Taller 5

Johana Tellez - Michael Caicedo

1. Con base en el programa Clase_NNA3, desarrolle dos redes neuronales con 2 capas oculta, una que aprenda a reconocer una compuerta NAND y la otra aprenda a reconocer una compuerta XOR.

Se guardó el código en el documento “Primer_punto.ipynb”. El programa define y entrena dos redes neuronales con **dos capas ocultas** (W1/b1, W2/b2, W3/b3). Cada red toma entradas de 2 dimensiones (las entradas lógicas X1, X2), pasa por dos capas ocultas con activación sigmoide y produce una salida (una probabilidad entre 0 y 1) que representa la salida de la puerta lógica (NAND o XOR). El entrenamiento usa **descenso de gradiente** por retropropagación y minimiza el error cuadrático medio (MSE) sobre los 4 ejemplos de la tabla de verdad.
El código genera los siguientes resultados: Cada resultado es un número entre 0 y 1 (salida de la sigmoide). Para interpretar como lógica booleana aplica un umbral: si `y_pred >= 0.5` → 1 (True), si `< 0.5` → 0 (False). Por ejemplo, para NAND esperamos algo cercano a `[1, 1, 1, 0]` para las entradas `[[0,0],[0,1],[1,0],[1,1]]`. Si ves valores como `0.98, 0.99, 0.97, 0.03` la red está bien entrenada. Para XOR queremos `~[0,1,1,0]`. Si alguna predicción está en un rango intermedio (por ejemplo 0.4–0.6), la red no terminó de aprender o los hiperparámetros requieren ajuste.

```
Entrenando red NAND...
Iter 0, Error: 0.2178
Iter 2000, Error: 0.0735
Iter 4000, Error: 0.0031
Iter 6000, Error: 0.0012
Iter 8000, Error: 0.0007

Resultados NAND:
[0 0] -> 0.998
[0 1] -> 0.981
[1 0] -> 0.981
[1 1] -> 0.036
```

Durante el entrenamiento de la red neuronal para la compuerta NAND, lo primero que se observa es la evolución del **error** en distintas iteraciones. En la iteración 0, con los pesos inicializados de manera aleatoria, el error comienza en 0.2178, lo cual indica que la red no sabe aún cómo relacionar las entradas con las salidas esperadas. A medida que pasan las iteraciones, ese error va disminuyendo: en la iteración 2000 ya se reduce a 0.0735, en la 4000 baja a 0.0031, en la 6000 a 0.0012 y finalmente en la 8000 alcanza 0.0007. Esta reducción progresiva demuestra cómo el algoritmo de retropropagación va ajustando los pesos para que la red se acerque cada vez más al comportamiento correcto de la compuerta lógica.

Cuando se evalúan los resultados finales con cada posible combinación de entradas, se observa que para `[0 0]` la salida de la red es 0.998, muy cercana al valor esperado de 1.

Para `[0 1]` y `[1 0]`, la red produce 0.981 en ambos casos, que también se aproxima mucho al valor correcto de 1. Finalmente, para `[1 1]` la salida calculada es 0.036, que aunque no es exactamente 0, está muy próxima y se interpreta como tal.

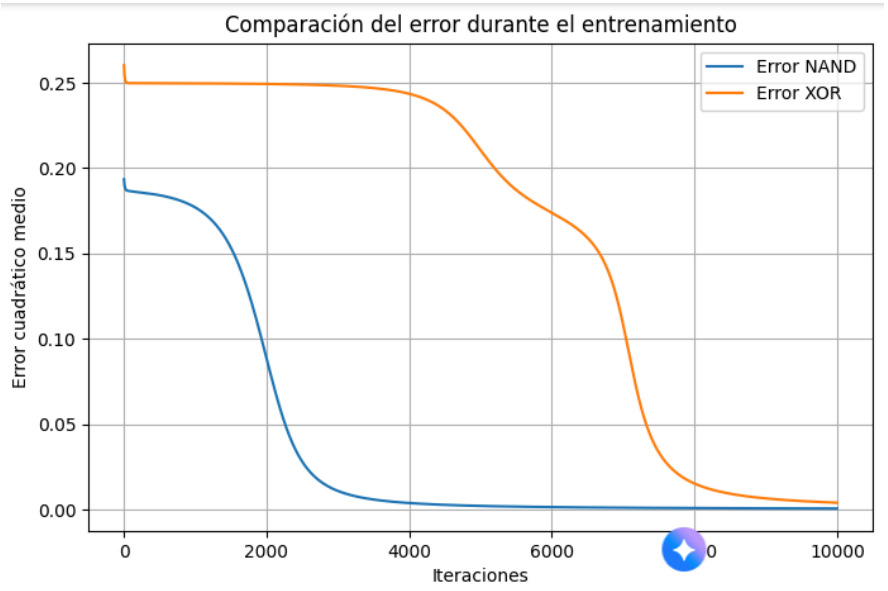
De la misma manera se interpretan los resultados para el entrenamiento de la red neuronal para la compuerta XOR

```
Entrenando red XOR...
Iter 0, Error: 0.2559
Iter 2000, Error: 0.2497
Iter 4000, Error: 0.2469
Iter 6000, Error: 0.1694
Iter 8000, Error: 0.0037
Iter 10000, Error: 0.0012
Iter 12000, Error: 0.0007
Iter 14000, Error: 0.0005

Resultados XOR:
[0 0] -> 0.020
[0 1] -> 0.980
[1 0] -> 0.980
[1 1] -> 0.020
```

Lo importante de estos resultados es que reflejan que la red logró **aprender el comportamiento de las compuertas**. Si se aplica un umbral simple (por ejemplo, considerar cualquier valor mayor o igual a 0.5 como 1 y menor que 0.5 como 0), entonces se puede configurar para que las predicciones se aproximen cada vez más al resultado esperado.

Al final también se muestra una gráfica del error calculado como la diferencia entre el valor obtenido y el valor esperado de cada iteración, se puede observar como disminuye el error tanto para el XOR como para el NAND en cada iteración.



2. Con base en la librería tensorflow, descargue el data set fashion MNIST. Haga una clasificación de prendas de vestir. Explique cada una de las funciones y las principales instrucciones, saque conclusiones

En este proyecto se trabajará con la **plataforma Google Colab**, un entorno gratuito en la nube que no requiere instalación local y que permite ejecutar código en Python directamente desde el navegador. Una de sus principales ventajas es que facilita el uso de librerías de **Machine Learning** como **Keras** y **TensorFlow**, ampliamente empleadas para el desarrollo de modelos de redes neuronales.

El objetivo principal es construir un **modelo de red neuronal artificial** capaz de reconocer diferentes tipos de prendas de vestir utilizando la base de datos **Fashion MNIST**. Este conjunto de datos contiene **60.000 imágenes de entrenamiento** y **10.000 de prueba**, todas en escala de grises, con un tamaño de **28 × 28 píxeles**. Las imágenes están clasificadas en **10 categorías**, cada una correspondiente a un tipo de prenda como camisetas, pantalones, zapatos, entre otros.

Antes de entrenar el modelo, es necesario realizar la **preparación de los datos**. En este caso, se aplica la **normalización** de los valores de los píxeles dividiéndolos entre 255, de modo que queden en un rango entre 0 y 1. Este paso es fundamental, ya que mejora el rendimiento del modelo y facilita el proceso de entrenamiento.

El modelo que se construirá es de tipo **secuencial**, lo que significa que las capas están organizadas una tras otra:

- **Primera capa (Flatten):** transforma cada imagen de 28×28 en un vector unidimensional de 784 valores, que servirá como entrada a la red.

- **Segunda capa (oculta):** compuesta por 10 neuronas que utilizan la función de activación **ReLU**, la cual permite que la red aprenda patrones relevantes en los datos.

- **Tercera capa (de salida):** con 10 neuronas, una por cada clase de prenda, que utilizan la función de activación **Softmax** para generar un vector de probabilidades y clasificar la imagen en la categoría con mayor probabilidad.

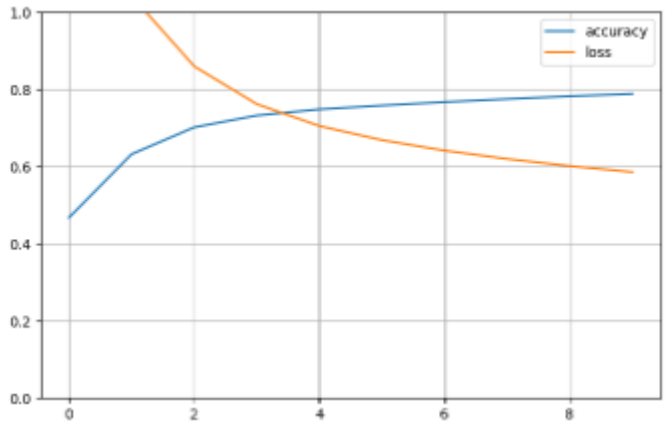
El **entrenamiento del modelo** consiste en ajustar los pesos de la red neuronal a través de varios ciclos (épocas). Para lograrlo, se emplean tres elementos clave:

- Una **función de pérdida**, que mide el error entre la predicción del modelo y el valor real.

- Un **optimizador**, que ajusta los pesos para reducir el error.

- Una **métrica de precisión (accuracy)**, que indica el porcentaje de aciertos en las predicciones.

A continuación se muestra el **porcentaje de aciertos (accuracy) frente al error (loss)** a lo largo de las distintas iteraciones del entrenamiento.



Durante este proceso, se realiza la **propagación hacia adelante** (forward propagation), donde el modelo calcula predicciones, y la **retropropagación** (backpropagation), en la cual se ajustan los parámetros para mejorar el desempeño. Este procedimiento se repite por varias épocas hasta que el modelo alcanza una buena capacidad de clasificación.

Resultados:

Con un porcentaje de error de **0.7846**, se puede observar que en la mayoría de las prendas el modelo alcanza una **tasa de aciertos superior al 90%**. Sin embargo, en otras categorías, como en las **zapatillas**, el modelo presenta desaciertos. Esto se debe a que los **zapatos comparten similitudes visuales entre sí y tienden a solaparse con otras clases**. Prueba de ello es que, mientras el modelo identifica correctamente

sneaker-sneaker con un 97% de precisión, también muestra confusión al clasificar **sneaker-sandalia con un 21%**, lo que evidencia la dificultad que tiene para diferenciar entre calzados con características similares.

3. Un data set de cualquier tipo, puede ser de https://www.kaggle.com/datasets, estudie sus características (features) y su rótulo. Diseñe una red neuronal y haga ejemplos con base en los pesos aprendidos

En este proyecto se trabajó con la plataforma **Google Colab**, un entorno gratuito en la nube que no requiere instalación local y que permite ejecutar código en Python directamente desde el navegador. Una de sus principales ventajas es que facilita el uso de librerías de **Machine Learning** como **Keras** y **TensorFlow**, ampliamente empleadas para el desarrollo de modelos de redes neuronales.

El objetivo principal fue construir un **modelo de red neuronal artificial** capaz de reconocer diferentes tipos de flores utilizando la base de datos **TF Flowers**. Este conjunto de datos contiene más de 18.000 imágenes distribuidas en 5 categorías: **daisy (margarita), dandelion (diente de león), roses (rosas), sunflowers (girasoles) y tulips (tulipanes)**. Las imágenes son a color y de diferentes dimensiones, por lo que fue necesario redimensionarlas a **64 × 64 píxeles** antes de utilizarlas en el entrenamiento.

Antes de entrenar el modelo, se realizó la **normalización de los valores de los píxeles** dividiéndolos entre 255, de modo que quedaran en un rango entre 0 y 1. Este paso es fundamental, ya que mejora el rendimiento del modelo y facilita el proceso de entrenamiento.

El modelo implementado fue de tipo **secuencial**, con las siguientes capas:

1. **Capa de entrada (Flatten):** transforma cada imagen de 64×64×3 en un vector unidimensional de 12.288 valores, que sirve como entrada a la red.

2. **Capa oculta:** compuesta por 128 neuronas que utilizan la función de activación **ReLU**, la cual permite que la red aprenda patrones relevantes en los datos.

3. **Capa oculta:** compuesta por 64 neuronas con activación **ReLU**.

4. **Capa de salida:** con 5 neuronas, una por cada clase de flor, que utilizan la función de activación **Softmax** para generar un vector de probabilidades y clasificar la imagen en la categoría con mayor probabilidad

El entrenamiento del modelo consistió en ajustar los pesos de la red neuronal a lo largo de varias **épocas (iteraciones)**. Para lograrlo se emplearon tres elementos clave:

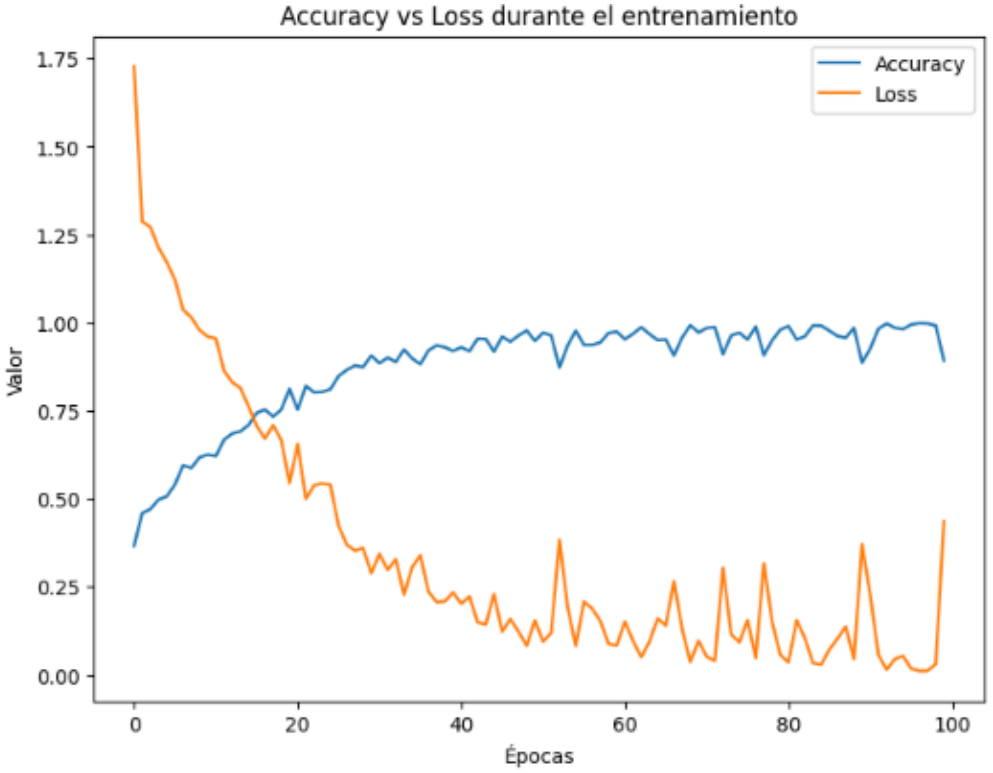
- **Función de pérdida:** *sparse categorical crossentropy*, que mide el error entre la predicción del modelo y el valor real.

- **Optimizador:** *Adam*, que ajusta los pesos de manera eficiente para reducir el error.

- **Métrica de precisión (accuracy):** indica el porcentaje de aciertos en las predicciones.

Durante este proceso, se aplicó **propagación hacia adelante (forward propagation)**, donde el modelo genera predicciones, y **retropropagación (backpropagation)**, en la cual se ajustan los parámetros de la red. Este procedimiento se repitió durante varias épocas hasta alcanzar una capacidad adecuada de clasificación.

Se graficaron las curvas de **accuracy** y **loss**, lo que permitió observar el comportamiento del modelo y comprobar que, a medida que aumentaba el número de iteraciones, mejoraba el desempeño.



Resultados

Tras el entrenamiento, el modelo alcanzó un **porcentaje de precisión aceptable** en la clasificación de las flores. No obstante, debido a que la red estaba compuesta únicamente por **tres capas densas**, su capacidad de representación era limitada frente a la complejidad de las imágenes a color.

Como consecuencia, el modelo **necesitó aproximadamente 100 iteraciones** para obtener resultados satisfactorios. Esto evidencia que, aunque fue capaz de aprender patrones generales y clasificar correctamente varias imágenes, una arquitectura más profunda y con capas convolucionales permitiría un mejor rendimiento y una convergencia más rápida.

```
Clase real: daisy
Clase predicha: daisy
```