Implementación de un Motor de Redes Neuronales Modular con Optimización Heurística

RADOSŁAWA ŻUKOWSKA[1,*] AND ÓSCAR RICO RODRÍGUEZ[2,*]

[*] Universidad de Las Palmas de Gran Canaria, Grado en Ciencia e Ingeniería de Datos
[1] radoslawa.zukowska101@alu.ulpgc.es
[2] oscar.rico101@alu.ulpgc.es
Compiled January 10, 2025

Este proyecto se centra en la creación de un motor de redes neuronales diseñado para optimizar el rendimiento sin depender de bibliotecas avanzadas de aprendizaje profundo. Partiendo de una red neuronal implementada desde cero, exploramos algoritmos de optimización heurística como el Descenso por el Gradiente (GD) y Adam para mejorar la eficiencia en el ajuste de los parámetros del modelo. Nuestra implementación se enfocó en crear una arquitectura modular que facilite la reutilización y personalización de componentes clave, tales como las funciones de activación, la función de pérdida y el propio modelo de red neuronal. Para evaluar el rendimiento de esta red, utilizamos el conjunto de datos MNIST Fashion para clasificación, y Caliornia Housing para regresión, dividiendo los datos en subconjuntos de entrenamiento, validación y prueba. Este enfoque permite verificar la capacidad del motor para generalizar y optimizar parámetros de una manera robusta, sin la necesidad de herramientas externas complejas.

1. INTRODUCCIÓN

En los últimos años, el uso de redes neuronales se ha incrementado de manera exponencial, impulsado en gran medida por bibliotecas avanzadas que facilitan su implementación. Sin embargo, comprender los fundamentos de estos modelos resulta fundamental para la investigación y la enseñanza de los principios básicos del aprendizaje profundo. Con esta motivación en mente, en este proyecto se desarrolló un motor de redes neuronales totalmente desde cero, sin depender de herramientas externas, a fin de explorar en detalle los componentes esenciales y ofrecer un entorno modular que permita personalizar y ajustar distintos elementos de la red.

Para lograrlo, se implementaron capas como *DenseLayer*, *FlattenLayer* y *ConvLayer*, acompañadas de diversas funciones de activación (*ReLU*, *Softmax*, *Linear*, *Softplus*, *Tanh*, *Sigmoid*) y funciones de pérdida específicas (*Entropía Cruzada*, *Mean Squared Error [MSE]*). Asimismo, se integraron optimizadores clave como *Adam* y *Descenso por el Gradiente* para realizar el ajuste eficiente de los parámetros. Esta variedad de componentes garantiza la flexibilidad del motor y la posibilidad de configurar la red según diferentes objetivos de aprendizaje.

Adicionalmente, se incluyeron características como la normalización de los datos, el entrenamiento en *batches* para optimizar tanto la estabilidad como el rendimiento del proceso de aprendizaje y la técnica de *early stopping* para prevenir el sobreajuste (overfitting). Para evaluar la eficacia de estas implementaciones, se realizaron experimentos de clasificación y regresión con diferentes conjuntos de datos. En clasificación, se emplearon el *accuracy score* y la *matriz de confusión* para medir el desempeño del modelo, mientras que en regresión se recurrió al coeficiente R^2 y *MSE*.

La memoria se organiza en varias secciones que describen progresivamente el proyecto: comenzando por los objetivos, se continúa con la metodología empleada y los conjuntos de datos utilizados, para luego detallar la arquitectura y cada uno de los componentes implementados. Posteriormente, se presentan los experimentos realizados y sus resultados, finalizando con las conclusiones extraídas y las posibles líneas de trabajo futuro. Esta estructura permite no solo mostrar los logros obtenidos, sino también reflexionar sobre los retos enfrentados y las áreas de mejora.

Nota sobre las funciones de activación: Para evitar extender innecesariamente la memoria, los experimentos se centraron únicamente en la función de activación *ReLU*, a pesar de haberse implementado varias alternativas. De esta manera, se facilita tanto la comparación de resultados como la claridad en la exposición de los mismos.

2. OBJETIVOS

Los principales objetivos de este proyecto son:

1. Desarrollar un motor de redes neuronales modulares desde cero, sin depender de bibliotecas avanzadas de aprendizaje profundo.
2. Proporcionar una estructura flexible que permita personalizar y adaptar componentes clave, como funciones de activación y optimizadores.
3. Evaluar la eficacia de algoritmos de optimización como GD y Adam en términos de rendimiento y capacidad de generalización.

Para la segunda entrega los objetivos son especialmente:

1. Desarrollar una Red Neuronal Convolucional (CNN).
2. Introducir procesamiento por lotes (batches) para entrenamiento.
3. Implementar *Early stopping*.
4. Usar una red neuronal para la tarea de regresión.

3. METODOLOGÍA

A. Arquitectura de la Red Neuronal Densa

La red neuronal desarrollada es una red neuronal fully connected, o completamente conectada, lo que significa que cada neurona de una capa está conectada a todas las neuronas de la siguiente capa. Este tipo de red permite capturar relaciones complejas entre las entradas y salidas del modelo, y su estructura modular permite una implementación flexible donde cada componente puede ajustarse o reemplazarse según las necesidades.

La arquitectura de esta red incluye varias capas densas (dense layers), que son las capas en las que cada neurona recibe la información de todas las neuronas de la capa anterior. Estas capas están distribuidas de la siguiente manera:

1. Capa de Entrada (input layer): La primera capa de la red, encargada de recibir las características o variables de entrada. Su tamaño depende del número de atributos o dimensiones en los datos de entrada (véase Fig. 1).

2. Capas Ocultas (hidden layer): La red incluye una o varias capas ocultas, que son capas intermedias entre la entrada y la salida. Cada capa oculta en esta arquitectura está formada por neuronas completamente conectadas y usa funciones de activación, como la ReLU (Rectified Linear Unit), que introduce no linealidad en el modelo. Estas funciones de activación son fundamentales para que la red pueda aprender patrones complejos (véase Fig. 1).

3. Capa de Salida (output layer): La última capa de la red produce el resultado final del modelo. El número de neuronas en la capa de salida depende de la tarea de la red (véase Fig. 1):

   (a) Para tareas de clasificación, puede haber una única neurona (para problemas binarios) o varias neuronas con una función de activación softmax (para clasificación multiclase).

   (b) Para tareas de regresión, la capa de salida suele tener una neurona con una activación lineal.

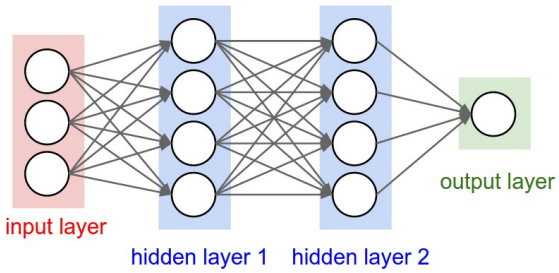

Fig. 1. Fully Connected Neural Network. [2]

B. Arquitectura de la Red Convolucional

Redes Convolucionales son un tipo de redes utilizadas para las imágenes en las que adicionalmente se utilizan capas convolucionales. Estas capas son capaces de capturar las características de las imágenes. Es importante tener en cuenta que, con diferencia de las redes densas, las convolucionales acceptan imágenes como entrada. No es necesario aplanarlas antes. También se pueden usar imágenes con muchas dimensiones, por ejemplo en colores.

La arquitectura de estas redes pueden ser complejas, con diferentes capas. Aquí implementamos su forma más básica y sencilla. Que consiste en las capas convolucionales, capas de aplanamiento y luego se finalizamos la red con capas densas, ya descritas anteriormente. (véase Fig. 1)

1. Capas Convolucionales: En estas capas se hace una operación de convolución. Que consiste en usar un filtro (kernel) que calcula una salida. Se pone un filtro en la entrada, lo multiplica por los valores correspondientes de la entrada, se añade un bias y se obtiene el valor de la matriz de salida, luego se mueve el filtro a un cierto paso y se repite esta operación hasta llegar al final. (véase Fig. 2)

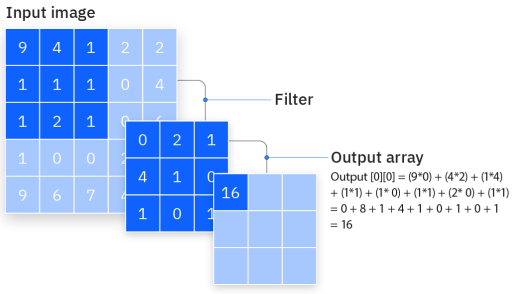

Fig. 2. Filtro en capa convolucional [3]

2. Capa de Aplanamiento (flatten): Puesto que en la salida de las capas convolucionales se obtienen las matrices, antes de procesarlas por capas densas (que requieren un vector como entrada) es necesario cambiar el formato de datos. Esta capa hace este aplanamiento. Con la diferencia no tienen pesos que aprender, ya que, su objetivo es sólo ajustar el formato de datos entre dos tipos de capas. (véase Fig. 3)

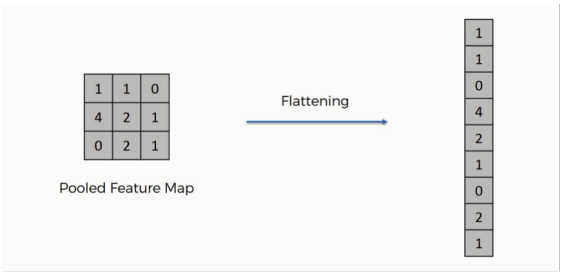

Fig. 3. Ejemplo de aplanamiento [5]

C. Forward Pass y Backpropagation

El proceso de entrenamiento de una red neuronal consta de dos etapas principales: Forward Pass y Backpropagation. Durante el Forward Pass, se calculan las salidas de cada capa a partir de las entradas iniciales, generando una predicción. A continuación, el algoritmo de Backpropagation se utiliza para ajustar los pesos de la red neuronal mediante el cálculo de los gradientes de la función de pérdida con respecto a cada parámetro de la red, lo que permite reducir el error de predicción en sucesivas iteraciones.

C.1. Forward Pass

La etapa de Forward Pass implica pasar los datos de entrada a través de la red, capa por capa, hasta obtener una predicción en la capa de salida. En cada capa, se aplica una transformación lineal mediante los pesos y sesgos correspondientes, seguida de una función de activación no lineal (como ReLU, sigmoid o tanh). Formalmente, para una capa l, el proceso de Forward Pass se puede describir de la siguiente manera:

Algorithm 1. Forward Pass con Múltiples Capas

procedure FORWARD(X)
  for layer in layers **do** ▷ Iterar sobre cada capa en la red
    $X \leftarrow$ layer.forward(X) ▷ Propagar la entrada X a través de la capa actual
  return X ▷ Retornar la salida de la última capa

Este pseudocódigo (véase Algorithm. 1) representa el proceso de forward para nuestra red neuronal con múltiples capas (self.layers). La entrada X se propaga sucesivamente a través de cada capa en layers mediante la llamada a layer.forward(X). La salida de una capa se convierte en la entrada de la siguiente, y al final se retorna la salida de la última capa.

Dentro de layer ocurre lo siguiente (véase Algorithm. 2):

Algorithm 2. Forward Pass para DenseLayer

procedure FORWARD(x)
  input $\leftarrow x$ ▷ Guardar la entrada para backpropagation
  $z \leftarrow x \cdot \text{weights} + \text{biases}$ ▷ Calcular salida lineal
  output $\leftarrow$ activation.forward(z) ▷ Aplicar función de activación a la salida lineal
  return output ▷ Retornar la salida activada

1. Entrada y Almacenamiento: La entrada x se almacena en self.input para usarla posteriormente en la retropropagación.
2. Cálculo de la Salida Lineal: Se calcula una salida lineal z a partir de x, los pesos de la capa (weights) y los sesgos (biases) mediante la operación $z = x \cdot weights + biases$.
3. Aplicación de la Activación: La función de activación (especificada por activation) se aplica a z para obtener la salida activada output, que es la salida final de esta capa.
4. Retorno de la Salida: Finalmente, se retorna output, que es la salida activada de la capa.

Este pseudocódigo (véase Algorithm. 3) representa el proceso de forward pass para una capa convolucional (ConvLayer):

Algorithm 3. Forward Pass para ConvLayer

procedure FORWARD(x)
  input $\leftarrow x$ ▷ Guardar la entrada para backpropagation
  $z \leftarrow$ signal.correlate2d($x, kernels$) ▷ Aplicar los filtros (convolucion)
  output $\leftarrow z +$ biases ▷ Sumar con biases
  return output ▷ Retornar la salida activada

C.2. Cálculo del error

La etapa de cálculo del error implica comparar las predicciones generadas en el *forward pass* con los valores reales para obtener la pérdida que indica qué tan precisa es la red en sus predicciones. En este caso, para problemas de clasificación se utiliza la *Cross Entropy Loss*, mientras que para problemas de regresión se emplea la *Mean Squared Error* (**MSE**).

Cross Entropy Loss o Entropía Cruzada

La entropía cruzada es común en problemas de clasificación. El cálculo del error se basa en la diferencia entre las probabilidades predichas (y_{pred}) y las etiquetas verdaderas (y_{true}). Para propósitos de *backpropagation*, calculamos el gradiente de la pérdida de entropía cruzada con respecto a las predicciones. Este gradiente será usado para actualizar los pesos durante el *backpropagation*. El cálculo se describe a continuación:

Algorithm 4. Gradiente de la Pérdida de Entropía Cruzada

procedure CROSSENTROPYLOSSGRAD($y_{\text{true}}, y_{\text{pred}}$)
  $\epsilon \leftarrow 1 \times 10^{-10}$ ▷ Pequeño valor para evitar división por cero
  $y_{\text{pred}} \leftarrow \text{clip}(y_{\text{pred}}, \epsilon, 1-\epsilon)$ ▷ Limitar valores de y_{pred} entre ϵ y $1-\epsilon$
  grad $\leftarrow \frac{y_{\text{pred}} - y_{\text{true}}}{y_{\text{pred}} \cdot (1 - y_{\text{pred}})}$ ▷ Calcular el gradiente de la pérdida
  return grad / y_{true}.shape[0] ▷ Retornar gradiente promedio por tamaño de *batch*

Este pseudocódigo (véase Algorithm. 4) calcula el gradiente de la pérdida de entropía cruzada. El gradiente obtenido indica cómo ajustar las predicciones para reducir la pérdida, proporcionando la base para ajustar los pesos de la red en *backpropagation*.

Mean Squared Error (MSE)

Para problemas de regresión, se utiliza la *Mean Squared Error* (**MSE**), definida como:

$$\text{MSE}(y_{\text{true}}, y_{\text{pred}}) = \frac{1}{N} \sum_{i=1}^{N} (y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)})^2$$

donde N es el número de muestras, $y_{\text{true}}^{(i)}$ son los valores reales y $y_{\text{pred}}^{(i)}$ las predicciones del modelo. A continuación, se muestra el cálculo de su gradiente con respecto a las predicciones:

Algorithm 5. Gradiente de la Pérdida de Error Cuadrático Medio (MSE)

procedure MSELOSSGRAD($y_{\text{true}}, y_{\text{pred}}$)
  $error \leftarrow y_{\text{pred}} - y_{\text{true}}$
  grad $\leftarrow \frac{2 \times error}{y_{\text{true}}.\text{shape}[0]}$
  return grad

Este pseudocódigo (véase Algorithm. 5) muestra como se utiliza el gradiente de manera análoga al de entropía cruzada para llevar a cabo la retropropagación en redes orientadas a tareas de regresión, indicando la dirección en la que deben ajustarse los parámetros del modelo para minimizar el *MSE*.

C.3. Backpropagation

Una vez que se ha obtenido la salida a través del Forward Pass y se ha calculado la pérdida, el siguiente paso es ajustar los parámetros de la red para minimizar este error, utilizando el algoritmo de Backpropagation. Este algoritmo calcula gradientes de los parámetros mediante el método backward de la clase 'NeuronalNetwork', basándose en la derivada parcial de la función de pérdida con respecto a cada parámetro.

El proceso de Backpropagation comienza en la última capa de la red (la capa de salida) y se propaga hacia atrás, capa por capa, hasta llegar a la capa inicial.

Algorithm 6. Backward Pass con Múltiples Capas

```
procedure BACKWARD(grad_output, layers)
    for layer in reversed(layers) do                ▷ Iterar sobre las capas en orden inverso
        grad_output ← layer.backward(grad_output)   ▷ Propagar el gradiente hacia atrás a través de la capa actual
```

Este pseudocódigo (véase Algorithm. 6) representa el proceso de backward pass para una red neuronal con múltiples capas. Durante el backward pass:

1. Gradiente de Salida: El gradiente grad_output que se recibe en el backward pass es el gradiente de la pérdida con respecto a la salida de la última capa.
2. Iteración en Orden Inverso: Se recorren las capas de la red en orden inverso (reversed(layers)) para propagar los gradientes hacia atrás, desde la última capa hasta la primera.
3. Propagación del Gradiente: Para cada capa, se llama a su método backward, pasando el gradiente actual grad_output. La salida de este método se convierte en el nuevo grad_output que se pasa a la capa anterior en la red.
4. Gradiente Final: Al final del bucle, grad_output contiene el gradiente ajustado que corresponde a la entrada de la red.

Algorithm 7. Backpropagation para DenseLayer

```
procedure BACKPROPAGATION(input, weights, biases, output, grad_output, activation)
    grad_output ← activation.backward(output, grad_output)   ▷ Aplicar derivada de la activación
    grad_weights ← input^T · grad_output                     ▷ Calcular gradiente de los pesos
    grad_biases ← ∑ grad_output (a lo largo de la dimensión batch)   ▷ Calcular gradiente de los biases
    return grad_output · weights^T                           ▷ Retornar gradiente de la entrada para la capa anterior
```

Este pseudocódigo (véase Algorithm. 7) representa el proceso de backpropagation para una capa densa (DenseLayer). Durante este paso:

1. Aplicación de la Derivada de la Activación: La derivada de la función de activación se aplica a grad_output, lo que ajusta el gradiente para la activación específica.
2. Gradiente de los Pesos: El gradiente con respecto a los pesos se calcula multiplicando la entrada transpuesta por grad_output.
3. Gradiente de los Biases: El gradiente con respecto a los biases se obtiene sumando grad_output sobre la dimensión de batch.
4. Gradiente para la Capa Anterior: Finalmente, se calcula el gradiente de entrada para esta capa, que será usado en la capa anterior durante la retropropagación. Este gradiente es el producto de grad_output con la transposición de los pesos, y es el valor retornado.

Algorithm 8. Backward Pass for ConvLayer

```
procedure BACKWARD(grad_output, layers)
    input_gradient ← zeros
    biases_gradient ← zeros
    for i ← 0 to self.depth − 1 do
        for j ← 0 to self.input_depth − 1 do
            kernels_gradient[i, j] ← signal.correlate2d(self.input[j], grad_output[i], "valid")
            input_gradient[j] ← input_gradient[j] + signal.correlate2d(grad_output[i], self.kernels[i, j], "full")
        biases_gradient[i] ← biases_gradient[i] + grad_output[b, i]
    return input_gradient
```

Este pseudocódigo (véase Algorithm. 8) representa el proceso de backpropagation para una capa convolutiva (ConvLayer). Durante este paso:

1. Inicialización de gradientes: Se inicializa gradientes de kernels y biases a zeros.
2. Convolución: Se hace convolución de input con gradient output de la capa para obtener gradientes de los kernels.
3. Gradientes de biases: Se obtienen al sumar los gradientes de biases con gradientes de la salida.

D. Optimizadores

Para optimizar los parámetros de la red neuronal, implementamos dos métodos:

D.1. GD

Actualiza los parámetros mediante la tasa de aprendizaje fija. Se utiliza como una línea de base para evaluar la efectividad del modelo.

Algorithm 9. Actualización de Parámetros con GD

procedure GD(layer, grad, learning_rate)
  for param in PARAMS **do** ▷ Iterar sobre cada parámetro en la capa
    new_value $\leftarrow$ $layer_{param}$ − learning_rate × $grad_{param}$ ▷ Calcular nuevo valor restando el gradiente escalado por la tasa de aprendizaje
    $layer_{param} = new\_value$ ▷ Actualizar el parámetro con el nuevo valor calculado

Este pseudocódigo (véase Algorithm. 9) representa el proceso de actualización de parámetros de una capa con GD. Durante este paso:

1. Iteración sobre los Parámetros: Para cada parámetro en PARAMS (que puede incluir pesos, biases, etc.), el optimizador realiza una actualización.
2. Cálculo de la Nueva Valor del Parámetro: Para cada parámetro param, se calcula su nuevo valor restando el gradiente ($grad_{param}$) multiplicado por la tasa de aprendizaje (learning_rate).
3. Actualización del Parámetro: Se establece el nuevo valor calculado en el atributo correspondiente de la capa layer.
4. Este proceso permite ajustar cada parámetro de la capa en la dirección que minimiza la pérdida, utilizando el gradiente y la tasa de aprendizaje.

D.2. Adam

Utiliza la primera y segunda derivadas acumuladas para ajustar dinámicamente la tasa de aprendizaje.

Algorithm 10. Actualización de Parámetros con Adam

procedure ADAM(layer, grad, t)
  if layer $\notin$ m **then**
    m[layer] $\leftarrow$ {param : zeros_like(getattr(layer, param)) for each param in PARAMS}
    v[layer] $\leftarrow$ {param : zeros_like(getattr(layer, param)) for each param in PARAMS}
  for param in PARAMS **do**
    m[layer][param] $\leftarrow \beta_1 \cdot$ m[layer][param] $+ (1 - \beta_1) \cdot$ grad[param] ▷ Actualizar primer momento (promedio móvil de gradientes)
    v[layer][param] $\leftarrow \beta_2 \cdot$ v[layer][param] $+ (1 - \beta_2) \cdot$ (grad[param])^2 ▷ Actualizar segundo momento (promedio móvil de gradientes al cuadrado)
    $\hat{m} \leftarrow \frac{\text{m[layer][param]}}{1-\beta_1^t}$ ▷ Calcular estimación sesgada corregida para el primer momento
    $\hat{v} \leftarrow \frac{\text{v[layer][param]}}{1-\beta_2^t}$ ▷ Calcular estimación sesgada corregida para el segundo momento
    new_value $\leftarrow$ getattr(layer, param) $- \frac{\text{learning\_rate} \cdot \hat{m}}{\sqrt{\hat{v}}+\epsilon}$ ▷ Calcular nuevo valor utilizando los momentos corregidos
    setattr(layer, param, new_value) ▷ Actualizar el parámetro con el nuevo valor calculado

Este pseudocódigo (véase Algorithm. 10) representa el proceso de actualización de parámetros para una capa utilizando el optimizador Adam. A continuación se detallan los pasos:

1. **Inicialización de Momentos**: Si la capa `layer` aún no tiene valores iniciales en los diccionarios de momentos m y v, se inicializan a ceros para cada parámetro en `PARAMS`.
2. **Actualización de Momentos**:
   - **Primer Momento (m)**: Se calcula el promedio móvil del gradiente (primer momento) como:
   $$m[\text{layer}][\text{param}] = \beta_1 \cdot m[\text{layer}][\text{param}] + (1 - \beta_1) \cdot \text{grad}[\text{param}]$$
   - **Segundo Momento (v)**: Se calcula el promedio móvil del cuadrado del gradiente (segundo momento) como:
   $$v[\text{layer}][\text{param}] = \beta_2 \cdot v[\text{layer}][\text{param}] + (1 - \beta_2) \cdot (\text{grad}[\text{param}])^2$$
3. **Corrección del Sesgo**:
   - **Primer Momento Corregido ($\hat{m}$)**: El primer momento m se ajusta para eliminar el sesgo inicial:
   $$\hat{m} = \frac{m[\text{layer}][\text{param}]}{1 - \beta_1^t}$$

- **Segundo Momento Corregido ($\hat{v}$)**: El segundo momento v se ajusta de manera similar:

$$\hat{v} = \frac{v[\text{layer}][\text{param}]}{1 - \beta_2^t}$$

4. **Actualización del Parámetro**: Usando los momentos corregidos, el parámetro `param` se actualiza como:

$$\text{new\_value} = \text{getattr}(\text{layer}, \text{param}) - \frac{\text{learning\_rate} \cdot \hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

Luego, el nuevo valor se asigna al parámetro en la capa `layer`.

4. CONJUNTO DE DATOS UTILIZADO

Para poner a prueba el motor de redes neuronales, empleamos tres conjuntos de datos muy utilizados en el ámbito de aprendizaje automático: **Fashion MNIST**, para tareas de clasificación, y **California Housing**, para tareas de regresión, y **MNIST** para tareas de clasificación con redes convolucionales. Estas fuentes de datos ofrecen la posibilidad de evaluar el comportamiento del modelo en escenarios distintos, proporcionando un panorama amplio de sus capacidades.

A. MNIST

El conjunto de datos MNIST es uno de más populares conjuntos de datos utilizados para redes neuronales. Que consiste imágenes con una resolución de 28x28 pixeles en una dimensión. Los píxeles son en tonos de gris y tienen valores desde 0 hasta 255. Los imágenes muestran dígitos escritos a mano y tienen etiquetas de 0 a 9. Las imágenes se divide en tres subconjuntos en la misma manera que los otros conjuntos de datos (80% entrenamiento, 10%validación y 10% prueba).

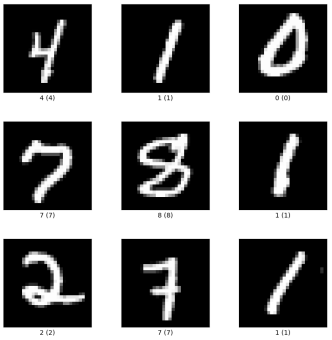

Fig. 4. Ejemplos de imágenes presentes en MNIST. [7]

B. Fashion MNIST

El conjunto de datos Fashion MNIST, diseñado como una alternativa más desafiante al MNIST original, contiene imágenes de artículos de ropa en una resolución de 28x28 píxeles. Cada imagen se convierte en un vector de 784 valores para su procesamiento en la red neuronal. Este conjunto de datos cuenta con 10 categorías de productos (camisetas, pantalones, zapatos, etc.) y se divide en subconjuntos de entrenamiento (80%), validación (10%) y prueba (10%). Fashion MNIST sirve para evaluar la capacidad de la red en tareas de clasificación de imágenes más complejas y menos estructuradas que las de dígitos.

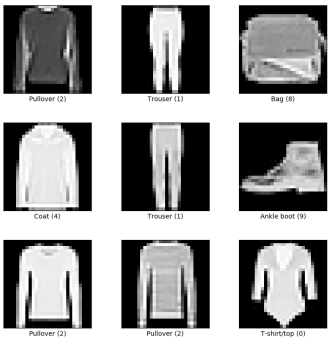

Fig. 5. Ejemplos de imágenes presentes en Fashion MNIST. [8]

C. California Housing

El conjunto de datos California Housing se centra en la estimación del valor de las viviendas en distintas localidades de California. Cada muestra incluye características como la media de ingresos, el promedio de número de habitaciones, la antigüedad de las viviendas, entre otras variables que proporcionan información socioeconómica y demográfica. Para el entrenamiento y evaluación, se emplea la misma estrategia de división (80% entrenamiento, 10% validación y 10% prueba). Este conjunto permite evaluar el rendimiento de la red neuronal en tareas de regresión.

Table 1. Estadísticas descriptivas de algunas variables en el conjunto de datos California Housing [6]

Variable	Media	Desv. Est.	Mín.	Máx.
MedInc	3.87	1.91	0.50	14.00
HouseAge	28.64	12.58	1.00	52.00
AveRooms	5.43	2.47	0.80	141.00
AveBedrms	1.07	0.25	0.30	34.07
Population	1425.47	1132.46	3.00	35682.00
Latitude	35.07	2.14	32.54	41.95
Longitude	-119.57	2.00	-124.35	-114.31
MedianHouseValue	2.07	1.15	0.14	5.0

En la Tabla 1 se muestra un resumen estadístico de algunas de las principales características del conjunto California Housing. Con estos dos conjuntos de datos, combinamos tanto la tarea de clasificación de imágenes (con Fashion MNIST) como la de regresión (con California Housing), ofreciendo una base sólida para medir, ajustar y comparar el rendimiento del modelo de redes neuronales en diferentes ámbitos.

5. DETALLES DE IMPLEMENTACIÓN

La implementación modular se organiza en varias clases y archivos, cada uno de los cuales define un aspecto específico del modelo de red neuronal. Esta estructura modular permite la integración de diferentes capas, funciones de activación, optimizadores y métodos de evaluación en un solo flujo de entrenamiento. A continuación, se describen los componentes principales:

1. **NeuronalNetwork.py**: Define la clase principal `NeuronalNetwork`, que organiza el flujo de datos y el proceso de aprendizaje mediante retropropagación.
   - `NeuronalNetwork` inicializa la red neuronal mediante la configuración de capas especificadas en `layers_config`. Para cada capa, crea un objeto `DenseLayer` con el tamaño de entrada, el tamaño de salida y la función de activación correspondiente.
   - `forward`: Realiza el forward pass pasando los datos de entrada a través de cada capa secuencialmente.
   - `backward`: Ejecuta la retropropagación invocando el método `backward` de cada capa en orden inverso, calculando los gradientes necesarios para la actualización de los pesos.
2. **LossFunction.py**: Define funciones de pérdida y sus gradientes, necesarios para evaluar el error y ajustarlo en cada iteración.
   - `cross_entropy_loss`: Calcula la pérdida de entropía cruzada, que es ideal para problemas de clasificación. Recorta valores extremos de `y_pred` para evitar errores numéricos.
   - `cross_entropy_loss_grad`: Calcula el gradiente de la función de pérdida de entropía cruzada, necesario para la retropropagación. Este gradiente se utiliza en el *backward pass* para ajustar los parámetros del modelo en la dirección de minimización del error.
   - `mean_squared_error (MSE)`: Mide la diferencia cuadrática media entre las salidas predichas y las verdaderas, utilizada principalmente en problemas de regresión.
   - `mean_squared_error_grad`: Calcula el gradiente de la función de pérdida MSE, necesario para la retropropagación. Este gradiente se emplea para ajustar los parámetros del modelo en dirección a la minimización de la pérdida.
3. **DenseLayer.py**: Define la clase `DenseLayer`, que representa una capa completamente conectada (o densa) en la red.
   - `DenseLayer` inicializa los pesos de la capa y los sesgos, además de la función de activación.
   - `forward`: Calcula la salida de la capa mediante una transformación lineal seguida de la aplicación de una función de activación.
   - `backward`: Calcula los gradientes de los pesos y sesgos para la retropropagación. Devuelve el gradiente con respecto a la entrada de la capa para propagarlo hacia capas anteriores.
4. **ConvLayer.py**: Define la clase `ConvLayer`, que representa una capa de convolución en la red neuronal.
   - `ConvLayer` inicializa los kernels (filtros) y los sesgos necesarios para las operaciones de convolución.

- `forward`: Realiza operaciones de convolución en las entradas utilizando los filtros, añade los sesgos, y genera la salida de la capa.
- `backward`: Calcula los gradientes de los kernels y los sesgos para la retropropagación. También calcula el gradiente con respecto a las entradas para propagarlo hacia capas anteriores.

5. **FlattenLayer.py**: Define la clase `FlattenLayer`, que aplana una entrada multidimensional en una dimensión única para conectarla a capas densas.
   - `FlattenLayer` no tiene parámetros que aprender, pero guarda las formas de entrada y salida para realizar el aplanado correctamente.
   - `forward`: Convierte la entrada multidimensional en una representación de una sola dimensión, preservando el tamaño del lote.
   - `backward`: Restaura la entrada original desde el gradiente, devolviendo el gradiente con la forma original.
6. **ActivationFunction.py**: Define clases para funciones de activación como ReLU, Softmax, Linear, Softplus, Tanh y Sigmoid.
   - `ReLU`: Una función de activación común que introduce no linealidad al "activar" solo valores positivos.
   - `Softmax`: Convierte las salidas de la capa en probabilidades, típicamente usada en la última capa de clasificación. También se implementa el cálculo del gradiente de Softmax, que se utiliza en retropropagación.
   - `Linear`: Devuelve el valor de entrada sin cambios (función de activación lineal).
   - `Softplus`: Una versión "suave" de ReLU, definida como $\log(1 + e^x)$.
   - `Tanh`: Aplica la función tangente hiperbólica, con rango de salida entre $(-1, 1)$.
   - `Sigmoid`: Transforma la entrada en valores entre $(0, 1)$ mediante la función $\sigma(x) = \frac{1}{1+e^{-x}}$.
7. **Optimizer.py**: Define clases de optimización, las cuales aplican los gradientes calculados a los pesos de las capas.
   - `GDOptimizer`: Implementa el algoritmo de Descenso de Gradiente (GD) básico, actualizando cada parámetro según el gradiente y una tasa de aprendizaje fija.
   - `AdamOptimizer`: Implementa el optimizador Adam, que ajusta dinámicamente los parámetros utilizando promedios móviles de primer y segundo momento. Este optimizador permite una convergencia más rápida y estable en comparación con GD.
8. **Trainer.py**: Define la clase `Trainer`, que gestiona el proceso de entrenamiento de la red.
   - `Trainer` inicializa la red neuronal, el optimizador y las funciones de pérdida. Este módulo ejecuta el entrenamiento completo, iterando por cada época.
   - `train`: Entrena la red en múltiples épocas, almacenando la pérdida y la precisión en cada iteración. Evalúa el rendimiento en el conjunto de validación y muestra los resultados en intervalos de épocas especificados.
9. **Test.py**: Contiene funciones para la evaluación y visualización de resultados.
   - `predict`: Realiza predicciones para datos de entrada utilizando el modelo entrenado (función `nn.forward`).
   - `accuracy_evaluate`: Calcula la precisión de las predicciones en comparación con las etiquetas verdaderas, medido como la proporción de aciertos.
   - `mse_evaluate`: Calcula el error cuadrático medio (MSE) para casos de regresión o métricas adicionales en clasificación.
   - `confusion_matrix`: Genera la matriz de confusión para analizar el rendimiento de clasificación por clase. Incluye la opción de rotar las etiquetas en el eje `x` para mayor legibilidad.
   - `plot_training_history_with_validation`: Grafica de forma comparativa la pérdida (*loss*) y la precisión (*accuracy*) tanto en entrenamiento como en validación para monitorear sobreajuste.
   - `calculate_metrics`: Calcula métricas como precisión, *recall* y *F1-score* por clase, además de mostrar los valores *weighted average* sobre todas las clases.
   - `print_metrics`: Imprime la precisión global y una tabla con las métricas por clase (mediante `pandas.DataFrame`).
10. **NN.ipynb**: Este archivo de Jupyter Notebook es donde se carga y prepara el conjunto de datos y se realizan todas las pruebas de la red neuronal. Contiene las siguientes etapas clave:
    - **Carga de Datos**: Se cargan los conjuntos de datos de MNIST o Fashion MNIST, y se preprocesan las imágenes para convertirlas en vectores adecuados para la red neuronal.

- **Configuración del Modelo**: Se establece la configuración de la red (capas y funciones de activación) y se inicializa el optimizador.
- **Entrenamiento del Modelo**: Se entrena la red utilizando la clase `Trainer`, que organiza el proceso de aprendizaje y evalúa el rendimiento en los conjuntos de validación.
- **Evaluación del Modelo**: Se realizan pruebas en el conjunto de prueba utilizando las funciones de `Test.py`, incluyendo predicciones, matriz de confusión y cálculo de métricas de precisión, recall y F1-score.
- **Visualización de Resultados**: Se muestran gráficas de la evolución de la pérdida y precisión en el entrenamiento, así como resultados específicos del conjunto de prueba, proporcionando una visión completa del rendimiento de la red.

11. **CNN.ipynb**: Este archivo de Jupyter Notebook es donde se realiza la ejecución y entrenamiento de la red neuronal convolutiva. Este archivo es similar a NN.ipynb.

- **Carga de Datos:** Se cargan el conjunto MNIST y se preprocesan las imágenes en la misma manera con el otro archivo, sólo las de entrada (X) se pone luego en la forma de matriz 2D, no vectores.
- **Configuración de Modelo:** Se configura el modelo con una pequeña diferencia del otro, porque se pasa las capas directamente a la red, no como un diccionario de configuración.
- **Entrenamiento del Modelo**: Para entrenar se utiliza la clase `Trainer`. Sólo es que capas convolucionales se actualizan dentro de la funcion `ConvLayer.backward`, otras capas se actualiza con `Optimizer`.
- **Evaluación y visualización de Resultados:** Esto se hace en la misma manera con en el otro archivo.

Esta estructura modular organiza el flujo de datos y permite que cada componente funcione en conjunto de manera eficiente. Además, facilita ajustes y pruebas de diferentes configuraciones y algoritmos, proporcionando flexibilidad para investigar mejoras en el rendimiento del modelo.

6. EXPERIMENTOS

En esta sección se describen los experimentos realizados para evaluar el rendimiento del modelo en dos escenarios diferentes: clasificación de imágenes con *Fashion MNIST*, regresión con *California Housing*, y CNN con *MNIST*. Para cada configuración se registraron las métricas de pérdida y, cuando correspondía, la métrica de desempeño (precisión en clasificación y error cuadrático medio en regresión) tanto en entrenamiento como en validación, lo que permitió un análisis comparativo detallado. *Nota: Aunque se implementaron múltiples funciones de activación, en estos experimentos se optó por usar ReLU para simplificar su presentación.*

En el caso de **clasificación con Fashion MNIST**, se utilizó una arquitectura compuesta por:

$$512 \text{ (ReLU)} \rightarrow 256 \text{ (ReLU)} \rightarrow 10 \text{ (Softmax)},$$

Mientras que para la **regresión con California Housing** se emplearon las siguientes capas:

$$128 \text{ (ReLU)} \rightarrow 64 \text{ (ReLU)} \rightarrow 32 \text{ (ReLU)} \rightarrow 1 \text{ (Linear)}.$$

Además de la configuración de capas, se exploraron distintas estrategias de optimización (como *Adam* y *Gradient Descent*) y se incorporó la técnica de *early stopping* para controlar el sobreajuste. En las próximas secciones se presentan los resultados obtenidos bajo cada escenario, así como un análisis de los valores de pérdida y precisión/error que evidencian la capacidad de generalización del modelo.

A. Dataset: MNIST Fashion

A.1. Experimento #1: Optimizador GD

Table 2. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Test	Batch Size	Early Stopping
0.1	24	0.146	0.67	Full size	Yes
0.1	14	0.056	0.857	128	Yes
0.01	29	0.062	0.877	128	Yes

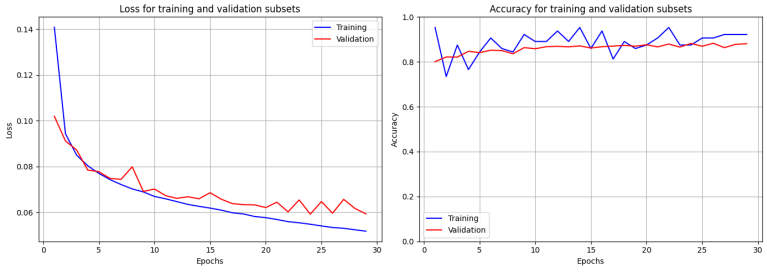

Fig. 6. Cambio del error y precision para subconjuntos de entrenamiento y validación para la red densa con tasa de aprendizaje de 0.01 y tamaño de lote 128

A.2. Experimento #2: Optimizador Adam

Table 3. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Test	Batch Size	Early Stopping
0.005	80	0.071	0.855	Full size	Yes
0.005	19	0.054	0.89	128	Yes
0.0005	31	0.052	0.895	128	Yes

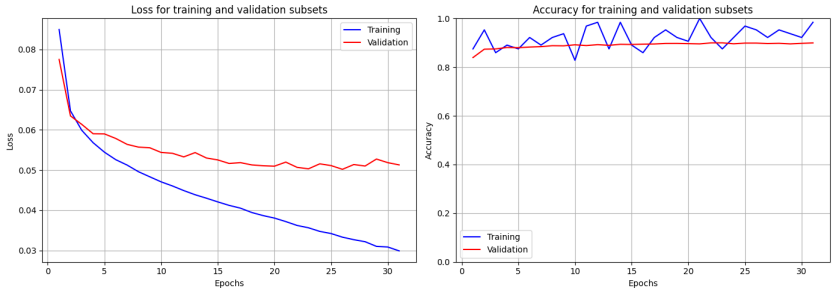

Fig. 7. Cambio del error y precision para subconjuntos de entrenamiento y validación para la red densa con tasa de aprendizaje de 0.0005 y tamaño de lote 128

B. Dataset: MNIST con CNN

Para estos experimentos se utilizó una arquitectura compuesta por:

$$\text{ConvLayer(kernel\_size=3,depth=5)} \rightarrow \text{FlattenLayer} \rightarrow 100\ \text{(ReLU)} \rightarrow 10\ \text{(Softmax)},$$

Table 4. Resultados de los experimentos con CNN usando optimizador GD

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Test	Batch Size	Early Stopping
0.001	20	0.021	0.963	32	No
0.001	20	0.029	0.951	64	No
0.005	20	0.017	0.969	64	No

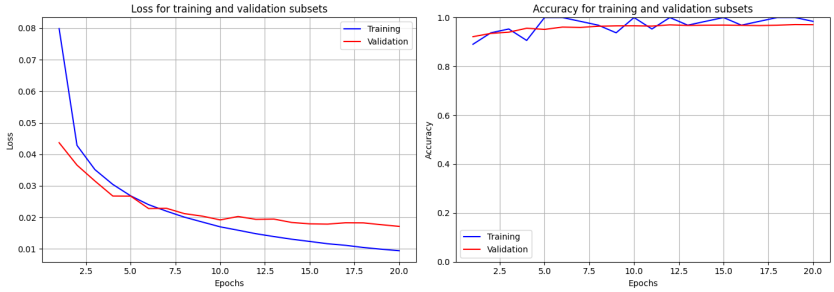

Fig. 8. Cambio del error y precision para subconjuntos de entrenamiento y validación para la red convolucional con tasa de aprendizaje de 0.005 y tamaño de lote 64

C. Dataset: California Housing

C.1. Experimento #1: Optimizador GD

Table 5. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	R^2	MSE	Batch Size	Early Stopping
0.01	11	0.33	2.35	Full size	Yes
0.01	7	0.7	2.39	128	Yes

C.2. Experimento #2: Optimizador Adam

Table 6. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	R^2	MSE	Batch Size	Early Stopping
0.01	9	0.3	2.45	Full size	Yes
0.01	8	0.75	2.36	128	Yes

7. RESULTADOS

A continuación se presenta un resumen de los principales hallazgos de cada uno de los experimentos, seguido de las conclusiones generales que se pueden extraer a partir de los resultados.

A. Clasificación con MNIST Fashion

Observaciones principales (véanse Table. 2 y Table. 3):

- Utilizar un *batch size* de 128 mejoró notablemente la precisión, en comparación con el entrenamiento usando el conjunto completo (*full batch*).
- Reducir la tasa de aprendizaje de 0.1 a 0.01 en GD permitió una convergencia más estable, alcanzando una mayor precisión de test (0.877).
- Con Adam, la mejor precisión (0.895) se obtuvo usando una tasa de aprendizaje de 0.0005 y *batch size* de 128.
- Frente a Gradient Descent, Adam alcanzó valores similares o mejores de precisión cuando la tasa de aprendizaje se eligió apropiadamente.

B. Clasificación con CNN con MNIST

Observaciones principales (véase Table. 4):

- La red convolucional se entrena correctamente con las imágenes de MNIST y da resultados muy buenos.
- El mejor resultado se logró con tasa de aprendizaje 0.005 y batch size de 64, alcanzado una precisió de 0.969 para el subconjunto de test.
- La red convolucional ya que no utilizamos GPU, toma más tiempo para aprender, una época puede durar entre 30-60s.

C. Regresión con California Housing

Observaciones principales (véanse Table. 5 y Table. 6):

- Un *batch size* menor (128) permitió un R^2 mucho mayor ([0.7 vs. 0.33],[0.75 vs. 0.3]), lo que indica un mejor ajuste del modelo a los datos.
- Pese a que el MSE en *batch=128* no mejoró significativamente respecto al *full batch*, el incremento en R^2 evidencia una mayor capacidad de explicar la varianza de los datos.
- El mejor resultado en regresión se logró con Adam, tasa de aprendizaje de 0.01 y *batch size* de 128, alcanzando un R^2=0.75 y un MSE=2.36.
- Comparado con Gradient Descent, Adam mostró una mejor capacidad de ajuste usando configuraciones apropiadas (en este caso, *batch size* de 128).

8. CONCLUSIONES

- **Importancia de la tasa de aprendizaje y *batch size*:**
  - En Fashion MNIST, tanto con GD como con Adam, reducir la tasa de aprendizaje y utilizar *batch sizes* moderados (128) permitió mejoras importantes en la métrica de desempeño (precisión).
  - Para California Housing, usar *batch size* de 128 también resultó más beneficioso que entrenar con el conjunto completo.
- **Rendimiento comparativo de los optimizadores:**
  - **GD vs. Adam en clasificación (Fashion MNIST):** Adam alcanzó mejores precisiones con menos épocas, sobre todo con tasas de aprendizaje bajas (0.0005).
  - **GD vs. Adam en regresión (California Housing):** El mejor R^2 (0.78) y MSE (2.31) se consiguieron con Adam, confirmando su efectividad para esta tarea.
- **Beneficio del *early stopping*:**
  - En todos los experimentos, el uso de *early stopping* ayudó a frenar el sobreajuste al detener el entrenamiento cuando no había mejoras en la pérdida de validación. Esto permitió tiempos de entrenamiento más cortos y modelos con mejor capacidad de generalización.
- **Arquitecturas profundas simples:**
  - Aun con arquitecturas de profundidad intermedia (2 y 3 capas ocultas), se obtuvo un rendimiento competitivo, demostrando que la elección del optimizador y la tasa de aprendizaje puede ser incluso más determinante que la complejidad del modelo.
- **Desempeño de CNNs en clasificación:**
  - La red convolucional aplicada a MNIST alcanzó una precisión notable del 96.9%, destacando la efectividad de las arquitecturas convolucionales para el reconocimiento de patrones en imágenes.
  - Aunque el entrenamiento fue más lento debido a la ausencia de GPU, los resultados muestran que configuraciones como una tasa de aprendizaje de 0.005 y un *batch size* de 64 son óptimas para este tipo de tareas.
  - El desempeño superior de la CNN refuerza su idoneidad para tareas de clasificación basadas en datos visuales, en comparación con arquitecturas densamente conectadas.

9. TRABAJO FUTURO

Las posibles extensiones de este proyecto incluyen:

1. **Implementación de arquitecturas CNN completas:** Para optimizar el rendimiento de las redes convolucionales (CNN), es posible incorporar capas adicionales como *Pooling* (por ejemplo, *MaxPooling* o *AveragePooling*) para reducir las dimensiones espaciales y resaltar características relevantes. Además, se pueden diseñar redes convolucionales con múltiples capas convolucionales apiladas, lo que permite capturar patrones más complejos y jerárquicos en los datos. Estas mejoras pueden contribuir significativamente a mejorar la precisión y la capacidad de generalización del modelo.
2. **Implementación de diferentes tipos de redes:** Más allá de las CNN, se pueden explorar otros tipos de arquitecturas como las redes neuronales recurrentes (RNN), que son especialmente útiles para trabajar con datos secuenciales o dependientes en el tiempo, como texto, series temporales o datos de audio.

REFERENCES

1. Optimization-and-Heuristics, *NN-Project*, Repositorio GitHub, 2024. [En línea]. Disponible en: https://github.com/Optimization-and-Heuristics/NN-Project. [Accedido: 10-jan-2025].
2. CS231n, *Convolutional Networks*. [En línea]. Disponible en: https://cs231n.github.io/convolutional-networks/. [Accedido: 10-jan-2025].
3. IBM, *Convolutional Neural Networks*. [En línea]. Disponible en: https://www.ibm.com/think/topics/convolutional-neural-networks. [Accedido: 10-jan-2025].
4. YouTube, "Convolutional Neural Networks" [video]. [En línea]. Disponible en: https://www.youtube.com/watch?v=Lakz2MoHy6o. [Accedido: 10-jan-2025].
5. SuperDataScience, "Convolutional Neural Networks (CNN) - Step 3: Flattening". [En línea]. Disponible en: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening. [Accedido: 10-jan-2025].
6. Inria, "Datasets California Housing", *Scikit-learn MOOC*. [En línea]. Disponible en: https://inria.github.io/scikit-learn-mooc/python_scripts/datasets_california_housing.html. [Accedido: 10-jan-2025].
7. Kaggle, "MNIST dataset". [En línea]. Disponible en: https://www.kaggle.com/datasets/hojjatk/mnist-dataset. [Accedido: 10-jan-2025].
8. Kaggle, "FashionMNIST". [En línea]. Disponible en: https://www.kaggle.com/datasets/zalando-research/fashionmnist. [Accedido: 10-jan-2025].