Departamento de Electrónica
1er Cuatrimestre de 2024

Redes Neuronales (86.54)
Trabajo Práctico 1

Lucas Burdman

Padrón: 104310

eMail: lburdman@fi.uba.ar

Profesores: Lew - Veiga - Mininni

Índice

1. Introducción

Este trabajo práctico abarca los siguientes temas clave en redes neuronales y aprendizaje automático:

- **Perceptrón Simple**: se busca entrenar una neurona para resolver los problemas linealmente separables AND y OR de 2 y 4 entradas, analizando la evolución del entrenamiento.
- **Capacidad del Perceptrón**: se analiza la capacidad del perceptrón simple en función de patrones enseñados.
- **Perceptrón Multicapa**: se implementa y entrena una red multicapa para la función lógica XOR linealmente *no* separable.
- **Backpropagation**: se utiliza el algoritmo para el entrenamiento de las redes multicapa.
- **Entrenamiento por minibatches**: se analizan las ventajas de entrenar una red con distintos tamaños de *minibatch* para el aprendizaje de una función compleja.
- **Algoritmos Genéticos**: se usan los beneficios de algoritmos genéticos para optimizar un perceptrón que resuelva la función lógica XOR de 2 entradas.

En el presente informe se detalla el método de resolución de los enunciados, así como el análisis y las conclusiones de los resultados obtenidos.

2. Desarrollo

2.1. Ejercicio 1: Perceptrón Simple

***Enunciado**: Implemente un perceptrón simple que aprenda la función lógica AND de 2 y de 4 entradas. Lo mismo para la función lógica OR. Para el caso de 2 dimensiones, grafique la recta discriminadora y todos los vectores de entrada de la red.*

Para la resolución, se entreno una red neuronal de una neurona. Para entrenarla se inicializaron sus pesos sinápticos y luego se realizó un aprendizaje, actualizando sus pesos w_i hasta encontrar una solución con error nulo.

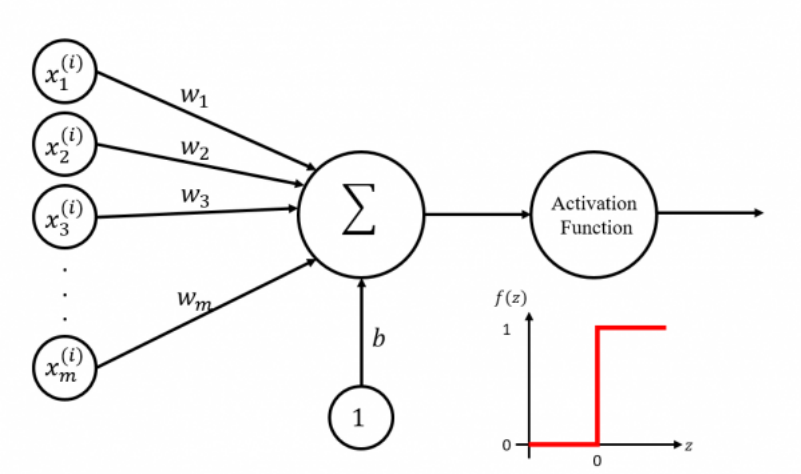

Figura 2.1.1: Topología de red de una neurona

Cada entrada se multiplica por su peso sináptico y luego se calcula h como $h = \sum_{j=1}^{M} w_j x_j$. Así, se evalúa el signo de h a través de la función de activación, que evaluará el signo de h. Si es positivo la salida será 1 y si es negativo -1.

Para actualizar los pesos sinápticos, se resta el valor de salida obtenido con la salida deseada del patrón de entrada que esta siendo evaluado. Si el error del patrón no es nulo, cada coeficiente w_i se actualiza como:

$$w_0(k) = \eta \cdot w_0(k-1)$$

$$\Delta w_j(k) = \eta \cdot x_j \cdot (y_{deseada} - y_{obtenida}) \tag{2.1}$$

$$w_j(k) = w_j(k-1) + \Delta w_j(k)$$

Donde 2.1 es la regla general de aprendizaje del perceptrón. Se repite este procedimiento hasta que el error total de todos los patrones de entrada sea nulo.

El parámetro η es la tasa de aprendizaje. Mientras sea menor, toma más iteraciones en converger a una solución posible, dado que determina que tanto se modifica el valor de los pesos sinápticos (es decir, cuánto se rota el plano de la recta discriminadora).

Es importante notar que esta topología encuentra solución **solo** para problemas linealmente separables, de no ser así el perceptrón será incapaz de encontrar una solución con error total nulo.

Para el desarrollo de este ejercicio se elige un valor de $\eta = 0,3$ y los pesos sinápticos se inicializaron con valores entre 10 y -10 aleatoriamente.

2.1.1. Perceptrón simple para AND de 2 entradas

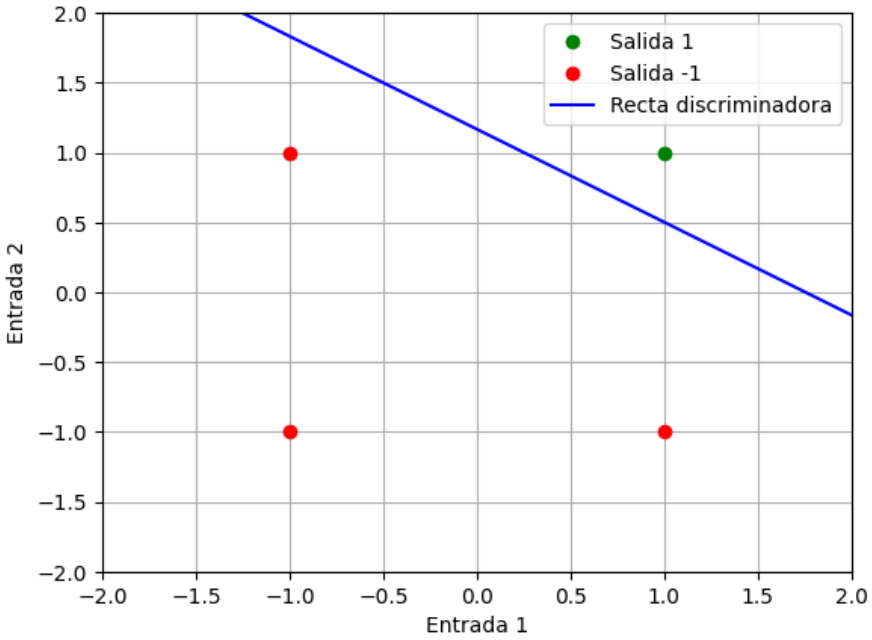

Figura 2.1.1.1: Resultado de perceptrón simple para AND de 2 entradas

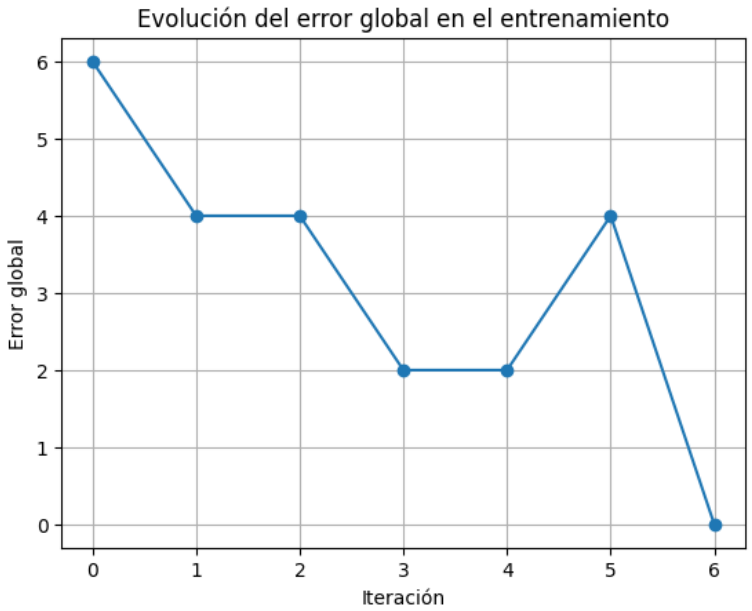

Figura 2.1.1.2: Evolución del error global en cada iteración para AND de 2 entradas

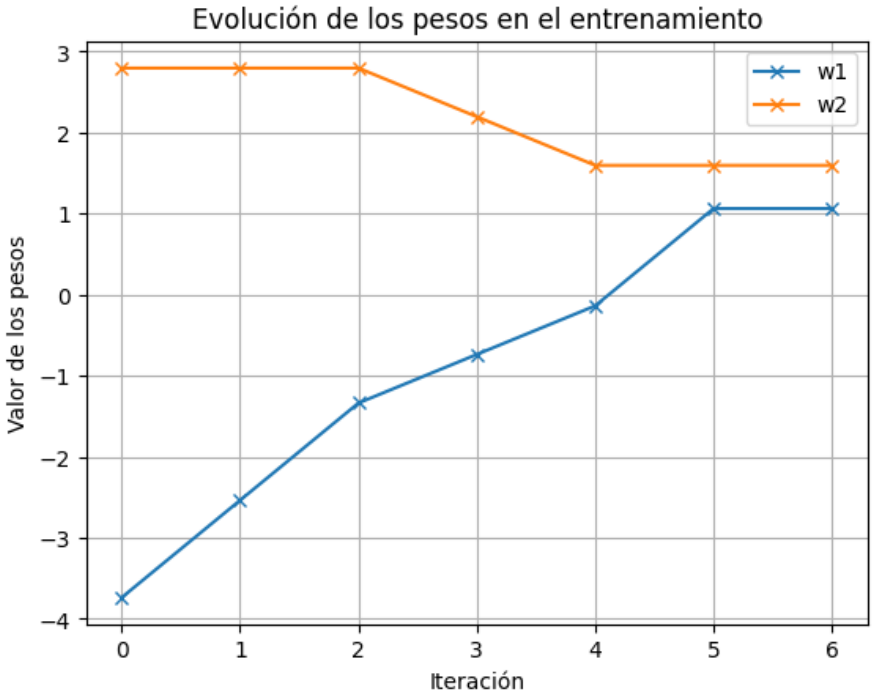

Figura 2.1.1.3: Evolución de los pesos sinápticos en cada iteración para AND de 2 entradas

2.1.2. Perceptrón simple para AND de 4 entradas

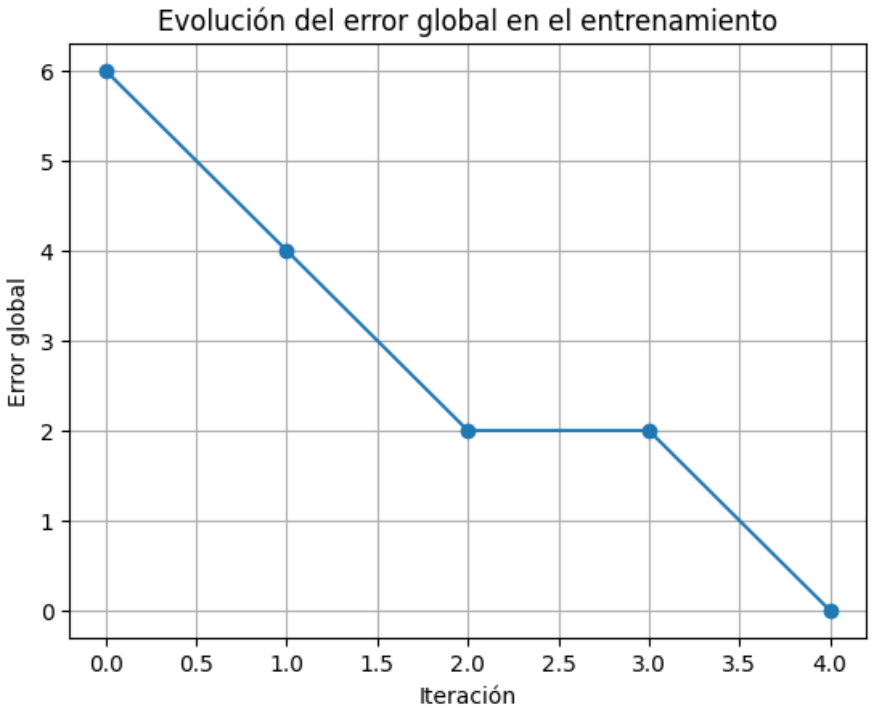

Figura 2.1.2.1: Evolución del error global en cada iteración para AND de 4 entradas

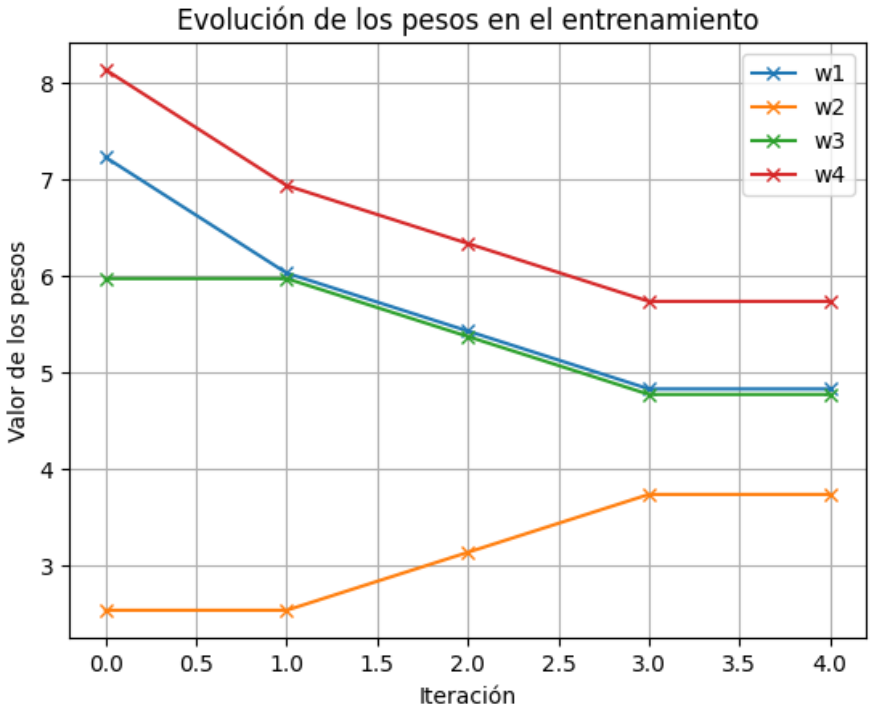

Figura 2.1.2.2: Evolución de los pesos sinápticos en cada iteración para AND de 4 entradas

2.1.3. Perceptrón simple para OR de 2 entradas

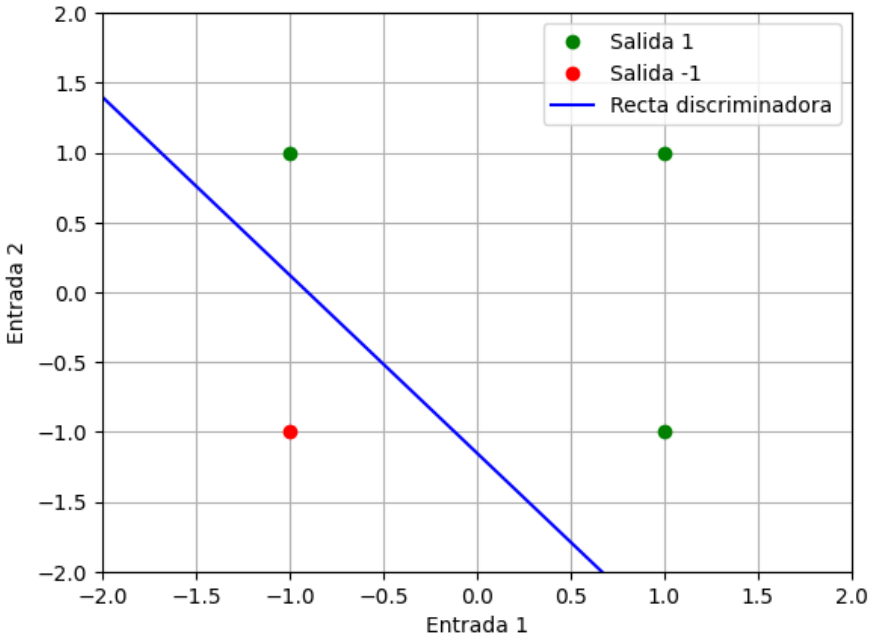

Figura 2.1.3.1: Resultado de perceptrón simple para OR de 2 entradas

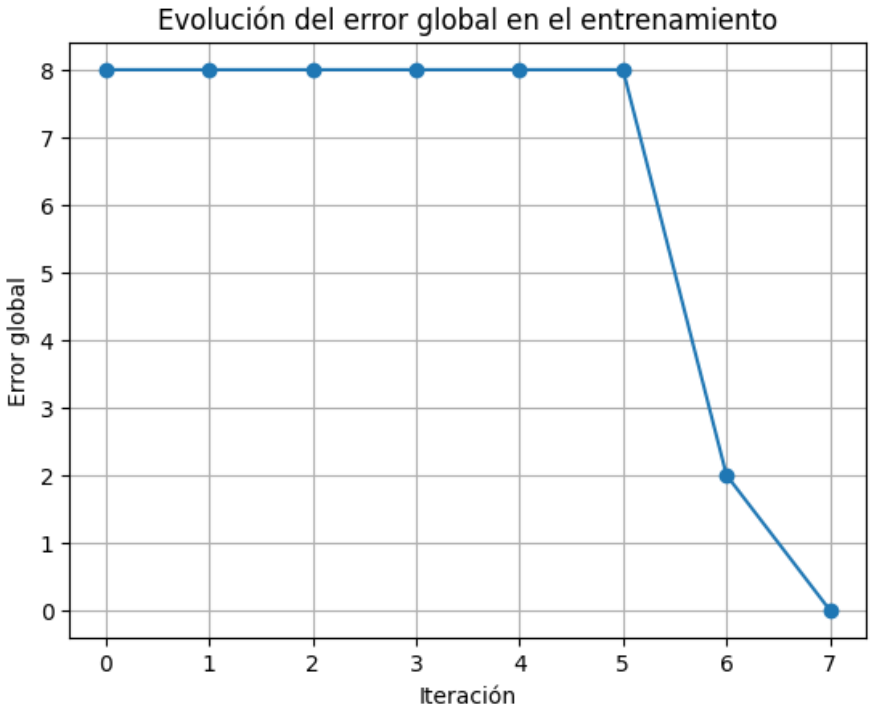

Figura 2.1.3.2: Evolución del error global en cada iteración para OR de 2 entradas

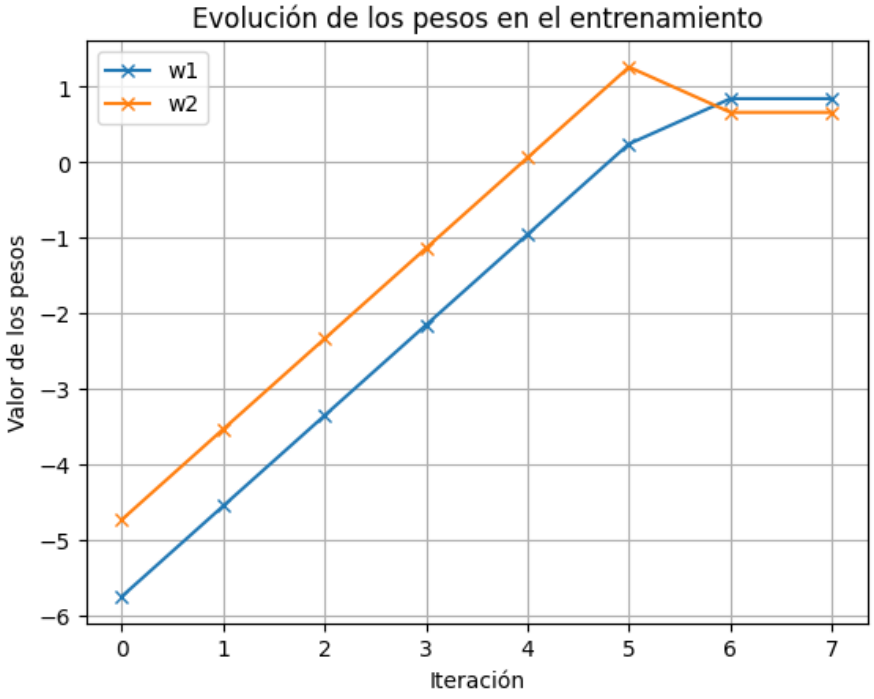

Figura 2.1.3.3: Evolución de los pesos sinápticos en cada iteración para OR de 2 entradas

2.1.4. Perceptrón simple para OR de 4 entradas

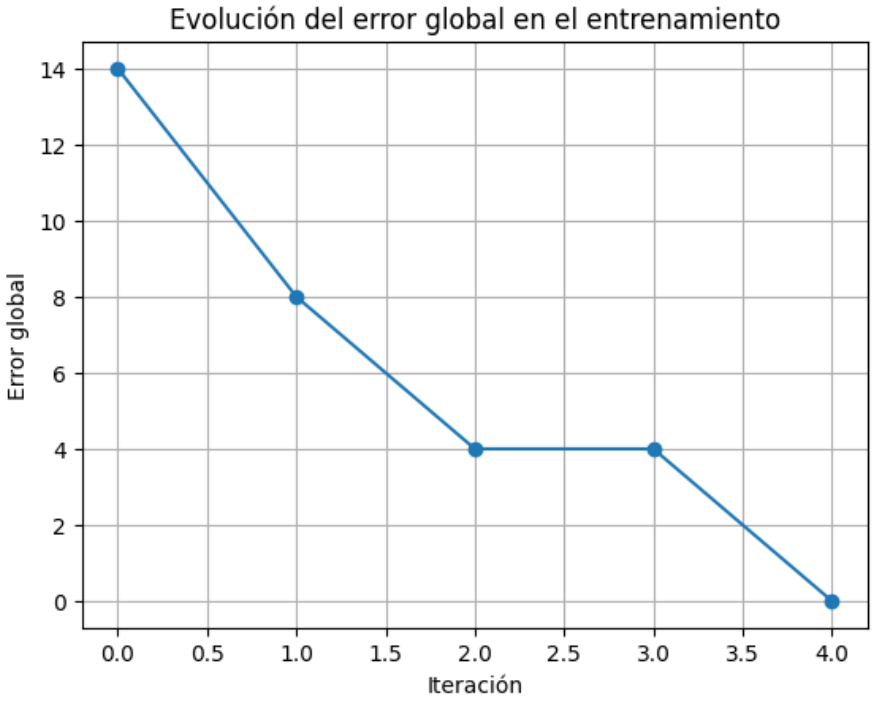

Figura 2.1.4.1: Evolución del error global en cada iteración para OR de 4 entradas

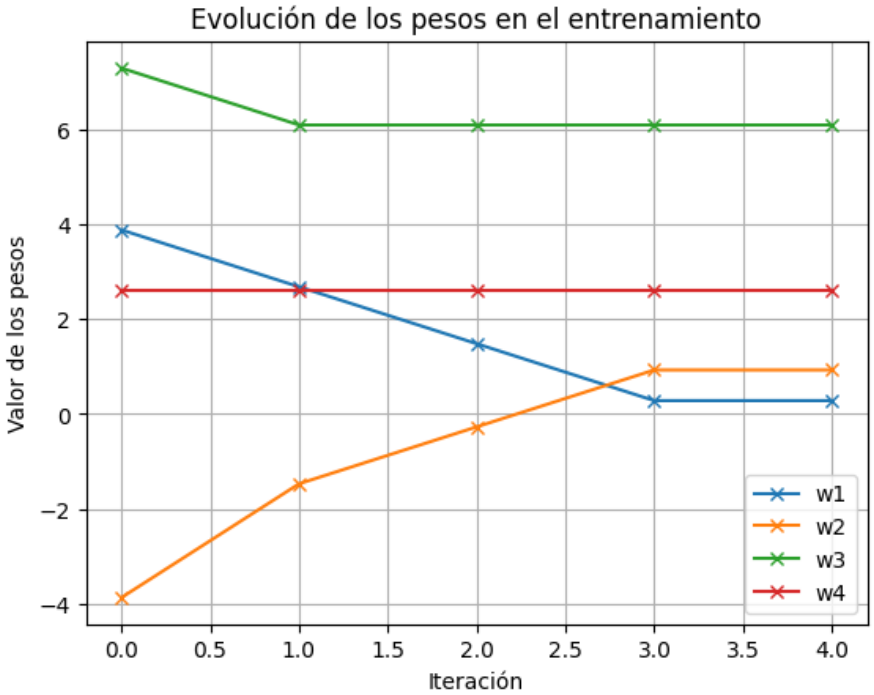

Figura 2.1.4.2: Evolución de los pesos sinápticos en cada iteración para OR de 4 entradas

2.2. Ejercicio 2: Capacidad del perceptrón

Enunciado: *Determine numéricamente cómo varia la capacidad del perceptrón simple en función del número de patrones enseñados.*

La capacidad del perceptrón da idea de que cantidad de patrones aleatorios podemos esperar que aprenda el perceptrón en una red neuronal de cierto tamaño.

Para el ejercicio se entrena un modelo simple con patrones de diferentes tamaños, evaluando si el perceptrón pudo aprender sin errores. A medida que se aumenta el número de patrones, se observa cómo cambia la capacidad, reflejando la efectividad del perceptrón para clasificar los patrones. Los resultados muestran la relación entre el tamaño de los patrones y el rendimiento del perceptrón.

Utilizando $N = 10, 20, 40$ entradas, se evalúa si un perceptrón, con $\eta = 0,3$ y pesos aleatorios entre -5 y 5, es capaz de encontrar una solución con error nulo para problemas de dimensión de hasta $4N$.

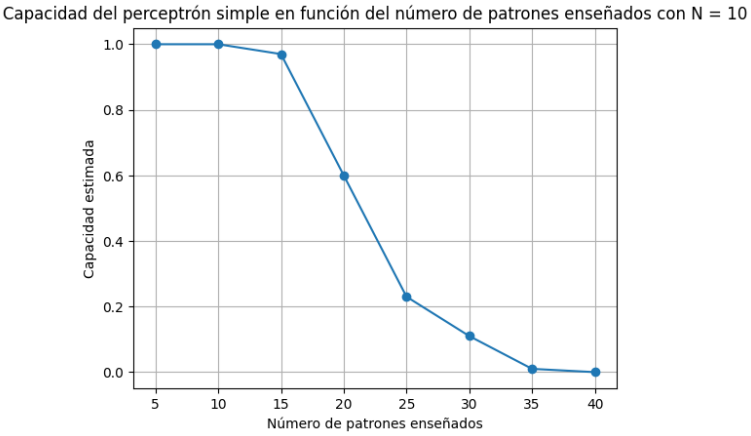

Figura 2.2.1: Capacidad del perceptrón para N = 10

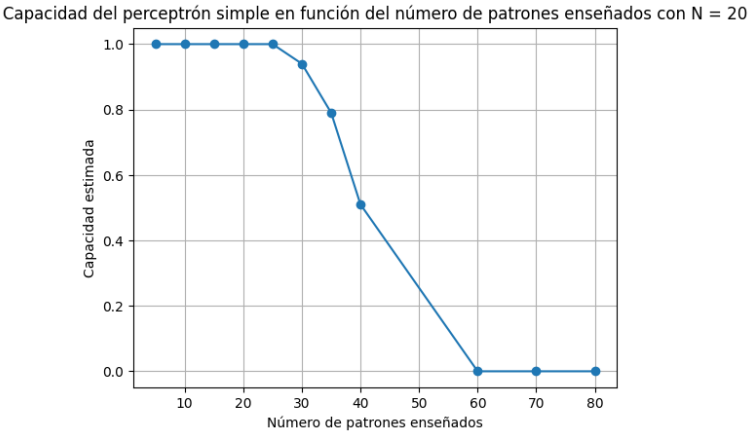

Figura 2.2.2: Capacidad del perceptrón para N = 20

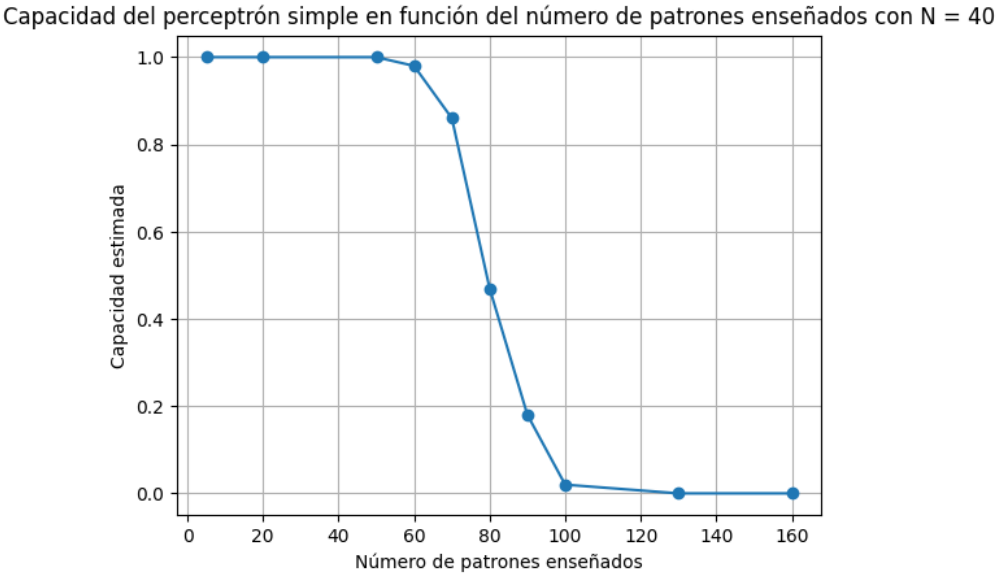

Figura 2.2.3: Capacidad del perceptrón para N = 40

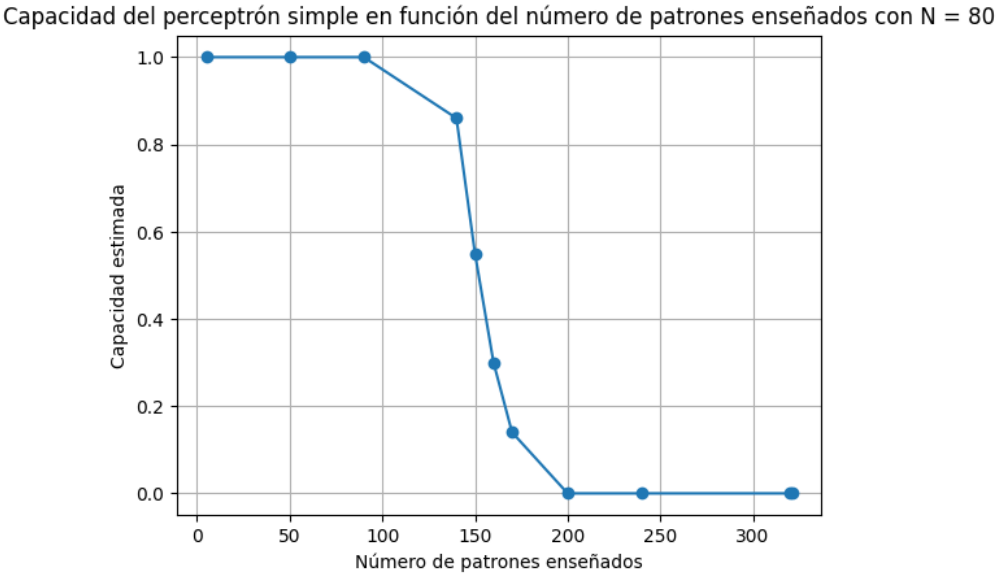

Figura 2.2.4: Capacidad del perceptrón para N = 80

Se puede observar que en $2N$ patrones enseñados se alcanza una capacidad de aproximadamente 0,5, como se predice según la teoría.

2.3. Ejercicio 3: Perceptrón Multicapa

Enunciado:

a) Implemente un perceptrón multicapa que aprenda la función lógica XOR de 2 y de 4 entradas (utilizando el algoritmo Backpropagation).

b) Muestre cómo evoluciona el error durante el entrenamiento.

c) Para una red entrenada en la función XOR de dos entradas, muestre cómo varía el error en función del cambio en dos pesos de la red. De ejemplos de mínimos locales y mesetas.

d) Idem (c) pero mostrando el error para cada patrón de entrada por separado.

El perceptrón multicapa se debe implementar para resolver problemas *no* linealmente separables, como lo es la función lógica XOR. Para esto se implementa un entrenamiento que haga propagación hacia adelante (*forward propagation*), para luego propagar el error hacia atrás *backpropagation*. Así, se logra actualizar los pesos sinápticos de las capas de la red, para encontrar una solución.

En la etapa *forward*, las entradas se pasan a través de las capas de la red. Cada neurona de una capa recibe entradas de todas las neuronas de la capa anterior, que se suman ponderadamente según los pesos de las conexiones. Esta suma se pasa a través de una función de activación.

Una vez que se obtiene la salida de la red, se calcula el error entre la salida obtenida y la deseada. Este error se utiliza para ajustar los pesos de la red mediante el algoritmo de *backpropagation*. El error se propaga de vuelta a través de la red, desde la capa de salida hasta las capas de entrada, ajustando los pesos en cada conexión para minimizar el error en futuras predicciones.

Así, la red se entrena iterativamente en múltiples pasadas sobre el conjunto de datos, ajustando continuamente los pesos para reducir el error de salida.

Este proceso asegura que el perceptrón multicapa aprenda de manera efectiva, ajustando sus pesos sinápticos para que la red pueda resolver el problema y clasifique correctamente.

2.3.1. Perceptrón multicapa para XOR de 2 entradas

Para el problema XOR de 2 entradas, se implementa una red neuronal de 2 neuronas de entrada, una capa oculta con 5 neuronas y 1 neurona de salida. Así, con una tasa de aprendizaje de $0,1$, se realizaron 1000 iteraciones.

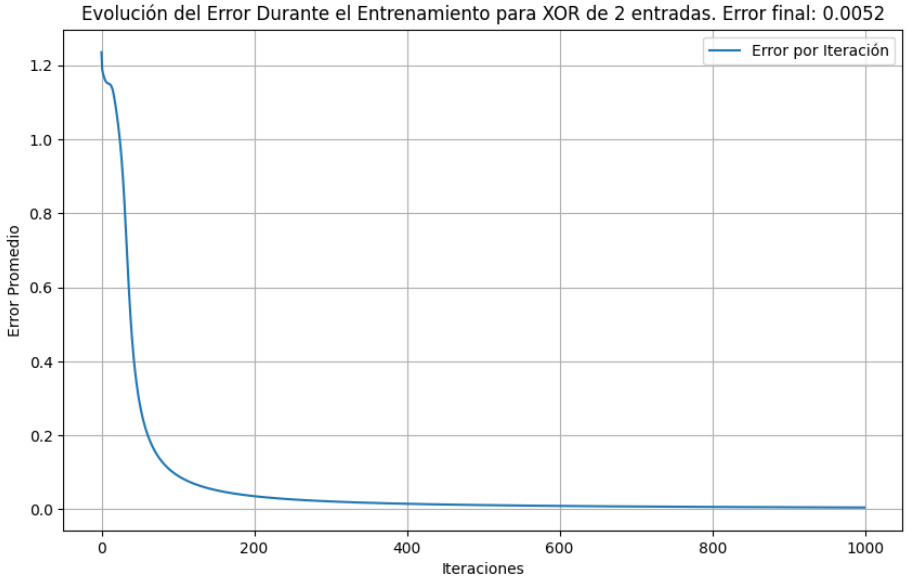

Figura 2.3.1.1: ECM para XOR de 2 entradas usando la función de activación *tanh*

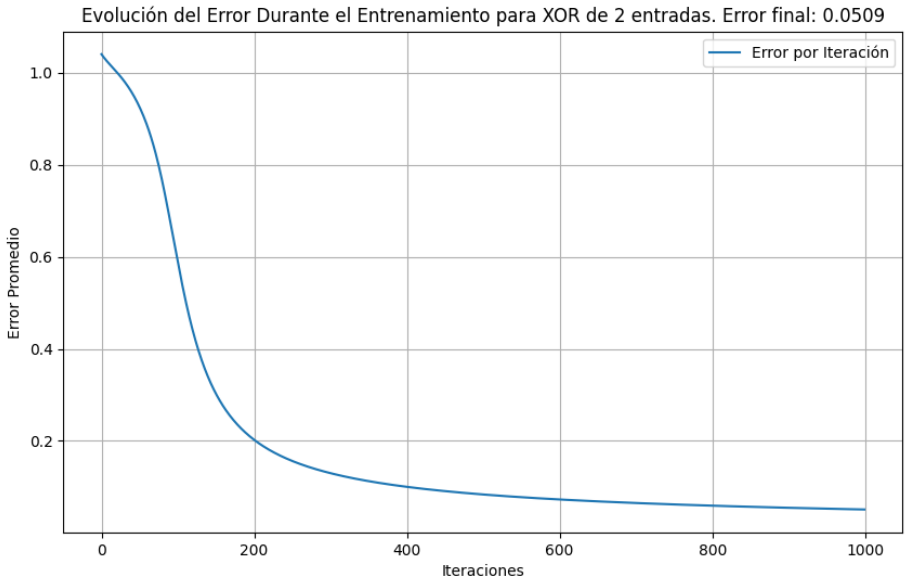

Figura 2.3.1.2: ECM para XOR de 2 entradas usando la función de activación *sigmoid*

La evolución del error utilizando los mismos parámetros de entrenamiento convergen a valores cercanos a 0 más lento al utilizar la función de activación *sigmoid*. Esto se puede deber a que los gradientes de *tanh* son más fuertes en comparación a *sigmoid*, lo que se traduce en aprendizajes más lentos y un error final mayor.

2.3.2. Perceptrón multicapa para XOR de 4 entradas

Para el problema XOR de 4 entradas, se implementa una red neuronal de 4 neuronas de entrada, una capa oculta con 10 neuronas y 1 neurona de salida. Así, con una tasa de aprendizaje de $0,1$, se realizaron 1000 iteraciones.

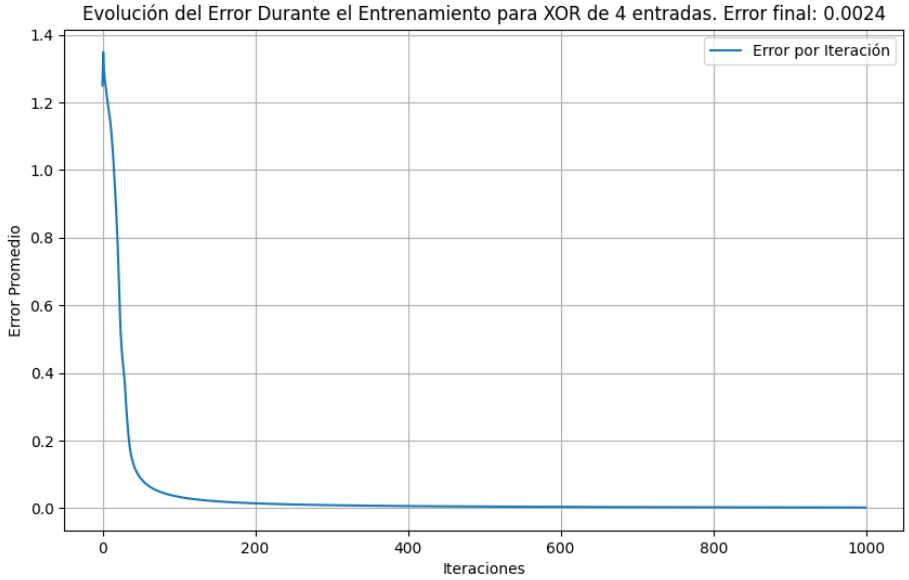

Figura 2.3.2.1: ECM para XOR de 4 entradas usando la función de activación *tanh*

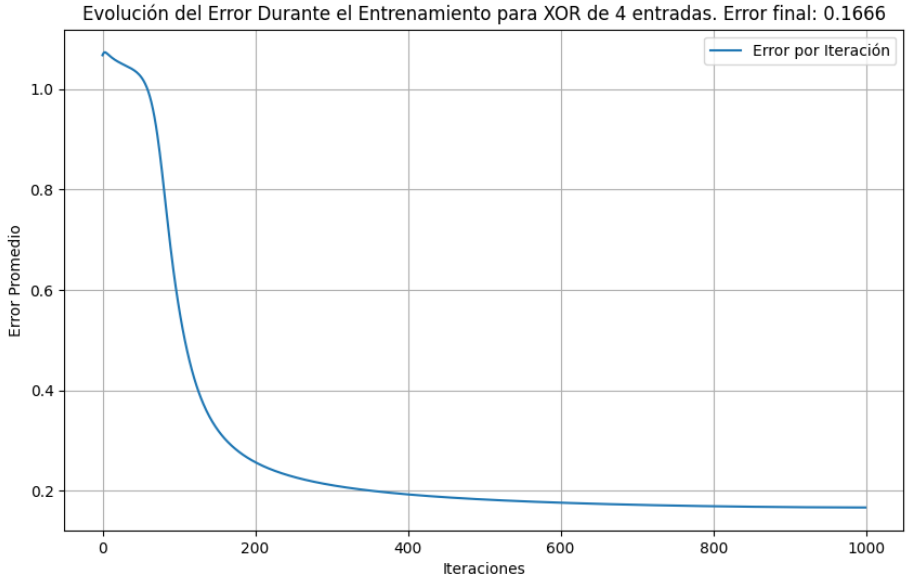

Figura 2.3.2.2: ECM para XOR de 4 entradas usando la función de activación *sigmoid*

Nuevamente, la evolución del error al utilizar la función de activación *sigmoid* es más lenta, siendo el error final alcanzado mucho mayor.

2.3.3. Error en función del cambio en dos pesos sinápticos

Ahora para la función XOR de 2 entradas se entrena una red neuronal de 2 neuronas de entrada, una capa oculta con 5 neuronas y 1 neurona de salida, con una tasa de aprendizaje de $0,2$, logrando un error de $0,0024$.

Se eligen dos pesos sinápticos que logren un mapa que posea un centro donde el error no se vea muy modificado, para visualizar así un mínimo local. Se varia su valor con una amplitud de ± 15 con 200 valores entre ellos

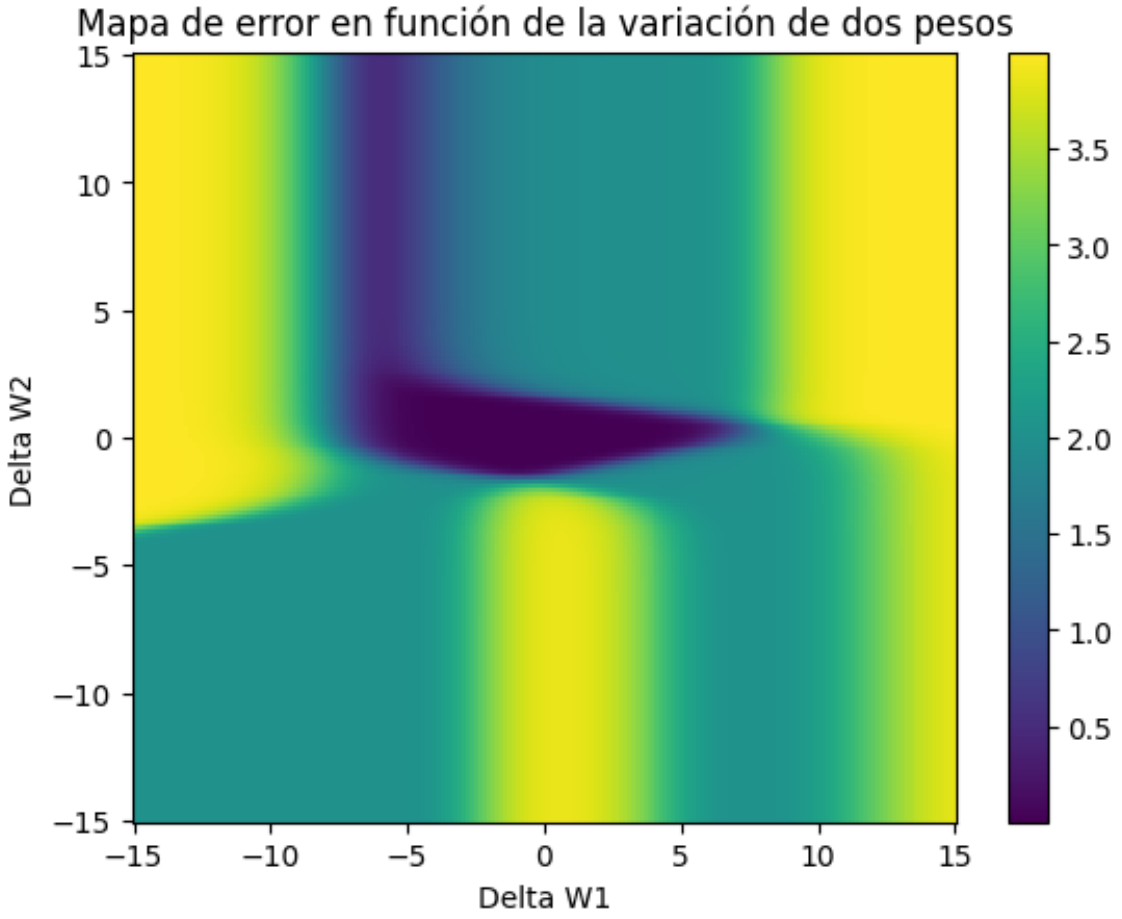

Figura 2.3.3.1: Mapa de error al variar dos pesos sinápticos

Para visualizar mejor el efecto de modificar los pesos sinápticos, se realiza un gráfico en 3D, viéndose claramente una zona plana en los valores correspondientes a $\Delta w = -5$ y $\Delta w = -15$ para ambos pesos elegidos, y el mínimo local donde el error casi no se ve afectado al variar $\Delta w_1 = \pm 5$, y sin tener tanto margen de variación para Δw_2.

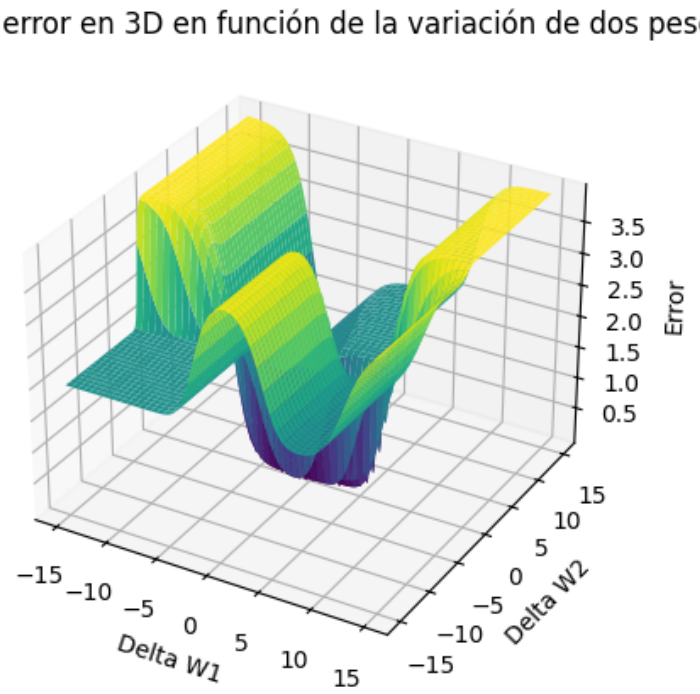

Figura 2.3.3.2: Mapa de error en 3D

Por otro lado, el error para cada patrón es diferente. Siendo la superposición de estos el mapa de error de la Figura 2.3.3.1.

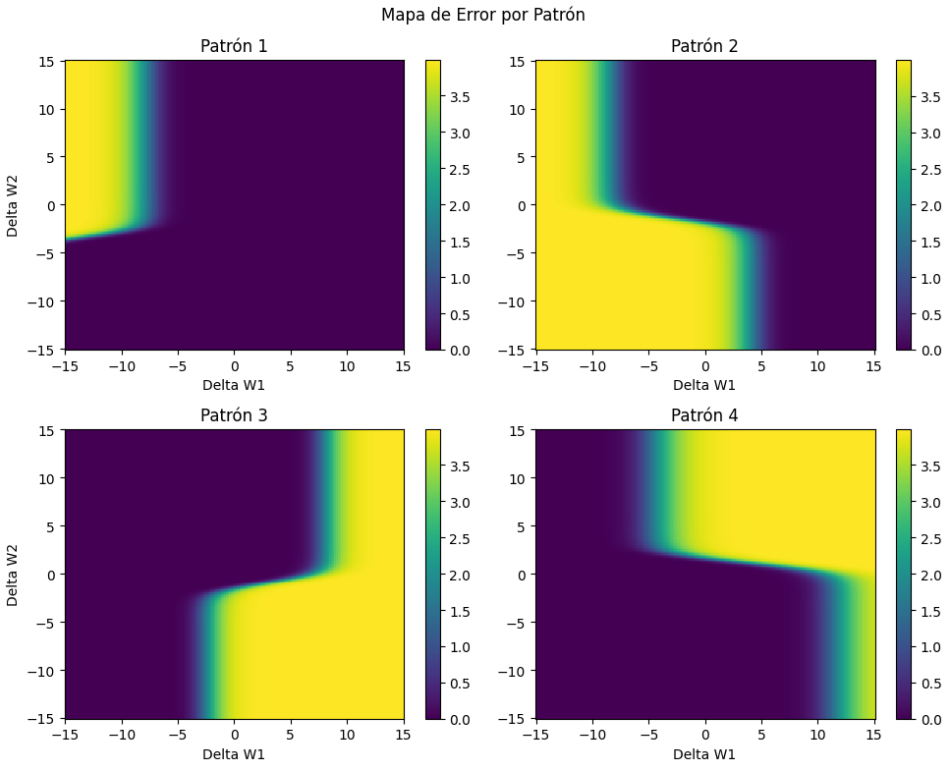

Figura 2.3.3.3: Mapa de error para cada patrón de entrada por separado

2.4. Ejercicio 4: Perceptrón Multicapa para una función

Enunciado:

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

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

donde: $x, y \in [0, 2\pi]$ *y* $z \in [-1, 1]$. *Para ello construya un conjunto de datos de entrenamiento y un conjunto de evaluación. Muestre el error en función de las épocas de entrenamiento.*

b) *Analice, mediante simulaciones, el efecto que tiene el tamaño de minibatch y la constante de aprendizaje en el número de iteraciones y el tiempo total de entrenamiento necesario para obtener un buen desempeño de la red.*

Para aprender la función $f(x, y, z)$ se utilizan minibatches. El entrenamiento por minibatches es una técnica de optimización que consiste en dividir el conjunto total de datos de entrenamiento en pequeños subconjuntos, denominados *minibatches*, y actualizar los parámetros del modelo iterativamente con cada minibatch.

El entrenamiento con minibatches permite que el modelo se actualice con más frecuencia. En este caso, al generar un gran volumen de datos, evaluando cada minibatch por separado se converge a una solución mas rápidamente que si se utilizara el conjunto de datos completo. Además, este método permite manejar mejor la variabilidad y complejidad de la función. Trabajando con muestras pequeñas en cada iteración es mas sencillo explorar el espacio de soluciones, pudiendo encontrar una mejor aproximación a la función.

Para la implementación se utilizaron un total de 6 minibatches de 200 datos cada uno. Luego se generaron 1200 datos para el entrenamiento y otro 30 % adicional para los datos de test (un total de 1560).

Luego, con una red neuronal con las 3 entradas de la función, una capa oculta con 5 neuronas y una neurona de salida, se realizo el aprendizaje por minibatches con una tasa de aprendizaje de $0,1$ y 10000 iteraciones.

Estos datos no fueron escogidos al azar, sino conforme a un análisis de buenos resultados, tiempo de entrenamiento y pruebas con diferentes configuraciones. Con la configuración elegida se llega a una buena solución, tomando poco tiempo de ejecución. Mientras valores bajos de la tasa de aprendizaje y de tamaño de batch tomaban mas de 10 minutos en finalizar el entrenamiento, este entrenamiento termina en 7 segundos.

Para actualizar los datos, en cada iteración se mezclan los datos para que cada minibatch tenga una selección aleatoria de las muestras. Luego, por cada uno de ellos, se propaga hacia adelante y luego se propaga el error hacia atrás, actualizando los pesos y calculando el error, y se guarda el error correspondiente.

2.4.1. Resultados obtenidos

Luego del entrenamiento se evalúa la red neuronal entrenada para la función con los datos de testeo y se calcula el error cuadrático medio entre las salidas calculadas y las reales, obteniendo un valor de $MSE = 0,0031$.

En el siguiente gráfico se comparan los valores reales correspondientes a los datos destinados para testeo (negro), comparado con los valores obtenidos de ingresar las entradas de dichos datos, siendo claro que se logra una red neuronal muy bien entrenada para la función dada, la cual da valores muy cercanos a los reales, con error muy bajo.

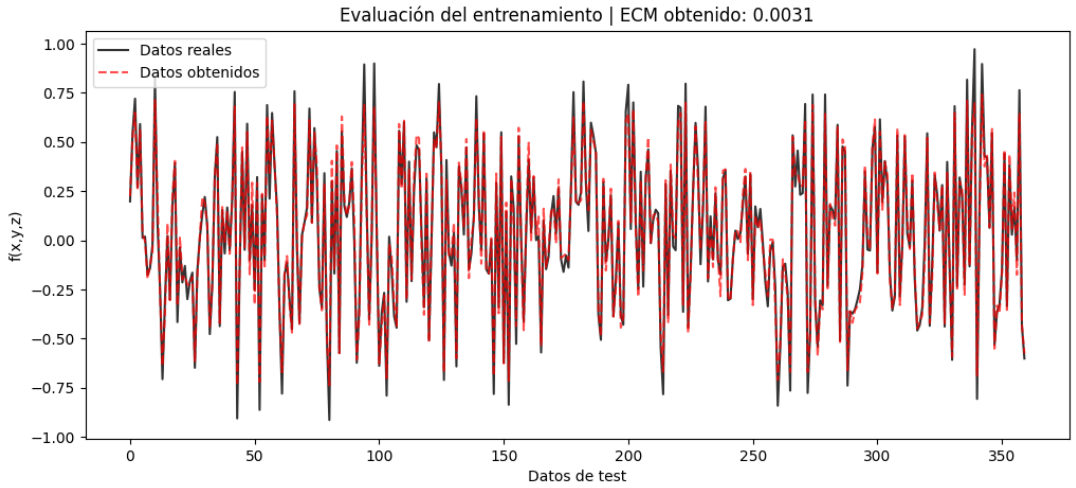

Figura 2.4.1.1: Datos obtenidos vs Datos reales

Por otro lado, el en el siguiente gráfico se puede observar esta comparación de otra manera. En este se puede notar como los valores en el entrenamiento son muy similares a los reales, pero en los valores extremos (máximos y mínimos) se desvía un poco del valor real, sobrestimando los mínimos y subestimando los máximos.

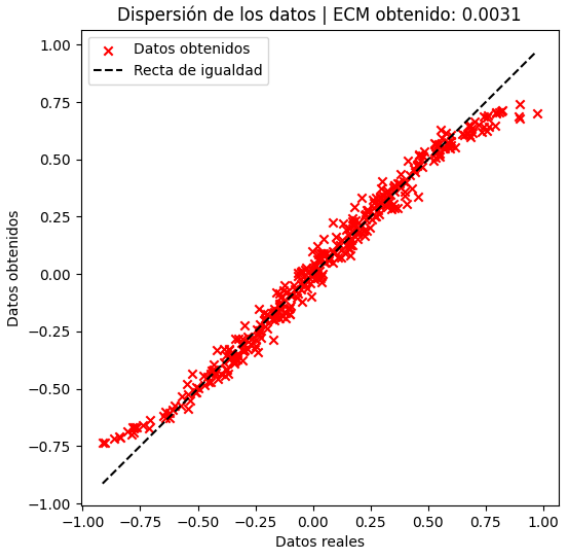

Figura 2.4.1.2: Datos obtenidos vs Datos reales

Por ultimo, la evolución del error en cada iteración muestra como va disminuyendo conforme la red esta mejor entrenada, con picos en algunas iteraciones en las que el aprendizaje no fue tan bueno.

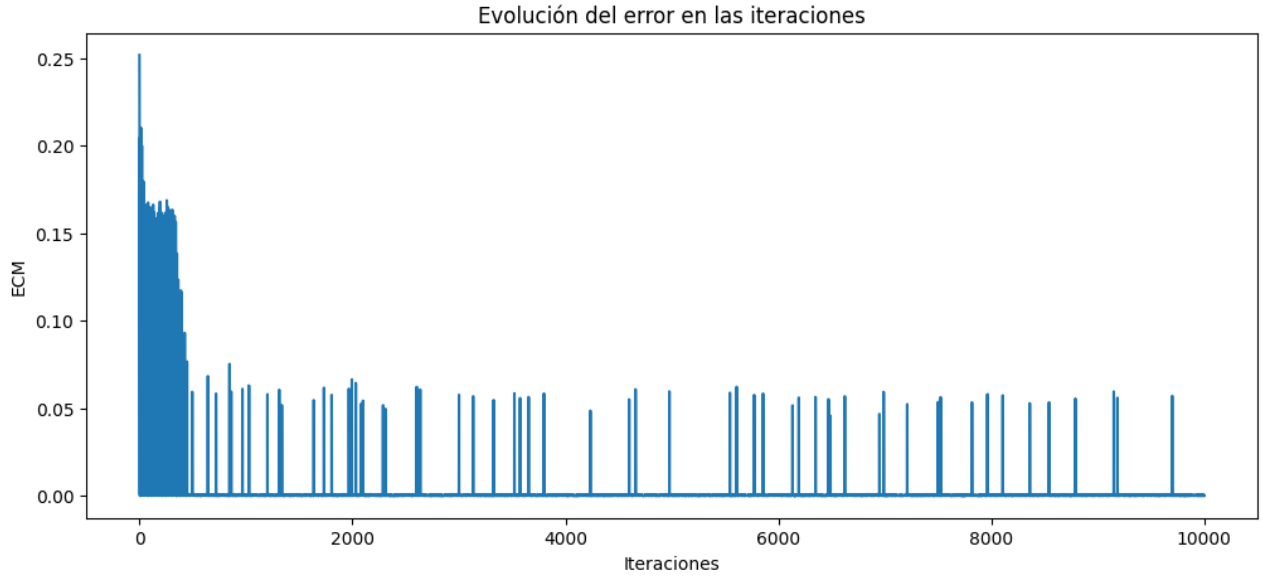

Figura 2.4.1.3: Error en función de las iteraciones

2.4.2. Efectos del tamaño del *minibatch* y la tasa de aprendizaje

Para diferentes valores de tamaño de minibatch y la tasa de aprendizaje, se simula el entrenamiento para ver su efecto sobre el error cuadrático medio y el tiempo de entrenamiento.

Para esto se elige un umbral de error del $0,01$ y entrenando el algoritmo con todas las combinaciones de valores de:

$$\text{Tasa de aprendizaje } (\eta) : 1e-4, 1e-3, 1e-2, 1e-1, 1$$
$$\text{Tamaño de minibatches: } 1, 5, 10, 20, 50, 200, 500$$

A partir de esto, luego de recopilar los datos se llega a los mapas de calor.

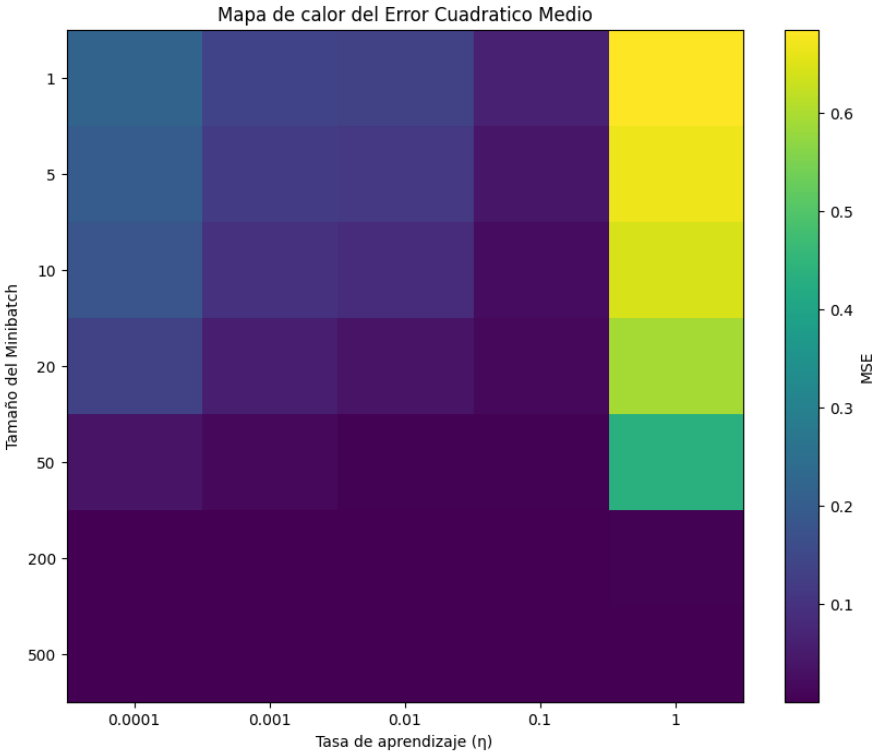

Figura 2.4.2.1: ECM para diferentes combinaciones de tasa de aprendizaje y tamaño de batches

En el mapa de calor del ECM se observa una disminución de este conforme se agranda el tamaño del minibatch, esto quiere decir que utilizar más datos por actualización ayuda a una estimación mejor.

Por otro lado, una tasa de aprendizaje muy pequeña muestra peor desempeño, sugiriendo que se converge muy lentamente hacia una solución óptima.

Además, una taza muy grande junto a tamaños pequeños de minibatch muestran el peor ECM, esto es por la pobre representación del problema además de una corrección en las actualizaciones muy grande, lo que hace que no pueda converger a buenos resultados.

Los mejores resultados se muestran con una tasa de aprendizaje de 0.1 y tamaños de minibatch grandes.

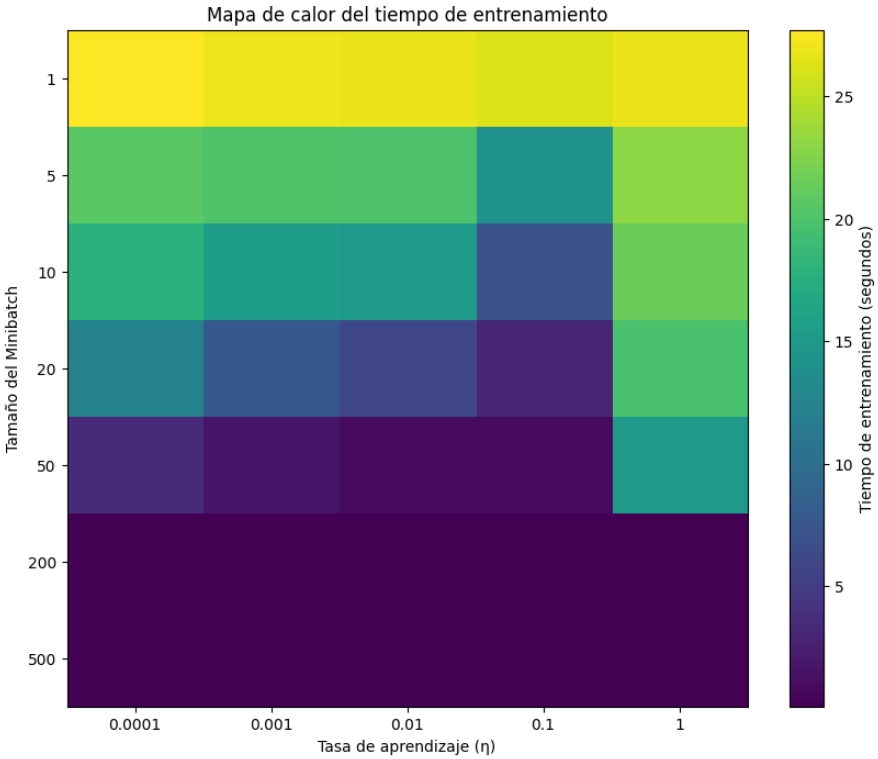

Figura 2.4.2.2: Tiempo de entrenamiento para diferentes combinaciones de tasa de aprendizaje y tamaño de batches

En el mapa de calor correspondiente al tiempo de entrenamiento, se observa un tiempo mayor al seleccionar minibatches pequeños, esto es debido al mayor numero de actualizaciones de pesos que deben realizarse.

Por otro lado, en cuanto a la tasa de aprendizaje, los mejores resultados nuevamente son en valores del orden de $\eta = 0,1$, con minibatches grandes.

2.5. Ejercicio 6: Optimización con algoritmo genético

Enunciado:

a) *Encontrar un perceptrón multicapa que resuelva una XOR de 2 entradas con un algoritmo genético. Graficar el fitness a lo largo del proceso de evolución.*

b) *¿Cómo impacta en el aprendizaje la constante de mutación, la probabilidad de crossover y el tamaño de la población?*

Para una función XOR de 2 entradas, una forma de encontrar una buena red neuronal que resuelva la función lógica, es a partir de un algoritmo genético.

Este se basa en el proceso de selección natural, manipulando una población con cierta cantidad de individuos. Estos representan soluciones potenciales al problema en cuestión y son evaluados por una función de fitness que determina qué tan buena es la solución.

El algoritmo tiene los siguientes pasos:

- **Inicialización**: Se genera una población inicial de individuos de manera aleatoria.
- **Evaluación**: Cada miembro de la población es evaluado mediante una función de fitness, que asigna un valor entre 0 y 1 indicando qué tan eficaz es la solución que el individuo representa. El fitness se calcula como:

$$\frac{E_{max} - E}{E_{max}}$$

- **Selección**: Basado en los resultados de la función de fitness, se seleccionan individuos para la reproducción. Primero se asegura la permanencia del individuo *elite*. Este será el que represente la mejor solución (mayor fitness). Luego se seleccionan los individuos restantes para completar la población, comparando un numero aleatorio con la probabilidad de selección de cada uno, que se basa en el fitness de la forma:

$$p(IND_i) = \frac{fitness(IND_i)}{\sum_{j=0}^{N-1} fitness(IND_j)}$$

- **Crossover**: Los individuos seleccionados se combinan para formar descendientes, con una probabilidad de cruce, incorporando características de ambos individuos.
- **Mutación**: Con una probabilidad baja, los pesos de los nuevos individuos experimentan mutaciones, que son cambios aleatorios en uno o más pesos sinápticos. La mutación introduce variabilidad en la población y ayuda a evitar los mínimos locales en la búsqueda de soluciones.

Repitiendo estos pasos una determinada cantidad de veces se busca una solución óptima, mejorando los fitness generación a generación.

2.5.1. Desarrollo del algoritmo genético

Para implementar el algoritmo genético se utilizaron redes de 2 neuronas de entrada (por la XOR de 2 entradas), 5 neuronas en la capa oculta y una neurona de salida. Igualmente el algoritmo esta preparado para otras configuraciones de red, las cuales fueron probadas y continúan funcionando correctamente.

Componen a cada población de la implementación un total de 50 individuos. La función de activación utilizada es la *tanh*. El crossover tiene una probabilidad de ocurrencia de $0,7$ y la mutación una probabilidad menor, de $0,1$.

El corte del bucle del algoritmo puede ser al encontrar un individuo que tenga un fitness de al menos $0,9999$, asegurando una solución óptima, o bien que el promedio de todos los fitness de los individuos sea de al menos $0,9$, asegurando buenas soluciones de todos los individuos. En caso de no encontrar solución, el algoritmo corta en 2000 iteraciones, eligiéndose un valor grande para asegurar el buen desempeño del algoritmo.

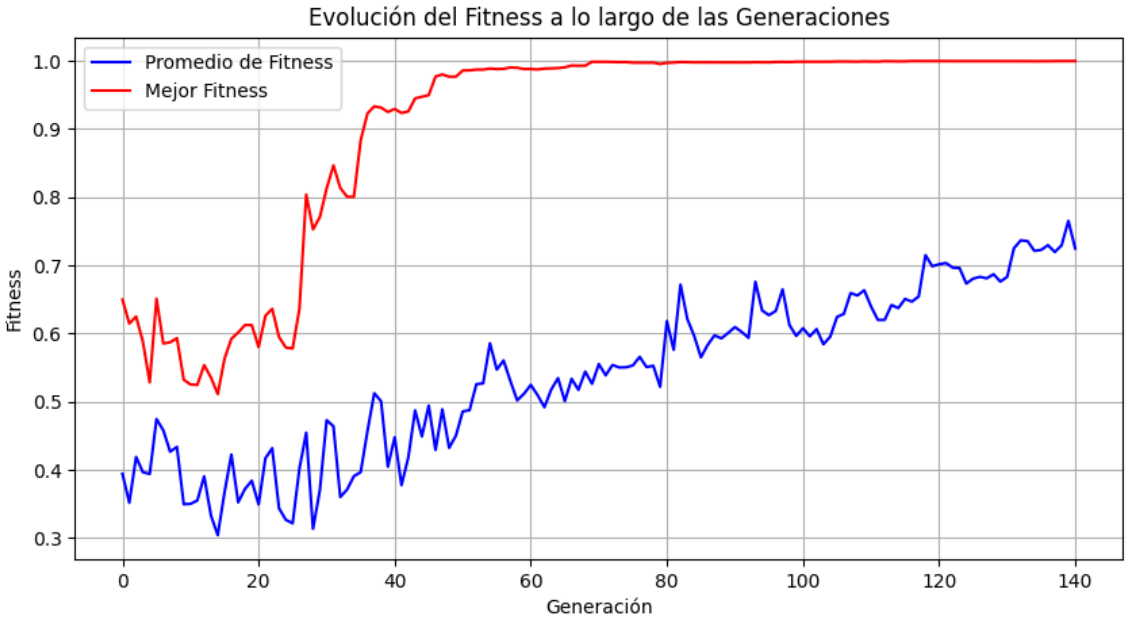

Figura 2.5.1.1: Fitness en función de las iteraciones - Corte por mejor fitness

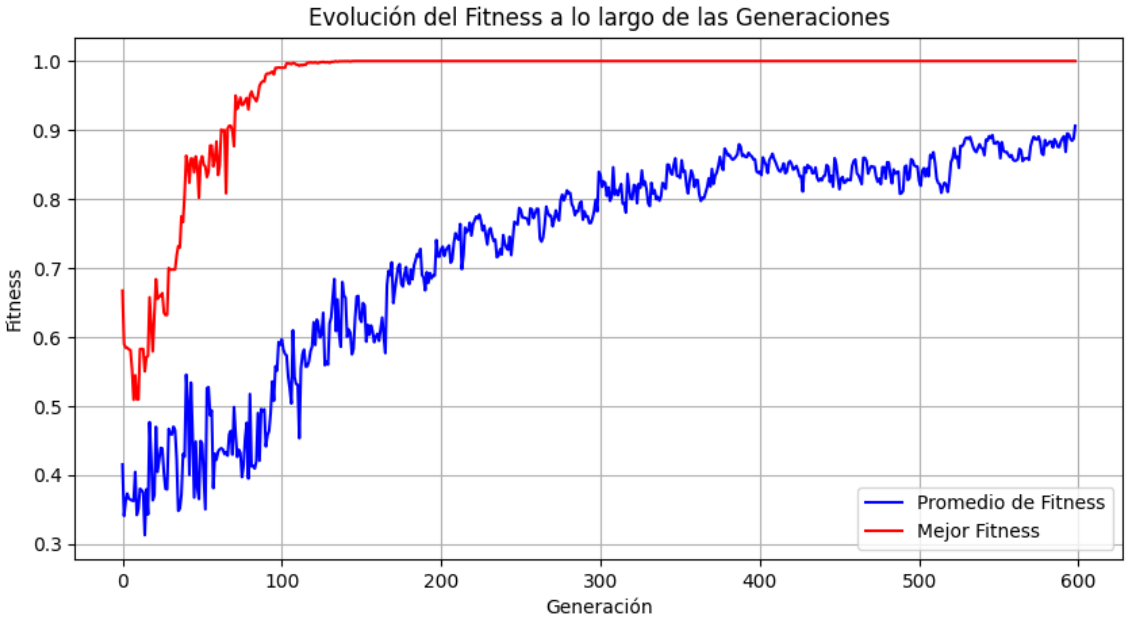

Figura 2.5.1.2: Fitness en función de las iteraciones - Corte por promedio del fitness

Claramente, mientras que el algoritmo con corte por promedio de fitness necesita alrededor de 600 iteraciones para finalizar, el corte por solución óptima precisa alrededor de 140 iteraciones.

2.5.2. Impacto de los parámetros en el aprendizaje

La elección de los parámetros del algoritmo puede impactar en el aprendizaje y su funcionalidad. Probando diferentes configuraciones se observa que:

- **Constante de Mutación**: Este parámetro varía aleatoriamente los pesos sinápticos de cada individuo de la población. Por lo visto en el Ejercicio 3, este paso puede ayudar a encontrar partes del espacio de soluciones que no se toman en cuenta, evitando mínimos locales y encontrando una solución óptima en menos iteraciones.

  Si se elige una probabilidad grande, se puede encontrar una solución óptima en menos iteraciones:

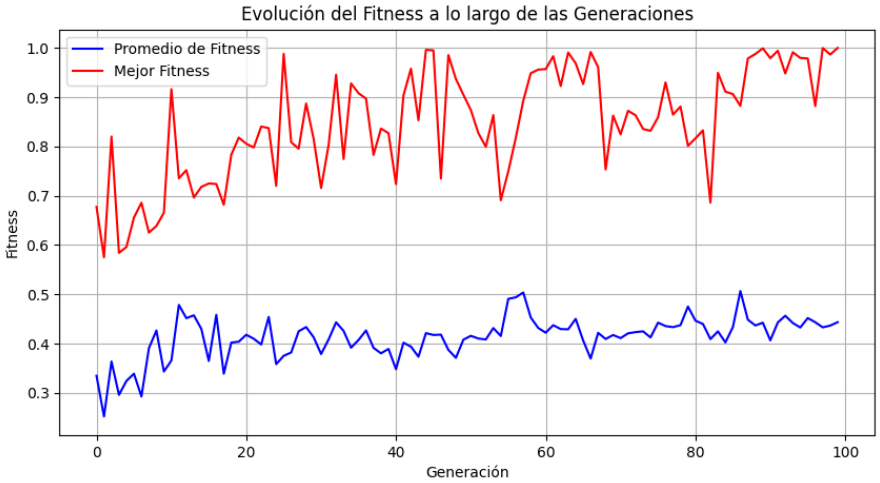

Figura 2.5.2.1: Evolución del Fitness - Probabilidad de mutación: 0,8 - Solución óptima

Ahora bien, con esta constante de mutación, se introducen demasiadas variaciones, haciendo que no se converja a una buena solución en promedio.

En el siguiente gráfico, si bien encuentra una solución óptima rápidamente, el promedio no logra llegar a $0,9$ en 2000 iteraciones.

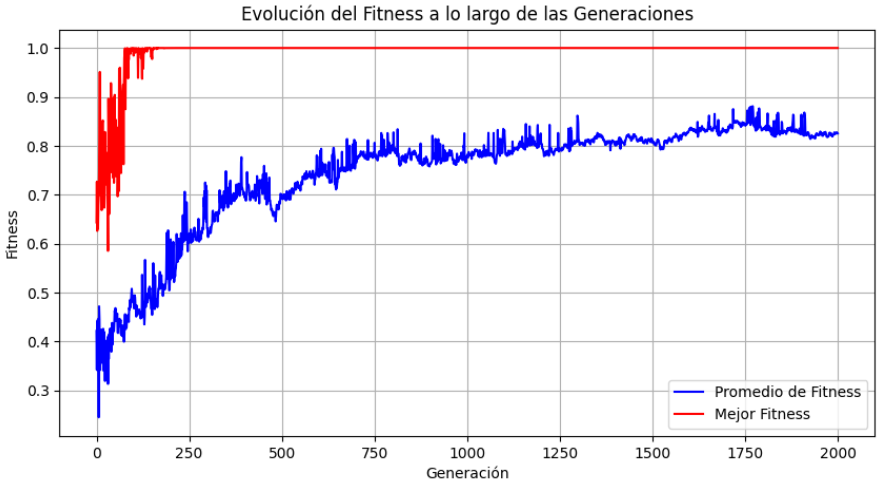

Figura 2.5.2.2: Evolución del Fitness - Probabilidad de mutación: 0,8 - Promedio

- **Probabilidad de Crossover**: Este parámetro determina que tan a menudo se combina el material genético de dos individuos para crear descendientes.

  Si está bien ajustada puede llevar a soluciones más óptimas, pero si es muy baja, no habrá tanta mezcla genética, lo que puede limitar la exploración del espacio de soluciones, y si es muy alta, las buenas soluciones estarán intervenidas, sin permitir que llegue a una buena solución por el constante desorden.

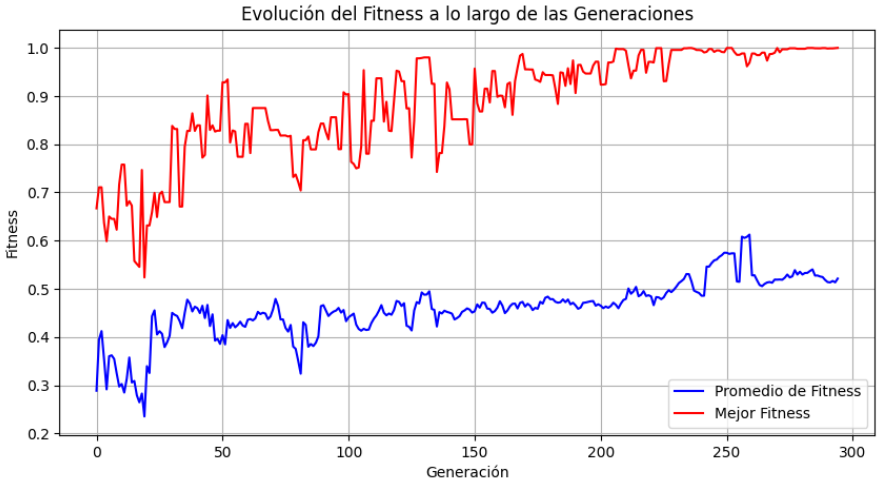

Figura 2.5.2.3: Evolución del Fitness - Probabilidad de crossover: 0,1

Aquí se ve claramente como eligiendo un valor pequeño, toma más del doble de iteraciones en encontrar una solución óptima.

- **Tamaño de la Población**: determina cuántos individuos tendrán las poblaciones, es decir, cuántas soluciones potenciales se consideran en cada generación.

  Un tamaño de población adecuado garantiza una buena representación de la diversidad genética, lo que permite explorar múltiples soluciones al mismo tiempo.

  Si el tamaño de la población es demasiado pequeño, no habrá suficiente diversidad genética para explorar eficazmente el espacio de soluciones. Si es demasiado grande, el costo computacional es alto, sin necesariamente mejorar los resultados.

  En la compilación, mientras que un tamaño de 5 individuos toma $0,8s$, uno de 50 toma $3,0s$, uno de 500 individuos toma un total de $12,9s$ en completar el algoritmo.