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 November 3, 2024

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, 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

El aprendizaje profundo ha avanzado de manera significativa, impulsado por herramientas que permiten construir redes neuronales complejas. Estas bibliotecas, aunque poderosas, pueden limitar la flexibilidad y la comprensión de los detalles internos de los modelos.

En este proyecto, hemos desarrollado un motor de redes neuronales desde cero, enfatizando en la implementación modular y en la optimización heurística. Con este enfoque buscamos reducir la dependencia de bibliotecas externas y promover un entendimiento profundo del funcionamiento de los modelos.

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.

3. METODOLOGÍA

A. Arquitectura de la Red Neuronal

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.

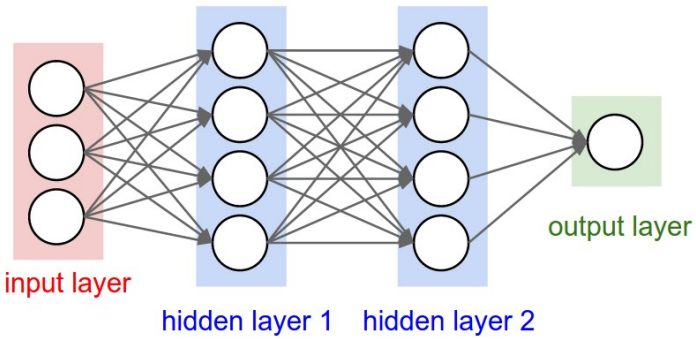

Fig. 1. Fully Connected Neuronal Network.

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: 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.

2. Capas Ocultas: 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.

3. Capa de Salida: 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:

   (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.

B. 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.

B.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 ← layer.forward(X)              ▷ Propagar la entrada X a través de la capa actual
    return X                              ▷ Retornar la salida de la última capa
```

Este pseudocódigo 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:

Algorithm 2. Forward Pass para DenseLayer

```
procedure FORWARD(x)
    input ← x                             ▷ Guardar la entrada para backpropagation
    z ← x · weights + biases              ▷ Calcular salida lineal
    output ← activation.forward(z)        ▷ Aplicar función de activación a la salida lineal
    return output                         ▷ Retornar la salida activada
```

Este pseudocódigo representa el proceso de forward pass para una capa densa (DenseLayer). Durante este paso:

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.

B.2. Cálculo del error

La etapa de cálculo del error implica comparar las predicciones generadas en el forward pass con las etiquetas reales para obtener el error (pérdida) que indica qué tan precisa es la red en sus predicciones. En este caso, se utiliza la pérdida de entropía cruzada, 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, también 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 3. 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 $1-\epsilon$
  $\text{grad} \leftarrow \frac{y_{\text{pred}} - y_{\text{true}}}{y_{\text{pred}} \cdot (1 - y_{\text{pred}})}$ ▷ Calcular el gradiente de la pérdida
  return $\text{grad} / y_{\text{true}}.\text{shape}[0]$ ▷ Retornar gradiente promedio por tamaño de batch

Este pseudocódigo 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.

B.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 4. 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 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 5. Backpropagation para DenseLayer

procedure BACKPROPAGATION(input, weights, biases, output, grad_output, activation)
  grad_output ← activation.backward(output, grad_output) ▷ Aplicar derivada de la activación
  $\text{grad\_weights} \leftarrow \text{input}^T \cdot \text{grad\_output}$ ▷ Calcular gradiente de los pesos
  $\text{grad\_biases} \leftarrow \sum \text{grad\_output}$ (a lo largo de la dimensión batch) ▷ Calcular gradiente de los biases
  return $\text{grad\_output} \cdot \text{weights}^T$ ▷ Retornar gradiente de la entrada para la capa anterior

Este pseudocódigo 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.

C. Optimizadores

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

C.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 6. 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 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.

C.2. Adam

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

Algorithm 7. Actualización de Parámetros con Adam

procedure ADAM(layer, grad, t)
  if layer $\notin$ m **then**
    m[layer] $\leftarrow$ {param : zeros_like(getattr(layer, param)) para cada param en PARAMS}
    v[layer] $\leftarrow$ {param : zeros_like(getattr(layer, param)) para cada param en PARAMS}
  for cada param en 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 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 probar el motor de redes neuronales, empleamos los conjuntos de datos MNIST y Fashion MNIST, ambos ampliamente utilizados en el ámbito de aprendizaje automático para evaluar el rendimiento de modelos en tareas de clasificación de imágenes.

A. MNIST

El conjunto de datos MNIST contiene imágenes de dígitos escritos a mano, representadas en una resolución de 28x28 píxeles. Cada imagen se convierte en un vector de 784 valores para su procesamiento en redes neuronales. Los datos se dividen en subconjuntos de entrenamiento (80%), validación (10%) y prueba (10%), permitiendo una evaluación precisa del rendimiento del modelo en distintos contextos.

B. Fashion MNIST

El conjunto de datos Fashion MNIST, diseñado como una alternativa más desafiante a MNIST, contiene imágenes de artículos de ropa en lugar de dígitos. Al igual que MNIST, cada imagen tiene una resolución de 28x28 píxeles, convirtiéndose también en un vector de 784 valores. Este conjunto de datos cuenta con las mismas divisiones en entrenamiento (80%), validación (10%) y prueba (10%), y presenta categorías como camisetas, pantalones, zapatos, entre otros. Fashion MNIST es útil para evaluar la capacidad del modelo en tareas de clasificación de imágenes más complejas y menos estructuradas que las de dígitos.

Ambos conjuntos de datos proporcionan una base sólida para comparar y ajustar el rendimiento del modelo de redes neuronales en tareas de reconocimiento de patrones.

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.

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. **ActivationFunction.py**: Define clases para funciones de activación como ReLU y Softmax.
   - `ReLU`: Una función de activación común que introduce no linealidad activando 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.
5. **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.
6. **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.
7. **Test.py**: Contiene funciones para la evaluación y visualización de resultados.
   - `predict`: Realiza predicciones para datos de entrada utilizando el modelo entrenado.
   - `evaluate`: Calcula la precisión de las predicciones en comparación con las etiquetas verdaderas.
   - `plot_training_history`: Grafica la evolución de la pérdida y la precisión a lo largo del entrenamiento.
   - `confusion_matrix`: Genera la matriz de confusión para analizar el rendimiento de clasificación por clase.
   - `plot_training_history_with_validation`: Permite comparar la pérdida y precisión de entrenamiento con los valores de validación, detectando posibles problemas de sobreajuste.
   - `calculate_metrics` y `print_metrics`: Calculan y muestran métricas como precisión, recall y F1-score, brindando un análisis detallado del rendimiento en cada clase.
8. **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.

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

Esta sección detalla los experimentos realizados para evaluar el rendimiento del modelo bajo diferentes configuraciones de entrenamiento y optimización. Cada experimento se registró con una métrica de pérdida y precisión en entrenamiento y validación, permitiendo un análisis comparativo.

A. Dataset: MNIST

A.1. Experimento #1: Ajuste de la Tasa de Aprendizaje GD

- **Objetivo**: Analizar cómo afecta la tasa de aprendizaje al rendimiento del modelo y su velocidad de convergencia.
- **Parámetros de Configuración**:
  - **Optimizador**: GD
  - **Tasas de Aprendizaje Probadas**: 0.1, 0.5, 1
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 1. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Validación
0.1	260	0.050	0.916
0.5	260	0.025	0.958
1.0	160	4.186	0.091

- **Observaciones**:
  - La tasa de aprendizaje de 1.0 provoca inestabilidad.
  - La tasa de aprendizaje de 0.5 muestra el mejor rendimiento en precisión y pérdida.
  - La tasa de aprendizaje de 0.1 es estable pero con convergencia más lenta.

A.2. Experimento #2: Ajuste de la Tasa de Aprendizaje Adam

- **Objetivo**: Analizar cómo afecta la tasa de aprendizaje al rendimiento del modelo y su velocidad de convergencia.
- **Parámetros de Configuración**:
  - **Optimizador**: Adam
  - **Tasas de Aprendizaje Probadas**: 0.0005, 0.001, 0.005, 0.01, 0.05
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 2. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Validación
0.0005	260	0.022	0.961
0.001	260	0.018	0.970
0.005	260	0.020	0.970
0.01	260	0.025	0.967
0.05	260	1.836	0.305

- **Observaciones**:
  - Tasa de aprendizaje óptima alrededor de 0.001 y 0.005.
  - Tasa de aprendizaje muy alta (0.05) resulta en inestabilidad.
  - Tasas de aprendizaje bajas ofrecen estabilidad pero pueden ralentizar la convergencia.

A.3. Experimento #3: Modificar estructura de la Red para el mejor modelo y learning rate

- **Objetivo**: Comparar el rendimiento de diferentes dimensiones de la capa oculta.
- **Parámetros de Configuración**:
  - **Optimizador**: Adam
  - **Tasa de Aprendizaje**: 0.005
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 3. Resultados de Comparación de Dimensiones de la Capa Oculta

Capas	Taza de Aprendizaje	Época	Pérdida de Validación	Precisión de Validación
{ReLU, Softmax}	0.005	260	0.020	0.970
{ReLU, ReLU, Softmax}	0.005	260	0.028	0.970

- **Observaciones**:
  - Precisión similar con diferentes configuraciones de capas ocultas.
  - Ligera diferencia en la pérdida de validación.
  - Recomendación de la estructura más simple para este caso.

B. Dataset: MNIST Fashion

B.1. Experimento #1: Ajuste de la Tasa de Aprendizaje GD

- **Objetivo**: Analizar cómo afecta la tasa de aprendizaje al rendimiento del modelo y su velocidad de convergencia.
- **Parámetros de Configuración**:
  - **Optimizador**: GD
  - **Tasas de Aprendizaje Probadas**: 0.01, 0.1, 0.5
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 4. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Validación
0.01	260	0.122	0.775
0.1	260	0.083	0.833
0.5	80	4.118	0.106

- **Observaciones**:
  - La tasa de aprendizaje de 0.5 provoca inestabilidad.
  - La tasa de aprendizaje de 0.1 muestra el mejor rendimiento en precisión y pérdida.
  - La tasa de aprendizaje de 0.01 es estable pero con convergencia más lenta.

B.2. Experimento #2: Ajuste de la Tasa de Aprendizaje Adam

- **Objetivo**: Analizar cómo afecta la tasa de aprendizaje al rendimiento del modelo y su velocidad de convergencia.
- **Parámetros de Configuración**:
  - **Optimizador**: Adam
  - **Tasas de Aprendizaje Probadas**: 0.0005, 0.001, 0.005, 0.01, 0.05
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 5. Resultados del Ajuste de Tasa de Aprendizaje

Tasa de Aprendizaje	Época	Pérdida de Validación	Precisión de Validación
0.0005	260	0.057	0.891
0.001	260	0.054	0.894
0.005	260	0.054	0.896
0.01	260	0.058	0.886
0.05	80	3.665	0.203

- **Observaciones**:
  - Tasa de aprendizaje óptima alrededor de 0.005.
  - Divergencia a tasas de aprendizaje altas (0.05)
  - Estabilidad y menor convergencia con tasas de aprendizaje bajas (0.0005 y 0.001).

B.3. Experimento #3: Modificar estructura de la Red para el mejor modelo y learning rate

- **Objetivo**: Comparar el rendimiento de diferentes dimensiones de la capa oculta.
- **Parámetros de Configuración**:
  - **Optimizador**: Adam
  - **Tasa de Aprendizaje**: 0.005
  - **Épocas de Entrenamiento**: 260
  - **Evaluación**: Cada 20 épocas

Table 6. Resultados de Comparación de Dimensiones de la Capa Oculta

Capas	Época	Pérdida de Validación	Precisión de Validación
{ReLU, Softmax}	260	0.054	0.896
{ReLU, ReLU, Softmax}	260	0.062	0.884

- **Observaciones**:
  - Mejor desempeño con la configuración de una sola capa oculta ReLU, Softmax.
  - Recomendación de la configuración más simple.

7. RESULTADOS

Los resultados de los experimentos se presentan por dataset, destacando los mejores resultados obtenidos para cada caso.

A. Resultados en MNIST

El modelo alcanzó los mejores resultados en el conjunto de datos MNIST utilizando el optimizador Adam con una tasa de aprendizaje de 0.005 y la función de activación ReLU en las capas ocultas.

- **Precisión Global en Conjunto de Prueba**: 0.971
- **Métricas por Clase**:

Table 7. Métricas de Precisión, Recall y F1-Score en MNIST

Clase	Precisión	Recall	F1-Score	Soporte
0	0.9861	0.9861	0.9861	432
1	0.9895	0.9853	0.9874	476
2	0.9691	0.9808	0.9749	416
3	0.9661	0.9596	0.9629	446
4	0.9659	0.9754	0.9707	407
5	0.9704	0.9652	0.9678	374
6	0.9690	0.9831	0.9760	413
7	0.9779	0.9693	0.9736	456
8	0.9672	0.9568	0.9620	370
9	0.9537	0.9537	0.9537	410
Promedio Ponderado	0.9719	0.9719	0.9719	4200

B. Resultados en Fashion MNIST

Para el conjunto de datos Fashion MNIST, el modelo obtuvo el mejor rendimiento también con Adam y una tasa de aprendizaje de 0.005. A continuación se presentan las métricas.

- **Precisión Global en Conjunto de Prueba**: 0.900
- **Métricas por Clase**:

Table 8. Métricas de Precisión, Recall y F1-Score en Fashion MNIST

Clase	Precisión	Recall	F1-Score	Soporte
Tshirt/top	0.8440	0.8773	0.8603	709
Trouser	0.9810	0.9753	0.9782	689
Pullover	0.7904	0.8746	0.8304	694
Dress	0.8775	0.9256	0.9009	712
Coat	0.8578	0.8042	0.8301	720
Sandal	0.9627	0.9642	0.9635	670
Shirt	0.7997	0.7034	0.7484	681
Sneaker	0.9406	0.9527	0.9466	698
Bag	0.9815	0.9691	0.9753	712
Ankle boot	0.9688	0.9538	0.9612	715
Promedio Ponderado	0.9004	0.9001	0.8996	7000

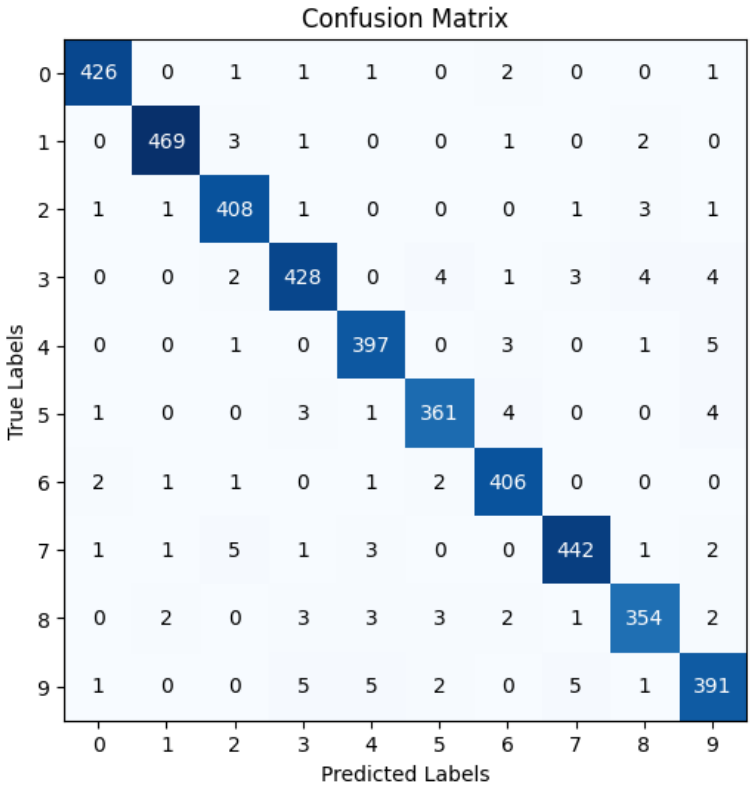

Fig. 2. Matriz de Confusión en el Conjunto de Prueba MNIST

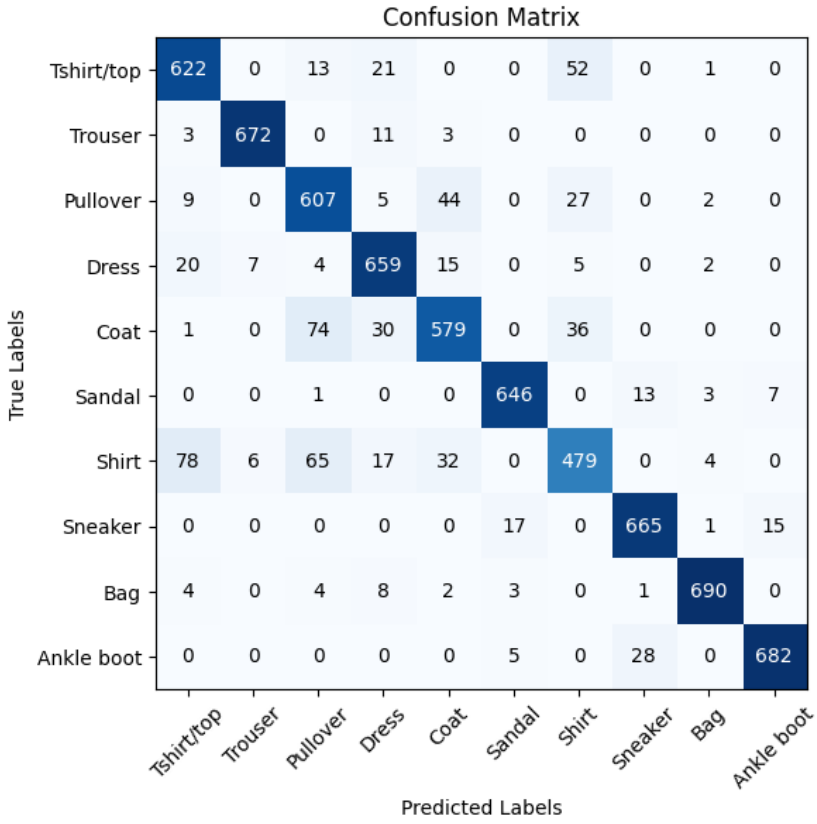

Fig. 3. Matriz de Confusión en el Conjunto de Prueba Fashion MNIST

8. CONCLUSIONES

A partir de los experimentos realizados, se obtienen las siguientes conclusiones clave:

1. Eficiencia del Optimizador Adam: A lo largo de los experimentos, el optimizador Adam demostró ser superior en términos de velocidad de convergencia y precisión final en comparación con el descenso de gradiente (GD). La capacidad de Adam para ajustar dinámicamente la tasa de aprendizaje mediante el uso de promedios móviles le permite adaptarse mejor a los gradientes y alcanzar una precisión elevada con estabilidad. Esto lo convierte en una opción ideal para tareas de clasificación en redes neuronales simples, como las exploradas en este proyecto.

2. Importancia Crítica de la Tasa de Aprendizaje: La tasa de aprendizaje desempeña un papel esencial en el rendimiento del modelo. Se encontró que una tasa de aprendizaje moderada (alrededor de 0.001 a 0.005) ofreció un equilibrio óptimo entre precisión y estabilidad, mientras que tasas más altas causaron inestabilidad y tasas más bajas resultaron en una convergencia lenta. Esto subraya la importancia de seleccionar cuidadosamente la tasa de aprendizaje al entrenar redes neuronales, especialmente en modelos implementados desde cero.

3. Beneficios de la Modularidad en la Implementación: La arquitectura modular de este motor de redes neuronales permite una fácil incorporación de componentes adicionales, como nuevas funciones de activación y optimizadores. Este enfoque flexible facilita la experimentación y permite analizar el impacto de cada componente en el rendimiento general del modelo. Además, la modularidad contribuye a mejorar la comprensión y el control del flujo de datos y del proceso de aprendizaje.

4. Versatilidad del Motor en Diferentes Conjuntos de Datos: El modelo fue evaluado en dos conjuntos de datos distintos (MNIST y Fashion MNIST), demostrando su capacidad de generalizar en tareas de clasificación de imágenes más allá de los dígitos. Esto refleja la robustez del modelo y su adaptabilidad para otros problemas de clasificación, con ajustes mínimos en la arquitectura y los hiperparámetros.

5. Resultados Sólidos con una Arquitectura Simple: La red neuronal de arquitectura simple, con una capa oculta y funciones de activación como ReLU y Softmax, logró buenos resultados en los conjuntos de datos probados. Esto sugiere que, en tareas de clasificación relativamente simples, no siempre es necesario recurrir a redes profundas o a configuraciones complejas para alcanzar un buen rendimiento.

Estas conclusiones reflejan los beneficios de una implementación personalizada y modular para explorar diferentes configuraciones y algoritmos de optimización. El motor desarrollado proporciona una base sólida para el entrenamiento de redes neuronales con un alto grado de control sobre cada etapa del aprendizaje.

9. TRABAJO FUTURO

Las posibles extensiones de este proyecto incluyen:

1. Implementación de Redes Neuronales Convolucionales (CNN): Para mejorar el rendimiento en la clasificación de imágenes, una extensión natural sería implementar redes neuronales convolucionales. Las CNNs están mejor adaptadas para procesar datos en formato de imagen, ya que capturan patrones espaciales y características locales de manera más efectiva. Esta extensión permitiría abordar tareas de clasificación más complejas y obtener mejores resultados en conjuntos de datos de mayor resolución.

2. Ampliación a Otros Conjuntos de Datos: Incluir nuevos conjuntos de datos en diferentes dominios podría enriquecer los experimentos y proporcionar un análisis más completo de la capacidad de generalización del modelo. Esto ayudaría a evaluar su rendimiento en problemas variados, como reconocimiento facial, detección de objetos o procesamiento de datos no visuales.

3. Implementación de Algoritmos de Optimización Adicionales: Además de Adam y GD, el proyecto podría ampliarse con la incorporación de otros algoritmos de optimización como RMSprop, AdaGrad y AdaDelta. Estos optimizadores podrían mejorar la eficiencia de convergencia y adaptarse mejor a diferentes conjuntos de datos y arquitecturas. Al comparar estos optimizadores, se obtendría una visión más completa de cómo cada uno afecta el rendimiento en distintas configuraciones.

Estas mejoras potenciales reflejan el camino para hacer que el motor de redes neuronales sea más robusto y versátil, ampliando su aplicabilidad en distintos ámbitos y habilitando la investigación en técnicas de aprendizaje profundo más avanzadas.

REFERENCES

1. Optimization-and-Heuristics, NN-Project, GitHub repository, 2024. [En línea]. Disponible en: https://github.com/Optimization-and-Heuristics/NN-Project. [Accedido: 03-nov-2024].
2. R. Sardana, Building Neural Network from Scratch, Towards Data Science, Medium, 2019. [En línea]. Disponible en: https://towardsdatascience.com/building-neural-network-from-scratch-9c88535bf8e9. [Accedido: 03-nov-2024].
3. W. Salmon, Simple MNIST NN from Scratch (Numpy, no TF/Keras), Kaggle, 2021. [En línea]. Disponible en: https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras. [Accedido: 03-nov-2024].