

# Implementación de un Perceptrón Multicapa para la Detección de Patrones en Matrices 10x10

Aldo Omar Andres<sup>1</sup>, Sixto Feliciano Arrejin<sup>1</sup>, Agustín Nicolás Bravo Pérez<sup>1</sup>, Tobias Alejandro Maciel Meister<sup>1</sup>, André Leandro San Lorenzo<sup>1</sup>

<sup>1</sup> Universidad Tecnológica Nacional, Facultad Regional Resistencia, Argentina  
aldoandres045@gmail.com  
arrejinsixto@gmail.com  
anbravoperez@gmail.com  
tobiasmaciel03@gmail.com  
andreleandrosanlorenzo@gmail.com

**Abstract.** Este trabajo presenta el diseño, implementación, entrenamiento, validación y optimización de un Perceptrón Multicapa (MLP) desarrollado en Python para la clasificación de patrones correspondientes a las letras b, d y f en una matriz 10x10. Se entrenó el modelo con tres conjuntos de datos de 100, 500 y 1000 ejemplos. Se analizan tres optimizaciones sobre el entrenamiento de la red: aumento de la cantidad de neuronas ocultas, disminución progresiva del factor de aprendizaje, y descenso del gradiente estocástico con mini-batches.

**Keywords:** Multilayer Perceptron, Artificial Intelligence, Neural Networks.

## 1 Introducción

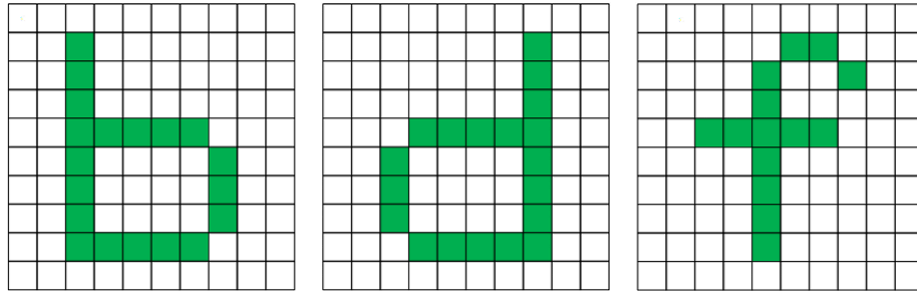
La inteligencia artificial generativa continúa ganando relevancia en múltiples dominios, y comprender su funcionamiento se vuelve cada vez más importante. Estos sistemas de IA se construyen sobre LLMs, modelos que son el último resultado de una evolución que comenzó hace décadas con las redes neuronales artificiales, las cuales se originan en la inteligencia artificial tradicional. Las redes neuronales son un modelo inspirado en la biología de los sistemas nerviosos, donde una neurona captura señales y se conecta con otras neuronas para procesar esa información.

La red neuronal más básica es el perceptrón, creado por Rosenblatt en 1958 [7]. Consiste en una sola neurona con varias entradas y una salida. A esta red se la entrena con un conjunto de datos para que aprenda patrones y pueda generalizarse a nuevos datos desconocidos. El perceptrón multicapa (*Multi-layer Perceptron*, MLP) es una evolución del perceptrón simple que agrega capas ocultas a la red para poder modelar relaciones más complejas. Es un paso hacia las redes neuronales profundas (*deep learning*), sistemas sobre los cuales se construyen los LLMs actuales.

El trabajo presenta el diseño, desarrollo, entrenamiento, validación y optimización de un MLP para la detección de patrones en una matriz 10x10. Fue realizado en el marco del cursado 2025 de la asignatura Inteligencia Artificial de la carrera de Ingeniería en Sistemas de Información de la Universidad Tecnológica Nacional, Facultad Regional Resistencia.

### 1.1 Escenario

Dada una matriz 10x10 donde cada celda puede estar marcada o vacía, se desea detectar el patrón de entrada y clasificarlo en una de las clases de la figura 1. Un patrón de entrada puede tener una distorsión del 1% al 30%, donde una distorsión del 30% significa que 15 de sus 100 celdas, elegidas al azar, han sido alternadas. También se debe contar con una interfaz de usuario que permita ingresar un patrón arbitrario.



**Fig. 1.** Patrones a detectar en la matriz 10x10 y clasificar como letra 'b', 'd' o 'f'.

Se codificó los patrones de entrada como un arreglo de 100 bits, donde cada bit representa una celda y un 1 indica que esa celda está marcada.

### 1.2 Generación del Conjunto de Datos

Se generaron tres conjuntos de datos (*datasets*). En cada uno, el 10% de los patrones no están distorsionados y el resto tienen una distorsión del 1% al 30% según una distribución uniforme. Estos datasets fueron divididos tres veces cada uno en datasets de entrenamiento y validación acorde a la tabla 1.

**Tabla 1.** Conjuntos de datos generados.

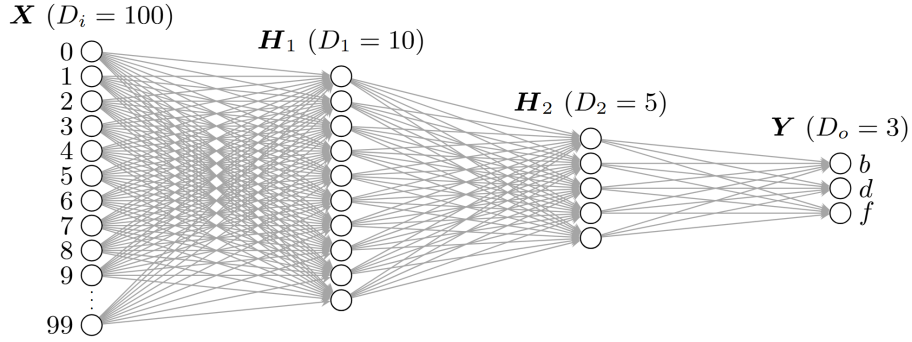
Dataset	Cantidad de Ejemplos	Porcentaje para Entrenamiento	Porcentaje para Validación
1	100	90%	10%
2	100	80%	20%
3	100	70%	30%
4	500	90%	10%
5	500	80%	20%
6	500	70%	30%
7	1000	90%	10%
8	1000	80%	20%
9	1000	70%	30%

La separación entre entrenamiento y validación es aleatoria. Los datasets deben ser representativos al definir la distribución de los ejemplos de entrenamiento.

## 2 Algoritmo

Inicialmente se diseñó un perceptrón multicapa tomando como referencia principal la red MADALINE presentada en el libro de Hiler & Ramirez [4][8]. El libro de Prince se consultó como material complementario [5]. El escenario define las siguientes restricciones de diseño para el MLP:

- 1 o 2 capas ocultas.
- De 5 a 10 neuronas por capa.
- Funciones de activación: lineal y sigmoideal.
- Coeficiente de aprendizaje entre 0 y 1.
- Término momento entre 0 y 1.



**Fig. 2.** Arquitectura del perceptrón multicapa diseñado.

La figura 2 representa una red neuronal feed-forward completamente conectada. Se la puede entender como una función  $f[X, \phi] = Y$  que clasifica el vector de entrada  $X = [x_1, x_2, \dots, x_{100}]^T$  en la clase asociada a la activación de mayor valor del vector de salida  $Y = [y_b, y_d, y_f]^T$ . El conjunto de parámetros  $\phi = \{B_k, W_k\}_{k=0}^K$  contiene:

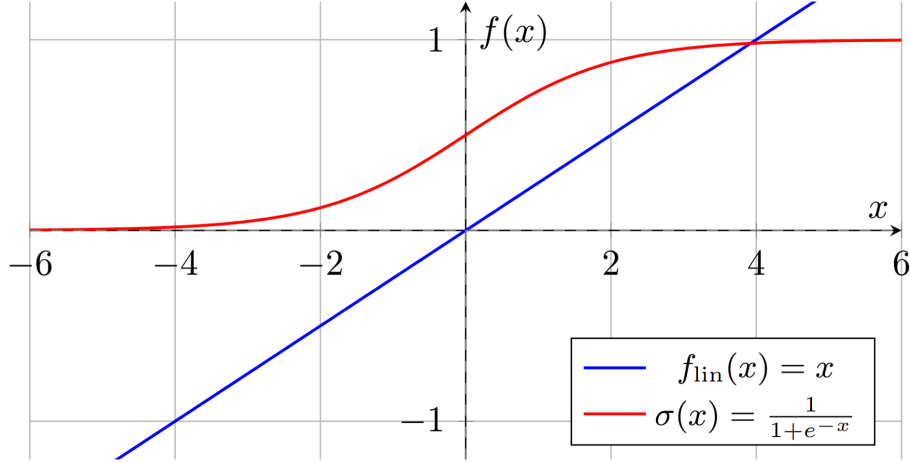
- El vector de umbrales (*biases*)  $B_k$  que contribuyen a la capa  $k + 1$ .
- La matriz de pesos (*weights*)  $W_k$  aplicados a la capa  $k$  y que afectan a  $k + 1$ .

La red neuronal se puede representar utilizando notación matricial: [5]

$$\begin{aligned} X &= [x_1, x_2, \dots, x_{100}]^T \\ H_1 &= a[B_1 + W_1 X] \\ H_2 &= a[B_2 + W_2 H_1] \\ Y &= B_3 + W_3 H_2 \end{aligned}$$

Donde  $a[\bullet]$  es una función que aplica la función de activación por separado a cada elemento de su vector de entrada. En este trabajo se implementaron las funciones de activación lineal y sigmoideal (figura 3), aunque solo se utilizó la función sigmoideal.

### Funciones de activación: lineal y sigmoideal



**Fig. 3.** Funciones de activación disponibles. En los experimentos solo se usó la sigmoideal.

Al diseñar una red neuronal se debe elegir la regla de aprendizaje o *loss function* a utilizar. En este trabajo se utilizó la regla *Least Mean Squared* o regla delta presentada por Hilera & Martínez: [4]

$$\epsilon_k^2 = \frac{1}{2L} \sum_{k=1}^L (d_k - s_k)^2$$

Esta regla compara la salida obtenida contra la salida deseada para obtener el error, costo o pérdida. El entrenamiento del MLP consiste en ajustar iterativamente los parámetros del modelo para reducir ese error en cada iteración, tal que el conocimiento de la red se almacena en los pesos y umbrales.

Al tratarse de una red neuronal multicapa, se utiliza el algoritmo *backpropagation* del gradiente descendiente para ajustar los umbrales y pesos del modelo luego de cada ejecución del algoritmo *feedforward*. Este es el mecanismo de aprendizaje del modelo, y se repite hasta llegar a una condición de freno. La condición de freno utilizada en el MLP indica terminar el entrenamiento cuando la disminución del error entre una iteración y la siguiente resulte menor a una tolerancia igual a 0.000001.

El *momento* es una modificación al algoritmo *backpropagation* que evita oscilaciones y suaviza el progreso del algoritmo a lo largo del entrenamiento. Introduce un término momento en el cálculo del nuevo valor de los pesos. El cálculo del peso  $w_{ji}$  es: [4]

$$\begin{aligned} w_{ji}(t+1) &= w_{ji}(t) + \alpha \delta_{pj} y_{pi} + \beta (w_{ji}(t) - w_{ji}(t-1)) \\ \Delta w_{ji}(t+1) &= \alpha \delta_{pj} y_{pi} + \beta \Delta w_{ji}(t) \end{aligned}$$

El coeficiente  $\beta$  es un nuevo hiper-parámetro que determina la relevancia de los valores anteriores del peso en el cálculo del nuevo valor del peso.

### 3 Desarrollo

Para el desarrollo del trabajo se utilizaron las siguientes herramientas:

- Python como lenguaje de programación.
- Pandas, NumPy y matplotlib como paquetes principales de Python.
- Jupyter Notebook y Google Colab como entorno de ejecución.
- Git, GitHub y Trello para la colaboración y el seguimiento del proyecto.
- Gradio como paquete de UI para desarrollar la interfaz gráfica.

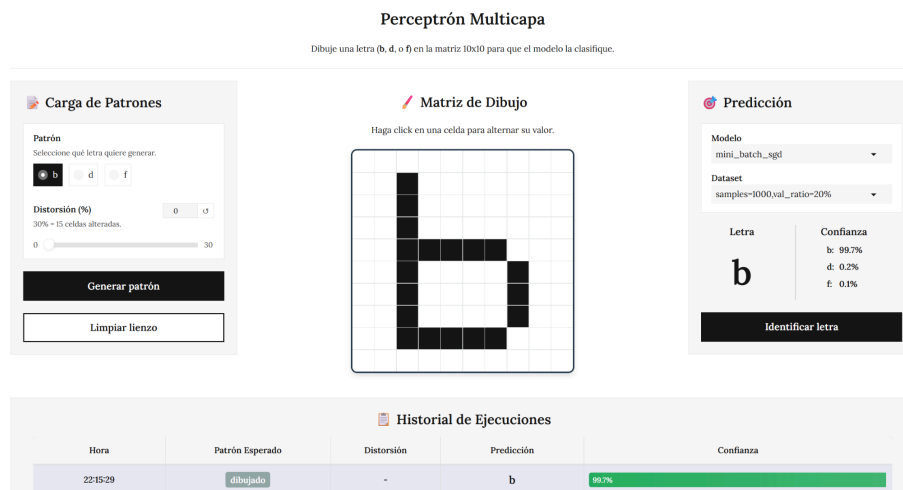
En Trello se trabajó con un tablero Kanban que permitió visualizar el estado del proyecto en todo momento. Resultó ser una herramienta útil para el seguimiento del trabajo dentro del equipo. [2]

La mayoría del trabajo se realizó en un solo Jupyter Notebook para que sea fácil de compartir. Este notebook está disponible en un repositorio público de GitHub y se puede abrir desde Google Colab. [1][3][6]

La ejecución completa del notebook demora varios minutos dado que se generan los conjuntos de datos y luego se entrenan múltiples modelos con distintas variaciones de hiper-parámetros. Se utilizó un generador de números aleatorios con una semilla predefinida para garantizar la reproducibilidad de los experimentos.

Se implementó cada dataset como un DataFrame de pandas con 101 columnas, donde las columnas 0-99 representan las celdas de la matriz 10x10 y la columna *class* indica la clase a la cual pertenece el ejemplo. Cada fila del DataFrame es un ejemplo o patrón de entrada. En la clase **MLP** de Python se implementó un perceptrón multicapa con la arquitectura presentada en la sección anterior. Luego se desarrolló una clase **MLP2** más sofisticada que permite variar la arquitectura de la red, lo cual fue útil para realizar experimentos y optimizar el entrenamiento del modelo.

Finalmente, se construyó una interfaz gráfica que permite al usuario ingresar patrones arbitrarios y ejecutar el MLP para clasificarlos, validando así el diseño y entrenamiento de la red neuronal. La figura 4 ilustra el resultado final.



**Fig. 4.** Interfaz gráfica para utilizar el perceptrón multicapa.

## 4 Experimentos y Resultados

Cada experimento o simulación en el notebook implicó cuatro pasos:

1. Configurar los parámetros del modelo (ya sea la clase MLP o MLP2).
2. Entrenar un modelo con esos parámetros para cada par de datasets de entrenamiento y validación. Esto da un total de nueve modelos entrenados. El entrenamiento termina cuando la diferencia entre el error actual y el error de la época anterior es menor a una tolerancia igual a 0.000001.
3. Graficar la evolución del error global y el error de validación a lo largo de los nueve entrenamientos. Se utilizó matplotlib para realizar los gráficos.
4. Calcular la precisión de validación para cada modelo entrenado. La precisión del modelo está dada por la cantidad de predicciones correctas dividida por la cantidad de ejemplos a predecir. Se utilizó esta medida para comparar entre experimentos.

### 4.1 Modelo Original

Se entrenó un modelo con la clase MLP del notebook, que implementa la arquitectura presentada en secciones anteriores. Hiper-parámetros seleccionados:

- Arquitectura: [100, 10, 5, 3], es decir una primera capa oculta con diez neuronas ocultas y una segunda capa oculta con cinco neuronas ocultas.
- Función de activación: sigmoideal.
- Factor de aprendizaje: 0.1.
- Factor momento ( $\beta$ ): 0.1.

Al finalizar el entrenamiento se obtuvo el gráfico de la figura 5, que muestra la evolución del error. Estos resultados motivaron nuevos experimentos con variaciones en los hiper-parámetros y ajustes en la arquitectura para mejorar el entrenamiento de la red neuronal. A estos efectos se desarrolló una segunda implementación del perceptrón multicapa en la clase MLP2. Esta nueva clase permitió parametrizar la arquitectura de la red y definir ciertos hiper-parámetros adicionales necesarios para las optimizaciones que se realizaron.

### 4.2 Aumento de las Neuronas Ocultas

El primer experimento consistió en incrementar la cantidad de neuronas de la segunda capa oculta de 5 a 10, alcanzando así la configuración máxima según las restricciones del escenario. La arquitectura resultante es: [100, 10, 10, 3]. El resto de los hiper-parámetros se mantuvieron iguales para aislar el efecto de esta modificación sobre el desempeño del modelo.

La intuición detrás de este ajuste radica en que aumentar la cantidad de neuronas ocultas incrementa la capacidad de la red para representar funciones más complejas, lo que le permite capturar relaciones no lineales más profundas en los datasets.

Durante las pruebas, se observó una mejora en el aprendizaje de todos los modelos, incluso aquellos para los cuales el modelo original aprendía poco. También se observó una tendencia al sobreajuste.

### 4.3 Disminución Progresiva del Factor de Aprendizaje

Un factor de aprendizaje constante puede provocar que la red oscile en el entorno del mínimo local si sucede que los pasos de actualización son muy grandes para etapas tardías del entrenamiento. En el segundo experimento se implementó una estrategia de disminución progresiva del factor de aprendizaje (*learning rate*), implementando un *learning rate decay* que disminuye el valor del factor de aprendizaje en un 20% cada 25 épocas. El objetivo de esta técnica fue reducir las oscilaciones en la función del error durante las últimas etapas del entrenamiento para estabilizar la convergencia del modelo. Se mantuvieron los mismos parámetros y la arquitectura con neuronas ocultas adicionales del experimento anterior.

Finalizadas las pruebas con este nuevo parámetro, se observó una leve mejora en la convergencia y una menor tendencia al sobreajuste, posiblemente porque la red llega al mínimo local de manera más directa. Comparando con el resto de optimizaciones, se consideró que esta optimización tiene un impacto notable pero no importante en el entrenamiento de la red.

### 4.4 Descenso del Gradiente Estocástico con Mini-batches

La clase MLP entrena de un ejemplo a la vez, lo que se puede considerar como un descenso del gradiente estocástico (*Stochastic Gradient Descent*, SGD) puro. Existe una variante de SGD que utiliza mini-batches para entrenar la red con varios ejemplos a la vez. Un *batch\_size* igual a 1 sería SGD puro. La implementación de MLP2 admite un parámetro que indica el tamaño de estos batches, y para este experimento se definió un *batch\_size* igual a 16. También se mantuvieron los parámetros de los experimentos anteriores.

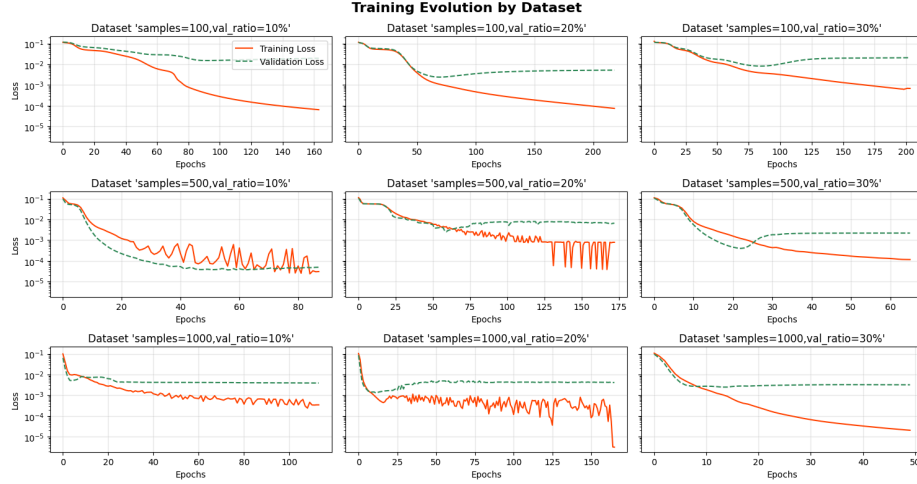
La evolución del entrenamiento de este modelo se presenta en la figura 6. Se observó una gran reducción del sobreajuste, aunque existe una desventaja importante: el error disminuye muy poco en los datasets de 100 ejemplos. Se consideró que el aprendizaje es deficiente en estos datasets debido a que un batch de 16 ejemplos sobre un datasets de tan solo 100 ejemplos reduce la variabilidad de los gradientes y dificulta la exploración del espacio de parámetros.

### 4.5 Búsqueda Exhaustiva de Hiper-parámetros

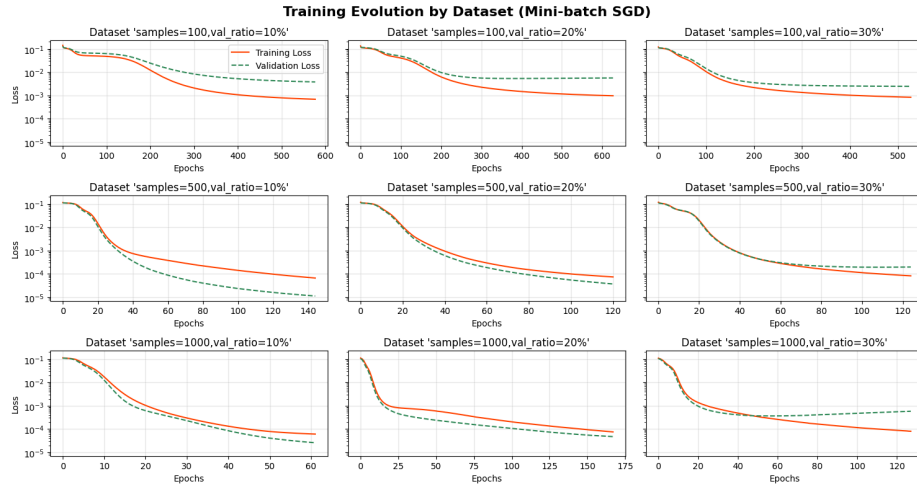
Finalmente se implementó una estrategia de *grid search* para encontrar mejores hiper-parámetros mediante una búsqueda exhaustiva por fuerza bruta. Este enfoque es ineficiente ya que requiere una enorme cantidad de entrenamientos, pero genera resultados estadísticos que pueden ser interesantes de analizar. Se explora el siguiente espacio de hiper-parámetros:

- Tamaño de la capa oculta 1: [50, 40, 30, 20, 10].
- Tamaño de la capa oculta 2: [25, 20, 15, 10, 5].
- Disminución progresiva del factor de aprendizaje: [0.01, 0.02, 0.05].
- Tamaños de los batches: [16, 8].

La búsqueda exhaustiva en ese espacio da un total de 150 entrenamientos necesarios. Se decidió realizar este experimento en otro Jupyter Notebook para separarlo del trabajo principal. Sus resultados son demasiado extensos y se omiten del informe.



**Fig. 5.** Evolución del entrenamiento para el modelo original. Las curvas naranjas indican el error global (de entrenamiento) y las curvas verdes indican el error de validación. Se observan oscilaciones en algunos datasets, poco aprendizaje en otros, y sobreajuste en algunos. Estas observaciones motivaron nuevos experimentos con variaciones en los hiper-parámetros y la arquitectura del modelo con el objetivo de mejorar el entrenamiento del MLP.



**Fig. 6.** Evolución del entrenamiento del modelo con descenso de gradiente estocástico (SGD) en mini-batches de 16 ejemplos por batch. Este modelo incluye las demás optimizaciones: el aumento del número de neuronas ocultas a 10 neuronas por capa y la disminución progresiva del factor de aprendizaje en un 20% cada 25 épocas. Se observa una curva de error suavizada por la disminución progresiva del factor de aprendizaje y un aprendizaje más estable gracias a la mayor cantidad de neuronas ocultas. El SGD con mini-batches reduce el sobreajuste presente en entrenamientos anteriores. Los datasets de 100 ejemplos presentan en general un error mayor al error que tienen en el modelo original, lo que sugiere que utilizar mini-batches es inconveniente cuando la cantidad de ejemplos es baja en relación al tamaño de los batches.

#### 4.6 Comparación entre Experimentos

La tabla 2 presenta las precisiones de validación calculadas para cada modelo con el cual se han realizado experimentos.

**Tabla 2.** Precisiones obtenidas por cada modelo entrenado.

Dataset	Modelo original	Aumento de neuronas ocultas	Disminución del factor de aprendizaje	SGD con mini-batches
samples=100,val_ratio=10%	<b>0.900</b>	1.000	1.000	1.000
samples=100,val_ratio=20%	<b>0.950</b>	<b>0.950</b>	<b>0.950</b>	<b>0.950</b>
samples=100,val_ratio=30%	<b>0.933</b>	1.000	1.000	1.000
samples=500,val_ratio=10%	<b>0.990</b>	1.000	1.000	1.000
samples=500,val_ratio=20%	<b>0.975</b>	1.000	1.000	1.000
samples=500,val_ratio=30%	<b>0.986</b>	<b>0.996</b>	<b>0.996</b>	<b>0.996</b>
samples=1000,val_ratio=10%	1.000	1.000	1.000	1.000
samples=1000,val_ratio=20%	<b>0.980</b>	1.000	1.000	1.000
samples=1000,val_ratio=30%	<b>0.993</b>	1.000	1.000	1.000

El experimento de SGD con mini-batches reduce el sobreajuste del entrenamiento, logrando así la mejor precisión en todos los datasets excepto en el de 100 ejemplos con 30% de ejemplos en el dataset de validación, donde el aprendizaje es deficiente. Se cree que este fenómeno se debe a que un dataset de entrenamiento con 70 ejemplos es muy pequeño para utilizar SGD con un tamaño de batch de 16 ejemplos.

En conjunto, los resultados demuestran que la interacción entre los hiper-parámetros tiene un efecto no lineal sobre la precisión y la generalización del modelo. Sin embargo, la complejidad del modelo debe ser coherente con el tamaño y la diversidad del conjunto de datos, en este caso priorizando arquitecturas simples pero bien ajustadas.

Se analizó que el aumento de neuronas ocultas es la optimización más simple de implementar y más efectiva para este escenario. Ignorando las restricciones de diseño dadas por el escenario, sería recomendable probar una arquitectura más profunda, es decir aumentar incluso más la cantidad de capas ocultas y la cantidad de neuronas por capa oculta, con el objetivo de mejorar la tolerancia al ruido y la capacidad del modelo de generalizar a datos nuevos.

Si bien el escenario es relativamente sencillo, para continuar optimizando el rendimiento de la red neuronal se puede:

- Aumentar significativamente el tamaño y la diversidad del conjunto de datos, Incluir ejemplos con los patrones desplazados del centro de la matriz. Se podrían incorporar nuevas clases correspondientes a otras letras del alfabeto.
- Implementar un mecanismo de parada temprana (*early stopping*) que detecte el punto de convergencia antes del sobreentrenamiento.
- Experimentar con otras funciones de activación como ReLU.
- Implementar optimizadores más sofisticados como Adam.

## 5 Conclusiones

A lo largo del trabajo se diseñó e implementó un perceptrón multicapa capaz de reconocer tres tipos de patrones en matrices  $10 \times 10$  con distintos niveles de distorsión. Los experimentos demostraron que los hiper-parámetros influyen de forma decisiva en la precisión del modelo, evidenciando mejoras en el entrenamiento al utilizar una arquitectura más profunda y un conjunto de datos más grande.

El aumento de neuronas ocultas fue la optimización más efectiva, mientras que la disminución progresiva del factor de aprendizaje y el SGD con mini-batches dieron beneficios específicos: mayor estabilidad y menor sobreajuste en datasets amplios, pero bajo impacto en los datasets más pequeños. Un modelo más complejo requiere más ejemplos para aprender. Otra limitación del modelo actual es que no reconoce patrones desplazados del centro de la matriz. Se plantearon mejoras futuras para optimizar la red neuronal ignorando las restricciones de diseño del escenario.

El MLP implementado cumple las necesidades del escenario dentro de las restricciones de diseño definidas. Se desarrolló una interfaz gráfica que permite al usuario ingresar patrones arbitrarios para clasificarlos con el modelo.

Aunque el campo de la inteligencia artificial ha avanzado hacia arquitecturas mucho más complejas que el perceptrón multicapa, este modelo sencillo permite explorar fundamentos que siguen vigentes. Comprender el funcionamiento interno de un MLP permite apreciar la evolución de las redes neuronales hacia el origen de los LLMs actuales. Conceptos como el aprendizaje supervisado, la retropropagación del error y la optimización de hiper-parámetros constituyen la base de las redes neuronales profundas que hoy impulsan la IA generativa.

## Referencias

1. Andres, A. O., Arrejin, S. F., Bravo Pérez, A. N., Maciel Meister, T. A., San Lorenzo, A. L.: Multilayer Perceptron [Notebook de Jupyter]. Google Colab (2025), <https://colab.research.google.com/github/elepad-org/mlp/blob/main/model/notebook.ipynb>
2. Andres, A. O., Arrejin, S. F., Bravo Pérez, A. N., Maciel Meister, T. A., San Lorenzo, A. L.: TP Inteligencia Artificial [Tablero Kanban]. Trello (2025), <https://trello.com/b/KvPLKgKd/tp-inteligencia-artificial>
3. Andres, A. O., Arrejin, S. F., Bravo Pérez, A. N., Maciel Meister, T. A., San Lorenzo, A. L.: Trabajo Práctico MLP [Repositorio de GitHub]. GitHub (2025), <https://github.com/elepad-org/mlp>
4. Hiler, J., Martínez, R.: Redes neuronales artificiales: fundamentos y aplicaciones. Alfaomega, México (2000)
5. Prince, S. J. D.: Understanding Deep Learning, p. 48. MIT Press, Cambridge, MA (2023)
6. Project Jupyter: Jupyter Notebook Documentation. Jupyter (2024), <https://jupyter-notebook.readthedocs.io/en/latest>
7. Rosenblatt, F.: The Perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review 65(6), 386–408 (1958), <https://doi.org/10.1037/h0042519>
8. Widrow, B., Winter, R. W.: MADALINE Rule II: A training algorithm for neural networks, Technical Report. Stanford University, Stanford, CA (1988), <https://www-isl.stanford.edu/~widrow/papers/c1988madalinerule.pdf>