



Máster Universitario en Ingeniería Informática
de la Universidad de Granada

Práctica de redes neuronales

Reconocimiento óptimo de caracteres MNIST

Inteligencia Computacional (IC)

Autora

Marina Jun Carranza Sánchez

Índice

1. Introducción	4
1.1. Herramientas y entorno	4
1.2. Preparación de los datos	4
1.3. Metodología	5
2. Red neuronal simple	7
2.1. Fundamentos teóricos	7
2.2. Implementación	7
2.3. Análisis de resultados	8
3. Red neuronal multicapa	9
3.1. Fundamentos teóricos	9
3.2. Implementación	9
3.3. Análisis de resultados	10
4. Red neuronal convolucional	10
4.1. Fundamentos teóricos	11
4.2. Implementación	11
4.3. Análisis de resultados	12
5. Deep learning	13
5.1. Fundamentos teóricos	13
5.2. Implementación	14
5.3. Análisis de resultados	16
6. Red neuronal combinada mejorada	16
6.1. Fundamentos teóricos	17
6.2. Implementación	18
6.3. Análisis de resultados	20
7. Conclusiones	21
8. Bibliografía	22

Índice de figuras

1.	Funciones para cargar las imágenes de entrenamiento y prueba	4
2.	Funciones para cargar las etiquetas de los conjuntos de imágenes	5
3.	Preparación de los conjuntos de datos	5
4.	Fragmento de código para imprimir los resultados del modelo	6
5.	Modelado de la primera red neuronal simple	8
6.	Resultados de la primera red neuronal simple	8
7.	Modelado de la segunda red neuronal	10
8.	Resultados de la segunda red neuronal	10
9.	Modelado de la tercera red neuronal	12
10.	Resultados de la tercera red neuronal	12
11.	Modelado de la cuarta red neuronal (pt. 1)	14
12.	Modelado de la cuarta red neuronal (pt. 2)	15
13.	Modelado de la cuarta red neuronal (pt. 3)	16
14.	Resultados de la cuarta red neuronal	16
15.	Ejemplo del funcionamiento de los callbacks	18
16.	Modelado de la quinta red neuronal (pt. 1)	18
17.	Modelado de la quinta red neuronal (pt. 2)	19
18.	Modelado de la quinta red neuronal (pt. 3)	20
19.	Resultados de la cuarta red neuronal	20

Índice de cuadros

1. Introducción

En esta sección se comentan las herramientas y el entorno de desarrollo utilizado para la práctica, así como algunos detalles acerca de la preparación de los datos.

1.1. Herramientas y entorno

Se ha optado por utilizar bibliotecas disponibles públicamente, que implementan los algoritmos necesarios para la práctica. El entorno de desarrollo elegido ha sido un Jupyter Notebook en *Google Colab*, haciendo uso de las funcionalidades de *TensorFlow* y *Keras*.

Esta elección se debe a su acceso gratuito a potentes recursos de computación como GPUs y TPUs, lo que acelera el entrenamiento de modelos complejos. Además, Colab ofrece un entorno preconfigurado y una integración fluida con Google Drive, lo que simplifica el almacenamiento de los conjuntos de datos de las imágenes.

1.2. Preparación de los datos

El objetivo es la implementación de redes neuronales para la identificación (o clasificación) correcta de dígitos en el conjunto de imágenes MNIST.

Antes de nada, se han definido cuatro funciones: las dos primeras (Figura 1) se encargan de abrir los ficheros `train-images.idx3-ubyte` y `t10k-images.idx3-ubyte`, que contiene las imágenes para el entrenamiento y las pruebas respectivamente, mientras que las dos últimas (Figura 2) sirven para cargar las etiquetas correspondientes a los dos conjuntos de imágenes (`train-labels.idx1-ubyte` y `t10k-labels.idx1-ubyte`).

```
import numpy as np

def cargar_train_imgs():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/train-images.idx3-ubyte', 'rb') as f:
        f.read(16) # Descartar cabecera
        images = []
        while True:
            image = f.read(784)
            if len(image) != 784:
                break
            images.append([x for x in image])
        return np.array(images)

def cargar_test_imgs():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/t10k-images.idx3-ubyte', 'rb') as f:
        f.read(16) # Descartar cabecera
        images = []
        while True:
            image = f.read(784)
            if len(image) != 784:
                break
            images.append([x for x in image])
        return np.array(images)
```

Figura 1: Funciones para cargar las imágenes de entrenamiento y prueba

```
def cargar_train_labels():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/train-labels.idx1-ubyte', 'rb') as f:
        f.read(8) # Descartar cabecera
        etiquetas = [x for x in f.read()]
        return np.array(etiquetas)

def cargar_test_labels():
    with open('/content/drive/MyDrive/MÁSTER/IC/samples/t10k-labels.idx1-ubyte', 'rb') as f:
        f.read(8) # Descartar cabecera
        etiquetas = [x for x in f.read()]
        return np.array(etiquetas)
```

Figura 2: Funciones para cargar las etiquetas de los conjuntos de imágenes

Una vez definidas las funciones anteriores, se llaman para cargar los datos. A continuación, se preprocesan las imágenes de entrenamiento y prueba normalizándolas, dividiendo sus valores de píxel por 255.0, lo que escala los valores al rango [0, 1]. Las etiquetas de ambos conjuntos se convierten a formato categórico utilizando `to_categorical` con 10 clases, lo que transforma las etiquetas en vectores one-hot adecuados para la clasificación multiclase.

Finalmente, las imágenes de entrenamiento y prueba se reestructuran con la función `reshape` para tener dimensiones de 28x28 píxeles y un único canal (blanco y negro), preparando los datos en el formato requerido para ser ingresados en las redes neuronales. Estos pasos vienen implementados en la Figura 3.

```
# Cargar los datos
train_images = cargar_train_imgs()
train_labels = cargar_train_labels()
test_images = cargar_test_imgs()
test_labels = cargar_test_labels()

# Preprocesar los datos de entrenamiento (Normalizar y convertir etiquetas)
train_images = train_images / 255.0 # rango [0, 1]
train_labels = to_categorical(train_labels, 10)

# Preprocesar los datos de prueba (Normalizar y convertir etiquetas)
test_images = test_images / 255.0 # rango [0, 1]
test_labels = to_categorical(test_labels, 10)

train_images = train_images.reshape(-1, 28, 28, 1) # '1': el canal (blanco y negro)
test_images = test_images.reshape(-1, 28, 28, 1)
```

Figura 3: Preparación de los conjuntos de datos

1.3. Metodología

Se va a seguir la siguiente metodología para cada red neuronal a implementar:

1. **Creación del modelo:** se crea un modelo secuencial utilizando la clase `Sequential` de *Keras*. Este modelo consiste en una pila lineal de capas que se añaden en orden.

2. **Compilación del modelo:** se compila el modelo, pudiendo configurar el optimizador (como adam), la función de pérdida (como `categorical_crossentropy`) y las métricas de evaluación (como `accuracy`, para monitorizar la precisión del modelo).
3. **Entrenamiento del modelo:** `model.fit()` entrenará el modelo con los datos de entrenamiento, pudiendo indicarle el número de épocas (`epochs`), el tamaño de lotes (`batch_size`) y el nivel de detalle de las salidas por consola (`verbose`).
4. **Cálculo del tiempo de entrenamiento:** utilizando `time.time()` para registrar los tiempos de inicio y final del entrenamiento y así poder calcular la duración total.
5. **Evaluación del modelo:** se evalúa el rendimiento del modelo tanto en el conjunto de entrenamiento como en el de prueba, teniendo en cuenta la pérdida y precisión, y pudiendo indicar con `verbose` cuántos detalles de la evaluación mostrar por la consola.
6. **Cálculo del porcentaje de error:** se calculan los errores de entrenamiento y de prueba mediante la expresión $error\% = 100 - accuracy * 100$. Estos errores se imprimen como porcentajes con dos decimales, proporcionando una medida clara de las tasas de fallo del modelo en ambos conjuntos de dato.

Los tres últimos pasos anteriores vienen descritos en la Figura 4.

```
# Calcular el tiempo de entrenamiento
end_time = time.time()
training_time = end_time - start_time
print(f"Tiempo de entrenamiento: {training_time:.2f} segundos")

# Evaluar el modelo en el conjunto de entrenamiento y el de prueba
train_loss, train_acc = model.evaluate(train_images, train_labels, verbose=0)
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=0)

# Calcular y mostrar los errores
train_error = 100 - train_acc * 100
test_error = 100 - test_acc * 100
print(f"Error de entrenamiento: {train_error:.2f}%")
print(f"Error de prueba: {test_error:.2f}%")
```

Figura 4: Fragmento de código para imprimir los resultados del modelo

2. Red neuronal simple

En primer lugar, se ha implementado una red neuronal simple, compuesta por una capa de entrada y una de salida de tipo softmax.

2.1. Fundamentos teóricos

La red neuronal implementada está basada en un modelo de tipo feedforward, que utiliza una arquitectura sencilla con las características descritas a continuación.

Se inicia con una capa de entrada para convertir las imágenes de entrada de forma (28, 28, 1) en vectores unidimensionales de 784 valores, como recomiendan Goodfellow (Goodfellow et al., 2016) para garantizar compatibilidad con capas densas.

La capa de salida utiliza una activación *softmax*, que convierte los logits en probabilidades interpretables para clasificación multi-clase mediante la fórmula:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Esto asegura que las probabilidades sumen 1, facilitando su interpretación como probabilidades (Bishop, 2006). La función de pérdida empleada, `categorical_crossentropy`, mide la distancia entre las probabilidades predichas y las etiquetas reales usando la fórmula:

$$H(y, \hat{y}) = - \sum_{i=1}^K y_i \log(\hat{y}_i)$$

Maximizando así la probabilidad de la clase correcta (Murphy, 2012). Para la optimización, se utiliza Adam (Kingma & Ba, 2014), que combina gradientes adaptativos y momentos para acelerar la convergencia mediante actualizaciones eficientes basadas en primeros y segundos momentos de los gradientes. Finalmente, el aprendizaje supervisado se realiza mediante retropropagación (Rumelhart et al., 1986) en minibatches, ajustando los pesos para minimizar la pérdida total.

2.2. Implementación

En la Figura 5, se define una red neuronal secuencial (`Sequential`) con una capa de entrada `Flatten`, que aplana las imágenes de tamaño (28, 28, 1) en vectores. Luego, incluye una capa de salida `Dense` con 10 neuronas y activación *softmax*, adecuada para clasificar en 10 categorías.

El modelo se compila utilizando el optimizador *adam*, la función de pérdida `categorical_crossentropy`, y la métrica de precisión (*accuracy*). El entrenamiento se realiza con el método `fit`, utilizando 10 épocas y un tamaño de batch de 32, mientras se registran métricas como precisión

durante el proceso. La salida es configurada con `verbose=2` para mostrar información detallada en la consola.

```
# Crear modelo
model = Sequential([
    Flatten(input_shape=(28, 28, 1)), # Aplanar las imágenes
    Dense(10, activation='softmax')    # Capa de salida softmax
])

# Compilar modelo
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

start_time = time.time()

# Entrenar modelo
history = model.fit(train_images, train_labels,
                   epochs=10,
                   batch_size=32,
                   verbose=2)
```

Figura 5: Modelado de la primera red neuronal simple

2.3. Análisis de resultados

```
Epoch 9/10
1875/1875 - 3s - 2ms/step - accuracy: 0.9299 - loss: 0.2527
Epoch 10/10
1875/1875 - 4s - 2ms/step - accuracy: 0.9310 - loss: 0.2508
Tiempo de entrenamiento: 34.14 segundos
Error de entrenamiento: 6.67%
Error de prueba: 7.12%
```

Figura 6: Resultados de la primera red neuronal simple

La primera red neuronal, con una arquitectura simple y sin capas ocultas, logró un error de prueba del 7.12 %, con un tiempo de entrenamiento de solo 34.14 segundos.

Este modelo, aunque básico, ofrece una solución rápida para tareas sencillas de clasificación. Sin embargo, su bajo rendimiento en el conjunto de prueba refleja una capacidad limitada para generalizar, posiblemente debido a su simplicidad y la ausencia de técnicas de regularización o capas intermedias.

3. Red neuronal multicapa

En segundo lugar, se ha optado por una red neuronal con una capa oculta de 256 unidades logísticas y una capa de salida de tipo softmax.

3.1. Fundamentos teóricos

Esta red tiene una arquitectura algo más compleja que el modelo básico anterior, ya que introduce una capa oculta densa con 256 unidades logísticas y activación ReLU (*Rectified Linear Unit*). La función ReLU está definida como:

$$f(x) = \max(0, x),$$

Esta permite manejar de forma eficiente problemas de desvanecimiento del gradiente al solo activar las neuronas con valores positivos, lo que acelera la convergencia del entrenamiento (Nair & Hinton, 2010).

Esta capa oculta añade capacidad de representación al modelo, permitiendo aprender características más complejas de los datos de entrada. La capa de entrada y de salida son las mismas definidas en la anterior implementación.

3.2. Implementación

La implementación (Figura 7) es prácticamente idéntica a la de la red neuronal anterior, salvo por que se le ha añadido entre la capa de entrada y la de salida una capa oculta Dense con 256 neuronas y activación ReLU, que permite aprender características no lineales.

De nuevo, el modelo se compila utilizando el optimizador *adam*, la función de pérdida *categorical_crossentropy*, y la métrica de *accuracy*.

El entrenamiento se realiza con el método *fit*, configurado para 10 épocas, un tamaño de lote de 32, y *verbose=2* para obtener información del proceso de entrenamiento en consola.

```

model = Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(256, activation='relu'), # Capa oculta de 256 unidades logísticas
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

start_time = time.time()

history = model.fit(train_images, train_labels,
                    epochs=10,
                    batch_size=32,
                    verbose=2)

```

Figura 7: Modelado de la segunda red neuronal

3.3. Análisis de resultados

```

Epoch 9/10
1875/1875 - 11s - 6ms/step - accuracy: 0.9961 - loss: 0.0126
Epoch 10/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9968 - loss: 0.0102
Tiempo de entrenamiento: 92.66 segundos
Error de entrenamiento: 0.25%
Error de prueba: 2.07%

```

Figura 8: Resultados de la segunda red neuronal

La segunda red neuronal introdujo una capa oculta con 256 unidades y activación ReLU, lo que mejoró significativamente su capacidad para aprender características más complejas de los datos.

Esto se tradujo en una mayor precisión y un error de prueba de solo 2.07 %. El tiempo de entrenamiento aumentó, pero la mejora en el rendimiento lo justifica ampliamente. Comparada con la primera red, este modelo demuestra cómo una mayor complejidad puede mejorar la generalización en tareas de clasificación.

4. Red neuronal convolucional

A continuación, se ha considerado una red neuronal convolucional entrenada con gradiente descendente estocástico.

4.1. Fundamentos teóricos

La red neuronal implementada corresponde a una red convolucional (CNN), ampliamente utilizada para tareas de clasificación de imágenes debido a su capacidad para capturar patrones espaciales y características jerárquicas en los datos. Este modelo combina capas convolutivas, de *pooling* y densas, cada una diseñada para cumplir una función específica:

- **Capas convolutivas:** las capas Conv2D aplican filtros con tamaño (3, 3) para detectar características locales como bordes y texturas en las imágenes de entrada. Estas capas usan la activación ReLU vista antes, para introducir no linealidad y mejorar la eficiencia computacional (Nair & Hinton, 2010).
- **Capas de *Pooling*:** las capas MaxPooling2D con tamaño de ventana (2, 2) reducen la dimensionalidad espacial de las características extraídas, manteniendo la información más relevante y reduciendo la carga computacional. El *max-pooling* selecciona el valor máximo en cada ventana, lo que ayuda a captar las características más prominentes.
- **Capas densas:** la salida de las capas convolutivas es aplanada con Flatten y pasada a una capa completamente conectada (Dense) con 128 neuronas y activación ReLU, para integrar las características aprendidas. Finalmente, una capa de salida con activación *softmax* convierte las predicciones en probabilidades para las 10 clases del problema.

El modelo utiliza el optimizador SGD (*Stochastic Gradient Descent*) con tasa de aprendizaje de 0.01 y *momentum* de 0.9. El *momentum* acelera la convergencia al suavizar las actualizaciones de gradiente, evitando oscilaciones en direcciones ortogonales al gradiente principal (Sutskever et al., 2013). La función de pérdida utilizada es `categorical_crossentropy`, adecuada para clasificación multiclase, y se valida el rendimiento durante el entrenamiento utilizando un 20 % de los datos de entrenamiento como conjunto de validación interna.

Las CNN, como este modelo, son efectivas para la extracción automática de características espaciales y han demostrado gran éxito en tareas de visión computacional, como describen LeCun (LeCun et al., 1998) y Krizhevsky (Krizhevsky et al., 2012).

4.2. Implementación

El modelo secuencial implementado en la Figura 9 consta de las siguientes capas:

- Dos capas convolutivas (Conv2D) con 32 y 64 filtros respectivamente, tamaño de kernel (3, 3) y activación ReLU, para extraer características espaciales locales.

- Dos capas de *max-pooling* (MaxPooling2D) con tamaño de ventana (2, 2), para reducir la dimensionalidad espacial.
- Una capa de aplanamiento (Flatten) para transformar las características espaciales en un vector.
- Una capa densa (Dense) con 128 unidades y activación ReLU, para integrar las características aprendidas.
- Una capa de salida (Dense) con 10 unidades y activación *softmax*, para clasificación en 10 clases.

El modelo se compila utilizando el optimizador SGD con una tasa de aprendizaje de 0.01 y *momentum* de 0.9. El entrenamiento se realiza con el método `fit`, configurado para 15 épocas, tamaño de lote de 32 y un *split* del 20 % de los datos de entrenamiento como conjunto de validación. La salida se configura con `verbose=2` para mostrar el progreso en consola.

```
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)), # Primera capa convolutiva
    MaxPooling2D(pool_size=(2, 2)), # Capa de max-pooling
    Conv2D(64, kernel_size=(3, 3), activation='relu'), # Segunda capa convolutiva
    MaxPooling2D(pool_size=(2, 2)), # Capa de max-pooling
    Flatten(), # Aplanar la salida
    Dense(128, activation='relu'), # Capa densa con 128 unidades
    Dense(10, activation='softmax') # Capa de salida softmax
])

# Compilar el modelo usando SGD
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9), # SGD con momentum
              loss='categorical_crossentropy',
              metrics=['accuracy'])

start_time = time.time()

history = model.fit(train_images, train_labels,
                    epochs=15, # Ajustar para garantizar buena convergencia
                    batch_size=32,
                    validation_split=0.2, # Validación interna
                    verbose=2)
```

Figura 9: Modelado de la tercera red neuronal

4.3. Análisis de resultados

```
Epoch 14/15
1500/1500 - 81s - 54ms/step - accuracy: 0.9985 - loss: 0.0046 - val_accuracy: 0.9895 - val_loss: 0.0446
Epoch 15/15
1500/1500 - 82s - 55ms/step - accuracy: 0.9996 - loss: 0.0016 - val_accuracy: 0.9907 - val_loss: 0.0465
Tiempo de entrenamiento: 1026.72 segundos
Error de entrenamiento: 0.24%
Error de prueba: 0.93%
```

Figura 10: Resultados de la tercera red neuronal

La tercera red, una CNN con capas convolutivas y de *pooling*, alcanzó un rendimiento aún mejor, con un error de prueba reducido a 0.93 %.

Este modelo sobresale al extraer características espaciales jerárquicas de las imágenes, lo que lo hace especialmente efectivo para tareas relacionadas con visión computacional. Sin embargo, el tiempo de entrenamiento fue considerablemente mayor en relación con los anteriores, lo que refleja la alta demanda computacional de este tipo de arquitecturas.

5. Deep learning

Deep learning usando pre-entrenamiento de autoencoders para extraer características de las imágenes y una red neuronal con capa de salida softmax.

5.1. Fundamentos teóricos

Este modelo combina un autoencoder convolucional con una red neuronal supervisada para realizar una clasificación eficiente. El objetivo principal del autoencoder es extraer características relevantes y de menor dimensionalidad a partir de los datos de entrada, las cuales se utilizan posteriormente como entrada para una red neuronal densa para clasificación. Los componentes son los siguientes:

- **Autoencoder convolucional:** este a su vez consta de dos partes:
 - *Codificador:* reduce la dimensionalidad espacial de las imágenes utilizando capas convolutivas (Conv2D) con activación ReLU y capas de *max-pooling*. Este proceso permite capturar características jerárquicas de los datos.
 - *Decodificador:* reconstruye las imágenes originales a partir de las características comprimidas usando capas de *upsampling* y convolución. La última capa utiliza una activación sigmoid para garantizar que las salidas reconstruidas estén en el rango $[0, 1]$, apropiado para imágenes normalizadas (Vincent et al., 2010).
- **Red neuronal supervisada:** las características comprimidas extraídas por el autoencoder son aplanadas y usadas como entrada para una red neuronal densa. Este modelo consta de dos capas densas (Dense), con una capa oculta de 128 unidades y activación ReLU, y una capa de salida con activación softmax para clasificación multiclase.
- **Optimización:** el autoencoder se entrena usando el optimizador *adam*, que ajusta los pesos para minimizar la pérdida *binary_crossentropy*, adecuada para tareas de reconstrucción. Posteriormente, la red supervisada se entrena con el optimizador SGD y *momentum*, para mejorar la convergencia y evitar oscilaciones.

- **Entrenamiento supervisado tras la extracción de características:** las características comprimidas actúan como un conjunto de datos transformado con información relevante, lo que reduce la complejidad del modelo supervisado y mejora su capacidad de generalización (Hinton & Salakhutdinov, 2006).

5.2. Implementación

El código consta de tres partes principales:

1. Definición del autoencoder:

- Se utiliza `Input` para definir la forma de entrada (28, 28, 1).
- El codificador tiene dos capas convolutivas (`Conv2D`) con 32 y 64 filtros, seguidas de capas de *max-pooling* (`MaxPooling2D`). La activación es `ReLU` y el `padding` es `'same'` para preservar dimensiones.
- El decodificador invierte el proceso mediante capas de *upsampling* (`UpSampling2D`) y convoluciones para reconstruir la entrada original.
- El modelo es compilado con el optimizador *adam* y pérdida `binary_crossentropy`.

```
# Crear el autoencoder
input_img = Input(shape=(28, 28, 1))

# Codificador
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decodificador
x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Figura 11: Modelado de la cuarta red neuronal (pt. 1)

2. Entrenamiento del autoencoder:

- Se entrena el autoencoder usando `fit` con los datos de entrenamiento tanto como entrada como salida, durante 10 épocas, con un tamaño de lote de 256 y validación del 20 %.

- Una vez entrenado, el codificador es extraído y utilizado para transformar los datos en representaciones comprimidas.

```
# Entrenar el autoencoder
start_time = time.time()
autoencoder.fit(train_images, train_images,
                epochs=10,
                batch_size=256,
                validation_split=0.2,
                verbose=2)
end_time = time.time()

print(f"Tiempo de entrenamiento del autoencoder: {end_time - start_time:.2f} segundos")

# Extraer características del autoencoder
encoder = Model(input_img, encoded)
encoded_train = encoder.predict(train_images)
encoded_test = encoder.predict(test_images)

# Aplanar las características extraídas
X_train_flat = encoded_train.reshape(encoded_train.shape[0], -1)
X_test_flat = encoded_test.reshape(encoded_test.shape[0], -1)
```

Figura 12: Modelado de la cuarta red neuronal (pt. 2)

3. Red supervisada para clasificación:

- La red neuronal supervisada tiene una capa densa oculta con 128 neuronas y activación ReLU, seguida de una capa de salida con 10 neuronas y activación softmax.
- Se entrena utilizando SGD con tasa de aprendizaje 0.01 y *momentum* de 0.9, minimizando la pérdida `categorical_crossentropy`.
- El entrenamiento se realiza durante 20 épocas, con un tamaño de lote de 32 y validación del 20 %.


```

# Crear y entrenar la red neuronal simple
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_flat.shape[1],)),
    Dense(10, activation='softmax')
])

# Compilar el modelo usando SGD
model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo supervisado
start_time = time.time()
history = model.fit(X_train_flat, train_labels,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2,
                    verbose=2)
end_time = time.time()

```

Figura 13: Modelado de la cuarta red neuronal (pt. 3)

5.3. Análisis de resultados

```

Epoch 19/20
1500/1500 - 4s - 3ms/step - accuracy: 0.9851 - loss: 0.0460 - val_accuracy: 0.9763 - val_loss: 0.0912
Epoch 20/20
1500/1500 - 3s - 2ms/step - accuracy: 0.9852 - loss: 0.0439 - val_accuracy: 0.9761 - val_loss: 0.0828
Tiempo de entrenamiento total (autoencoder + red neuronal): 173.28 segundos
Error de entrenamiento: 1.45%
Error de prueba: 2.10%

```

Figura 14: Resultados de la cuarta red neuronal

El cuarto modelo, que combina un autoencoder con una red supervisada, logró un error de prueba de 2.10 %. Aunque no alcanzó el rendimiento de la CNN en términos de precisión, este modelo es útil para tareas que requieren una reducción de dimensionalidad previa, como la compresión de datos o la extracción de características relevantes, además requiere menos tiempo que el modelo anterior al utilizar una red neuronal más simple.

6. Red neuronal combinada mejorada

A continuación, se ha intentado combinar algunas de las técnicas anteriores para implementar una red neuronal que se acerque lo máximo posible a una *accuracy* de 99.7 %.

6.1. Fundamentos teóricos

Este modelo mejorado combina técnicas avanzadas de deep learning para optimizar la clasificación de imágenes. La arquitectura se basa en redes convolucionales (CNN), a las que se integran estrategias de regularización, aumento de datos y optimización, detalladas a continuación:

- **Aumento de datos:** se utiliza `ImageDataGenerator` para aplicar transformaciones como rotación, desplazamientos horizontales y verticales, y zoom. Estas técnicas aumentan artificialmente el tamaño del conjunto de datos, ayudando a prevenir el sobreajuste y mejorando la generalización del modelo.
- **Arquitectura del modelo:** la red incluye capas convolutivas (`Conv2D`) y de *max-pooling*, que extraen y reducen las características espaciales. Cada capa convolutiva está seguida de una capa de *dropout*, que aleatoriamente apaga un porcentaje de neuronas durante el entrenamiento para reducir el sobreajuste (Srivastava et al., 2014). Las capas densas al final permiten integrar características y realizar la clasificación final.
- **Regularización con *Dropout*:** las capas `Dropout` con tasas de 25 % y 50 % reducen el riesgo de sobreajuste al evitar que el modelo dependa excesivamente de características específicas. Esto mejora la capacidad del modelo para generalizar a datos no vistos (Srivastava et al., 2014).
- **Optimización avanzada:** el optimizador *adam* se utiliza con una tasa de aprendizaje inicial de 0.001, combinando la eficiencia de SGD con el cálculo de momentos adaptativos. La pérdida `categorical_crossentropy` mide la distancia entre las etiquetas reales y las predicciones del modelo, adecuada para clasificación multiclase.
- **Callbacks para optimización:**
 - `EarlyStopping` detiene el entrenamiento si la métrica de validación deja de mejorar después de 17 épocas consecutivas.
 - `ReduceLROnPlateau` ajusta dinámicamente la tasa de aprendizaje cuando el modelo alcanza un *plateau* en el rendimiento.
 - `ModelCheckpoint` guarda el mejor modelo durante el entrenamiento, permitiendo su uso posterior para predicciones (Bengio, 2012).

En la Figura 15 se muestra el funcionamiento de los tres callbacks anteriores: *ReduceLROnPlateau*, que disminuye la tasa de aprendizaje cuando `val_loss` deja de mejorar, ayudando a realizar ajustes más precisos; *EarlyStopping*, que detiene el entrenamiento tras varias épocas sin mejora

para evitar sobreajuste (aunque no se activa en esta imagen, sí lo hace en la Figura 19); y *ModelCheckpoint*, que guarda automáticamente el mejor modelo basado en el menor *val_loss*, asegurando que el modelo final sea el más óptimo alcanzado.

```
Epoch 8: val_loss did not improve from 0.02519
1500/1500 - 19s - 12ms/step - accuracy: 0.9773 - loss: 0.0740 - val_accuracy: 0.9919 - val_loss: 0.0278 - learning_rate: 0.0010
Epoch 9/85

Epoch 9: val_loss did not improve from 0.02519
1500/1500 - 21s - 14ms/step - accuracy: 0.9797 - loss: 0.0716 - val_accuracy: 0.9923 - val_loss: 0.0268 - learning_rate: 0.0010
Epoch 10/85

Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.

Epoch 10: val_loss did not improve from 0.02519
1500/1500 - 19s - 13ms/step - accuracy: 0.9791 - loss: 0.0708 - val_accuracy: 0.9924 - val_loss: 0.0263 - learning_rate: 0.0010
Epoch 11/85

Epoch 11: val_loss did not improve from 0.02519
1500/1500 - 19s - 12ms/step - accuracy: 0.9834 - loss: 0.0545 - val_accuracy: 0.9925 - val_loss: 0.0266 - learning_rate: 5.0000e-04
Epoch 12/85

Epoch 12: val_loss improved from 0.02519 to 0.02255, saving model to best_model.keras
1500/1500 - 20s - 13ms/step - accuracy: 0.9847 - loss: 0.0525 - val_accuracy: 0.9945 - val_loss: 0.0226 - learning_rate: 5.0000e-04
Epoch 13/85

Epoch 13: val_loss improved from 0.02255 to 0.01947, saving model to best_model.keras
1500/1500 - 19s - 13ms/step - accuracy: 0.9851 - loss: 0.0510 - val_accuracy: 0.9946 - val_loss: 0.0195 - learning_rate: 5.0000e-04
Epoch 14/85
```

Figura 15: Ejemplo del funcionamiento de los callbacks

6.2. Implementación

El código implementa las siguientes etapas para el entrenamiento y evaluación del modelo:

1. División y aumento de datos:

- Los datos de entrenamiento se dividen en conjuntos de entrenamiento y validación usando `train_test_split`.
- El aumento de datos se realiza con `ImageDataGenerator`, aplicando transformaciones como rotación (10°), desplazamientos (10 %) y zoom (10 %) en los datos de entrenamiento. Para los datos de validación, no se aplica aumento.

```
# Dividir los datos en entrenamiento y validación
train_images_split, val_images_split, train_labels_split, val_labels_split = train_test_split(
    train_images, train_labels, test_size=0.2, random_state=42
)

# Generador con aumentación para datos de entrenamiento
train_datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1
)

# Generador sin aumentación para datos de validación
val_datagen = ImageDataGenerator()

# Preparar los generadores
train_generator = train_datagen.flow(train_images_split, train_labels_split, batch_size=32)
val_generator = val_datagen.flow(val_images_split, val_labels_split, batch_size=32)
```

Figura 16: Modelado de la quinta red neuronal (pt. 1)

2. Definición del modelo:

- La arquitectura del modelo incluye:
 - Dos bloques convolutivos con 64 y 128 filtros, kernel (3, 3), y activación ReLU.
 - Capas de *max-pooling* (2, 2) para reducir dimensionalidad.
 - Capas de *dropout* para prevenir el sobreajuste.
 - Una capa densa con 256 unidades y activación ReLU.
 - Una capa de salida con activación *softmax* para clasificación multiclase.
- El modelo se compila usando el optimizador *adam* con tasa de aprendizaje inicial de 0.001, y pérdida *categorical_crossentropy*.

```
model = Sequential([
    Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Conv2D(128, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
```

Figura 17: Modelado de la quinta red neuronal (pt. 2)

3. Entrenamiento del modelo:

- El entrenamiento utiliza generadores para los datos de entrenamiento y validación, aplicando aumento de datos en los primeros.
- Se configuran *callbacks* como *EarlyStopping*, *ReduceLROnPlateau*, y *ModelCheckpoint* para mejorar la eficiencia del entrenamiento.
- El modelo es entrenado durante un máximo de 85 épocas (no suele alcanzar este límite, ya que se detiene normalmente en la época 40-50), con un tamaño de lote de 32.

```
# Callbacks para optimización
callbacks = [
    EarlyStopping(patience=17, restore_best_weights=True, verbose=1),
    ReduceLROnPlateau(factor=0.5, patience=3, min_lr=1e-6, verbose=1),
    ModelCheckpoint('best_model.keras', save_best_only=True, verbose=1)
]

start_time = time.time()

history = model.fit(
    train_generator,
    epochs=85,
    validation_data=val_generator,
    callbacks=callbacks,
    verbose=2
)
```

Figura 18: Modelado de la quinta red neuronal (pt. 3)

6.3. Análisis de resultados

```
Epoch 47: val_loss did not improve from 0.01800
1500/1500 - 21s - 14ms/step - accuracy: 0.9903 - loss: 0.0308 - val_accuracy: 0.9951 - val_loss: 0.0181 - learning_rate: 1.0000e-06
Epoch 48/85

Epoch 48: val_loss did not improve from 0.01800
1500/1500 - 19s - 13ms/step - accuracy: 0.9911 - loss: 0.0289 - val_accuracy: 0.9950 - val_loss: 0.0181 - learning_rate: 1.0000e-06
Epoch 48: early stopping
Restoring model weights from the end of the best epoch: 31.
Tiempo de entrenamiento: 1003.86 segundos
Entrenamiento. Accuracy: 1.0% - Error: 0.28%
Prueba. Accuracy: 1.0% - Error: 0.38%
```

Figura 19: Resultados de la cuarta red neuronal

Finalmente, la red mejorada, que incluyó técnicas avanzadas como aumento de datos, *dropout*, y *callbacks*, logró los mejores resultados entre todas las evaluadas. Con un error de prueba de solo 0.38 %, este modelo destaca por su capacidad de generalización y resistencia al sobreajuste. Aunque su tiempo de entrenamiento es elevado, demuestra cómo la integración de estrategias avanzadas puede maximizar el rendimiento, superando incluso a la CNN estándar en precisión y estabilidad.

7. Conclusiones

En conclusión, este

8. Bibliografía

- Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures. *Neural Networks: Tricks of the Trade*, 7700, 437-478.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786), 504-507.
- Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 25, 1097-1105.
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.
- Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 807-814.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56), 1929-1958.
- Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 1139-1147.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *Journal of Machine Learning Research*, 11, 3371-3408.