Redes neuronales y deep learning

UNIVERSIDAD DE GRANADA

Pablo Morenilla Pinos
morenillapablo@correo.ugr.es
IC Prácticas

Índice

1. Introducción. **3**
1.1. Descripción del problema. 3
1.2. MNIST. 3
1.3. Herramientas usadas. 4

2. Implementación. **6**
2.1. Lectura de Datos desde MNIST. 6
2.2. Construcción del modelo. 7
2.3. Entrenamiento del modelo. 10

3. Resultados. **16**
3.1. Gráficas obtenidas. 16
3.2. Resultados 20
3.3. Conclusiones. 21

Índice de figuras

1. Lectura números. 6
2. Representación de los números. 7
3. Construcción modelo simple. 8
4. Construcción modelo mlp. 9
5. Construcción modelo cnn. 10
6. Entrenamiento modelo simple. 11
7. modelo simple secuencias. 11
8. Entrenamiento modelo mlp. 12
9. modelo mlp secuencias. 13
10. Entrenamiento modelo cnn. 14
11. modelo cnn secuencias. 15
12. Accuracy modelo simple. 16
13. Loss modelo simple. 16
14. Accuracy modelo mlp. 17
15. Loss modelo mlp. 17
16. Accuracy modelo cnn. 17
17. Loss modelo cnn. 18
18. Resultados modelo simple. 18
19. Resultados modelo mlp. 19
20. Resultados modelo cnn. 19

1. Introducción.

1.1. Descripción del problema.

El reconocimiento de dígitos escritos es un problema en el campo de la inteligencia artificial, y aborda la tarea de convertir imágenes de números escritos a mano en representaciones digitales. Este problema, a menudo denominado Optical Character Recognition (OCR), es esencial en diversas aplicaciones, desde la clasificación automática de correos hasta la digitalización de documentos históricos.

En esta práctica, nos centramos en abordar el problema de OCR utilizando redes neuronales artificiales, una rama de la inteligencia artificial que imita el funcionamiento del cerebro humano para aprender patrones y realizar tareas específicas. Utilizaremos el conjunto de datos MNIST, que consiste en imágenes normalizadas de dígitos manuscritos.

El objetivo principal es explorar y evaluar diferentes enfoques de redes neuronales para resolver este desafío específico. Se ha convertido en un estándar de referencia en la evaluación de algoritmos de reconocimiento de dígitos, con aplicaciones que van desde sistemas de pago automáticos hasta la verificación de identidad.

En lugar de simplemente abordar el problema desde cero, consideramos la implementación de algoritmos de entrenamiento ya existentes, con la posibilidad de utilizar bibliotecas populares como TensorFlow o PyTorch. Sin embargo, las decisiones de implementación que se han tomado variarán en la calidad de nuestro modelo y, por ende, en la eficacia del reconocimiento de dígitos.

En el transcurso de esta práctica, también exploraremos estrategias de prueba que garanticen la precisión del cálculo del gradiente, un componente del entrenamiento de redes neuronales.

1.2. MNIST.

El conjunto de datos MNIST, abreviatura de "Modified National Institute of Standards and Technology", se ha consolidado como un pilar en la evaluación de algoritmos de reconocimiento de dígitos manuscritos. Fue creado por Yann LeCun, Corinna Cortes y Christopher J.C. Burges, y ha sido utilizado ampliamente en la investigación de aprendizaje automático y visión por computadora.

Este conjunto de datos se compone de imágenes normalizadas de dígitos manuscritos, extraídos de documentos de empleados del Instituto Nacional de Estándares y Tecnología de los Estados Unidos. La base de datos consta de 60,000 ejemplos de entrenamiento y 10,000 ejemplos de prueba, cada uno etiquetado con el dígito correspondiente del 0 al 9.

MNIST ha ganado popularidad debido a su simplicidad y relevancia. Las imágenes son monocromáticas y tienen una resolución de 28x28 píxeles, lo que las hace ideales para experimentar con algoritmos de reconocimiento de patrones. Además, el conjunto de datos ha sido utilizado como un estándar para comparar el rendimiento de diferentes enfoques de aprendizaje automático.

Su disponibilidad pública en el sitio web de Yann LeCun ha facilitado su acceso y uso en la comunidad científica. Dada su longevidad y confiabilidad, MNIST ha servido como punto de partida para muchos investigadores y profesionales que buscan desarrollar y probar algoritmos de clasificación de imágenes.

1.3. Herramientas usadas.

Durante la implementación de la práctica, se emplearon diversas herramientas, bibliotecas y el lenguaje de programación, las cuales se detallan a continuación:

Bibliotecas de Python:

- `keras.preprocessing.image`: Para la generación de imágenes de entrada.
- `numpy`: Para operaciones matriciales y numéricas.
- `time`: Para medir el tiempo de ejecución.
- Módulos personalizados (`model_builder`, `model_trainer`, `visualization`): Contienen funciones y clases específicas definidas en otros archivos.

Módulos Personalizados:

- `model_builder`: Contiene funciones para construir diferentes modelos de red neuronal.
- `model_trainer`: Contiene funciones para entrenar modelos de red neuronal.
- `visualization`: Contiene funciones para visualizar los resultados del modelo.

Bibliotecas Adicionales:

- `matplotlib.pyplot`: Para la generación de gráficos y visualización de datos.
- `os`: Para la manipulación de archivos y directorios.
- `shutil`: Para operaciones de manipulación de archivos y directorios.

Bibliotecas de Aprendizaje Automático:

- `keras.models`: Para construir modelos de red neuronal secuenciales.
- `keras.layers`: Contiene capas utilizadas en la construcción de modelos.
- `keras.optimizers`: Para configurar el optimizador durante la compilación del modelo.
- `keras.datasets.mnist` y `keras.datasets.fashion_mnist`: Para cargar conjuntos de datos de dígitos escritos a mano y conjuntos de datos de moda.

Bibliotecas de Aprendizaje Automático y Preprocesamiento de Datos:

- `keras.utils`: Para convertir etiquetas a matrices binarias y visualización del modelo.
- `sklearn.model_selection.train_test_split`: Para dividir el conjunto de datos en conjuntos de entrenamiento y validación.

Herramientas de Procesamiento de Imágenes y Aumento de Datos:

- `keras.preprocessing.image.ImageDataGenerator`: Para aplicar aumentos de datos durante el entrenamiento.

2. Implementación.

2.1. Lectura de Datos desde MNIST.

```python
from keras.datasets import mnist, fashion_mnist
from sklearn.model_selection import train_test_split

# Lectura de datos
def read_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    # Divide el conjunto de entrenamiento en train y val
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1, random_state=1)

    return (x_train, y_train), (x_test, y_test), (x_val, y_val)

# Preprocesa los datos de entrada
# - x_train: entrada del conjunto de entrenamiento.
# - y_train: etiquetas del conjunto de entrenamiento.
# - x_test: entrada del conjunto de test.
# - y_test: etiquetas del conjunto de test.
def summarize_dataset(x_train, y_train, x_test, y_test):
    print('Dimensión de x_train:', x_train.shape, 'y_train:', y_train.shape)
    print('Dimensión de x_test:', x_test.shape, 'y_test:', y_test.shape)
    print(x_train.shape[0], 'ejemplos de entrenamiento')
    print(x_test.shape[0], 'ejemplos de test\n')

# Preprocesa los datos de entrada
# - x_train: entrada del conjunto de entrenamiento.
# - x_test: entrada del conjunto de test.
def preprocess_data(x_train, x_test):
    x_train = x_train.astype('float32') # conversión a float32
    x_test = x_test.astype('float32')   # conversión a float32
    x_train /= 255                      # normalización a [0,1]
    x_test /= 255                       # normalización a [0,1]
```

Figura 1: Lectura números.

Explicación

El fragmento de código proporcionado tiene como objetivo principal la lectura de imágenes del conjunto de datos MNIST. A continuación, se explica de manera general el proceso:

a) **Selección del Conjunto de Datos:** La función `mnist.load_data()` se utiliza para cargar los datos de entrenamiento y prueba del conjunto MNIST.

b) **División del Conjunto de Entrenamiento:** Posteriormente, se realiza una división adicional del conjunto de entrenamiento en conjuntos de entrenamiento y validación mediante la función `train_test_split`. Esta división se realiza con el fin de tener un subconjunto para entrenar el modelo y otro para validar su desempeño durante el proceso de entrenamiento.

c) **Retorno de los Conjuntos:** La función retorna un conjunto de tres tuplas:

- `(x_train, y_train)`: Conjunto de entrenamiento que contiene las imágenes y las etiquetas asociadas.
- `(x_test, y_test)`: Conjunto de prueba que contiene las imágenes y las etiquetas asociadas.

- (*x_val, y_val*): Conjunto de validación que contiene las imágenes y las etiquetas asociadas.

Este fragmento establece las bases para la carga de datos, preparando los conjuntos necesarios para el posterior procesamiento y entrenamiento de modelos de aprendizaje automático.

Una vez hecho esto, podemos visualizar la imagen y su valor con show_data:

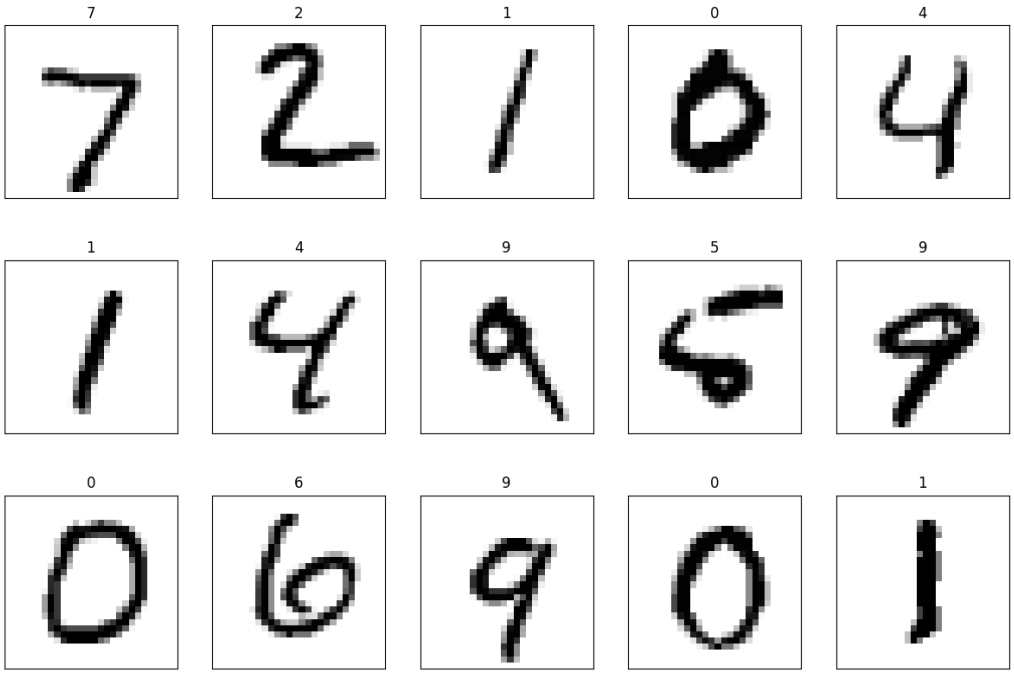


Figura 2: Representación de los números.

2.2. Construcción del modelo.

- **Red Neuronal Simple (`build_simple_neuronal_network_model`):**
  - Capas Densas: 128 neuronas ReLU, 64 neuronas ReLU y capa de salida con activación `softmax`.
  - Compilación del Modelo: Entropía cruzada categórica, optimizador Adam, métrica de precisión.
- **Red Neuronal Convolucional (CNN) (`construc_model`):**
  - Capas Convolucionales y de Pooling: Dos capas convolucionales seguidas de capas de max-pooling.

- Capas Densas: Capa de aplanado, capas densas ReLU.
- Regularización con Dropout: Capas de dropout para prevenir sobreajuste.
- Compilación del Modelo: Entropía cruzada categórica, optimizador Adam, métrica de precisión.

▪ **Perceptrón Multicapa (MLP) (`build_mlp_model`):**

- Capas Densas con Batch Normalization: Capas densas con activación ReLU y normalización por lotes.
- Regularización con Dropout: Capas de dropout para prevenir sobreajuste.
- Compilación del Modelo: Entropía cruzada categórica, optimizador Adam, métrica de precisión.

```python
# Función para construir y compilar un modelo de red neuronal simple.
# Construye y compila un modelo de red neuronal simple con capas densas.
#     Parameters:
#     - input_shape (tuple): Dimensiones de entrada de las imágenes.
#     - num_classes (int): Número de clases del problema.
#     Returns:
#     - model (Sequential): Modelo de red neuronal construido.
def build_simple_neuronal_network_model(input_shape, num_classes):

    # Crear un modelo secuencial
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(num_classes, activation='softmax')
    ])

    # Compilar el modelo con la función de pérdida, el optimizador y las métricas
    model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )

    return model
```

Figura 3: Construcción modelo simple.

```python
# Función para construir y compilar un modelo de red neuronal tipo perceptrón multicapa (MLP)
# Construye y compila un modelo de red neuronal tipo perceptrón multicapa (MLP).
#     Parameters:
#     - input_shape (tuple): Dimensiones de entrada de las imágenes.
#     - num_classes (int): Número de clases del problema.
#     Returns:
#     - model (Sequential): Modelo de perceptrón multicapa construido.
def build_mlp_model(input_shape, num_classes):
    # Crear un modelo secuencial
    model = Sequential([
        Flatten(input_shape=input_shape),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(256, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(128, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(num_classes, activation='softmax')
    ])

    # Compilar el modelo con la función de pérdida, el optimizador y las métricas
    model.compile(
        loss='categorical_crossentropy',
        optimizer=Adam(learning_rate=0.0001),
        metrics=['accuracy']
    )

    return model
```

Figura 4: Construcción modelo mlp.

```python
# Construye y compila un modelo de red neuronal convolucional (CNN).
#     Parameters:
#     - input_shape (tuple): Dimensiones de entrada de las imágenes.
#     Returns:
#     - model (Sequential): Modelo de red neuronal convolucional construido.
def cnn_model(input_shape):
    # Crear un modelo secuencial
    model = Sequential()

    # Agregar capas convolucionales y de pooling
    model.add(Conv2D(64, kernel_size=(5, 5),
                     activation='relu',
                     input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Agregar capa de dropout para regularización
    model.add(Dropout(0.3))

    # Aplanar la salida y agregar capas densas
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(128, activation='relu'))

    # Agregar capas de dropout adicionales
    model.add(Dropout(0.2))
    model.add(Dense(50, activation='relu'))
    model.add(Dropout(0.4))

    # Capa de salida con activación softmax para clasificación
    model.add(Dense(N_CLASSES, activation='softmax'))

    # Compilar el modelo con la función de pérdida, el optimizador y las métricas
    model.compile(loss=keras.losses.categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(lr=0.001),
                  metrics=['accuracy'])

    return model
```

Figura 5: Construcción modelo cnn.

2.3. Entrenamiento del modelo.

Entrenamiento de la Red Neuronal Simple:

El modelo simple se compila utilizando la entropía cruzada categórica como función de pérdida, el optimizador Adam y la métrica de precisión. Durante el entrenamiento, se utilizan datos de entrenamiento y validación con un tamaño de lote de 256 y se ejecutan 20 épocas.

```python
# Función para entrenar un modelo de red neuronal simple.
# Entrena un modelo de red neuronal simple y devuelve el historial de entrenamiento.
#     Parameters:
#     - model (Sequential): Modelo de red neuronal a entrenar.
#     - x_train (numpy.ndarray): Conjunto de entrenamiento de características.
#     - y_train (numpy.ndarray): Conjunto de entrenamiento de etiquetas.
#     - x_val (numpy.ndarray): Conjunto de validación de características.
#     - y_val (numpy.ndarray): Conjunto de validación de etiquetas.
#     Returns:
#     - history (History): Historial de entrenamiento del modelo.
def train_simple_neuronal_network(model, x_train, y_train, x_val, y_val):
    # Compilar el modelo
    model.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    # Entrenar el modelo
    history = model.fit(
        x_train, y_train,
        batch_size=BATCH_SIZE,
        epochs=EPOCHS,
        verbose=1,
        validation_data=(x_val, y_val)
    )

    return history
```

Figura 6: Entrenamiento modelo simple.

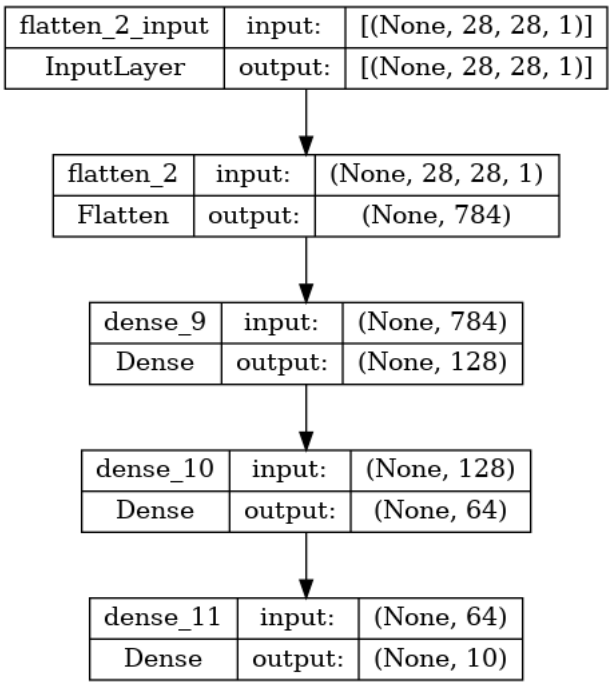


Figura 7: modelo simple secuencias.

Entrenamiento del Perceptrón Multicapa (MLP):

Para el MLP, se emplea un generador de datos que aplica aumentos como rotación, desplazamiento y zoom. El modelo se compila con entropía cruzada categórica, optimizador Adam y métrica de precisión. Durante el entrenamiento, se utiliza este generador con un tamaño de lote de 256 durante 20 épocas.

```python
# Función para entrenar un modelo de perceptrón multicapa (MLP) con aumento de datos.
# Entrena un modelo de perceptrón multicapa (MLP) con aumento de datos y devuelve el historial de entrenamiento.
#     Parameters:
#     - model (Sequential): Modelo de perceptrón multicapa a entrenar.
#     - x_train (numpy.ndarray): Conjunto de entrenamiento de características.
#     - y_train (numpy.ndarray): Conjunto de entrenamiento de etiquetas.
#     - x_val (numpy.ndarray): Conjunto de validación de características.
#     - y_val (numpy.ndarray): Conjunto de validación de etiquetas.
#     Returns:
#     - history (History): Historial de entrenamiento del modelo.
def train_mlp_model(model, x_train, y_train, x_val, y_val):
    # Configurar el generador de datos con aumento de datos
    datagen = ImageDataGenerator(
        rotation_range=10,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.1,
        zoom_range=0.1
    )
    datagen.fit(x_train)

    # Entrenar el modelo con el generador de datos
    history = model.fit(datagen.flow(x_train, y_train, batch_size=BATCH_SIZE),
                        steps_per_epoch=len(x_train) / BATCH_SIZE,
                        epochs=EPOCHS,
                        validation_data=(x_val, y_val),
                        verbose=1)

    return history
```

Figura 8: Entrenamiento modelo mlp.

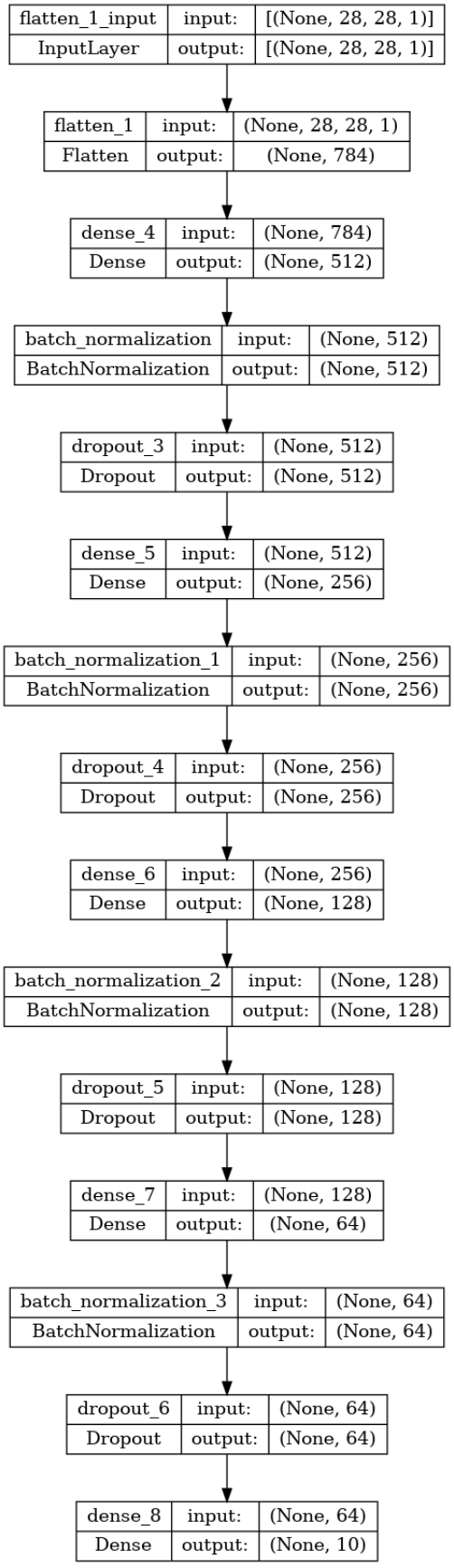
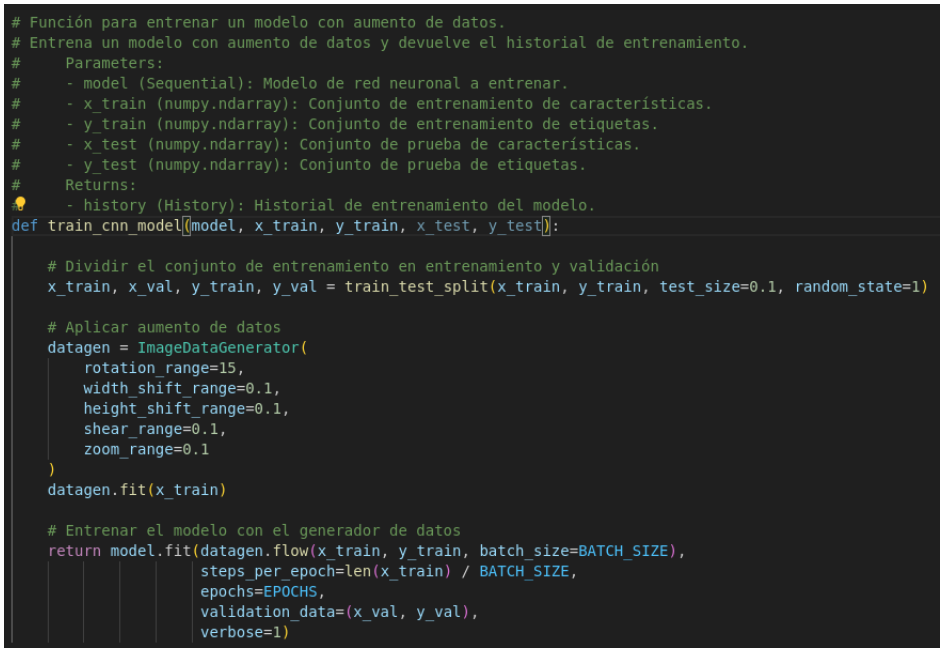


Figura 9: modelo mlp secuencias.

Entrenamiento de la Red Neuronal Convolucional:

En el caso de la red neuronal convolucional (CNN), el conjunto de entrenamiento se divide en datos de entrenamiento y validación. Se aplica un generador de datos con aumentos, como rotación y desplazamiento. El modelo se compila con entropía cruzada categórica, optimizador Adam y métrica de precisión. El entrenamiento se realiza mediante el generador de datos con un tamaño de lote de 256 durante 20 épocas.



```python
# Función para entrenar un modelo con aumento de datos.
# Entrena un modelo con aumento de datos y devuelve el historial de entrenamiento.
#     Parameters:
#     - model (Sequential): Modelo de red neuronal a entrenar.
#     - x_train (numpy.ndarray): Conjunto de entrenamiento de características.
#     - y_train (numpy.ndarray): Conjunto de entrenamiento de etiquetas.
#     - x_test (numpy.ndarray): Conjunto de prueba de características.
#     - y_test (numpy.ndarray): Conjunto de prueba de etiquetas.
#     Returns:
#     - history (History): Historial de entrenamiento del modelo.
def train_cnn_model(model, x_train, y_train, x_test, y_test):

    # Dividir el conjunto de entrenamiento en entrenamiento y validación
    x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.1, random_state=1)

    # Aplicar aumento de datos
    datagen = ImageDataGenerator(
        rotation_range=15,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.1,
        zoom_range=0.1
    )
    datagen.fit(x_train)

    # Entrenar el modelo con el generador de datos
    return model.fit(datagen.flow(x_train, y_train, batch_size=BATCH_SIZE),
                     steps_per_epoch=len(x_train) / BATCH_SIZE,
                     epochs=EPOCHS,
                     validation_data=(x_val, y_val),
                     verbose=1)
```

Figura 10: Entrenamiento modelo cnn.

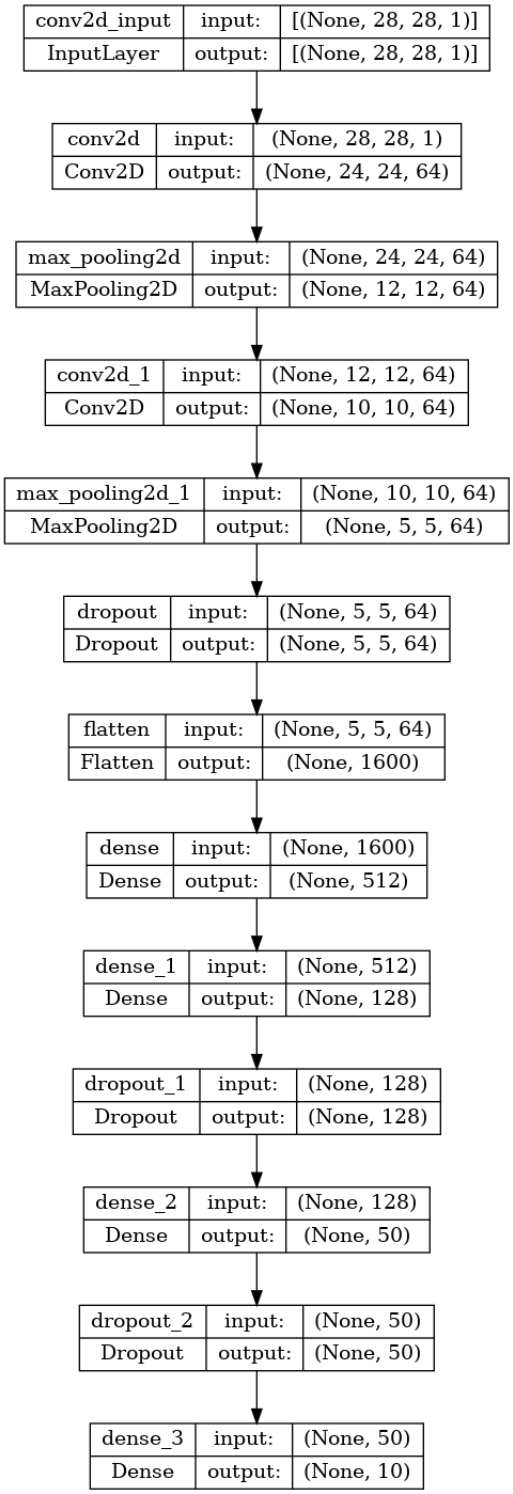


Figura 11: modelo cnn secuencias.

3. Resultados.

3.1. Gráficas obtenidas.

Se han generado gráficos que muestran la evolución de la precisión durante el entrenamiento de cada modelo, teniendo en cuenta la precisión de validación en cada época. Estas gráficas representan la relación entre el número de épocas y la precisión alcanzada por los modelos en el conjunto de entrenamiento.

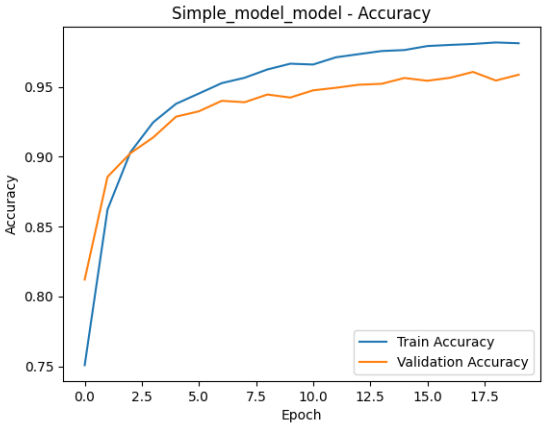


Figura 12: Accuracy modelo simple.

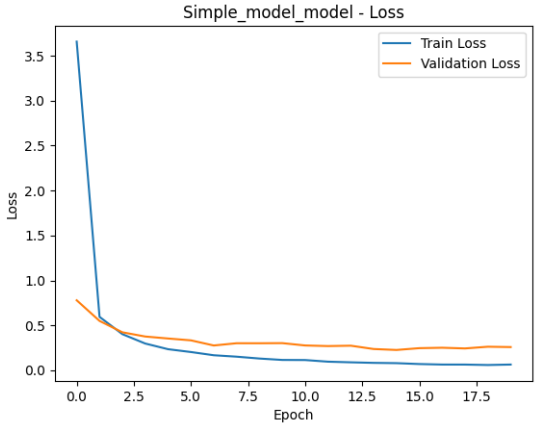


Figura 13: Loss modelo simple.

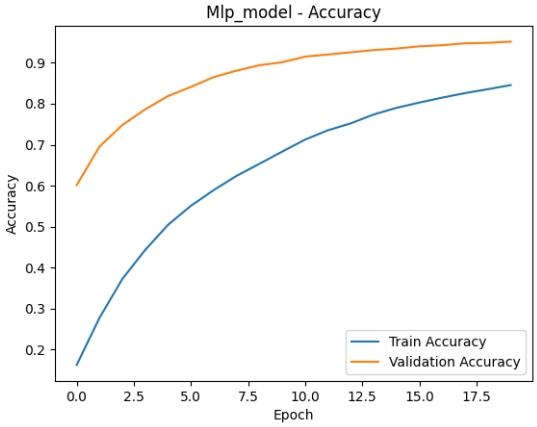


Figura 14: Accuracy modelo mlp.

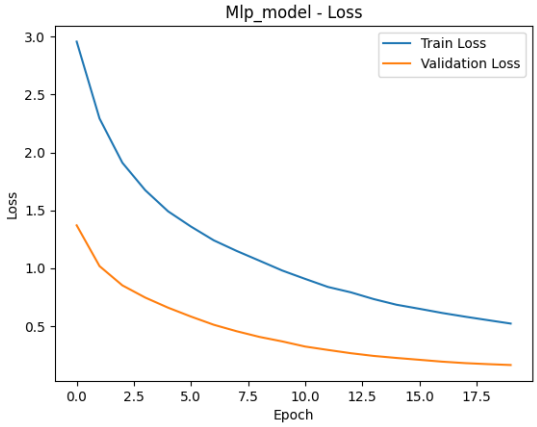


Figura 15: Loss modelo mlp.

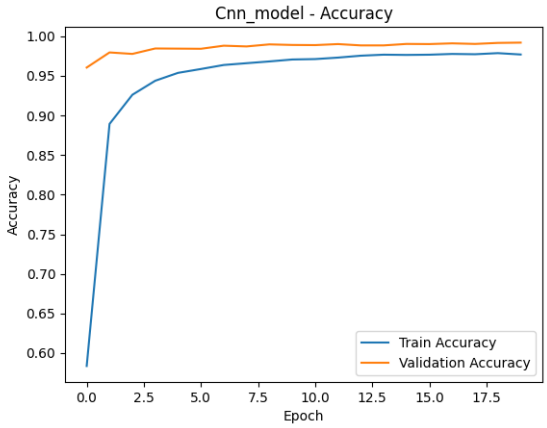


Figura 16: Accuracy modelo cnn.

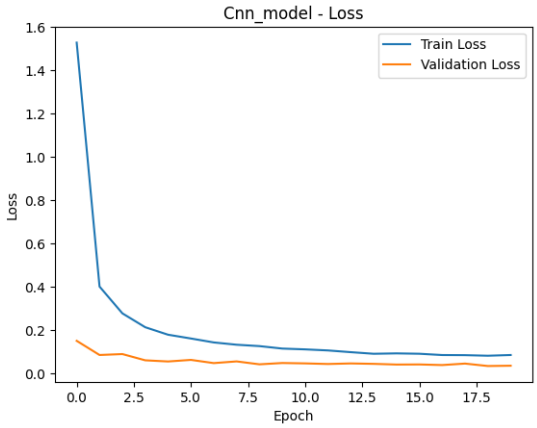


Figura 17: Loss modelo cnn.

Se han creado gráficos que representan la precisión de cada modelo tanto en el conjunto de entrenamiento como en el de prueba. Además, se incluyen las pérdidas asociadas a cada modelo, expresadas en porcentajes.

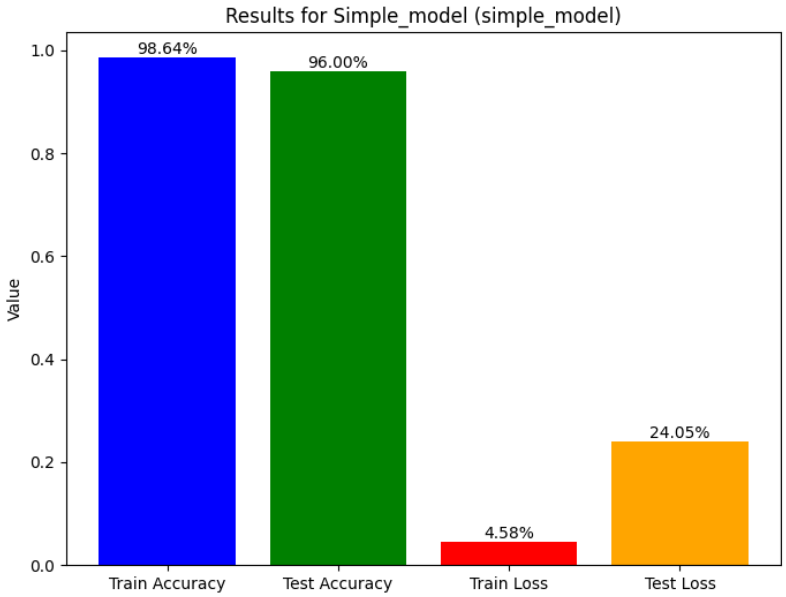


Figura 18: Resultados modelo simple.

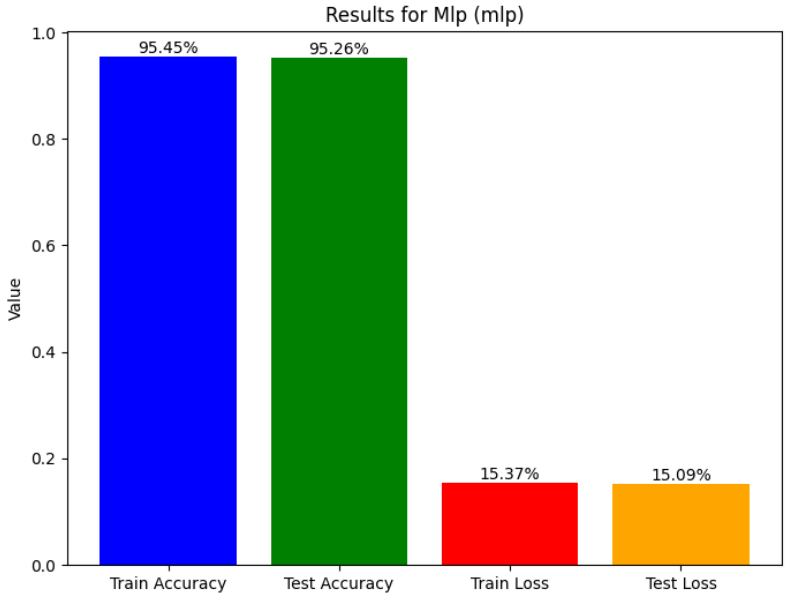


Figura 19: Resultados modelo mlp.

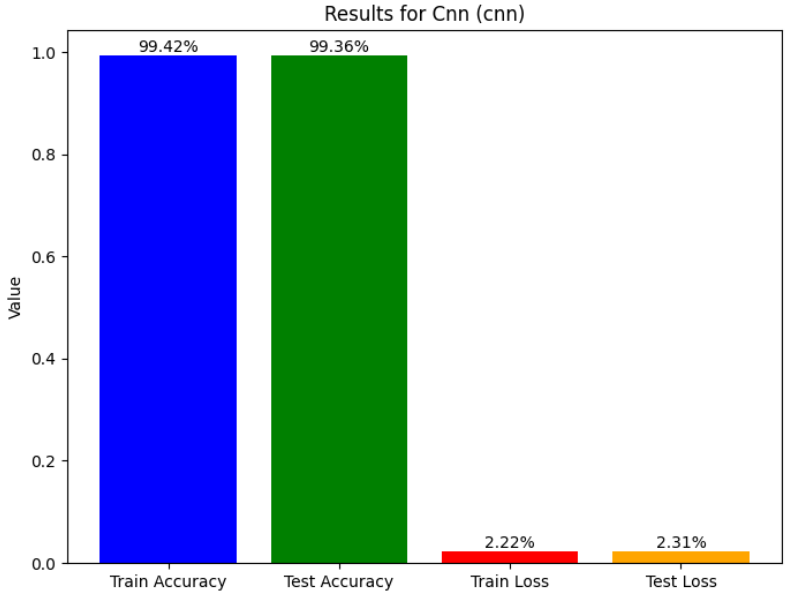


Figura 20: Resultados modelo cnn.

3.2. Resultados

Modelo: CNN

- Tiempo empleado: 159.91 segundos.
- Conjunto de entrenamiento:
  - Precisión: 99.42 %
  - Pérdida: 2.22 %
- Conjunto de prueba:
  - Precisión: 99.36 %
  - Pérdida: 2.31 %

Modelo: MLP

- Tiempo empleado: 113.32 segundos.
- Conjunto de entrenamiento:
  - Precisión: 95.45 %
  - Pérdida: 15.37 %
- Conjunto de prueba:
  - Precisión: 95.26 %
  - Pérdida: 15.09 %

Modelo: Simple Model

- Tiempo empleado: 7.57 segundos.
- Conjunto de entrenamiento:
  - Precisión: 98.64 %
  - Pérdida: 4.58 %
- Conjunto de prueba:
  - Precisión: 96.00 %
  - Pérdida: 24.05 %

3.3. Conclusiones.

Con base en los resultados obtenidos y los requisitos del problema abordado, se concluye que el modelo de red neuronal convolucional (CNN) se destaca como la opción más eficiente. La CNN ha demostrado una precisión excepcional tanto en el conjunto de entrenamiento como en el de prueba, manteniendo un tiempo de entrenamiento razonable.

Comparando con otros modelos como MLP y el modelo neuronal simple, la CNN ha logrado un rendimiento superior en términos de precisión y pérdida. Estos resultados sugieren que la CNN es la elección más adecuada para el problema específico de clasificación de imágenes.

Es importante tener en cuenta la complejidad del modelo y el tiempo de entrenamiento al evaluar opciones para aplicaciones del mundo real. En este caso, la CNN ha demostrado ser la mejor alternativa al lograr un equilibrio óptimo entre rendimiento y eficiencia.