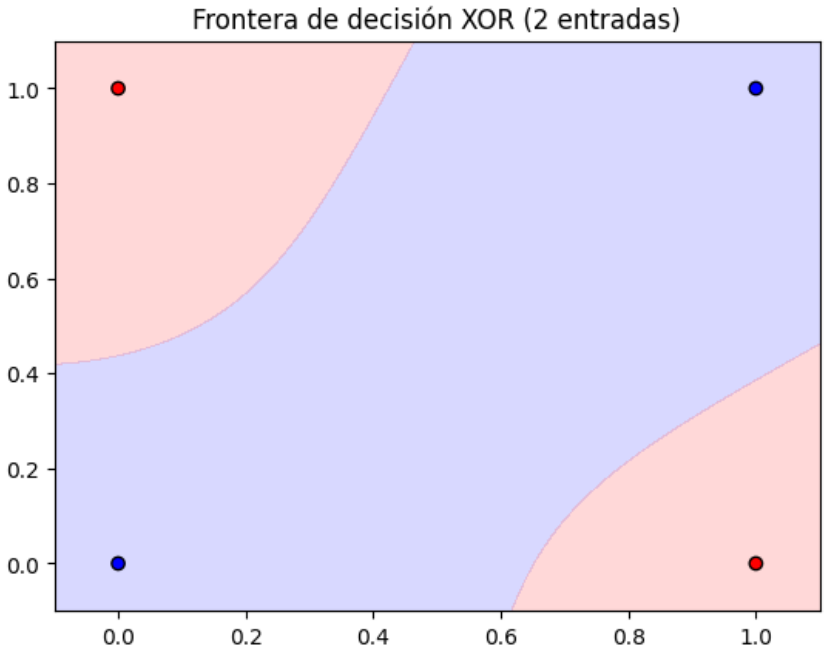

Figura 13: Frontera de decisión - XOR de 2 entradas - MLP 4 neuronas en capa oculta

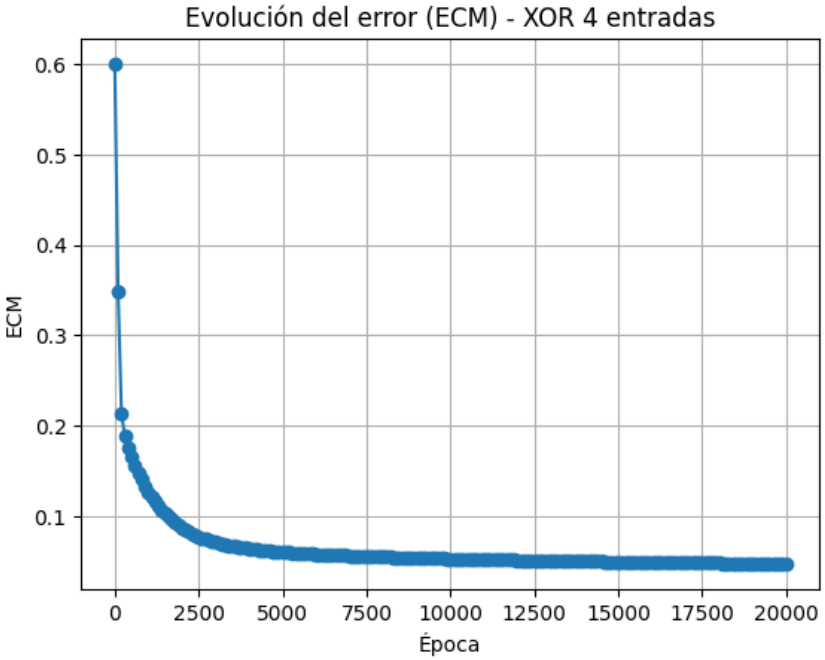

Figura 14: Error por época - XOR de 4 entradas - MLP 4 neuronas en capa oculta

Entradas	**Target**	**Salida sigmoide**	**Predicción (umbral 0.5)**
$[-1\ -1\ -1\ -1]$	0	0.727	1
$[1\ -1\ -1\ -1]$	1	0.879	1
$[-1\ 1\ -1\ -1]$	1	0.697	1
$[1\ 1\ -1\ -1]$	0	0.033	0
$[-1\ -1\ 1\ -1]$	1	0.863	1
$[1\ -1\ 1\ -1]$	0	0.054	0
$[-1\ 1\ 1\ -1]$	0	0.039	0
$[1\ 1\ 1\ -1]$	0	0.010	0
$[-1\ -1\ -1\ 1]$	1	0.696	1
$[1\ -1\ -1\ 1]$	0	0.028	0
$[-1\ 1\ -1\ 1]$	0	0.058	0
$[1\ 1\ -1\ 1]$	0	0.003	0
$[-1\ -1\ 1\ 1]$	0	0.034	0
$[1\ -1\ 1\ 1]$	0	0.007	0
$[-1\ 1\ 1\ 1]$	0	0.003	0
$[1\ 1\ 1\ 1]$	0	0.003	0

Cuadro 4: Resultados del perceptrón con cuatro entradas - MLP 4 neuronas en capa oculta

4.2.3. 10 Neuronas en capa oculta

Al aumentar el número de neuronas en la capa oculta a diez, se observa un cambio significativo en el comportamiento del MLP.

Para la XOR de dos entradas, la red sigue convergiendo sin problemas (Figura 15 y 16). Las fronteras de decisión siguen siendo curvas, cambiando en cada ejecución, pudiendo ser más o menos recta. Acá ya se entiende que la red tiene más que suficiente capacidad para aprender todos los patrones, y su espacio de soluciones posibles es grande y variado (relacionado con la cantidad de grados de libertad que le da la capa oculta de 10 unidades), por eso la gran variedad de fronteras.

Para la XOR de cuatro entradas, a diferencia de los casos anteriores con 2 o 4 neuronas, ahora la red logra aprender correctamente todos los patrones, alcanzando un error nulo (Figura 17). Esto confirma la sospecha de que las dificultades de aprendizaje de la XOR de 4 entradas en los experimentos anteriores estaba relacionada con la capacidad de la red, que se encontraba limitada por la cantidad de neuronas. De alguna manera, el aumento de grados de libertad hace que los subespacios de parámetros que son compatibles con el set de datos se vuelvan cada vez más accesibles, hasta que es lo suficientemente accesible como para que siempre haya error nulo post-aprendizaje.

El cuadro 6 muestra la tabla de verdad enseñada y aprendida. Nótese que, contrario a los casos para menos neuronas, la red aprendió bien la lógica, y esto se nota en que las salidas de la sigmoidea son o muy cercanas a 0, o muy cercanas a 1, indicando poca "confusión" (siempre está muy seguro de su decisión).

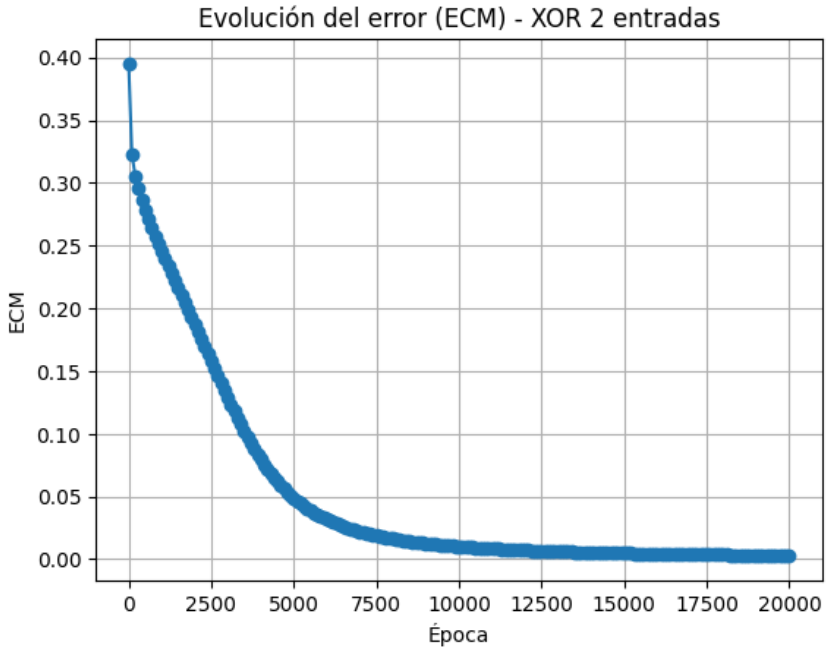

Figura 15: Error por época - XOR de 2 entradas - MLP 10 neuronas en capa oculta

Entradas	**Target**	**Salida sigmoide**	**Predicción (umbral 0.5)**
[0 0]	0	0.051	0
[0 1]	1	0.948	1
[1 0]	1	0.942	1
[1 1]	0	0.057	0

Cuadro 5: Resultados del perceptrón con dos entradas y 10 neuronas ocultas.

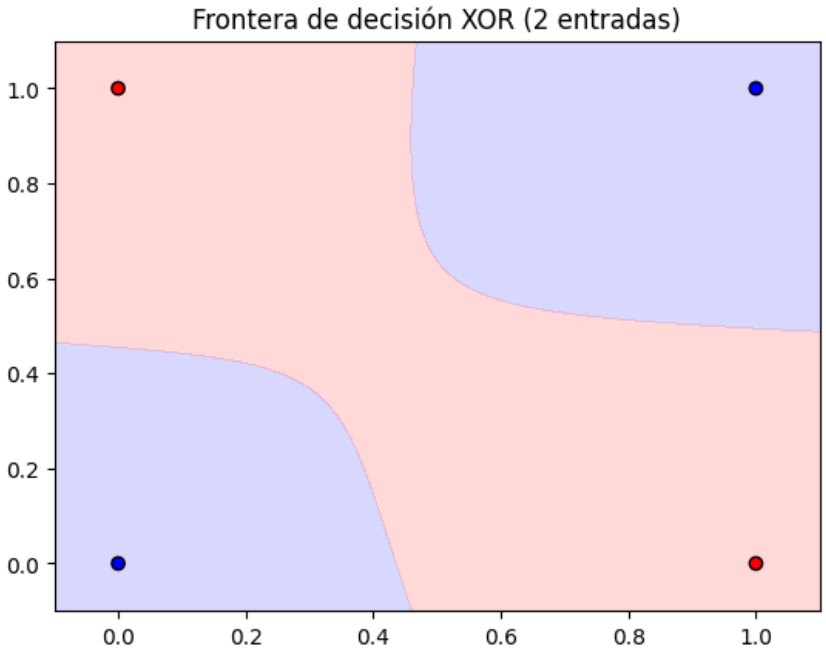

Figura 16: Frontera de decisión - XOR de 2 entradas - MLP 10 neuronas en capa oculta

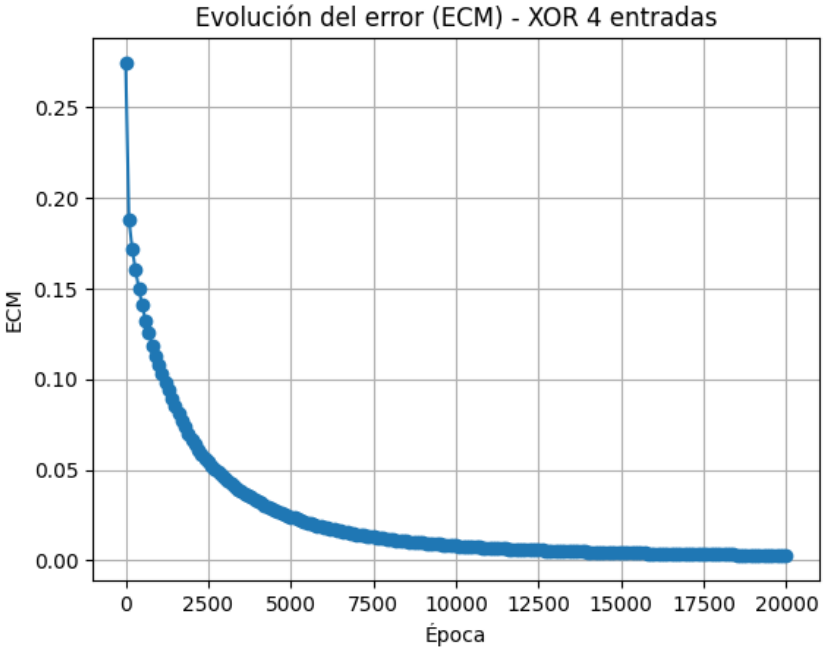

Figura 17: Error por época - XOR de 4 entradas - MLP 10 neuronas en capa oculta

Entradas	Target	Salida sigmoide	Predicción (umbral 0.5)
$[-1\ -1\ -1\ -1]$	0	0.113	0
$[1\ -1\ -1\ -1]$	1	0.950	1
$[-1\ 1\ -1\ -1]$	1	0.931	1
$[1\ 1\ -1\ -1]$	0	0.028	0
$[-1\ -1\ 1\ -1]$	1	0.912	1
$[1\ -1\ 1\ -1]$	0	0.018	0
$[-1\ 1\ 1\ -1]$	0	0.062	0
$[1\ 1\ 1\ -1]$	0	0.003	0
$[-1\ -1\ -1\ 1]$	1	0.921	1
$[1\ -1\ -1\ 1]$	0	0.042	0
$[-1\ 1\ -1\ 1]$	0	0.017	0
$[1\ 1\ -1\ 1]$	0	0.017	0
$[-1\ -1\ 1\ 1]$	0	0.012	0
$[1\ -1\ 1\ 1]$	0	0.010	0
$[-1\ 1\ 1\ 1]$	0	0.006	0
$[1\ 1\ 1\ 1]$	0	0.052	0

Cuadro 6: Resultados del perceptrón con cuatro entradas y 10 neuronas ocultas.

4.3. Análisis

El análisis ya se desarrolló casi completamente en cada subsección. Lo que se puede afirmar es que la función XOR de 2 entradas es relativamente simple de aprender para un MLP (3 perceptrones mínimo; 2 en la capa oculta), pero no tanto la XOR de 4 entradas, que requirió de una capa oculta bastante más grande (10 neuronas). Esto demuestra que siguen existiendo limitaciones a la cantidad de patrones que puede aprender una cierta red de perceptrones, y que parecería aumentar con la cantidad de neuronas agregadas.

Algo que no se mencionó es que los errores tienen forma de exponenciales decrecientes (aunque a veces deformes), que parecería normal para estos casos. Algunos entrenamientos presentaron como puntos silla, lo que indicaría que el algoritmo se encontró con una región de gradiente muy pequeño, como la figura 9, que se quedó varias epochs en 0.25 de ECM.

otro tema interesante para mencionar es que, como el método de gradiente usado no es estocástico, el resultado final del aprendizaje es determinístico. Esto trae como consecuencia que, si el espacio de soluciones de error mínimo es pequeño, sería difícil encontrar justo la condición inicial que caiga en el (más bien, no se encontró para menos de 10 neuronas). Claramente, el uso de más neuronas da más grados de libertad a la red, y más capacidad, y todo esto parecería ser acompañado por un incremento en accesibilidad para las regiones que son solución al problema. Volviendo al punto anterior, es posible que haya soluciones en espacios de menor dimensión que el de la red de 10 neuronas en la capa oculta, pero no fueron accedidos por la regla de aprendizaje. Opino que existe la posibilidad de que la función XOR de 4 entradas se pueda aprender con menos neuronas pero con el uso de alguna regla que meta estocacidad en el sistema, como el uso de un gradiente estocástico. Este punto se retoma en el inciso de *simulated annealing*, donde se comprueba esta hipótesis. También sería interesante saber si un perceptrón simple de 4 entradas con función de activación no monótona podría aprenderlo (intuyo que si).

5. Ejercicio 4

5.1. Consignas

a - Implemente una red con aprendizaje Backpropagation que aprenda la siguiente función:

$$f(x, y, z) = \sin(x) + \cos(y) + z$$

Donde: x e $y \in [0, 2\pi]$ y $z \in [-1, 1]$.

Para ello construya un conjunto de datos de entrenamiento y un conjunto de evaluación. Muestre la evolución del error de entrenamiento y de evaluación en función de las épocas de entrenamiento.

b - Estudie la evolución de los errores durante el entrenamiento de una red con una capa oculta de 30 neuronas cuando el conjunto de entrenamiento contiene 40 muestras. ¿Que ocurre si el *minibatch* tiene tamaño 40? ¿Y si tiene tamaño 1?

5.2. Desarrollo

El conjunto de datos se generó sin mayores problemas, fue solo evaluar la función $f(x, y, z)$ en muchos puntos.

Los datasets de entrenamiento y testeo se separaron 70/30 (56000 muestras de entrenamiento) y se entrenó variando el tamaño de *minibatch*. El uso de *minibatches* implica la utilización de un gradiente estocástico, que es más rápido (menos datos, menos cuentas) y ayuda con la convergencia ya que la aleatoriedad introducida por no usar todas las muestras causa que no siempre se caiga en las mismas cuencas de la función de costo.

En esta implementación se separaron los datos en tantos batches como corresponde según el tamaño, y todos se mostraron 10 veces en un orden aleatorio (pero nunca seguidos, todos se ostraban antes de volver a seleccionar uno ya mostrado). Se intentó balancear la cantidad de "veces" (en el código, "veces" sería la cantida de veces que se muestran todos los *minibatches*) para que sea justo entre modelos de tamaños de *minibatch* diferentes.

Los MLPs son de 30 neuronas en una sola capa oculta. Se mantuvieron las mismas condiciones para los 4 experimentos, para no sesgar los resultados. La validación se hizo con todo el set de testeo.

5.3. Inciso a

5.3.1. *minibatch* de 40 muestras

Las figuras 18 y 19 corresponden a un *minibatch* de 40 muestras.

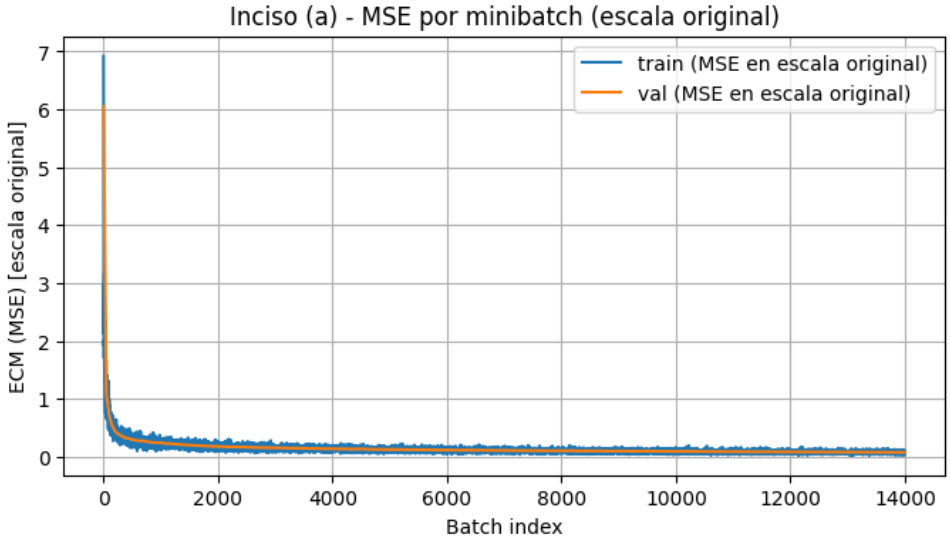

Figura 18: ECM de entrenamiento y validación - *minibatch* de 40 muestras, 10 veces, 1 epoch

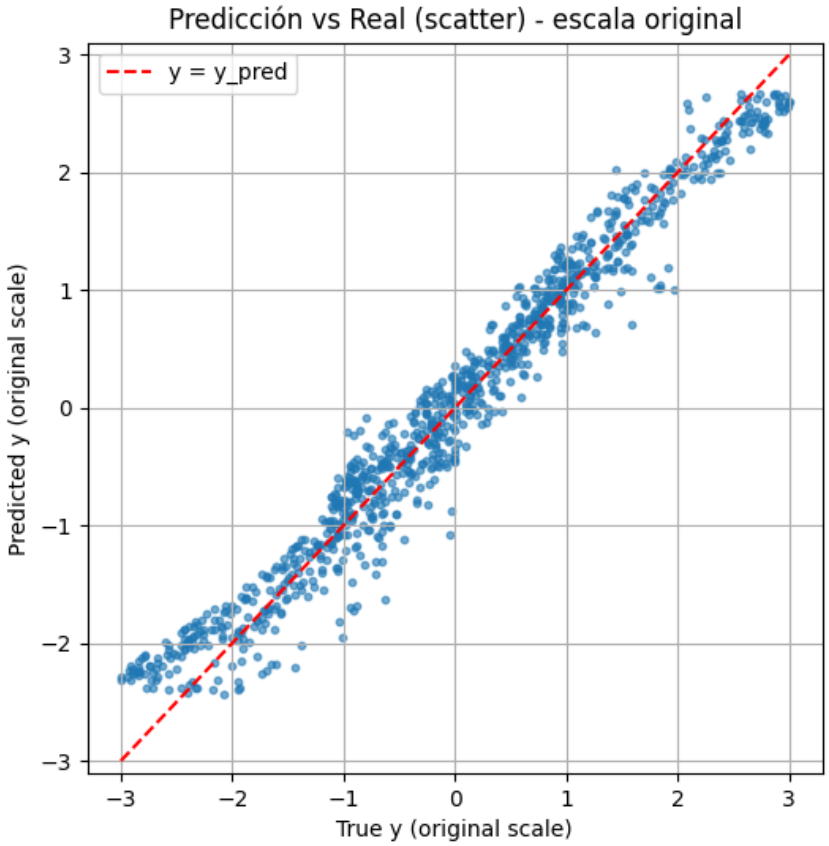

Figura 19: Predicción del modelo en base a (X,Y,Z) - *minibatch* de 40 muestras, 10 veces, 1 epoch

5.3.2. *minibatch* de 100 muestras

Las figuras 20 y 21 corresponden a un *minibatch* de 100 muestras. L

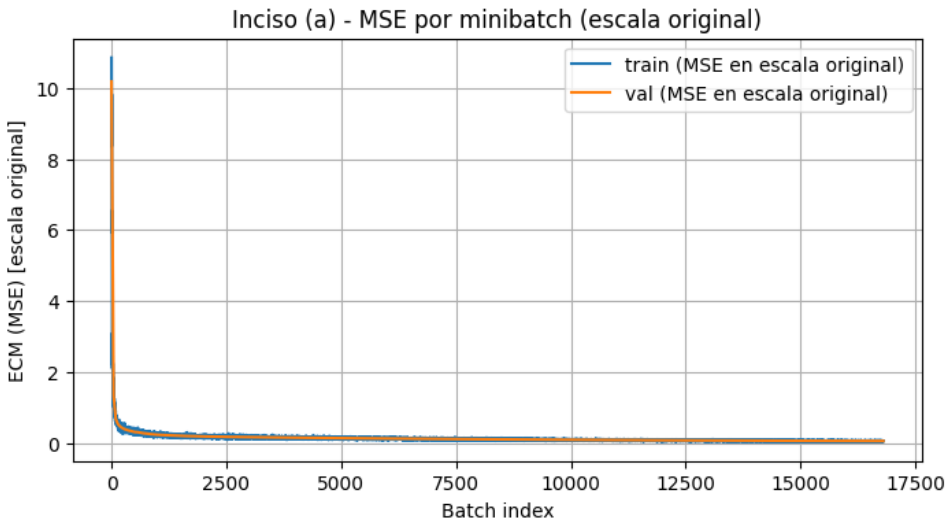

Figura 20: ECM de entrenamiento y validación - *minibatch* de 100 muestras

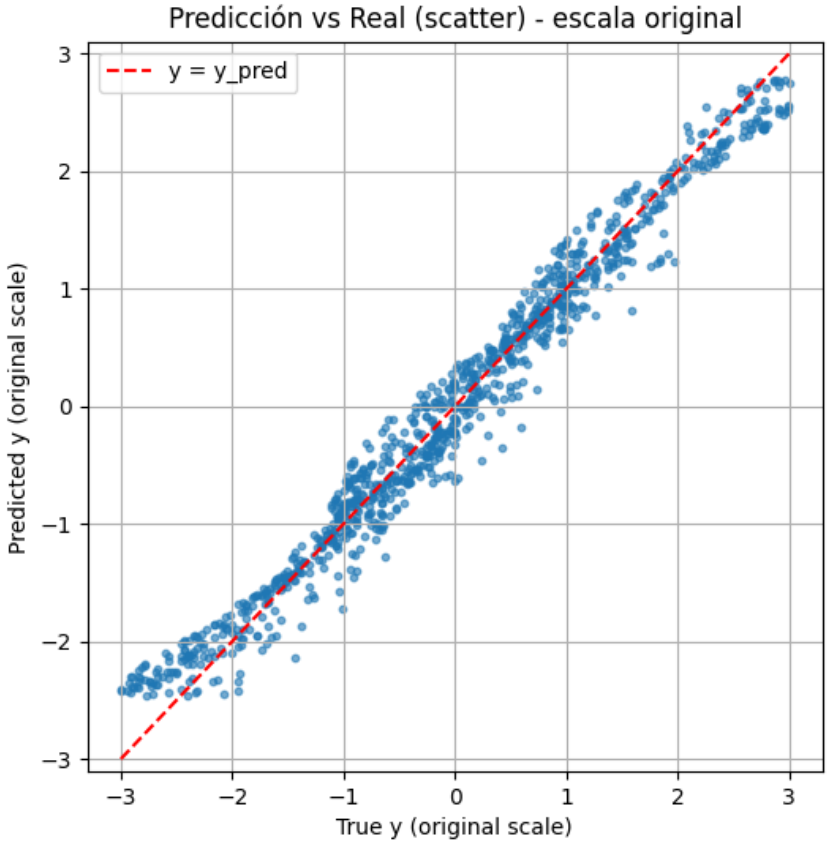

Figura 21: Predicción del modelo en base a (X,Y,Z) - *minibatch* de 100 muestras

5.3.3. *minibatch* de 1000 muestras

Las figuras 22 y 23 corresponden a un *minibatch* de 1000 muestras.

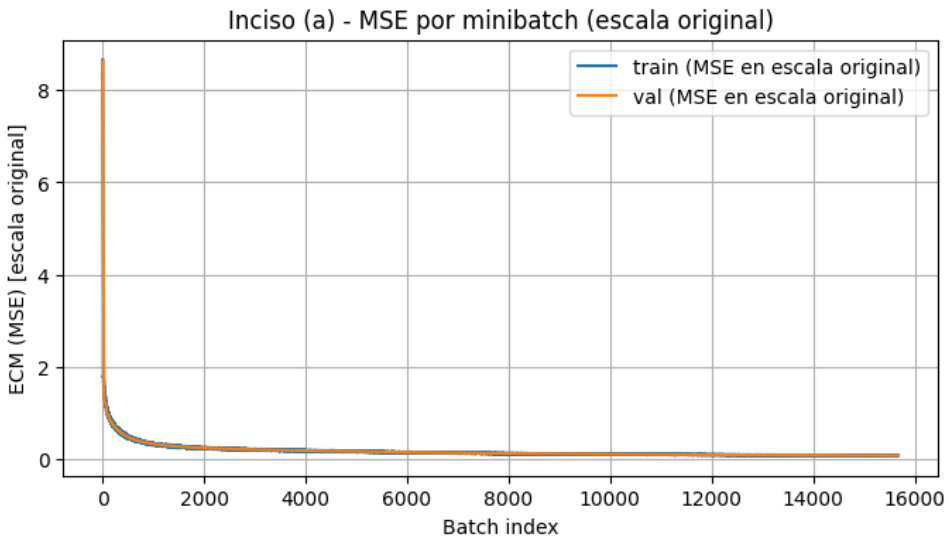

Figura 22: ECM de entrenamiento y validación - *minibatch* de 1000 muestras

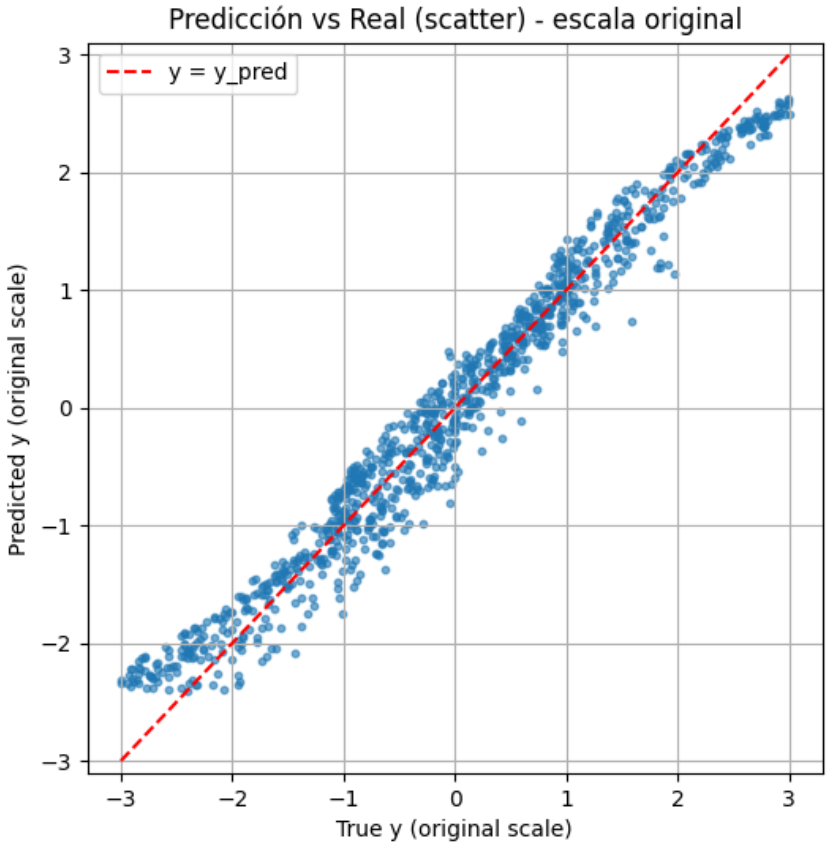

Figura 23: Predicción del modelo en base a (X,Y,Z) - *minibatch* de 1000 muestras

5.3.4. Análisis

El incremento del tamaño de *minibatch* manteniendo la cantidad de iteraciones constante hizo que se tardara mucho más en entrenar para los *minibatches* de 1000 que para el de 100 y el de 40. Claramente se entrenó con más iteraciones que las necesarias, evidente en el hecho de que el ECM converge para la iteración 4000 en los 3 casos.

El ECm de entrenamiento es más ruidoso con el *minibatch* más chico, lo que tiene sentido porque es el que tiene el gradiente más estocástico, de menos muestras. El ECM es ruidoso porque en cada pasada la dirección de descenso depende del *minibatch* de ese instante, que puede no coincidir con la anterior e incluso moverse en una dirección tal que aumenta el ECM.

Los 3 casos mostrados predicen con una forma aproximadamente igual, aunque considero que el mejor predictor es el de *minibatch* de 1000; parece el menos disperso de los 3 a lo largo de toda la recta. Los 3 casos presentan una aparente saturación en los extremos de la recta, que recuerda a una sigmoidea, y es muy probable que sea causado por la misma función de activación.

5.4. Inciso b

5.4.1. *minibatch* de 40 muestras

Las figuras 24 y 25 muestran el resultado de entrenar un MLP con *minibatch* de 40 muestras en base a solo 40 muestras de entrenamiento. El entrenamiento se ejecutó por la misma cantidad de tiempo que el de *minibatch* de 1 muestra, pero este no convergió a una solución adecuada, como se aprecia en la predicción.

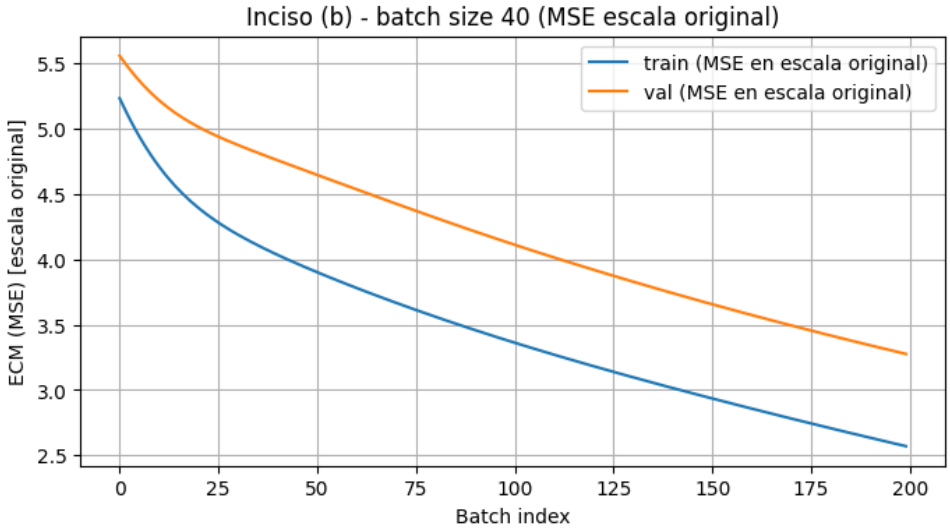

Figura 24: ECM de entrenamiento y validación - no hubo convergencia

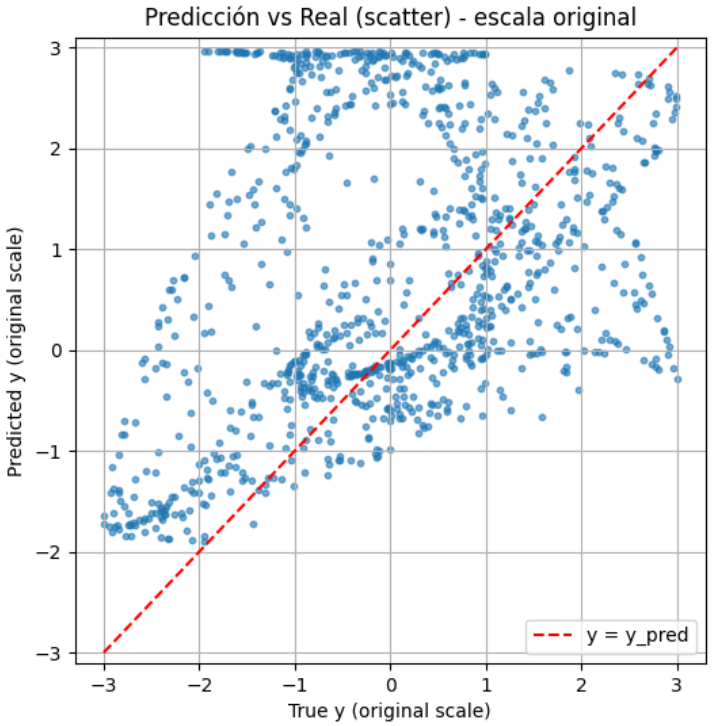

Figura 25: Predicción del modelo en base a (X,Y,Z)

Las figuras 26 y 27 son el resultado de tomar el modelo de las imágenes de antes pero darle más iteraciones de aprendizaje. Ahora si se observa convergencia en los EC; y la predicción parecería menos dispersa.

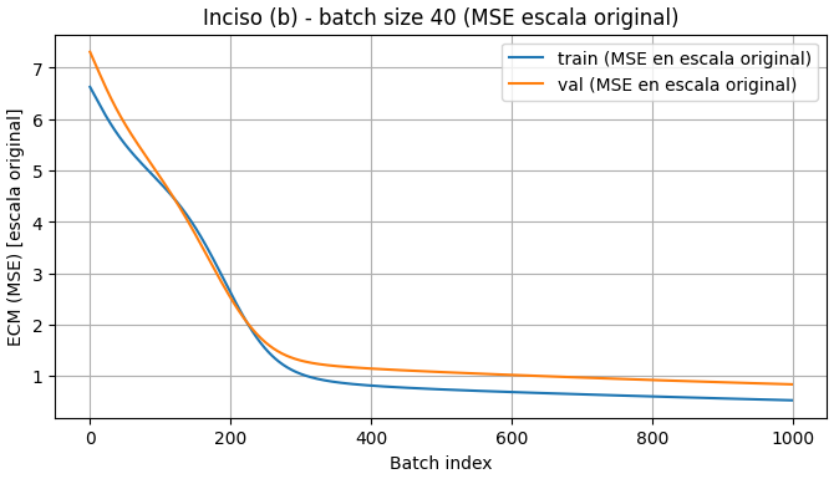

Figura 26

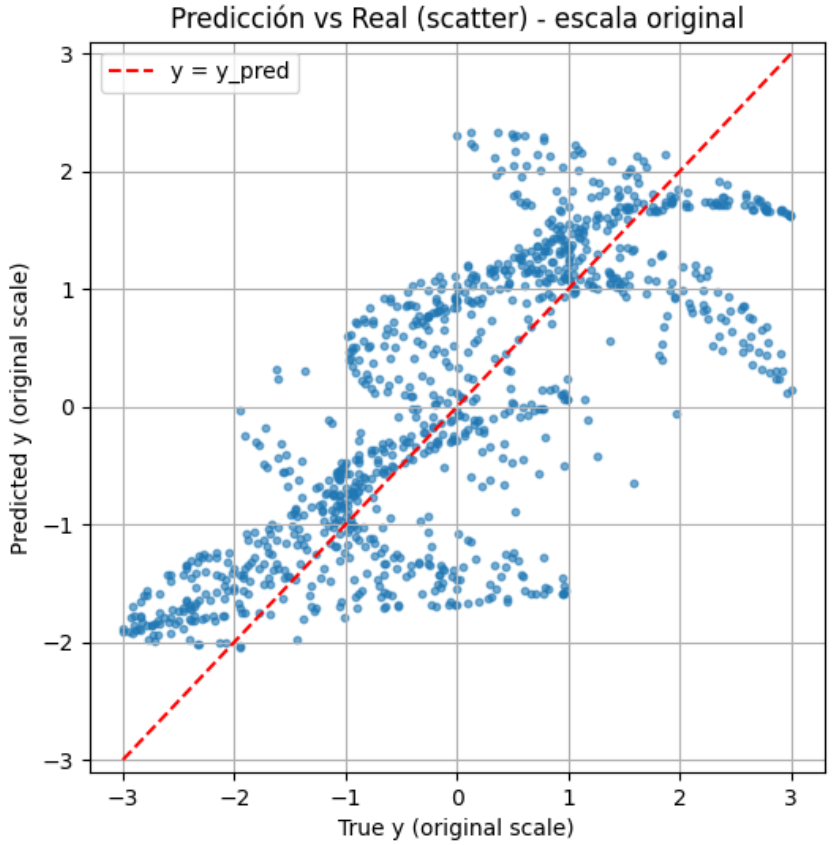

Figura 27

5.4.2. *minibatch* de 1 muestras

Las figuras 24 y 25 muestran el resultado de entrenar un MLP con *minibatch* de 1 muestra en base a solo 40 muestras de entrenamiento.

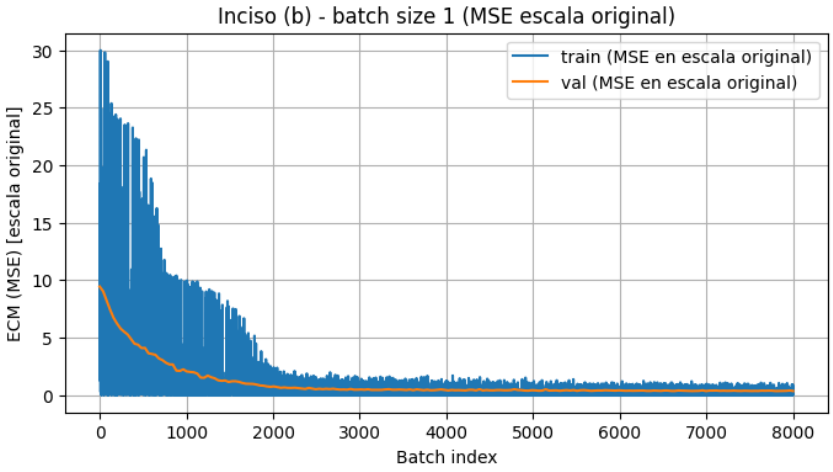

Figura 28: ECM de entrenamiento y validación - Hubo convergencia

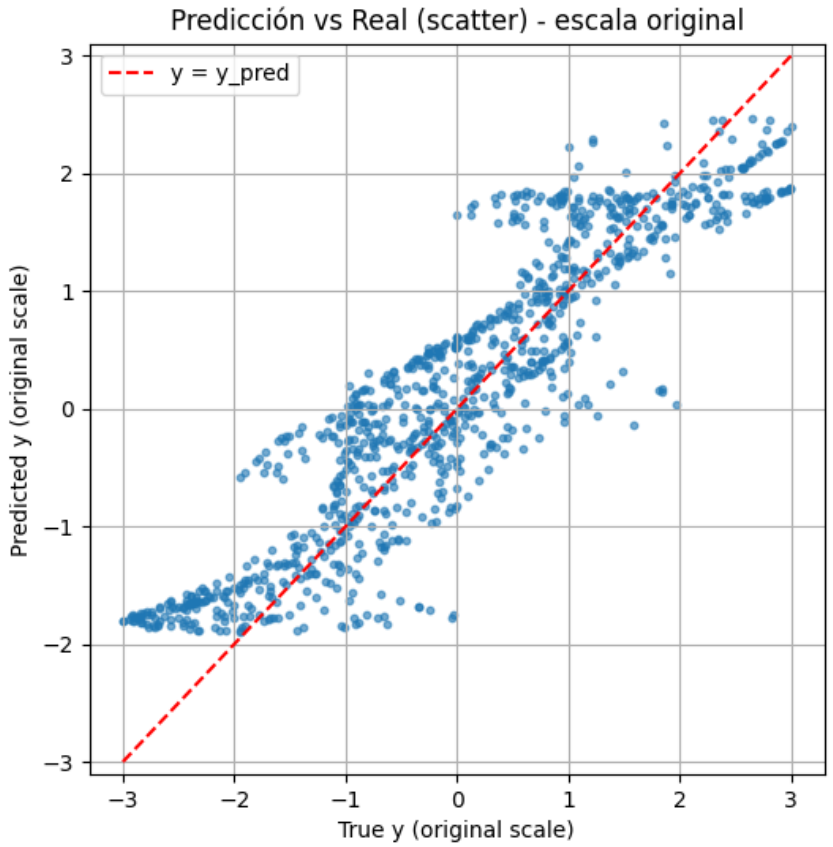

Figura 29: Predicción del modelo en base a (X,Y,Z)

5.4.3. Análisis

Los resultados presentan dos puntos interesantes de destacar. Por un lado, limitar el set de entrenamiento a 40 muestras hace que no sea representativo de la dinámica que se quiere aprender/modelar/predecir y, por ende, el modelo va a tener mucha dificultad prediciendo con exactitud. Los 3 modelos parecen generalizar, es decir, para las varias ternas de entrada parece dar una salida diferentes, pero no es cercana a la deseada. Por otro lado, el modelo entrenado con *minibatch* de 1 tiene mejor predicción que los otros, lo que puede deberse a que las muestras fueron más representativas (porque son tomadas aleatoriamente) y/o un *minibatch* de 40 con 40 muestras ya no es estocástico (usa las mismas 40 siempre) y entonces no se goza de los beneficios de un gradiente estocástico. Estos son varios pero el principal es que ayuda a la convergencia a mejores mínimos de la función de error.

5.5. Análisis global

Como resumen de los 2 análisis previos, se desea resaltar que el *minibatch* (indirectamente el gradiente estocástico) parecería aumentar la velocidad del proceso de aprendizaje, y produce resultados similarmente a los obtenidos con tamaños de lotes mayores. Esto es indispensable para la optimización en datasets grandes. Otro beneficio del uso de *minibatches* es que cada ejecución es única (por el orden de las muestras), por lo que el camino que recorre en la minimización varía con la ejecución, permitiendo evitar mínimos locales que posiblemente trabarían un aprendizaje por gradiente determinístico.

Otro punto es que, aunque la cantidad de muestras es importante, y condiciona la capacidad de predicción del modelo, también es imperativo elegir el método de aprendizaje correcto. Se pueden tener resultados muy diferentes según el tamaño de *minibatch*, como se notó en el segundo inciso del ejercicio.

En líneas generales, la consigna permitió entender que el resultado y proceso de entrenamiento no depende puramente de los datos (aunque si importan), sino que además de como se separan y usan dentro del proceso.

6. Ejercicio 5

6.1. Consignas

Siguiendo el trabajo de Hinton y Salakhutdinov (2006), entrene una máquina restringida de Boltzmann (RBM o MRB) con imágenes de la base de datos MNIST. Muestre el error de reconstrucción durante el entrenamiento, y ejemplos de cada uno de los dígitos reconstruidos.

6.2. Desarrollo

La MRB se puede pensar como una red *feedfoward* de 4 capas, en vez de pensarla como una de 2 capas con pesos simétricos. v es la capa visible y h es la oculta, y tenemos un $\hat{v}$ y $\hat{h}$, que son las estimaciones.

Las fórmulas que hay que seguir son las siguientes:

$$m_i = \text{pixel}_i$$

$$v_i \sim \mathcal{N}(m_i, 1)$$

$$P_r(j) = g(\sum_i w_{ij} v_i + b_j) \qquad g(x) = \frac{1}{1+e^{-x}}$$

$$h(j) = \begin{cases} 1 & \text{con probabilidad} \quad P_r \\ 0 & \text{con probabilidad} \quad 1 - P_r \end{cases}$$

$$\text{guardar} \quad (v_i, h_j)_{data}$$

Luego se repiten los mismos pasos pero para obtener las estimaciones:

$$m_i = \sum_j w_{ij} v_i + b_i$$

$$\hat{v}_i \sim \mathcal{N}(m_i, 1)$$

$$P_r(j) = g(\sum_i w_{ij} v_i + b_j) \qquad g(x) = \frac{1}{1+e^{-x}}$$

$$\hat{h}(j) = \begin{cases} 1 & \text{con probabilidad} \quad P_r \\ 0 & \text{con probabilidad} \quad 1 - P_r \end{cases}$$

$$\text{guardar} \quad (v_i, h_j)_{reconstruccion}$$

Finalmente se hace:

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle - \left\langle \hat{v_i} \hat{h_j} \right\rangle)$$

$$b_i = \eta(\langle v_i \rangle - \langle \hat{v_i} \rangle)$$

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle - \left\langle \hat{v_i} \hat{h_j} \right\rangle)$$

$$b_j = \eta(\langle h_j \rangle - \left\langle \hat{h_j} \right\rangle)$$

A grandes rasgos, la idea es que se tiene un proceso de aprendizaje no supervisado donde el modelo ajusta sus pesos (y biases) para que las correlaciones entre las unidades visibles y ocultas reconstruidas $(\hat{v}, \hat{h})$ se asemejen lo más posible a las del conjunto de datos original (v, h); es decir, las fórmulas buscan reducir la diferencia entre las estadísticas del modelo y las de los datos reales, permitiendo que la red aprenda a representar la distribución subyacente de las muestras de entrada. El código refleja toda esta lógica.

Una de las finalidades de la MRB es obtener una representación de los datos de menor dimensión que la entrada, de modo que las variables en la capa de menor dimensión capturen las dinámicas de mayor importancia (o varianza, como en PCA). La otra función es obtener un *autoencoder* no lineal (no es PCA), capaz de capturar y modelar relaciones complejas en los datos.

Se hicieron 3 experimentos, con 3 capas intermedias de 400, 200 y 100 neuronas. Todos se entrenaron hasta llegar a la convergencia.

Las figuras 30 y 31 muestran el ECM de reconstrucción y reconstrucciones para 400 neuronas en la capa más chica (también denominada intermedia). A modo de referencia, las imágenes de MNISTS son de 28x28, por lo que la capa de entrada es de 784.

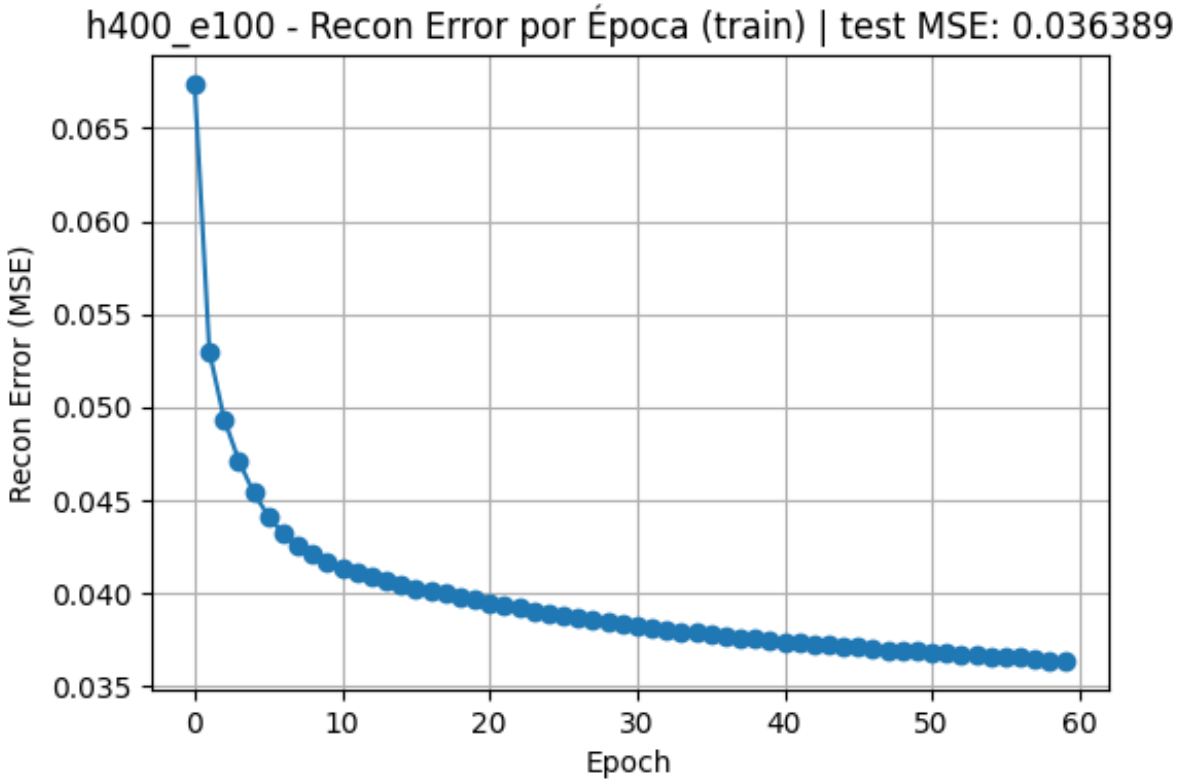

Figura 30: ECM de reconstrucción - 400 neuronas en capa intermedia de menor tamaño

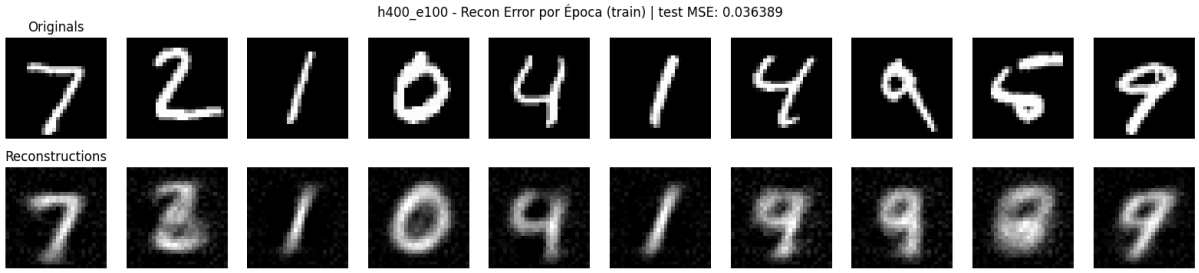

Figura 31: Reconstrucciones - 400 neuronas en capa intermedia de menor tamaño

Las figuras 32 y 33 muestran el ECM de reconstrucción y reconstrucciones para 200 neuronas en la capa más chica.

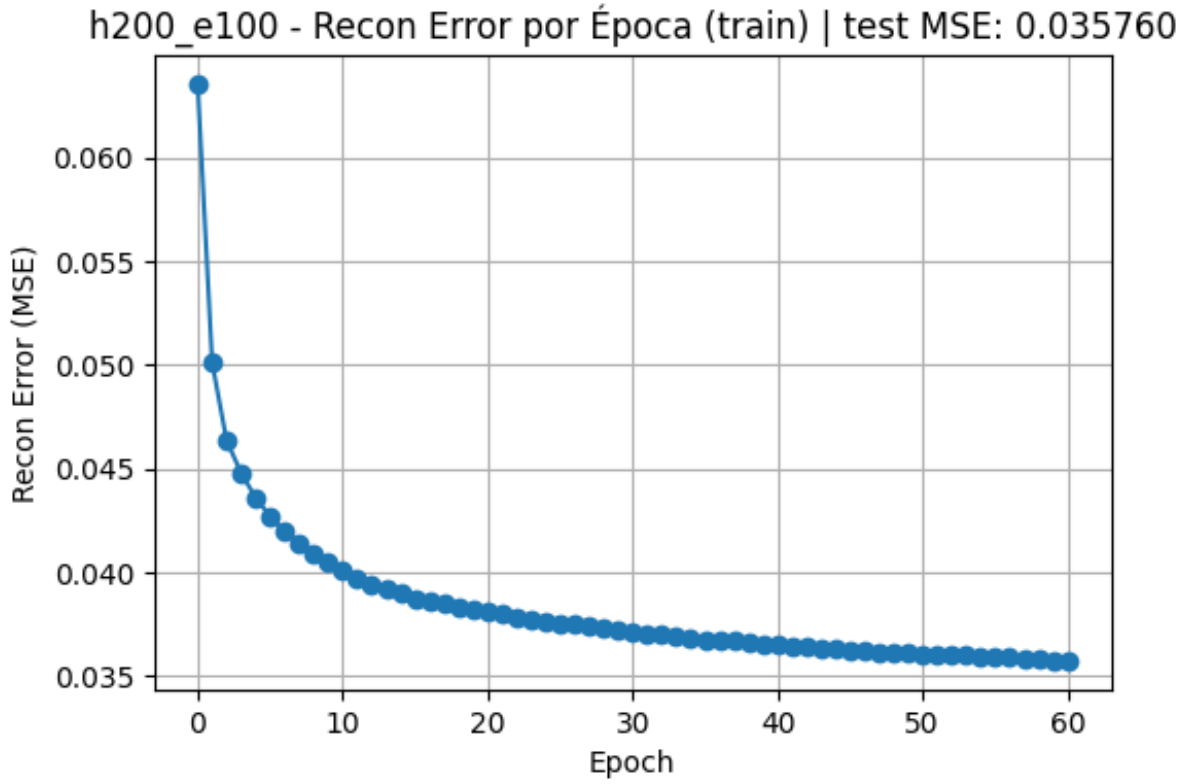

Figura 32: ECM de reconstrucción - 200 neuronas en capa intermedia de menor tamaño

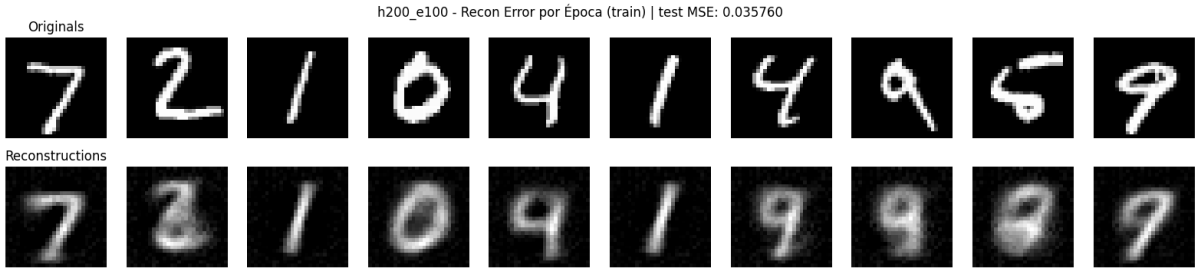

Figura 33: Reconstrucciones - 200 neuronas en capa intermedia de menor tamaño

Las figuras 34 y 35 muestran el ECM de reconstrucción y reconstrucciones para 100 neuronas en la capa más chica.

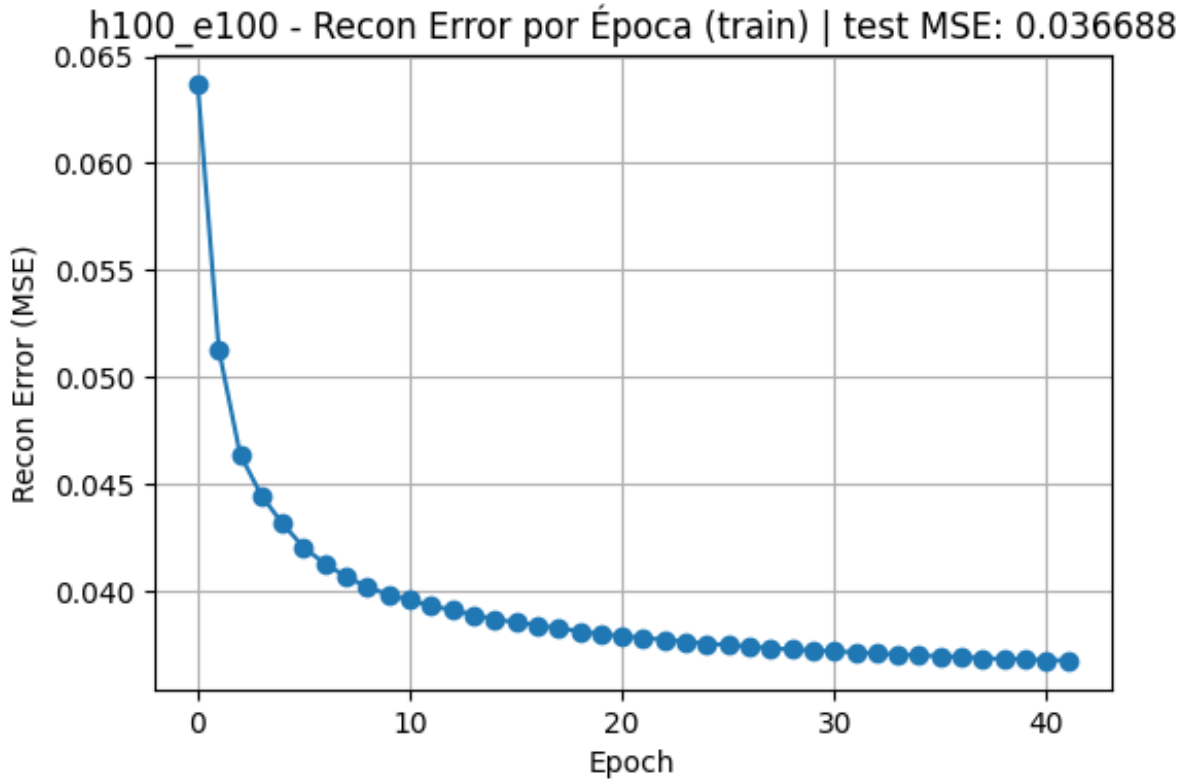

Figura 34: ECM de reconstrucción - 100 neuronas en capa intermedia de menor tamaño

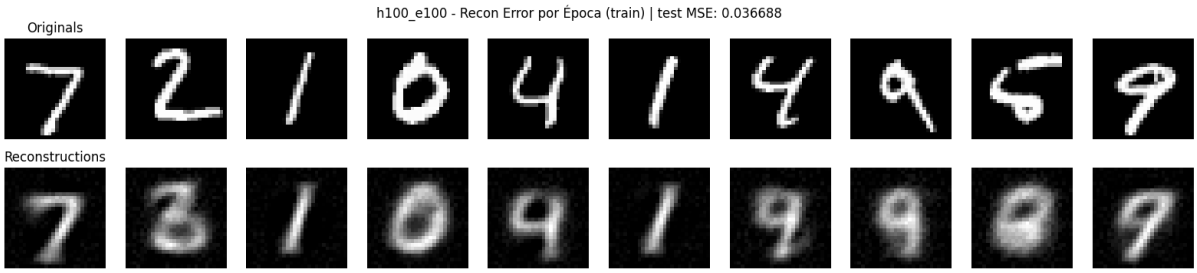

Figura 35: Reconstrucciones - 100 neuronas en capa intermedia de menor tamaño

Las figuras 36 y 37 muestran el ECM de reconstrucción y reconstrucciones para 50 neuronas en la capa más chica.

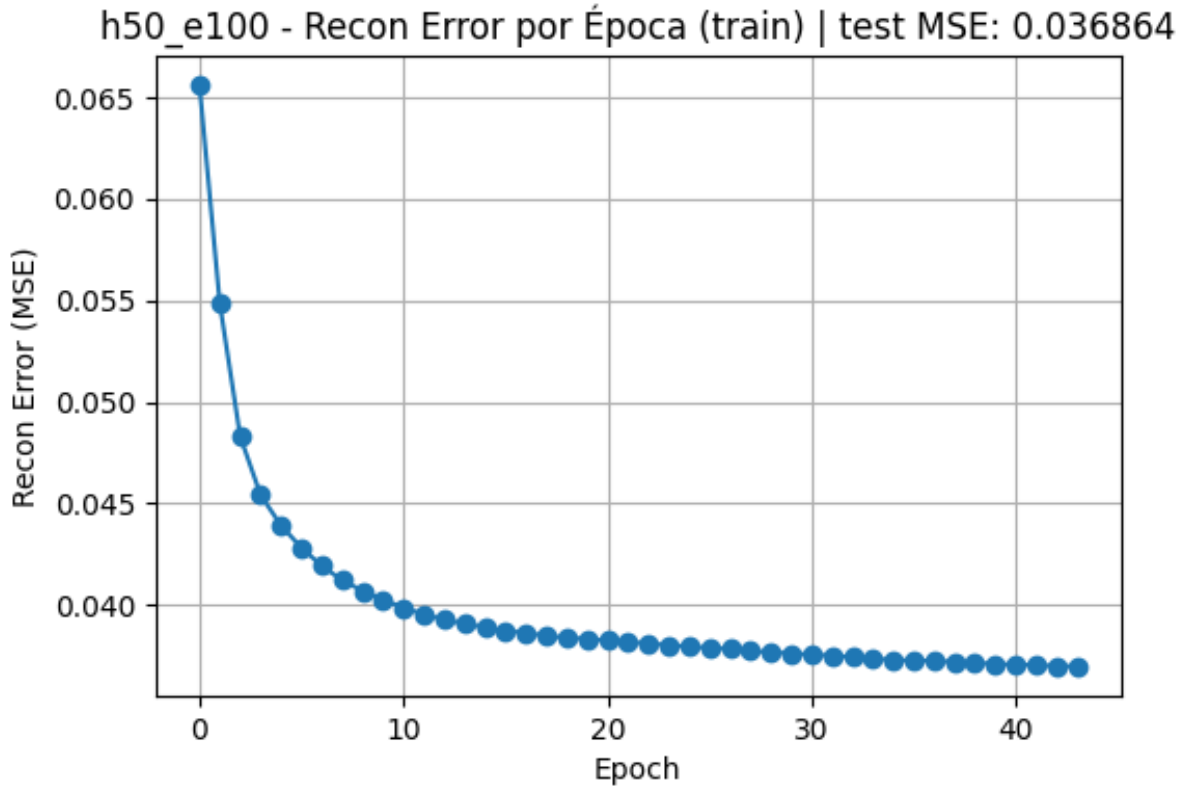

Figura 36: ECM de reconstrucción - 50 neuronas en capa intermedia de menor tamaño

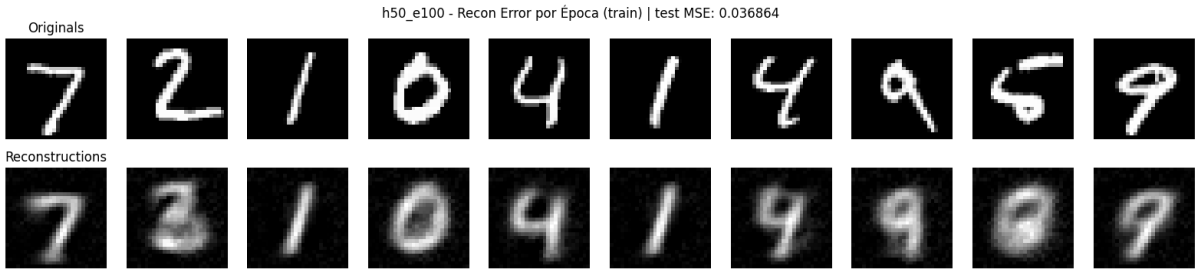

Figura 37: Reconstrucciones - 50 neuronas en capa intermedia de menor tamaño

Las figuras 38 y 39 muestran el ECM de reconstrucción y reconstrucciones para 10 neuronas en la capa más chica.

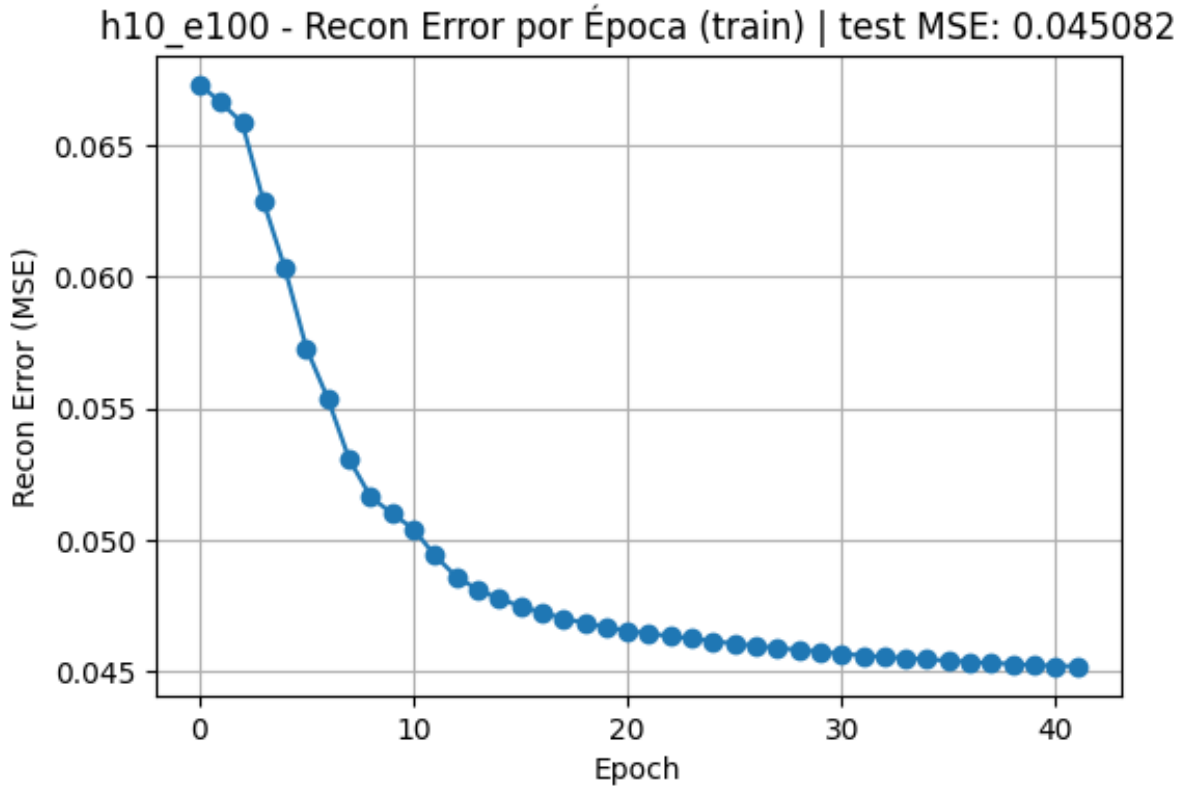

Figura 38: ECM de reconstrucción - 10 neuronas en capa intermedia de menor tamaño

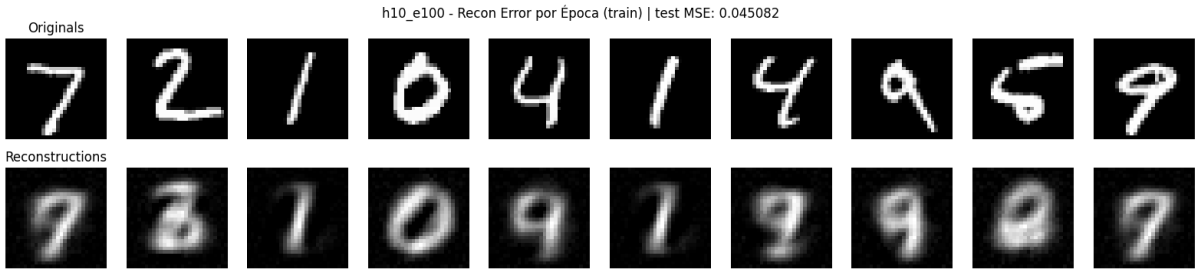

Figura 39: Reconstrucciones - 10 neuronas en capa intermedia de menor tamaño

Las figuras 40 y 41 muestran el ECM de reconstrucción y reconstrucciones para 1 neurona en la capa más chica.

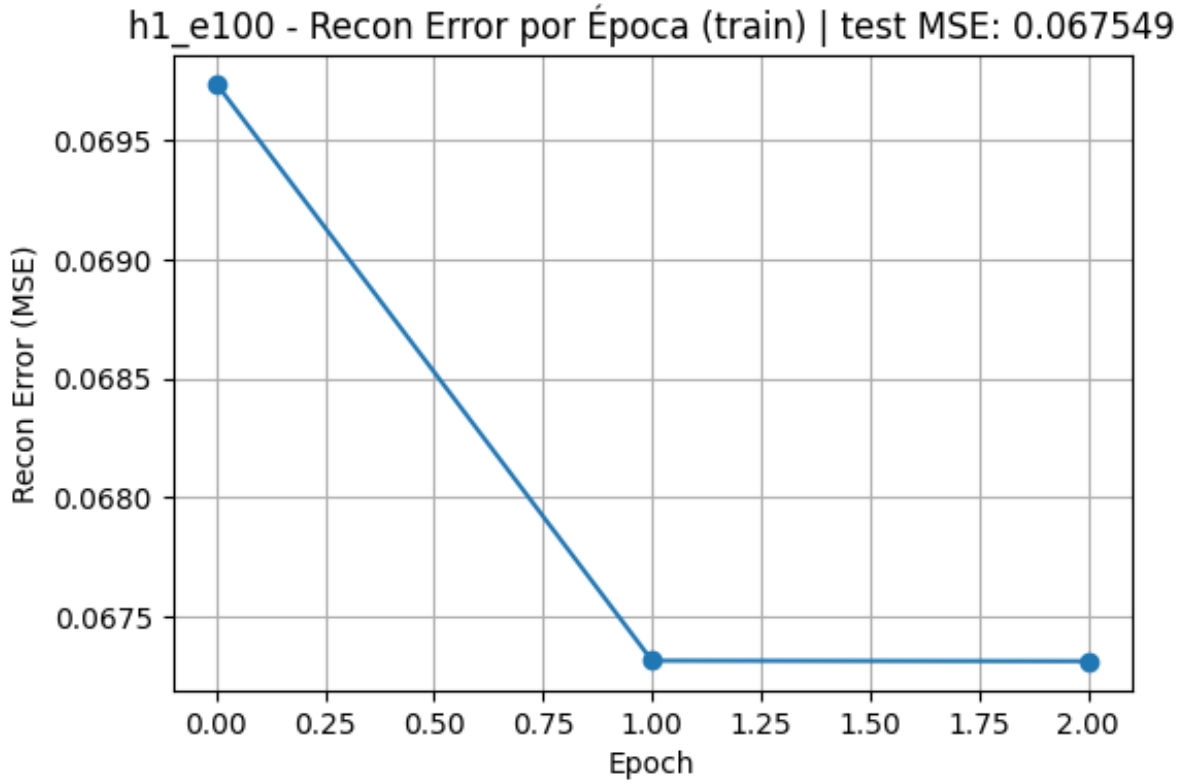

Figura 40: ECM de reconstrucción - 1 neurona en capa intermedia de menor tamaño

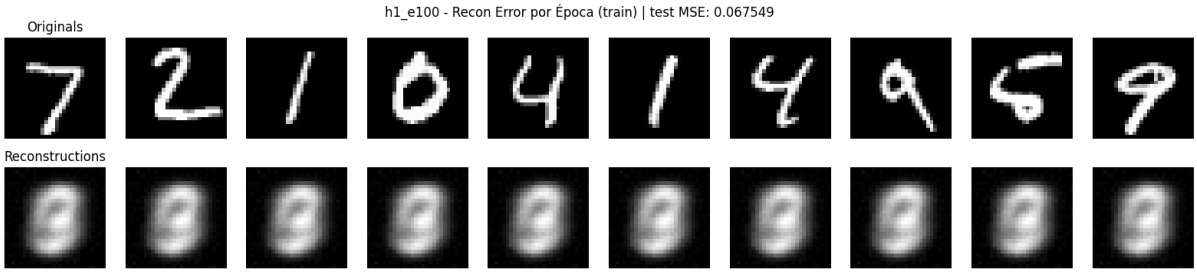

Figura 41: Reconstrucciones - 1 neurona en capa intermedia de menor tamaño

6.3. Análisis

Se entiende que, a más neuronas en la capa oculta mínima (sería la que buscamos que haga la reducción de dimensionalidad), más información/energía/varianza queda captada por el modelo reducido, a costas de una menor reducción de dimensionalidad. Como consecuencia de "quedarse" más dimensiones, las reconstrucciones deberían ser mejores con más neuronas en esta capa.

Todas las imágenes recién presentada muestran 2 tendencias claras:

- A medida que se achica la capa intermedia , las reconstrucciones se vuelven menos claras y similares al dígito original. En el extremo, la de 1 sola neurona reconstruye siempre la misma imagen.
- El ECM de reconstrucción aumenta a medida que se sacan neuronas de la capa intermedia.

Estos efectos observados tienen sentido, y suceden con PCA, que sería el método de reducción de dimensionalidad lineal. En este, cuando se quitan direcciones principales, se pierde información/energía/varianza de los datos de entrada, y para el caso de mantener solo la dirección de mayor varianza, el sistema siempre va a devolver el patrón más común a todos los datos, perdiéndose todo el detalle quye estaba en las direcciones de menos varianza.

7. Ejercicio 6

7.1. Consignas

Entrene una red convolucional para clasificar las imágenes de la base de datos MNIST. ¿Cuál es la red convolucional más pequeña que puede conseguir con una exactitud de al menos 90 % en el conjunto de evaluación? ¿Cuál es el perceptrón multicapa más pequeño que puede conseguir con la misma exactitud?

7.2. Desarrollo

Para este ejercicio se fueron generando modelos de CNN y de MLP cada vez más pequeños que lleguen al 90 % de *accuracy* de testeo/evaluación.

7.2.1. CNNs

La Figura 42 muestra la evolución de la pérdida de entrenamiento, la pérdida de validación y la precisión obtenida para la red neuronal convolucional. Esta red posee una capa convolucional que recibe un plano de entrada y genera 8 planos de salida, utilizando kernels de tamaño 5×5. Luego se aplica una capa de pooling con ventana de 2×2, que reduce a la mitad la resolución espacial, seguida de una capa completamente con 10 salidas. El modelo cuenta con 11,738 parámetros entrenables. Tras 15 épocas de entrenamiento se alcanzó una precisión del 98.46 %.

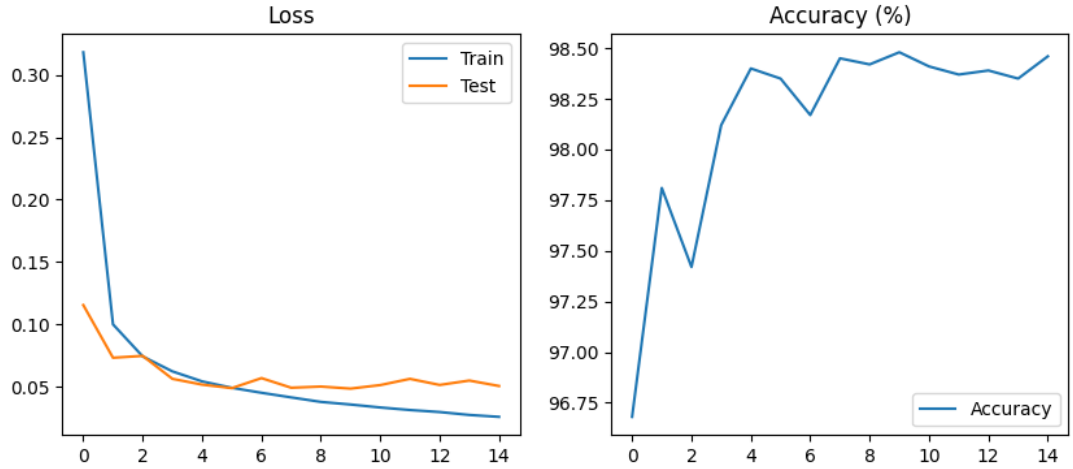

Figura 42: Evolución del error y precisión para la CNN N°1.

La Figura 43 corresponde a la segunda CNN entrenada, que posee una capa convolucional con un plano de entrada y 5 planos de salida, con kernels de tamaño 5×5. A continuación se aplica una operación de pooling con ventana de 2×2 y una capa completamente conectada de 10 salidas. El modelo tiene un total de 7,340 parámetros entrenables. Luego de 17 épocas, alcanzó una precisión del 98.10 %.

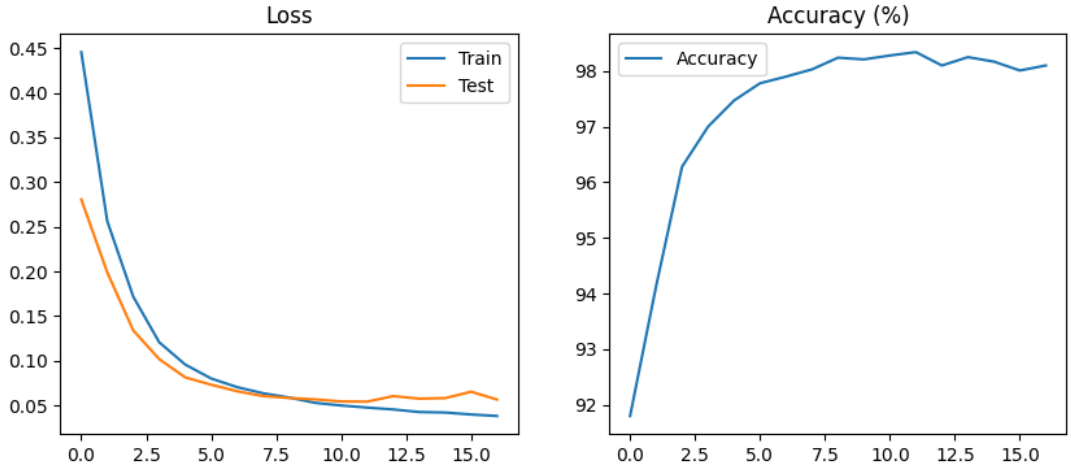

Figura 43: Evolución del error y precisión para la CNN N°2.

La Figura 44 muestra los resultados de la CNN número 3, compuesta por una capa convolucional que recibe un plano de entrada y produce 2 planos de salida mediante kernels de 5×5. Luego se aplica pooling con ventana de 2×2, y finalmente una capa lineal de 10 salidas. El modelo cuenta con 2,942 parámetros entrenables. Tras 30 épocas de entrenamiento, alcanzó una precisión del 97.01 %.

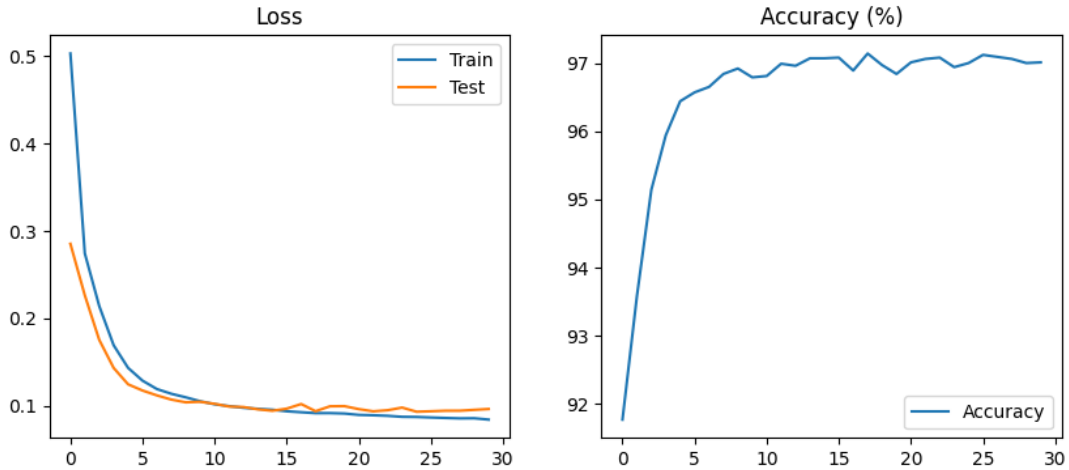

Figura 44: Evolución del error y precisión para la CNN N°3.

La Figura 45 presenta la cuarta CNN ideada, conformada por una capa convolucional con un plano de entrada y 2 planos de salida, con kernels de tamaño 9×9. Se aplica luego una capa de pooling de 2×2, seguida de una capa totalmente conectada de 10 salidas. El modelo tiene 2,174 parámetros entrenables. Tras 23 épocas, obtuvo una precisión del 97.09 %.

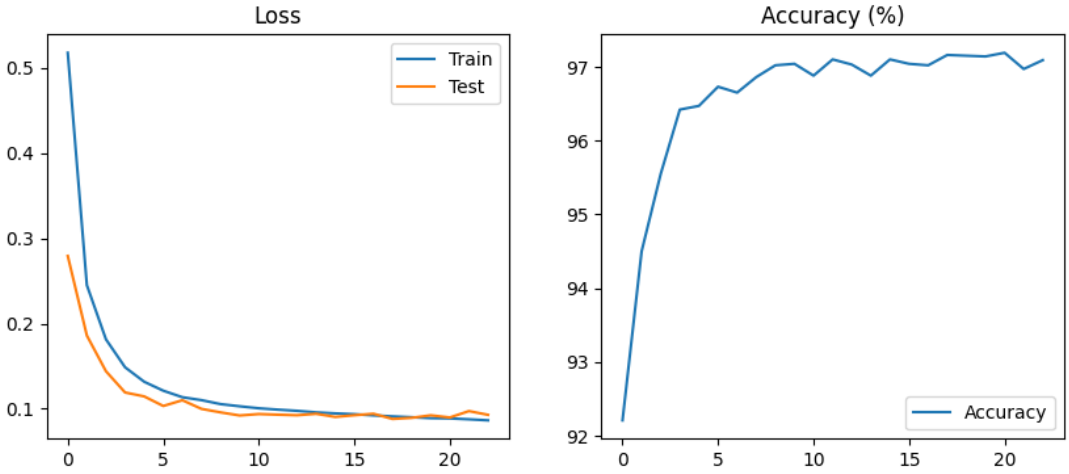

Figura 45: Evolución del error y precisión para la CNN N°4.

La Figura 46 corresponde a la CNN N°5, compuesta por una capa convolucional que recibe un plano de entrada y genera 2 planos de salida utilizando kernels de 15×15. Luego se aplica pooling con ventana de 2×2 y una capa lineal con 10 salidas. La red posee 1,442 parámetros entrenables. Tras 23 épocas, alcanzó una precisión del 95.32 %.

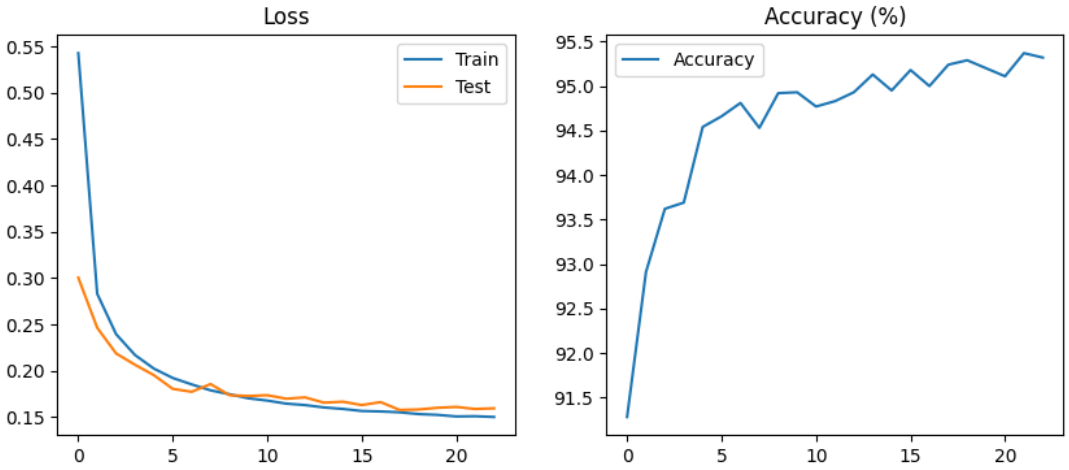

Figura 46: Evolución del error y precisión para la CNN N°5.

Finalmente, la Figura 47 muestra los resultados de la CNN más chica. Esta red posee una capa convolucional que recibe un plano de entrada y genera 2 planos de salida con kernels de 15×15, seguida de una operación de pooling con ventana de 4×4 y una capa completamente conectada con 18 entradas y 10 salidas. El modelo tiene 642 parámetros entrenables. Tras 30 épocas, se obtuvo una precisión del 91.58 %.

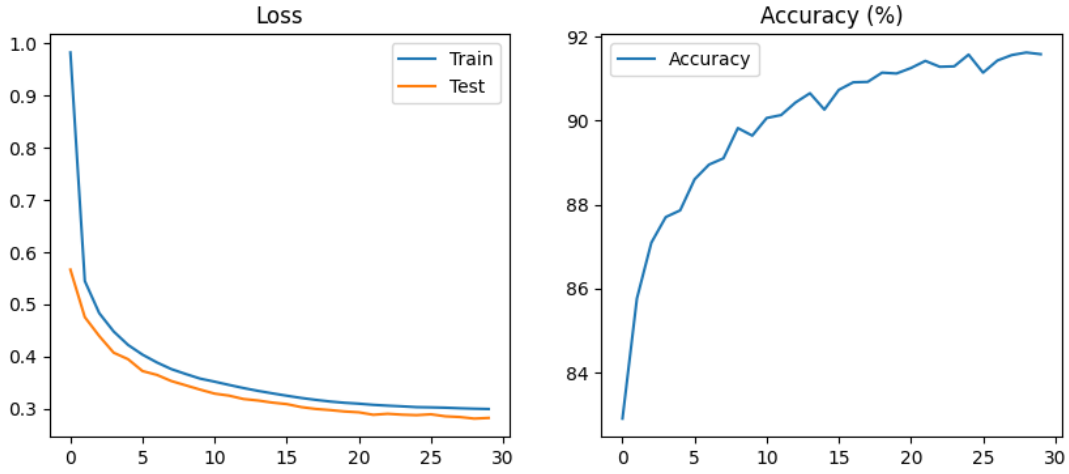

Figura 47: Evolución del error y precisión para la CNN N°6.

7.2.2. MLPs

La Figura 48 corresponde al MLP N°1, compuesto por dos capas lineales: una primera capa con 784 entradas y 64 salidas, y una segunda capa con 10 salidas. La red posee 50,890 parámetros entrenables. Tras 15 épocas, alcanzó una precisión del 97.36 %.

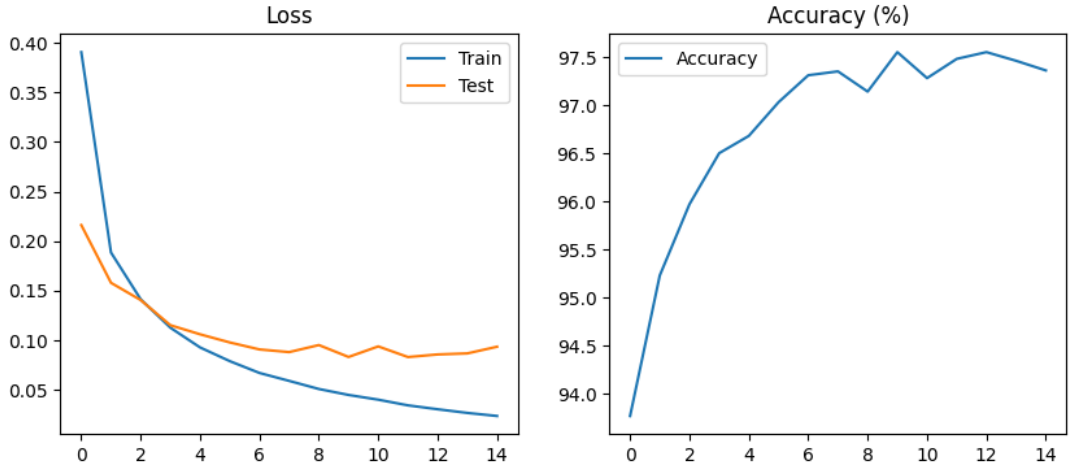

Figura 48: Evolución del error y precisión para el MLP1.

La Figura 49 presenta al MLP N°2, conformado por una única capa lineal con 784 entradas y 10 salidas. El modelo cuenta con 7,850 parámetros entrenables. Tras 18 épocas de entrenamiento, alcanzó una precisión del 92.86 %.

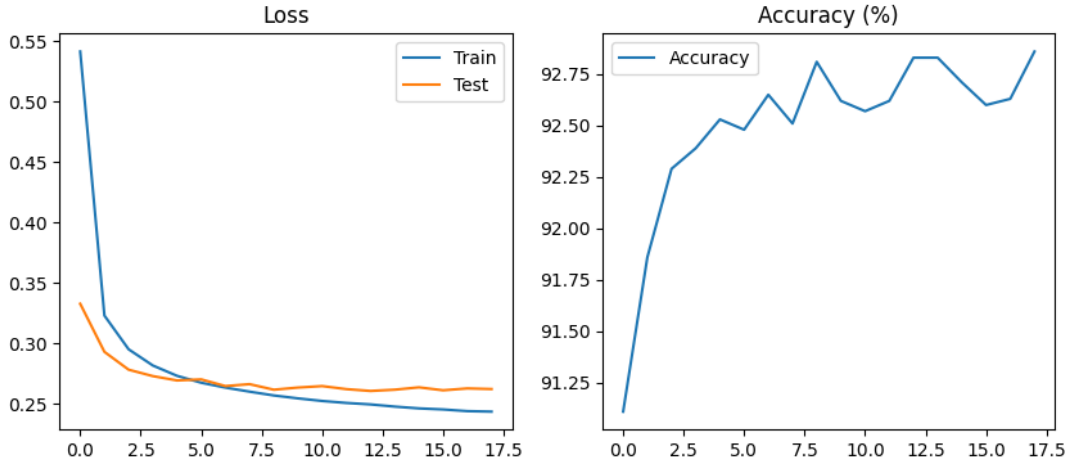

Figura 49: Evolución del error y precisión para el MLP2.

La Figura 50 muestra los resultados del MLP N°3, compuesto por una operación de *max-pooling* con ventana de 2×2, seguida de una capa lineal con 196 entradas y 10 salidas. El modelo tiene 1,970 parámetros entrenables. Tras 30 épocas, alcanzó una precisión del 92.17 %.

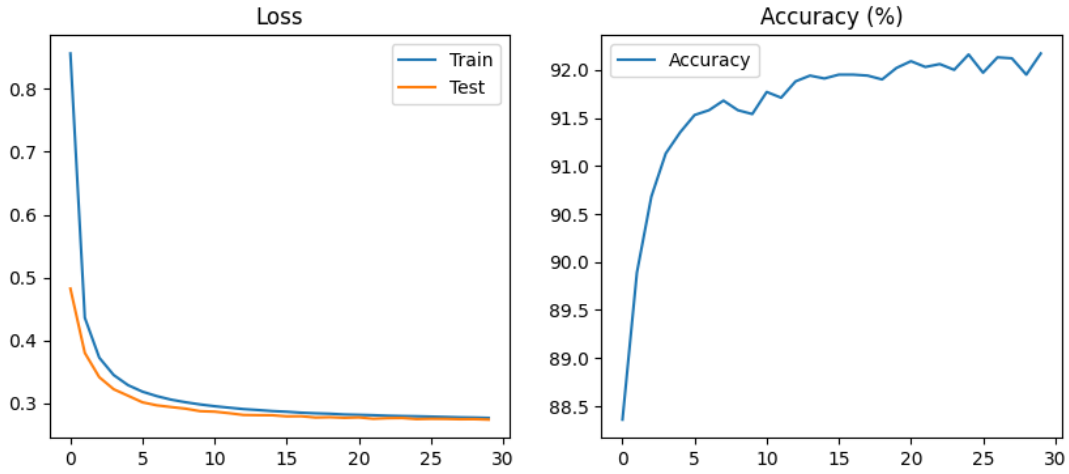

Figura 50: Evolución del error y precisión para el MLP3.

7.3. Análisis

Todos los gráficos se ven razonables: los errores disminuyen con las iteraciones y el *accuracy* tiende a subir. A veces el ECM de testeo/evaluación queda por debajo del de entrenamiento, que puede deberse a las imágenes del dataset de testeo. Se usó la proporción de train/test de 60000/10000 ($\sim$ 83/17) propia de MNIST, con *minibatches* de tamaño 64 (que aporta los beneficios ya mencionados en la consigna sobre el tema). Se usó la función de activación ReLu.

La CNN más chica encontrada es de 642 parámetros, y hace gran uso del pooling para disminuir fuertemente la dimensión de la capa lineal "*fully connected*" que nos da las 10 salidas (1 por dígito).

En cambio, el MLP más pequeño logrado fue de 1970 parámetros, y se hizo (según lo discutido en clase) haciendo "trampa" porque se aplicó pooling a los datos de entrada para poder reducir incluso más la dimensión del clasificador. Personalmente lo considero válido.

Es claro que el modelo con la capa convolucional es más eficiente en términos de cantidad de parámetros (son menos). La capa convolucional no solo es "barata" en términos de parámetros, sino que además permite hacer *feature extraction* local, es decir, sin perder la posición de la *feature*, y aparte aprende que características extaer para minimizar el error en la predicción/categorización. Esto último es crucial ya que evita tener que pensar de antemano que propiedades se deben extraer de lo que se desea clasificar, se deja en "manos" de la máquina para que lo aprenda. Ahora si, es posible que sea imposible para un humano entender que termina extrayendo cada plano de convolución [3].

[3] Algo así como cuando se obtiene una matriz para una representación de un sistema en espacio de estados. Es posible que la representación no esté dada en base a estados comprensibles (como velocidad, espacio, altura, etc.), sino que una mezcla indistinguible de ellos

8. Ejercicio 7

8.1. Consignas

Entrene un autoencoder para obtener una representación de baja dimensionalidad de las imágenes de MNIST. Use dichas representaciones para entrenar un perceptrón multicapa como clasificador. ¿Cuál es el tiempo de entrenamiento y la exactitud del clasificador obtenido cuando parte de la representación del autoencoder, en comparación con lo obtenido usando las imágenes originales?

8.2. Desarrollo

Primero se entrenó un *autoencoder* como los vistos en clase (pero sin hacer el proceso con las MRB). La finalidad es poder quedarse con la etapa de codificación, de *encoder*, que implicitamente ejecuta la función que transforma datos del dominio de la entrada al espacio de menor dimensión. Para este caso se eligió una reducción del 80 % de la dimensión hecho en 4 etapas: Se inició con una capa de 784 neuronas, que alimenta una de 392 y al final hay una de 156, una redución de 5 la dimensión de entrada. La función de activación es la ReLu y se usó en conjunto con "Adam". El entrenamiento del *autoencoder* fue no supervisado, solo requiere de los X, no de las *labels*.

La imagen 51 muestra la *loss* de entrenamiento y validación. Se observa convergencia. La figura 52 muestra los datos de entrada (los dígitos originales) y salida (reconstruidos) del *autoencoder*. Hay diferencias muy sutiles en la intensidad de los tonos. El *autoencoder* tomó ~ 8 mín en entrenarse.

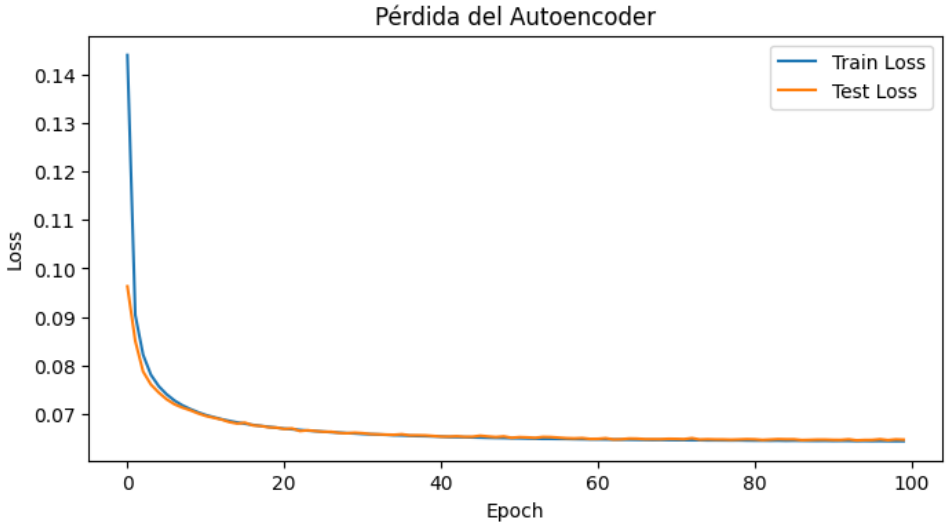

Figura 51: *Loss* de entrenamiento y testeo.

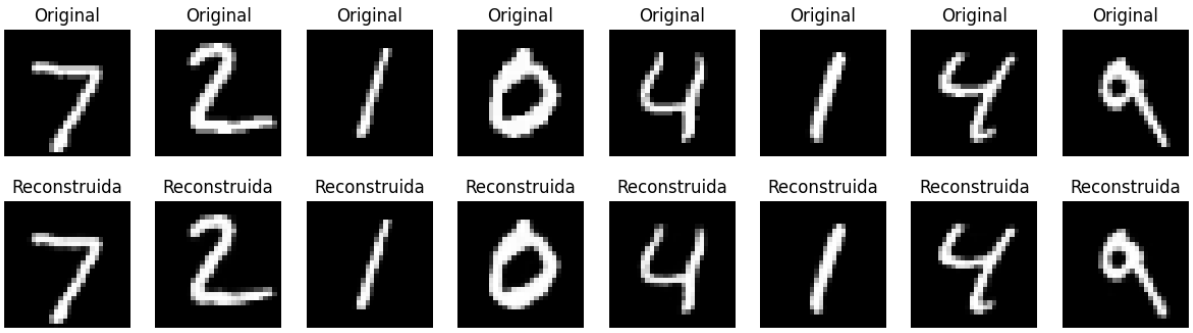

Figura 52: Dígitos originales y sus respectivas reconstrucciones.

Una vez entrenado el *autoencoder*, se extrajo el *encoder*, se congelaron sus pesos y se le acopló un MLP que pasa de esas 156 neuronas a 10. Una vez entrenada, esta etapa hizo la categorización de los dígitos de MNIST. EL entrenamiento de esta parte fue supervisado ya que requirió de las *labels* de las imágenes. El categorizador se entrenó por 100 epochs, y llegó a los resultados del cuadro 7. El entrenamiento tomó $\sim 6\,\text{mín}$ y $40\,\text{s}$

Cuadro 7: Precisión por dígito en el conjunto de prueba.

Dígito	**Accuracy [%]**	**Cantidad de muestras**
0	97.04	980
1	97.97	1135
2	90.50	1032
3	92.57	1010
4	90.84	982
5	81.95	892
6	94.15	958
7	90.76	1028
8	90.66	974
9	92.77	1009
Promedio	**91.92**	–
Global	**92.09**	**10,000**

Para comparar, se creó un MLP que pasa de la imagen de MNIST (tamaño 784) a 10 (la salida). La performance se ve en el cuadro 8 y el entrenamiento tomó ~ 7 mín.

Cuadro 8: Accuracy por dígito en el conjunto de prueba

Dígito	**Accuracy (%)**	**Cantidad de muestras**
0	96.94	980
1	97.53	1135
2	89.73	1032
3	90.10	1010
4	92.77	982
5	87.44	892
6	95.93	958
7	92.41	1028
8	90.97	974
9	92.57	1009
Promedio	**92.64**	–
Global	**92.73**	**10000**

También se probó con una red que usa el encoder pero no congela esas capas, sino que permite hacer cierto *fien-tuning* y otra con más capas ocultas en el MLP post *encoder*.

La red con *encoder fine-tuned* tardó SImín y logró los resultados del cuadro 9.

Cuadro 9: Accuracy por dígito en el conjunto de prueba (*encoder* con *fine-tuning*).

Dígito	**Accuracy (%)**	**Cantidad de muestras**
0	97.65	980
1	97.97	1135
2	90.41	1032
3	90.89	1010
4	93.28	982
5	85.87	892
6	95.09	958
7	91.54	1028
8	89.12	974
9	90.49	1009
Promedio	**92.23**	–
Global	**92.35**	**10000**

La red con otra capa oculta más y *encoder* congelado demoró $SI7$mín con 30 sec, y llegó a los resultados del cuadro 10.

Cuadro 10: Accuracy por dígito en el conjunto de prueba (*encoder* + FC intermedia 156→156→10).

Dígito	**Accuracy (%)**	**Cantidad de muestras**
0	99.08	980
1	99.65	1135
2	97.87	1032
3	98.71	1010
4	98.27	982
5	96.19	892
6	98.23	958
7	97.28	1028
8	97.64	974
9	96.13	1009
Promedio	**97.90**	–
Global	**97.94**	**10000**

8.3. Análisis

Los resultados obtenidos indican que el uso del *autoencoder* permitió hacerla reducción de dimensionalidad del 80 % que se buscaba, conservando gran parte de la información de la entrada. Sin embargo, esta reducción de dimensionalidad no garantizó la performance en los otros modelos.

El primer modelo (*encoder* congelado y 1 MLP de 156 a 10) alcanzó un *accuracy* global de 92.09 %, valor comparable pero ligeramente inferior al obtenido por el MLP entrenado directamente sobre las imágenes originales (92.73 %). Esto sugiere que, si bien el *encoder* logra una buena compresión, falta algo más para lograr igualar las performances.

Cuando se permitió el *fine-tuning* del encoder (es decir, ajustar parcialmente los pesos del encoder durante el entrenamiento del clasificador), el rendimiento mejoró levemente, alcanzando un *accuracy* global de 92.35 %. Esto indica que un ligero ajuste a los pesos del *encoder* ayuda a recuperar parte de la información perdida, o por lo menos llevar las representaciones a formas que hacen más exacta la clasificación.

Por otro lado, al agregar una capa oculta adicional de 156 neuronas (encoder congelado + FC intermedia 156→156→10), el desempeño aumentó notablemente, alcanzando una exactitud global de 97.94 %. Este resultado evidencia que, si bien la representación comprimida limita la información disponible, una arquitectura más elaborada permite explotar la información que sí está, que al fin y al cabo es la que representa más los datos originales en el espacio latente de menor dimensión.

En cuanto al tiempo de entrenamiento, si se considera únicamente la etapa de clasificación, los modelos basados en el encoder son más rápidos debido a la menor dimensionalidad de entrada (156 frente a 784). Sin embargo, al sumar el tiempo de entrenamiento previo del *autoencoder* (∼8 min), el proceso completo resulta más costoso en términos computacionales que entrenar un MLP directamente sobre los datos originales.

En conclusión, el uso del *encoder* puede ser beneficioso para reducir la dimensionalidad de los datos y acelerar el aprendizaje de etapas posteriores, pero en este caso, un MLP entrenado sobre la imagen de forma directa logra resultados equivalentes con menor complejidad arquitectónica. La combinación de *encoder* y clasificador más profundo (156→156→10) mostró ser la alternativa más efectiva y exacta dentro de los modelos derivados del *autoencoder*.

Una pregunta interesante para seguir con este tema sería investigar que es más rápido cuando se habla de ejecución en tiempo real. Un sistema como el MLP que va de 784 a 10 tiene muchas conexiones, y podría tomar más tiempo en ejecutarse que hacer la codificación de la imagen con el *encoder* y luego ejecutar un MLP de menor complejidad.

9. Ejercicio 8

9.1. Consignas

Encontrar un perceptrón multicapa que resuelva una XOR de 2 entradas mediante *simulated annealing*. Graficar el error a lo largo del proceso de aprendizaje.

9.2. Desarrollo

Para este ejercicio se reusó el código del ejercicio 3, cambiando la regla de aprendizaje a aquella del *simulated annealing*[4].

El algoritmo inicia con un set de pesos w, actualizados como $w(n+1) = w(n) + \mathcal{N}(0, \sigma^2)$. Es decir, el paso, o δw es aleatorio, y la varianza de la gaussiana determina que tan grande es ese paso. Luego de obtener un set de parámetros $w(n+1)$, se calcula el $\delta E = E(w(n+1)) - E(w(n))$, que es lo mismo que decir que se calcula el cambio del error después del cambio de parámetros.

Ahora viene lo interesante: si el error decrece ($\delta E < 0$) entonces se admite el cambio de parámetros y $w(n) = w(n+1)$. En cambio, si el error empeora ($\delta E > 0$), el cambio se admite con una probabilidad $e^{\frac{-\delta E}{T}}$, con T la temperatura del sistema en ese instante.

Como detalle final, la temperatura se debe decrecer para que el algoritmo vaya convergiendo (si es muy alta tal vez pasea por el espacio de estados sin límite). Esto generalmente se hace de forma exponencial como $T(n+1) = \alpha T(n)$. En el desarrollo hecho se usó un decaimiento de temperatura del tiempo exponencial y un σ constante. A continuación se detallan los varios casos simulados.

9.2.1. XOR de 2 entradas - 2 neuronas en capa oculta

$\alpha = 0{,}995$

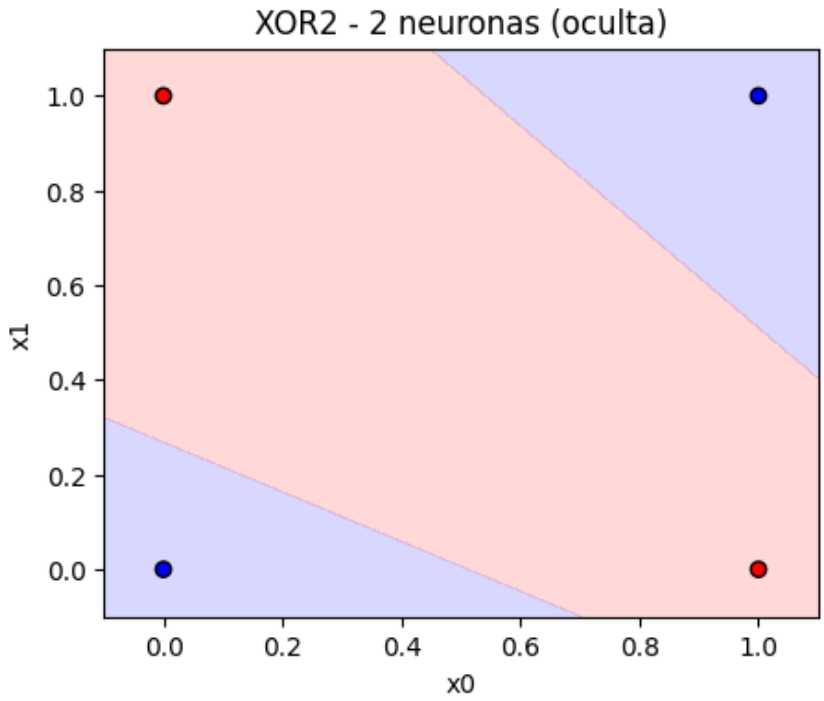

Figura 53: Frontera de decisión de MLP de 2 neuronas en capa oculta

[4]Emula el proceso físico de templar y revenir una pieza de metal

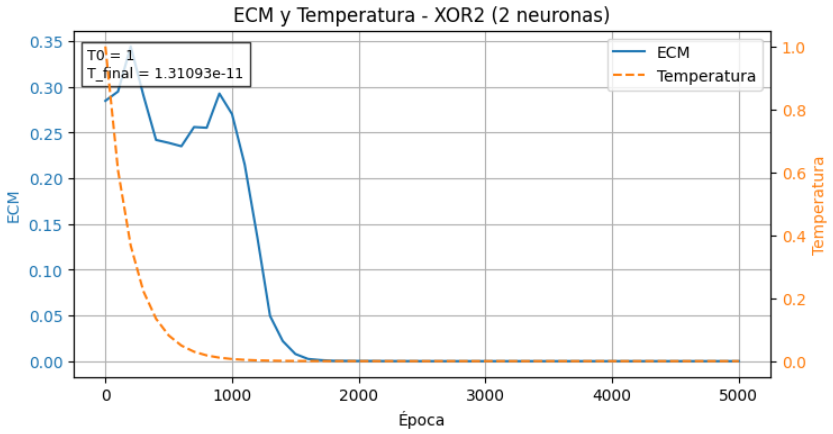

Figura 54: ECM y temperatura por interación de MLP de 2 neuronas en capa oculta

La XOR de 2 entradas fue aprendida perfectamente por una red de 2 neuronas en la capa oculta, igual que en la consigna 3. La imagen **??**uestra la frontera de decisiónfig:screenshot001, también recta como en el caso de gradiente descendiente. En el gráfico 54 se puede ver el ECM y temperatura. hay cierta estocacidad con el ECM (como es de esperar por la temperatura), pero termina disminuyendo hasta converger en 0.

9.2.2. XOR de 2 entradas - 4 neuronas en capa oculta

$\alpha = 0{,}995$

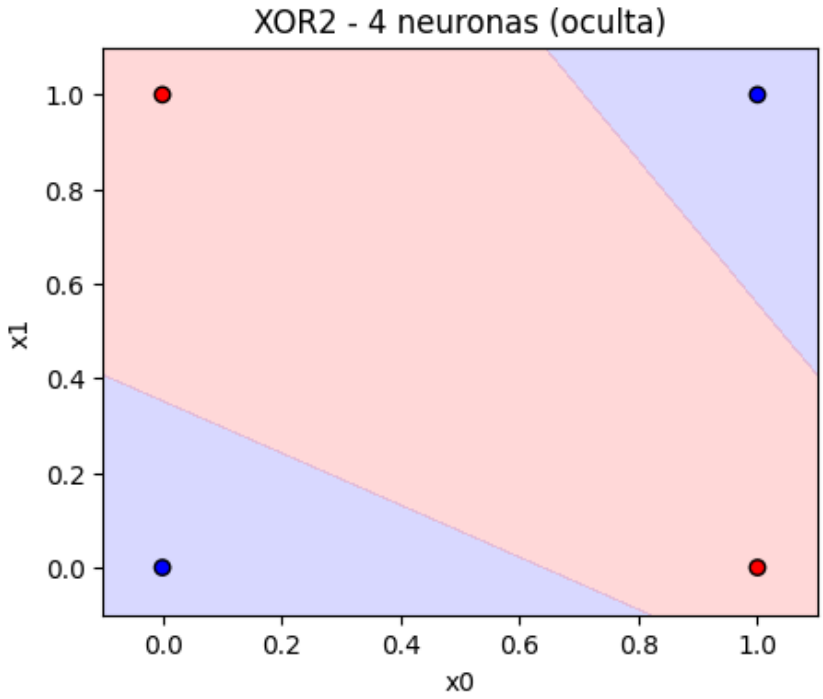

Figura 55: Frontera de decisión de MLP de 4 neuronas en capa oculta

La figura 55 muestra la frontera de decisión, y la 56 contiene el ECM y temperatura. La red aprendió el problema a la perfección. Su ECM es decreciente y la forntera es congruente con lo esperado y ya visto.

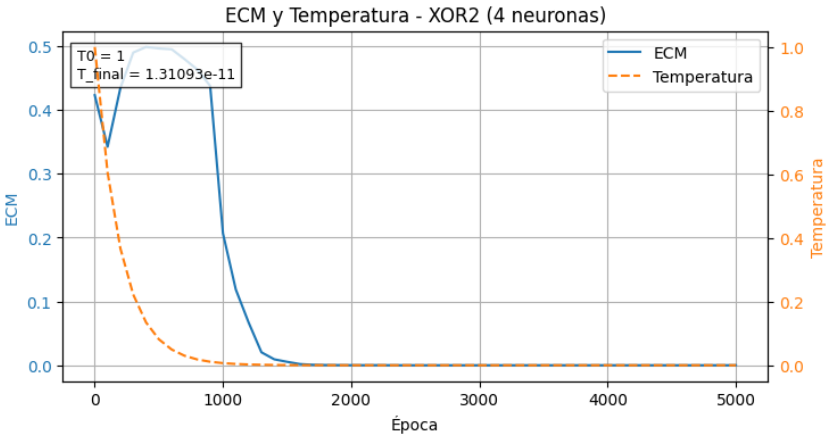

Figura 56: ECM y temperatura por interación de MLP de 4 neuronas en capa oculta

9.2.3. XOR de 2 entradas - 10 neuronas en capa oculta

$\alpha = 0{,}995$

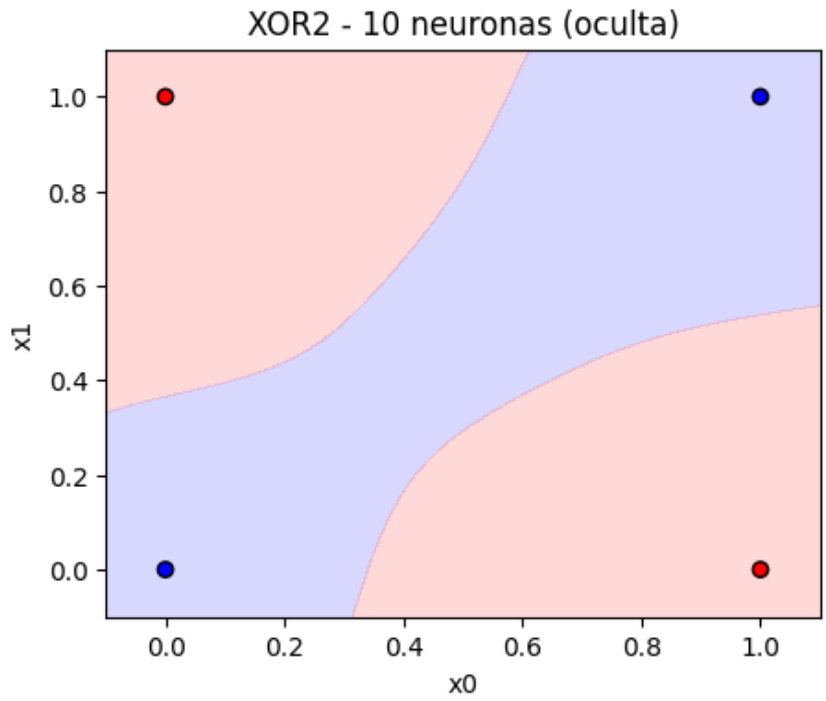

Figura 57: Frontera de decisión de MLP de 10 neuronas en capa oculta

La figura 57 muestra la frontera de decisión, y la 58 contiene el ECM y temperatura. la frontera de decisión es curva como en el caso de 10 neuronas con gradiente descendiente y el ECM disminuye como se esperaba.

9.2.4. XOR de 2 entradas - 2 neuronas en capa oculta

$\alpha = 0{,}98$

Para este caso (figuras 59 y 60), em uno la red aprendió el problema aún cuando la regla de la actualización de la temperatura era la de una exponencial rápida. En el otro no lo aprendió. Esto claramente es algo estocástico.

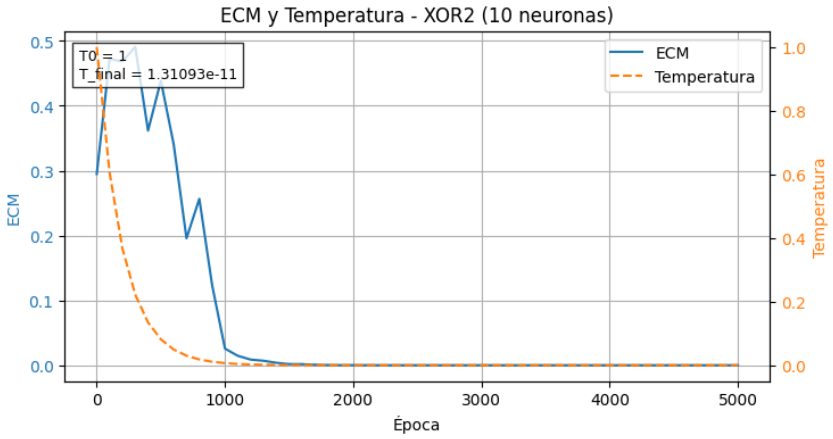

Figura 58: ECM y temperatura por interación de MLP de 10 neuronas en capa oculta

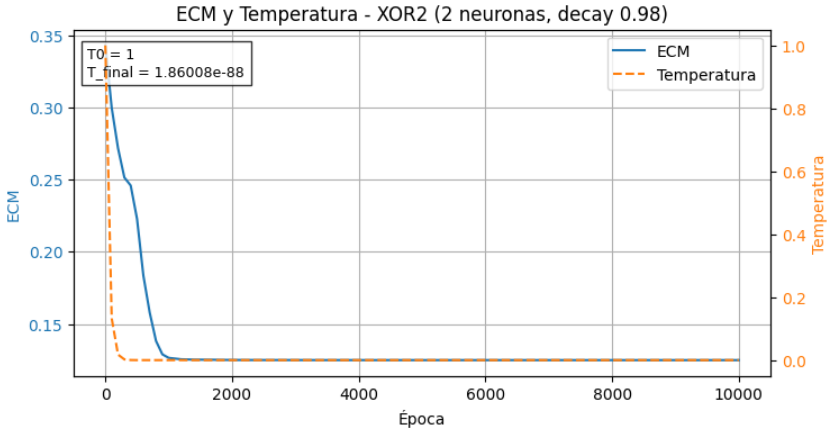

Figura 59: ECM y temperatura -Decay rate de 0.98 - caso 1

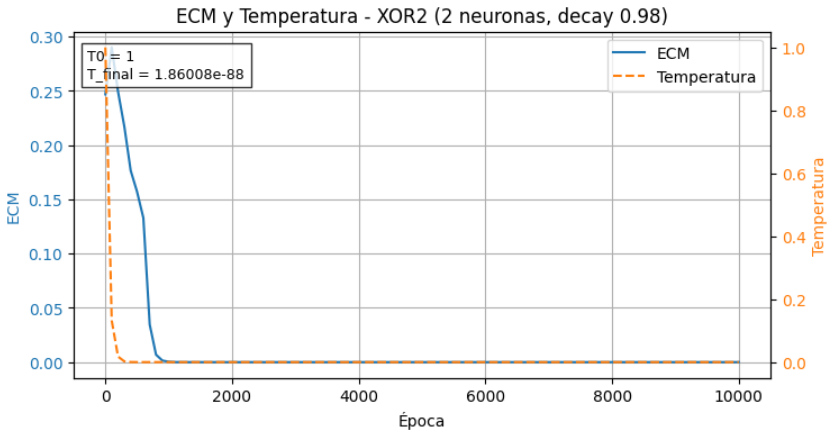

Figura 60: ECM y temperatura - Decay rate de 0.98 - caso 2

9.2.5. XOR de 2 entradas - 2 neuronas en capa oculta

$\alpha = 0{,}999$

Para este caso (figuras 61 y 62), la red claramente pasó demasiado tiempo con alta temperatura, cayendo en un mínimo no global. esto se ejecutó varias veces con el mismo resultado, aunque no es definitivo.

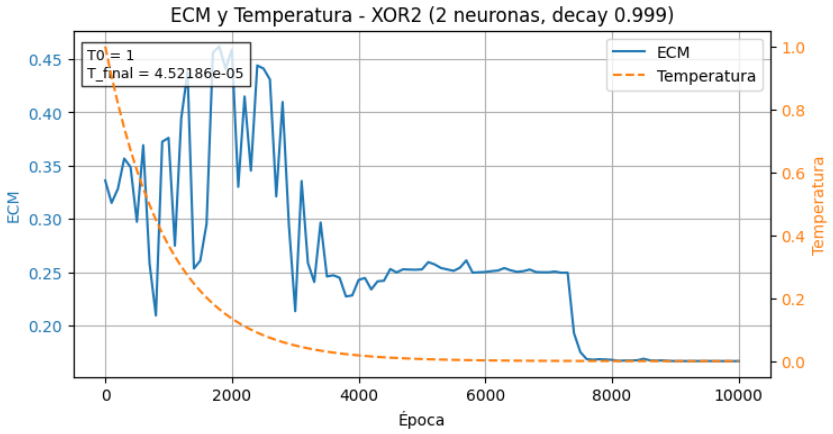

Figura 61: ECM y temperatura - Decay rate de 0.999

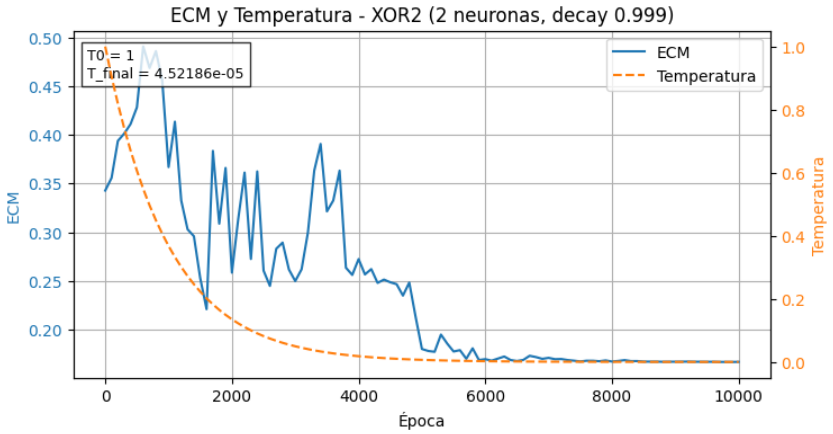

Figura 62: ECM y temperatura - Decay rate de 0.999

9.2.6. XOR de 4 entradas - 4 neuronas en capa oculta

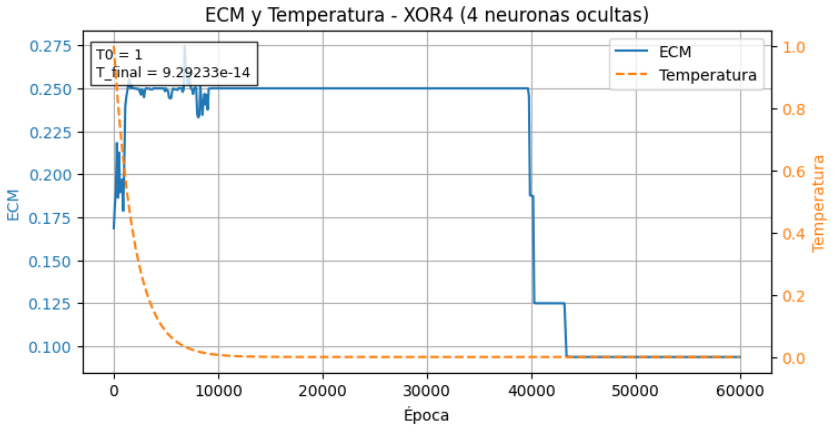

Figura 63

No aprendió

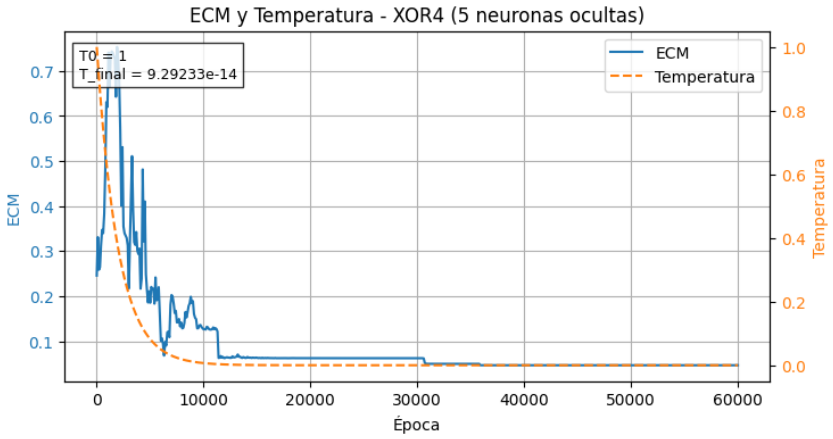

Figura 64

9.2.7. XOR de 4 entradas - 5 neuronas en capa oculta

aprendió

9.2.8. XOR de 4 entradas - 6 neuronas en capa oculta

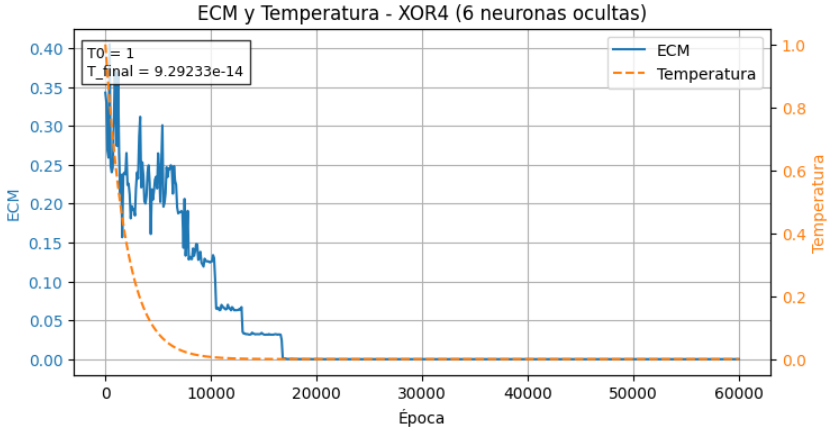

Figura 65

Sobre estas últimas, puede ser que le sea más fácil encontrar la solución pq el esacio de soluciones es más chico y este algoritmo se lo permite encontrar.

9.3. Análisis

10. Conclusiones