Escuela de
Ciencia y Tecnología
ECyT_UNSAM

REDES NEURONALES

"Predicción de la Calidad de Manzanas"

- **Alumna: Carolina Rimini**
- **Docentes: Josefina Bompensieri y Tomas Prudente**

2025

Parte 1: Análisis de la Base de Datos

Selección de la Base de Datos:

Para este trabajo práctico decidí usar un conjunto de datos sobre manzanas, que encontre en Kaggle, el cual incluye varias características físicas y sensoriales de las fruta. Elegí este dataset porque me pareció interesante para aplicar un modelo de clasificación binaria. El objetivo es predecir si una manzana es de buena o mala calidad según sus atributos. Esto va de la mano con la propuesta del proyecto, ya que se trata de implementar una red neuronal capaz de resolver tareas de clasificación mas realistas.

Descripción de cada columna

El conjunto de datos contiene las siguientes columnas:

- Size: Tamaño de la manzana. Variable continua.
- Weight: Peso en gramos. Variable continua.
- Sweetness: Nivel de dulzura (0-10). Variable discreta.
- Crunchiness: Crocancia (0-10). Variable discreta.
- Juiciness: Jugosidad (0-10). Variable discreta.
- Ripeness: Madurez (0-10). Variable discreta.
- Acidity: Nivel de acidez. Originalmente contenía valores no numéricos; la convertí a numérica. Variable continua.
- Quality: Variable categórica nominal ('good' o 'bad'). La transformé en una variable binaria llamada Quality_numeric (1 = buena, 0 = mala).

Estas columnas representan características medibles que pueden tener un impacto directo en la percepción de calidad de una manzana por parte de un consumidor.

Análisis de Correlaciones

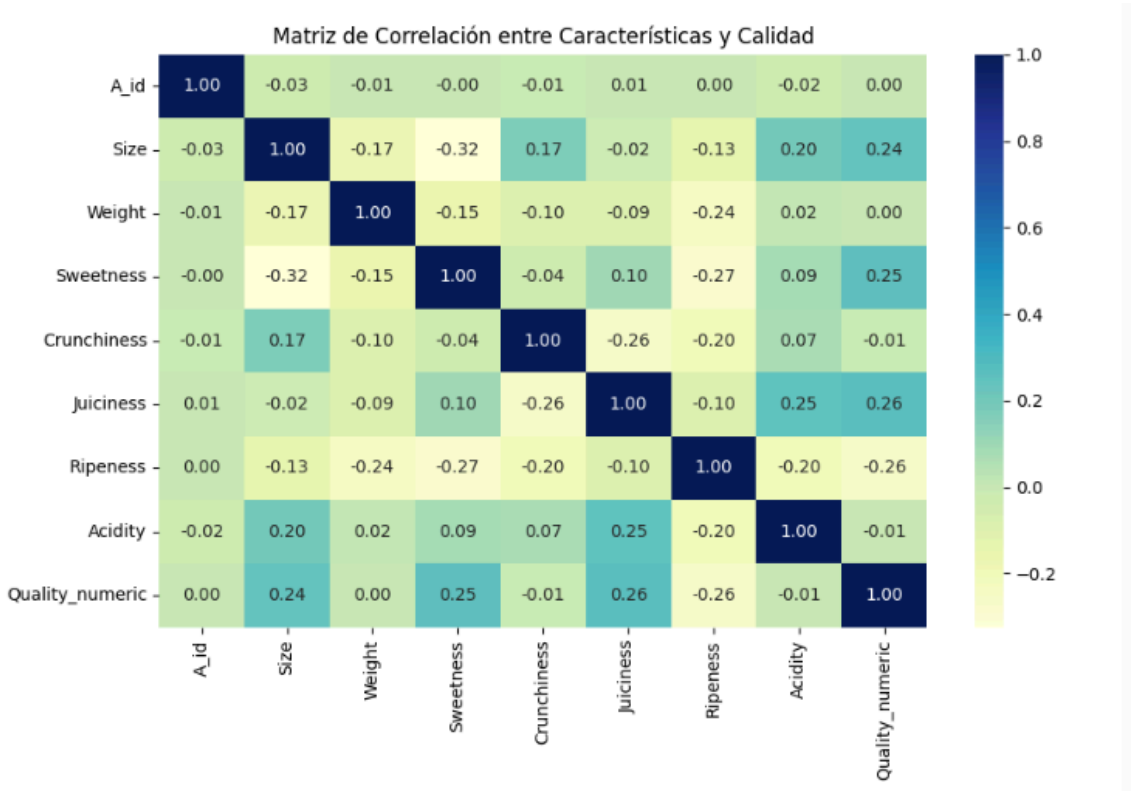
Realicé un mapa de calor de correlaciones, el cual me sirvio para observar la relación entre las características y la variable objetivo. Gracias a este encontre que:

- Weight y Size tenian alta correlación entre sí, lo cual es esperable porque el tamaño influye directamente en el peso.

- Sweetness, Juiciness y Crunchiness presentan correlación positiva moderada con la calidad. Lo cual tiene sentido porque una manzana dulce, jugosa y crocante probablemente es considerada de mejor calidad.

- Acidity tiene poca correlación, pero nom me parecio necesario sacarla porque pense que podría aportar información que me sirve. Por ejemplo, un nivel de acidez demasiado alto podría afectar negativamente la percepción.

El análisis de correlación fue muy importante para poder decidir con qué variables continuar y también mas que nada, para entender cómo interactúan entre ellas. Gráfico de la Matriz de Correlación:



Análisis de Factibilidad

Como mencione previamente, utilice un dataset con el objetivo de predecir la calidad de las manzanas. Después de hacer el analisis del mismo, noté que varias de las características tienen una buena correlación con la variable objetivo, que en este caso es la "calidad" (por ejemplo, si la manzana es buena o no). Esto fue muy útil, ya que trabajé con una red neuronal, que necesita que los datos ya estén etiquetados. Como en el dataset cada manzana ya viene clasificada según su calidad, la red pudo entrenarse bien y aprender a reconocer patrones que indican si la manzana es de buena calidad o no.

Esta red tiene como objetivo principal predecir si una manzana, con ciertas características físicas y químicas, cumple o no con los “estándares” necesarios para ser considerada de buena calidad. Por ejemplo, que tenga un color uniforme, buen peso, y que no esté golpeada, machucada. Si no cumple con esos criterios, la red la clasifica como una manzana común o de menor calidad.

Este tipo de sistema me parece que podria ser muy útil en la industria alimentaria, ya que permite automatizar el control de calidad. Al ingresar los datos de una nueva manzana, la red puede dar una predicción rápida y ayudar a decidir si la fruta está en buenas condiciones para el consumo o si debería destinarse a otro uso.

En resumen, esta base de datos me parecio adecuada para entrenar una red neuronal porque:

- Las variables predictoras están bien definidas y cuantificables.
- No hay muchas columnas categóricas, lo que facilita el preprocesamiento y el analisis.
- Hay una relación que es bastante visible entre los atributos y la variable objetivo.
- La cantidad de datos me parecio suficiente para poder entrenar una red sencilla.

Datos Atípicos y Limpieza de Datos

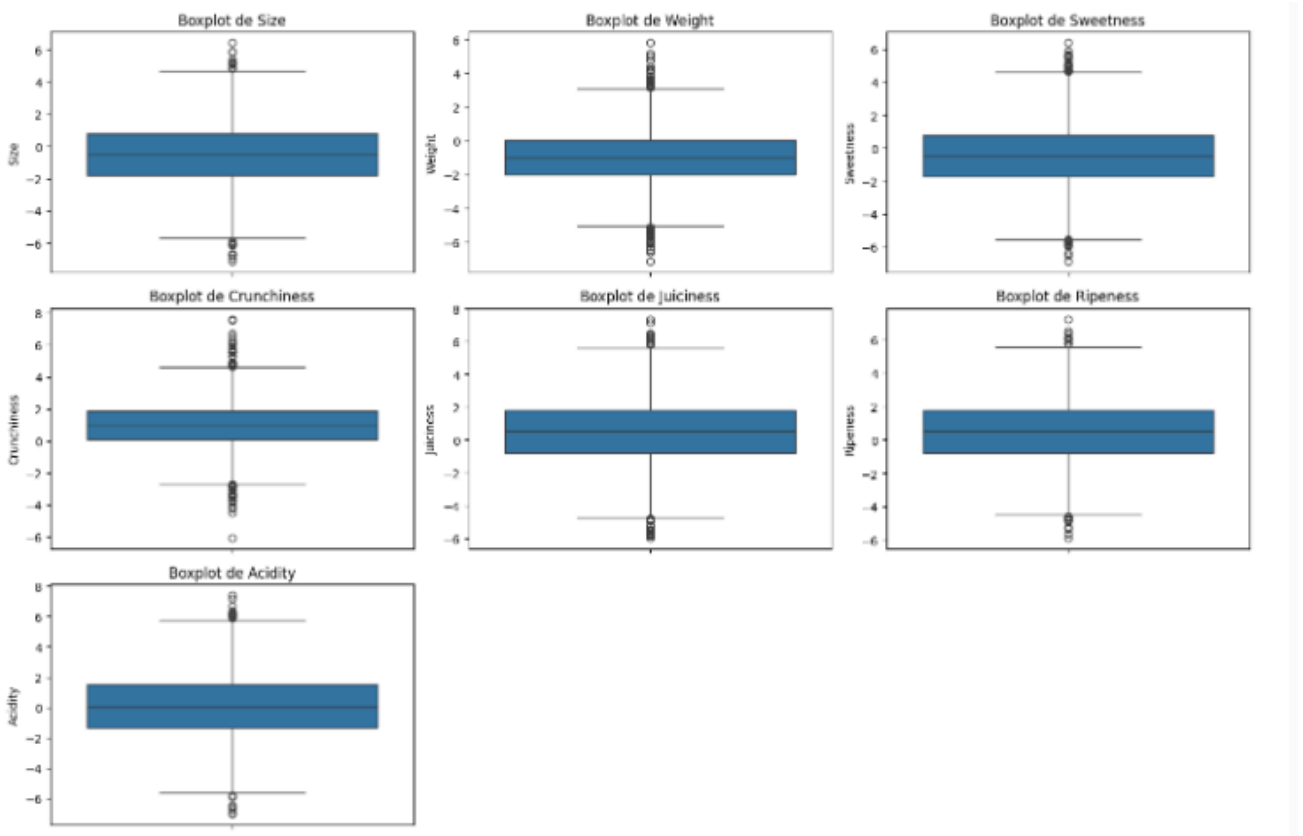
Mientras trabajaba con el dataset, uno de los primeras cosas que hice fue revisar si había errores en los datos. Por ejemplo, noté que la columna de *Acidez* no estaba en formato numérico, así que usé el comando pd.to_numeric(errors='coerce') para convertirla. Las celdas que no se pudieron

transformar quedaron como NaN, que básicamente indica que no hay dato. Como esas filas ya no eran útiles, las eliminé. Todo esto se puede ver en el archivo de apple(que es mi analisis). Después de eso, revisé si había más valores vacíos. Por suerte, no encontré ninguno en el resto de las columnas, así que no fue neceseria una limpieza tan grande.

En cuanto a los datos atípicos, decidí no eliminarlos. Me tomé el tiempo de analizar si realmente afectaban al modelo y vi que no eran muchos (aprox menos del 1% de los datos), por lo que no se me hacia un problema importante. Además, al comparar la media y la mediana de algunas columnas, noté que eran bastante similares, lo cual sugiere que no había grandes distorsiones.

Si bien habian algunos valores extremos, especialmente en el peso o el tamaño, no los consideré errores. Al contrario, me parecieron relevantes porque representan casos reales. Una manzana gigante o una muy chiquita existe, y me pareció importante que el modelo también pudiera aprender de esos ejemplos. Si los hubiera eliminado, la red no habría podido reconocer esas situaciones poco comunes, lo que haria que no sea tan precisa.

Para verlo en acción hice un boxplot de las últimas características clasificatorias:



Transformaciones Preliminares

Antes de entrenar la red neuronal, le hice algunas transformaciones a los datos para que el modelo pudiera interpretarlos correctamente. Estas modificaciones ayudaron a que todas las variables estuvieran en un formato adecuado y en condiciones similares para el análisis.

Estandarización: poner todas las variables en la misma escala

En lugar de aplicar una normalización Min-Max (que ajusta los valores entre 0 y 1), opté por estandarizar las variables numéricas, como el peso, el tamaño o la acidez. Basicamente que cada variable sea transformada para tener una media de 0 y una desviación estándar de 1.

Esta decisión se basó en el hecho de que algunas variables, como el peso, presentan valores numéricamente más altos que otras, como la acidez. De no ajustar estas diferencias, el modelo podia interpretar erróneamente que ciertas variables son más importantes que otras, simplemente por tener números más grandes. La estandarización me solucionaba este problema al poner todas las variables en la misma escala, lo que permite un aprendizaje más “equilibrado“ por parte de la red.

Además, consideré que la estandarización era más apropiada que la normalización en este caso, ya que los datos no siguen una distribución uniforme ni tienen límites claros.

Codificación binaria: transformar texto en valores numéricos

La variable "Calidad" (Quality), que indicaba si una manzana era “buena” o “mala”, estaba originalmente en formato de texto. Como los modelos de aprendizaje automático no pueden procesar texto directamente, convertí esta variable a un formato numérico. Para ello, creé una nueva columna llamada Quality_numeric, asignando el valor 1 a las manzanas “buenas” y 0 a las “malas”. Esta codificación **binaria** permite que la red neuronal entienda claramente qué debe predecir.

Reorganización del DataFrame: estructura clara para el modelado

Finalmente, reorganicé las columnas del conjunto de datos para ubicar la variable objetivo (Quality_numeric) al final. Esto la verdad que lo hice para que sea mas facil la visualización de las variables predictoras y de la etiqueta a

predecir, especialmente durante las etapas de entrenamiento y evaluación del modelo.

Parte 2: Desarrollo de la Red Neuronal

Arquitectura de la Red

Diseñé e implementé una red neuronal desde cero con numpy, compuesta por:

Capa de entrada:

La red comienza con una capa de entrada compuesta por 7 neuronas, una por cada característica numérica de las manzanas. Estas variables, como el peso, tamaño, color, dureza, acidez, entre otras, fueron previamente estandarizadas, y cada una alimenta directamente a una neurona de entrada. De esta forma, se asegura que toda la información del dataset esté disponible desde el inicio del procesamiento.

Capa oculta:

Hay una capa oculta con 16 neuronas, donde ocurre el procesamiento más complejo. Esta capa intermedia tiene como función aprender patrones y relaciones entre las variables de entrada. Elegí utilizar 16 neuronas como un punto medio, ya que me parecio una cantidad suficiente como para captar relaciones sutiles entre los datos (por ejemplo, la relación entre color y acidez en manzanas de buena calidad), pero no tan grande como para que el modelo corra el riesgo de overfitting. Probé con distintas y esta cantidad me dio muy buenos resultados.

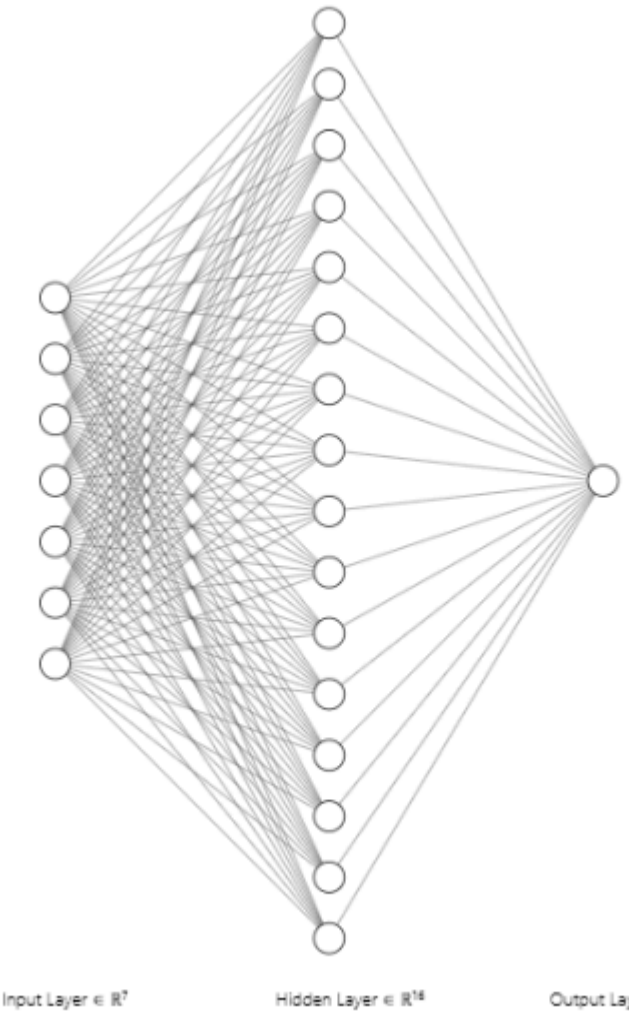
En esta capa empleé la función de activación ReLU. Esto me permitiio una propagación de señales más correcta y evito ciertos problemas asociados con funciones más antiguas, como la sigmoide en capas intermedias (por ejemplo, el desvanecimiento del gradiente).

Capa de salida: la predicción final

La red finaliza con una única neurona de salida, encargada de emitir la predicción final del modelo. Esta neurona utiliza la función de activación sigmoide, ya que se trata de un problema de clasificación binaria: la manzana puede ser “buena” (1) o “mala” (0).Esta funcion transforma la salida de la neurona en un valor entre 0 y 1, lo que puede interpretarse como una probabilidad. Por ejemplo, si el valor es cercano a 1, la red considera que la

manzana es probablemente de buena calidad; si es cercano a 0, indica una mayor probabilidad de que sea de mala calidad.

Aca dejo un grafico de la red.



En general, consideré esta arquitectura como equilibrada: ya que es lo suficientemente compleja como para aprender patrones en los datos, pero no tan pesada quizas, como para un entrenamiento manual con NumPy.

Implementación con NumPy

Implementé toda la red desde cero, incluyendo:

1. Inicialización de pesos y biases: Con el fin de garantizar la reproducibilidad, utilicé np.random.seed(). Los pesos comenzaron con

un pequeño valor con el fin de que no se sobrecargen.

2. Propagación hacia adelante:

   $Z1 = X \cdot W1 + b1$

   $A1 = ReLU(Z1)$

   $Z2 = A1 \cdot W2 + b2$

   $A2 = sigmoide(Z2)$

3. Función de costo: Use entropía cruzada binaria, ya que es la más adecuada para problemas de clasificación binaria.

4. Backpropagation: Implementé el algoritmo para poder calcular los gradientes de la función de costo respecto a cada peso y sesgo, y actualizarlos utilizando SGD.

5. Entrenamiento: Entrené el modelo durante 500 épocas (equivalentes a 50,000 iteraciones con un tamaño de batch de 32). En cada época registré la pérdida y la precisión para hacer un buen seguimiento de como iba aprendiendo.

Durante el entrenamiento noté una mejora notable constante en la precisión, lo cual me indicaba que la red cada vez aprendia mejor. Además, pude observar la curva de pérdida descendiendo de forma progresiva.

Evaluación del Modelo

Los parámetros son fundamentales, ya que determinan la velocidad y la efectividad con la que la red aprende de los datos. Ahora explico los parámetros más importantes que utilicé:

Número de Iteraciones (Epochs):

El número de iteraciones, lo fijé en **50.000**. Esto significa que la red vería y procesaría el conjunto de datos de entrenamiento 50.000 veces. Me parecio suficiente para aprender correctamente los patrones que relacionan las características de las manzanas con su calidad. En las primeras iteraciones, el modelo aprende rápido, pero con el tiempo, el progreso se hace más lento, por lo que se requiere seguir entrenando para que los ajustes sean más finos y

precisos. Elegí 50.000 iteraciones porque quería asegurarme de que la red tuviera suficiente tiempo para captar todos los detalles.

Tamaño del Batch (Batch Size):

Establecí un tamaño de batch de 32, lo que significa que en cada iteración la red no procesaba todo el conjunto de datos de entrenamiento de una sola vez, sino que procesaba grupos pequeños de 32 manzanas a la vez. Esto, permite que la red realice ajustes más estables y eficientes a medida que aprende.

Un tamaño de batch más pequeño es ventajoso porque evita que los ajustes sean demasiado abruptos, lo que podría llevar a un mal aprendizaje. Además, procesar datos en pequeños lotes me ayuda a que la red se mantenga equilibrada. De este modo, la red puede generalizar mejor, adaptándose a diferentes tipos de manzanas.

Tasa de Aprendizaje (Learning Rate):

Definí la tasa de aprendizaje en 0.01, que es un parámetro clave para ajustar el ritmo de corrección de los errores en cada iteración. Con esta, logré que la red aprendiera a un ritmo razonable, ni demasiado rápido ni demasiado lento, permitiendo que llegara a una solución óptima sin volverse inestable ni excesivamente lento.

Análisis de Overfitting

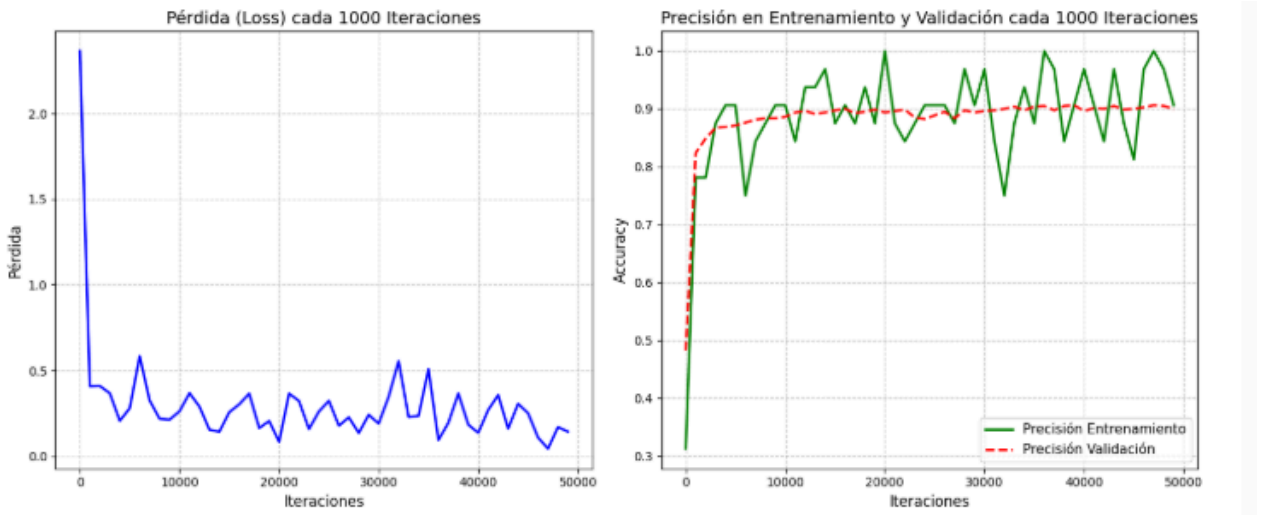
Para detectar overfitting:

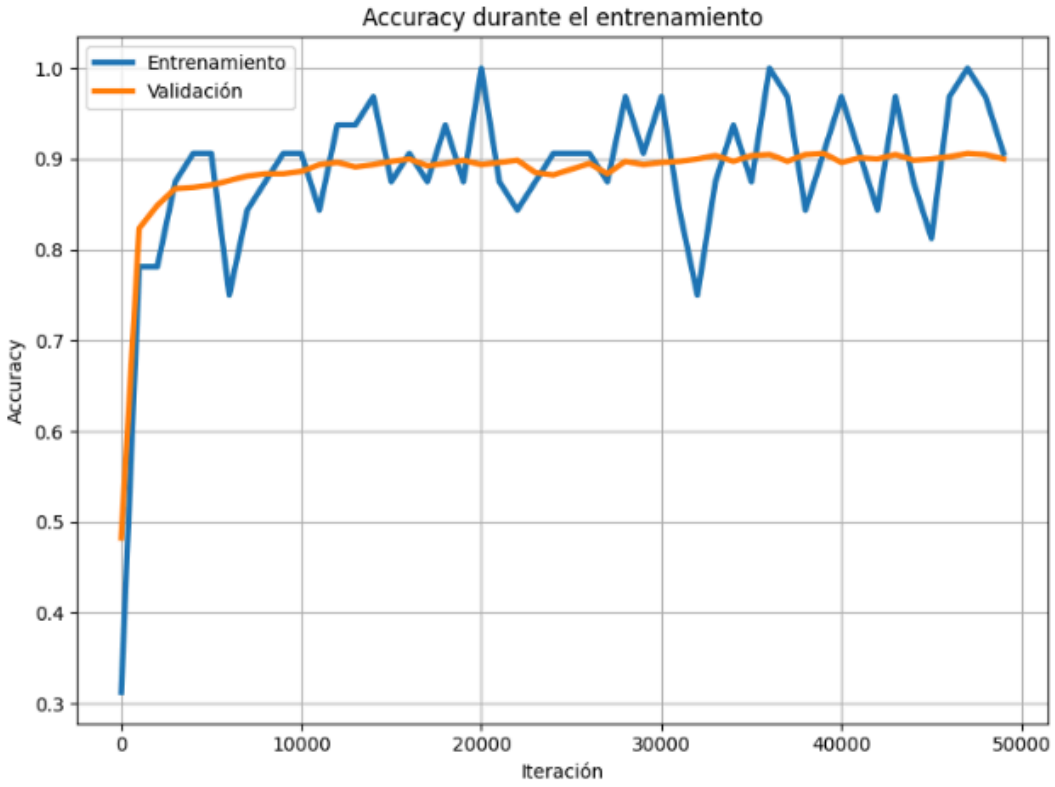
- Separé un 20% del dataset para lo que es validación.
- Y a partir de la iteración 40,000, el accuracy de validación se estancó mientras el de entrenamiento siguió subiendo, un leve overfitting.

Esto si que surgiere un leve sobreajuste. Aunque el efecto no fue dramático, podría mejorarse aplicando técnicas como:

- Early stopping: para detener el entrenamiento antes de que ocurra el sobreajuste.
- Regularización L2: me serviria para penalizar pesos excesivamente grandes.
- Dropout:en el caso d que usara una librería más avanzada.

Aca se puede ver cómo se comportaron la precisión y la pérdida durante el entrenamiento:





Parte 3: Comparación con Scikit-learn

Implementación con Scikit-learn

Utilicé el modelo MLPClassifier de la librería scikit-learn replicando la arquitectura:

from sklearn.neural_network import MLPClassifier

model = MLPClassifier(hidden_layer_sizes=(16,), activation='relu', max_iter=500, solver='sgd', random_state=42)

Esta implementación fue bastante más rápida y sencilla. No tarde tanto como lo hice al codear para hacer numpy. En cuanto a la precisión, la red neuronal manual implementada con NumPy alcanzó un 91.25% en el conjunto de entrenamiento y del 90.00% en el de validación.

Por otro lado, la red que fue entrenada con Scikit-learn (usando MLPClassifier) logró una precisión del 92.94% en entrenamiento y del 91.87% en validación, y supero levemente a la red manual.

En lo que es tiempo de ejecución, la diferencia fue significativa: la red manual tardó aproximadamente 23.11 segundos en entrenarse, mientras que la red de Scikit-learn completó su entrenamiento en solo 1.67 segundos.

Respecto a la flexibilidad, el enfoque con NumPy ofrece un mayor control sobre cada componente de la red, lo que permite personalizaciones detalladas y un entendimiento profundo del funcionamiento interno. Sin embargo, al mismo tiempo implica mayor complejidad. En cambio, Scikit-learn es menos flexible, pero mucho más sencilla de utilizar.

Finalmente, en cuanto a la facilidad de uso, la red manual requirió mas tiempo de trabajo, ajustes y depuración, mientras que la implementación con Scikit-learn se realizó extremadamente mas rapido.

Ambos modelos ofrecieron buenos resultados, pero con diferencias bastantes notables en esfuerzo, velocidad y precisión.

Conclusiones de la Comparación

- La red de scikit-learn es más rápida y precisa, gracias a optimizaciones internas.
- La red manual me permitió aprender cómo funcionan los pesos, las funciones de activación y el backpropagation paso a paso.
- Si bien para aplicaciones reales es práctico usar librerías, implementar una red desde cero es una experiencia muy formativa.

Parte 4: Conclusión Final

Este trabajo me ayudó a entender bien cómo funciona una red neuronal. Hacerla desde cero con NumPy fue un poco nose si dificil, pero siempre cuesta poner en practica de una todo lo que se vio en clase, senti que me sirvió mucho porque vi cómo se hace todo paso a paso.

Después, cuando usé Scikit-learn, fue mucho más fácil, pero porque ya entendía lo que estaba pasando gracias a lo que había hecho antes. Pude aprender sobre:

- La importancia de cada capa y función de activación.
- Cómo se ajustan los pesos para minimizar el error.
- Cómo evaluar el rendimiento de un modelo.

Comparar con Scikit-learn también me hizo dar cuenta de lo útiles que son las herramientas que ya existen.Por otro lado, ahora entiendo mucho mejor qué es lo que pasa "por detrás" cuando uso una.

Me voy con la sensación de haber hecho un proyecto completo, y pasar de trabajar con los datos hasta tener el modelo funcionando, y aprendiendo tanto la parte teórica como la práctica. La verdad que me ayudó a entender mucho mejor todo lo relacionado con el aprendizaje automático y las redes neuronales.