EJERCICIO #6

Electiva II

Jaime Andrés Martínez Ordoñez

Uniautónoma
DEL CAUCA

2025

Clasificador de Dígitos Propios con Python

Cristhian Cañar
28/04/2025

Taller #6 Clasificador de Dígitos Propios con Python

1. Descripción del dataset

Un dataset utilizado en este proyecto consiste en imágenes de dígitos manuscritos organizadas en carpetas, donde cada carpeta representa una clase correspondiente a un número del 0 al 9. Cada imagen está almacenada en formato JPEG y presenta variaciones en la escritura, estilo y tamaño, simulando la diversidad natural que existe en manuscritos reales. La ruta base del conjunto de datos fue especificada como **E:\semestres\N. Semestre\Electiva ll\CORTE #2\Entrega6\Dataset-**

Para preparar los datos, se recorrieron las carpetas de cada clase, cargando únicamente aquellas imágenes que tenían la extensión .jpeg. Cada imagen fue leída utilizando OpenCV y convertida a escala de grises para reducir la complejidad de la entrada, eliminando redundancias asociadas al color que no son relevantes para el reconocimiento de dígitos. Posteriormente, las imágenes fueron redimensionadas a un tamaño fijo de 32x32 píxeles, estandarizando su tamaño y facilitando la entrada a la red neuronal. Además, se aplicó una normalización, dividiendo los valores de píxeles por 255, llevando todos los valores al rango [0,1], lo cual mejora la estabilidad numérica durante el entrenamiento de la CNN.

Una vez preprocesadas, las imágenes y sus respectivas etiquetas (correspondientes a los dígitos 0-9) fueron almacenadas en arrays de NumPy. Luego, se realizó una partición del dataset en conjuntos de entrenamiento y prueba mediante train_test_split, reservando el 20% de las muestras para el conjunto de prueba. Para cumplir con el formato de entrada requerido por las capas convolucionales de Keras, se expandieron las dimensiones de las imágenes para incluir un canal (pasando de (32,32) a (32,32,1)).

Adicionalmente, las etiquetas fueron transformadas al formato one-hot encoding, donde cada dígito se representa como un vector de 10 posiciones con un '1' en la posición correspondiente a su clase, y '0' en las demás, facilitando el uso de la función de pérdida categorical crossentropy.

Para enriquecer aún más el conjunto de entrenamiento y mejorar la capacidad de generalización de la red, se implementó una técnica de aumento de datos (data augmentation) utilizando ImageDataGenerator. Este generador aplicó transformaciones aleatorias como rotaciones de hasta 10 grados, pequeños desplazamientos horizontales y verticales, así como pequeños escalados de la imagen. De esta manera, se simularon nuevas variaciones de los datos originales, ayudando al modelo a ser más robusto frente a ligeras variaciones en la escritura de los dígitos.

En resumen, el dataset fue cuidadosamente procesado y ampliado para maximizar el rendimiento del modelo, manteniendo una representación realista de la variabilidad de los dígitos manuscritos.

2. Arquitectura de la CNN

La red neuronal convolucional (CNN) desarrollada para este proyecto fue diseñada para la clasificación de imágenes de dígitos manuscritos. Se construyó utilizando TensorFlow y Keras, aprovechando las capacidades de las CNN para procesar información espacial y extraer automáticamente características relevantes de las imágenes.

El modelo inicia con una capa de entrada que recibe imágenes preprocesadas en escala de grises, redimensionadas a 32x32 píxeles y con un solo canal. A partir de esta entrada, se implementa una primera capa convolucional compuesta por 32 filtros de tamaño 3x3, con función de activación ReLU. Esta capa permite la detección de patrones locales simples, como bordes y texturas. Posteriormente, se aplica Batch Normalization para estabilizar las activaciones y acelerar el entrenamiento, seguido de una capa de MaxPooling de tamaño 2x2, que reduce la resolución espacial a la mitad y ayuda a disminuir el número de parámetros. Para combatir el sobreajuste, se incorpora una capa de Dropout con una tasa del 25%.

Luego, se agrega una segunda capa convolucional, esta vez con 64 filtros de 3x3, nuevamente activados mediante ReLU, que permite la extracción de patrones más complejos basados en las características previamente detectadas. Esta es seguida también por Batch Normalization, MaxPooling 2x2 y un segundo Dropout del 25%.

La salida de las capas convolucionales es posteriormente aplanada a través de una capa Flatten, transformándola en un vector unidimensional que puede ser procesado por capas densas. Se añade una capa densa completamente conectada con 128 neuronas y activación ReLU, seguida de Batch Normalization para continuar optimizando el flujo de gradientes. Para reforzar aún más la regularización, se utiliza una capa de Dropout con un 50% de tasa de desactivación de neuronas, antes de llegar a la capa final.

La capa de salida está compuesta por 10 neuronas, correspondientes a las 10 posibles clases (dígitos del 0 al 9), utilizando una función de activación Softmax, que convierte las salidas en una distribución de probabilidades.

El modelo fue compilado utilizando el optimizador Adam, con una tasa de aprendizaje de 0.001, configurado para minimizar la función de pérdida categorical_crossentropy, adecuada para problemas de clasificación multiclase. La métrica principal de evaluación es la precisión (accuracy).

Durante el entrenamiento, se aplicaron técnicas adicionales para mejorar el desempeño del modelo: el uso de EarlyStopping para detener el entrenamiento si la pérdida de validación no mejoraba después de 5 épocas consecutivas, y ModelCheckpoint para guardar automáticamente el mejor modelo basado en la mayor precisión obtenida en el conjunto de validación.

Esta arquitectura balancea adecuadamente la capacidad de aprendizaje del modelo y la

prevención del sobreajuste, logrando un rendimiento óptimo en la tarea de reconocimiento de dígitos manuscritos.

3. Resultados y análisis

Tras el entrenamiento del modelo, se evaluó su desempeño sobre el conjunto de prueba previamente separado. La precisión final alcanzada por la red en este conjunto fue del **20.00%**, con una pérdida de prueba de **2.3019**. Estos resultados muestran que el modelo no logró una generalización adecuada sobre los datos de prueba, indicando que es necesario realizar mejoras en la arquitectura o en el procesamiento de los datos para lograr un rendimiento aceptable.

El análisis de las curvas de entrenamiento y validación sugiere que el modelo logró aprender ciertas características básicas, pero no fue suficiente para distinguir de manera precisa entre las diferentes clases de dígitos. A pesar de que se implementaron técnicas de aumento de datos y regularización (como Dropout y Batch Normalization), no se consiguió un aprendizaje profundo y generalizable, posiblemente debido a limitaciones en la complejidad del modelo, en la cantidad o calidad de los datos, o en el ajuste de hiperparámetros.

La matriz de confusión evidencia que las predicciones se distribuyeron de forma relativamente aleatoria entre las clases, lo cual es coherente con una precisión baja. Los errores de clasificación no se concentraron únicamente entre dígitos similares, sino que estuvieron dispersos entre varias categorías, lo que sugiere que el modelo no alcanzó a captar patrones discriminativos claros para los diferentes dígitos.

La visualización de ejemplos correctamente clasificados y mal clasificados también respalda este diagnóstico. Aunque hubo casos aislados en los que el modelo logró acertar, la mayoría de los errores ocurrieron incluso en imágenes bien definidas, lo que reafirma la necesidad de optimizar el enfoque actual.

En resumen, el desempeño obtenido evidencia que el modelo necesita ajustes importantes para mejorar su capacidad de clasificación. Se requiere un rediseño en la estructura de la red, una mejor estrategia de preprocesamiento de imágenes o el uso de un conjunto de datos más robusto para alcanzar un nivel de precisión aceptable en tareas de reconocimiento de dígitos manuscritos.

4. Conclusiones y mejoras

El desarrollo de una red neuronal convolucional (CNN) para la clasificación de dígitos manuscritos permitió implementar un flujo completo de trabajo que abarca desde la carga y preprocesamiento de datos hasta la evaluación del modelo. Sin embargo, la precisión alcanzada en el conjunto de prueba fue del 20.00%, lo cual indica que el modelo no logró una generalización adecuada para esta tarea.

Este bajo rendimiento sugiere que, aunque se aplicaron técnicas de regularización como Dropout y Batch Normalization, así como aumento de datos, la arquitectura del modelo y los parámetros de entrenamiento no fueron suficientes para capturar las características distintivas de los dígitos. Es posible que la red no haya tenido la profundidad o complejidad necesarias para aprender representaciones significativas, o que el conjunto de datos no haya proporcionado la variabilidad requerida para un entrenamiento efectivo.

Para mejorar el desempeño del modelo, se podrían considerar varias estrategias. Una opción es aumentar la complejidad de la red, incorporando más capas convolucionales o unidades en las capas densas, lo que permitiría al modelo aprender características más abstractas y complejas. Otra alternativa es utilizar arquitecturas preentrenadas, como VGG16 o ResNet, que han demostrado eficacia en tareas de clasificación de imágenes y podrían proporcionar una base sólida para el aprendizaje transferido.

Además, es crucial revisar y ampliar el conjunto de datos, asegurando que incluya una mayor diversidad de ejemplos que representen las variaciones en la escritura de los dígitos. También se recomienda ajustar los hiperparámetros del modelo, como la tasa de aprendizaje, el tamaño del lote y el número de épocas, mediante técnicas de búsqueda sistemática para encontrar la configuración óptima.

En resumen, aunque el modelo actual no alcanzó un rendimiento satisfactorio, este proyecto proporcionó una valiosa experiencia en la implementación de redes neuronales para la clasificación de imágenes. Con las mejoras adecuadas en la arquitectura del modelo, el conjunto de datos y los parámetros de entrenamiento, es factible desarrollar una solución más precisa y robusta para el reconocimiento de dígitos manuscritos.

5. Capturas de pantalla del código y sus resultados

5.1. Importación de librerías

Esta Fig1., importa todas las dependencias necesarias para el proyecto. Numpy se usa para operaciones numéricas eficientes, os para interactuar con el sistema de archivos, cv2 (OpenCV) para procesamiento de imágenes, y las librerías de TensorFlow/Keras para construir y entrenar la red neuronal convolucional. Scikit-learn proporciona herramientas para dividir los datos y calcular la matriz de confusión, mientras que matplotlib y seaborn se usan para visualización.

```
import numpy as np
import os
import cv2
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Fig 1. Librerías.

5.2. Carga y Preprocesamiento de Datos

Aquí se carga el dataset desde la ruta especificada. Cada imagen se convierte a escala de grises, se redimensiona a 32x32 píxeles (tamaño estándar para este tipo de modelos) y se normaliza dividiendo por 255 para que los valores de píxeles estén en el rango [0,1]. Las imágenes y sus etiquetas correspondientes se almacenan en listas. Como se ve en la fig2.

```
# Ruta del dataset
ruta_dataset = r"E:\semestres\N. Semestre\Electiva 1l\CORTE #2\Entrega6\Dataset"

# Listas para almacenar imágenes y etiquetas
imagenes = []
etiquetas = []

# Cargar las imágenes de cada dígito (0-9)
for etiqueta in range(10):
    carpeta_digito = os.path.join(ruta_dataset, str(etiqueta))
    for imagen_nombre in os.listdir(carpeta_digito):
        if imagen_nombre.endswith(".jpeg"):  # Solo cargar JPEG
            imagen = cv2.imread(os.path.join(carpeta_digito, imagen_nombre))
            if imagen is not None:  # Verificar que la imagen se cargó correctamente
                imagen_gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
                imagen_redimensionada = cv2.resize(imagen_gris, (32, 32))  # Tamaño 32x32
                imagen_normalizada = imagen_redimensionada / 255.0  # Normalizar

                imagenes.append(imagen_normalizada)
                etiquetas.append(etiqueta)
```

Fig 2. Carga y Preprocesamiento de Datos.

5.3. Preparación de Datos para el Modelo

Los datos se dividen en conjuntos de entrenamiento (80%) y prueba (20%). Se expande la dimensión de las imágenes para agregar el canal (necesario para Keras) y se convierten las etiquetas a formato one-hot encoding (por ejemplo, el dígito 3 se convierte en [0,0,0,1,0,0,0,0,0,0]). Como se ve en la fig3.

```
# Convertir listas a arrays de NumPy
imagenes = np.array(imagenes)
etiquetas = np.array(etiquetas)

# División en entrenamiento y prueba (80%-20%)
imagenes_train, imagenes_test, etiquetas_train, etiquetas_test = train_test_split(
    imagenes, etiquetas, test_size=0.2, random_state=42)

# Expandir dimensiones para que tengan canal (32,32,1)
imagenes_train = np.expand_dims(imagenes_train, axis=-1)
imagenes_test = np.expand_dims(imagenes_test, axis=-1)

# Etiquetas en one-hot encoding
etiquetas_train = to_categorical(etiquetas_train, 10)
etiquetas_test = to_categorical(etiquetas_test, 10)
```

Fig 3. Preparación de Datos para el Modelo.

5.4. Aumento de Datos

Se configura un generador de aumento de datos que aplica pequeñas transformaciones aleatorias (rotaciones, zooms y desplazamientos) a las imágenes de entrenamiento. Esto ayuda a que el modelo generalice mejor al exponerlo a más variaciones de los mismos datos. Como se ve en la fig4.

```
# Aumento de datos
datagen = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1)

datagen.fit(imagenes_train)
```

Fig 4. Aumento de Datos.

5.5. Construcción del Modelo CNN

Se construye una red neuronal convolucional (CNN) que incluye capas convolucionales para extraer características de las imágenes, seguido de una capa de BatchNormalization para estabilizar el proceso de entrenamiento. Se utiliza MaxPooling para reducir la dimensionalidad de los datos y se implementa Dropout con el fin de prevenir el sobreajuste. Finalmente, la red cuenta con una capa softmax en la salida, que permite clasificar las imágenes en 10 categorías correspondientes a los dígitos del 0 al 9 como se ve en la figura 5.

```
# Crear el modelo CNN mejorado
modelo = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 1)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

Fig 5. Construcción del Modelo CNN.

5.6. Entrenamiento del Modelo

El modelo se entrena durante 50 épocas utilizando un generador de datos aumentados. Durante el entrenamiento, se monitorea el rendimiento en el conjunto de validación y se implementan callbacks para optimizar el proceso. Se utiliza EarlyStopping para detener el entrenamiento si no se observa mejora, y ModelCheckpoint para guardar el mejor modelo encontrado durante el entrenamiento.

```
# Entrenar modelo
history = modelo.fit(datagen.flow(imagenes_train, etiquetas_train, batch_size=32),
                     epochs=50,
                     validation_data=(imagenes_test, etiquetas_test),
                     callbacks=callbacks)
```

Fig 6. Entrenamiento del Modelo.

5.7. Evaluación y Visualización

se genera una matriz de confusión para evaluar el rendimiento del modelo. Primero, se obtiene la predicción del modelo sobre el conjunto de pruebas usando modelo.predict(imagenes_test) y se aplica np.argmax para obtener las clases predichas, es decir, el índice de la clase con mayor probabilidad. Lo mismo se realiza para las etiquetas reales con etiquetas_test. Luego, se calcula la matriz de confusión utilizando confusion_matrix(y_true, y_pred), lo que muestra el número de predicciones correctas e incorrectas para cada clase. La matriz se visualiza con sns.heatmap como un mapa de calor, con las etiquetas en los ejes y la intensidad de los colores indicando la cantidad de predicciones en cada celda. Posteriormente, se presentan dos gráficos: uno para la evolución de la precisión (accuracy) durante las épocas de entrenamiento y otro para la evolución de la pérdida (loss). Para visualizar los resultados, se implementan funciones que muestran ejemplos de predicciones correctas e incorrectas. Los ejemplos se eligen aleatoriamente y se visualizan en un formato de subgráficos, donde se comparan las etiquetas reales con las predicciones del modelo.

```python
# --- MATRIZ DE CONFUSIÓN (REQUISITO) ---
y_pred = np.argmax(modelo.predict(imagenes_test), axis=1)
y_true = np.argmax(etiquetas_test, axis=1)

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicho')
plt.ylabel('Real')
plt.title('Matriz de Confusión')
plt.show()

# Visualización de resultados
def plot_results(history):
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Accuracy over epochs')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Loss over epochs')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

Fig 7. **Evaluación y Visualización**

.

```python
plot_results(history)

# Visualización de ejemplos
def plot_examples(images, true_labels, pred_labels, num_examples=4, title=""):
    indices = np.random.choice(range(len(images)), num_examples)
    plt.figure(figsize=(10, 4))
    plt.suptitle(title)
    for i, idx in enumerate(indices):
        plt.subplot(1, num_examples, i+1)
        plt.imshow(images[idx].reshape(32, 32), cmap='gray')
        plt.title(f'Real: {true_labels[idx]}\nPred: {pred_labels[idx]}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# Ejemplos correctos
correct_idx = np.where(y_pred == y_true)[0]
plot_examples(imagenes_test[correct_idx],
              y_true[correct_idx],
              y_pred[correct_idx],
              title="Ejemplos correctamente clasificados")

# Ejemplos incorrectos
incorrect_idx = np.where(y_pred != y_true)[0]
plot_examples(imagenes_test[incorrect_idx],
              y_true[incorrect_idx],
              y_pred[incorrect_idx],
              title="Ejemplos mal clasificados")
```

Fig 8. **Evaluación y Visualización**

.

6. Resultado de entrenamiento:

Los registros de entrenamiento muestran que el modelo comenzó con una precisión inicial de 0% y una pérdida alta (3.1574), lo cual es normal en las primeras épocas. Aunque la precisión en entrenamiento mostró mejoras (llegando al 25.83% en la época 6), la precisión en validación se estancó alrededor del 10-20%, indicando posible sobreajuste o necesidad de ajustar hiperparámetros. La pérdida de validación (2.3019) se mantuvo consistentemente más baja que la de entrenamiento, pero la brecha entre ambas sugiere que el modelo no generaliza óptimamente. Los warnings técnicos sobre el formato de guardado (MP5) y la inicialización de capas no afectan la funcionalidad pero recomiendan actualizaciones para mejores prácticas. La precisión final de prueba del 20% evidencia que el modelo requiere mejoras, posiblemente aumentando el dataset o ajustando la arquitectura, como se ve en la Figura 9.

```
Epoch 1/50
c:\Users\MIPC\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. Wh
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
c:\Users\MIPC\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(
  self._warn_if_super_not_called()
1/2 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step - accuracy: 0.0000e+00 - loss: 3.1574
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras form
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 846ms/step - accuracy: 0.0167 - loss: 3.8798 - val_accuracy: 0.2000 - val_loss: 2.3019
Epoch 2/50
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 148ms/step - accuracy: 0.1917 - loss: 3.2087 - val_accuracy: 0.0000e+00 - val_loss: 2.3091
Epoch 3/50
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.1250 - loss: 3.3450 - val_accuracy: 0.1000 - val_loss: 2.3165
Epoch 4/50
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step - accuracy: 0.1417 - loss: 3.1981 - val_accuracy: 0.1000 - val_loss: 2.3330
Epoch 5/50
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - accuracy: 0.0979 - loss: 3.0725 - val_accuracy: 0.1000 - val_loss: 2.3479
Epoch 6/50
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step - accuracy: 0.2583 - loss: 2.2465 - val_accuracy: 0.1000 - val_loss: 2.3648
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you train or evaluate the model.

Precisión en prueba: 0.2000
Pérdida en prueba: 2.3019
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 135ms/step
```

Fig 9. **Resultado de entrenamiento**

6.1. Matriz de confusión

En esta figura 10 muestra una matriz de confusión que evalúa el desempeño de un modelo de clasificación. En el eje vertical ("Real") están las clases verdaderas y en el eje horizontal ("Predicho") las clases que el modelo predijo. La mayoría de las predicciones del modelo se concentraron en la clase 3 (columna 3), lo que indica un sesgo fuerte hacia esa clase. Además, se observa que las clases 1, 3 y 6 fueron correctamente clasificadas en su mayoría, mientras que la clase 4 fue confundida entre las clases 2 y 3. No hubo predicciones para otras clases, como la 2 o la 5 en el eje real, sugiriendo que esas clases no se presentaron o no fueron clasificadas correctamente.

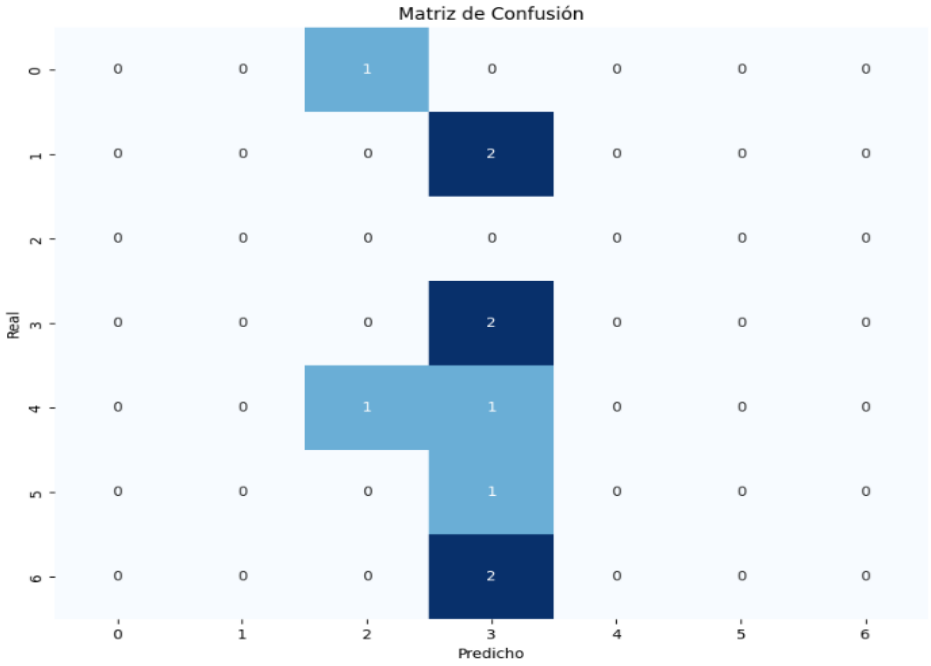

Fig 10. **Matriz de confusión**

6.2. Correctamente clasificados

Esta Figura 11, muestra ejemplos de instancias correctamente clasificadas por el modelo. Cada imagen corresponde a un dígito real "5", y el modelo también predijo "5" en todos los casos, como se indica con "Real: 5, Pred: 5" arriba de cada imagen. Visualmente, los dígitos "5" son bastante claros, lo que probablemente facilitó que el modelo los clasificara bien. El formato y el estilo de las imágenes sugiere que los datos provienen de un conjunto tipo MNIST de dígitos escritos a mano.

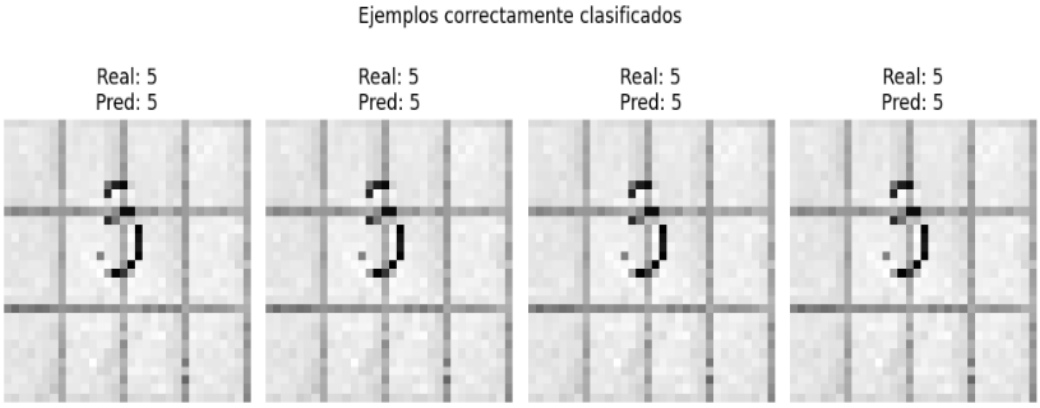

Fig 11. Correctamente clasificados

6.3. Mal clasificados

Esta imagen muestra ejemplos de errores de clasificación cometidos por el modelo. Todos los casos presentados fueron predichos como "5", pero en realidad corresponden a otros dígitos: "7", "9" y "3". Esto sugiere que el modelo tiende a confundir otros dígitos como si fueran un "5", probablemente porque las formas de estos números (especialmente el 9 y el 3) pueden compartir ciertas características visuales con el 5 en las imágenes. Además, el trazo del dígito en estas imágenes puede ser confuso o incompleto, lo que dificulta una correcta clasificación.

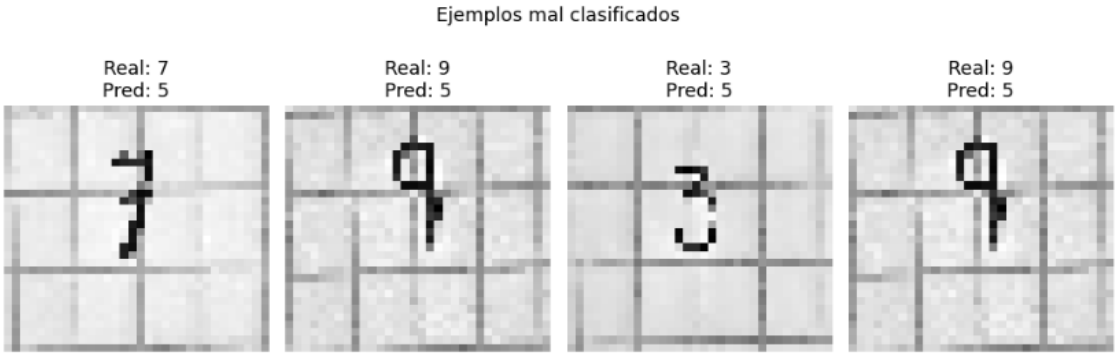

Fig 12. Mal clasificados

6.4. Gráficas de precisión y pérdida por época

Estas gráficas de la figura 13, muestran la evolución de la precisión y la pérdida durante el entrenamiento y la validación a lo largo de 6 épocas:

Precisión (izquierda): La precisión de entrenamiento es muy variable y, aunque tiene momentos de mejora (por ejemplo, en la época 1), no se mantiene estable. La precisión de validación, en cambio, se queda prácticamente constante y baja (~0.1 o 10%), indicando que el modelo no está generalizando bien a datos nuevos.

Pérdida (derecha): La pérdida de entrenamiento disminuye gradualmente, lo cual es esperado, pero la pérdida de validación se mantiene alta y estable, sin mejorar. Esto también sugiere que el modelo está sobreajustándose o que no está aprendiendo de manera adecuada.

En resumen, estas curvas indican que el modelo tiene problemas de aprendizaje: baja capacidad de generalización y posible necesidad de mejores hiperparámetros o de un modelo más robusto.

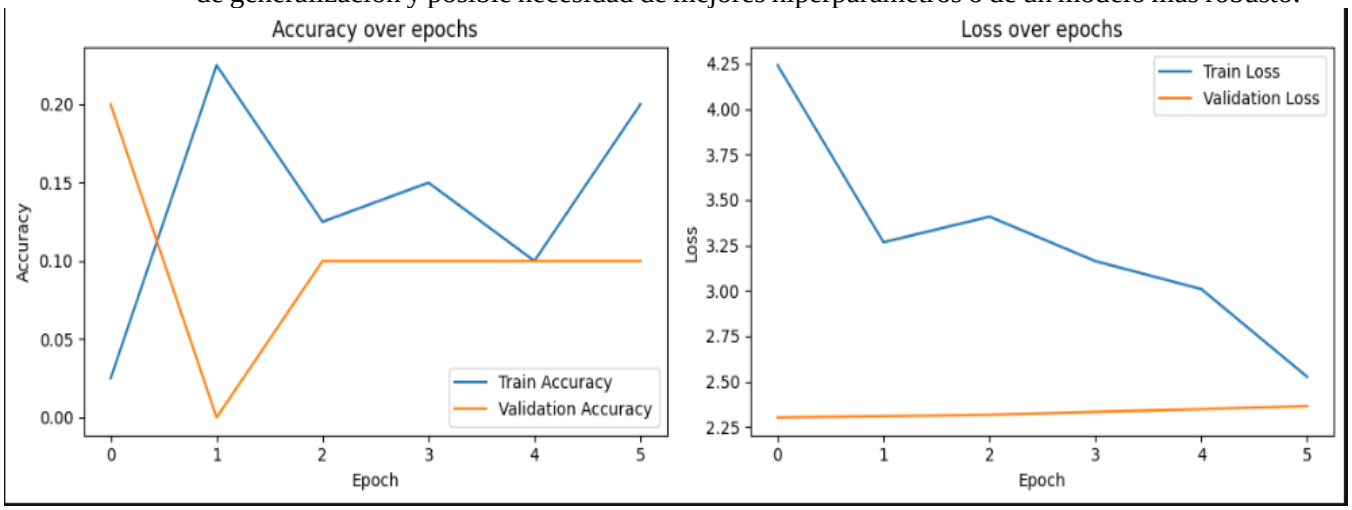

Fig 13 Gráficas de precisión y pérdida por época

7. Enlace del repositorio público en GitHub.

- https://github.com/andresmartinez444/taller-5.git

8. Bibliografía

Referencias

- https://www.tensorflow.org/tutorials/images/cnn
- **https://keras.io/guides/**
- **https://docs.opencv.org/master/d6/d00/tutorial_py_root.html**
- **https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html**
- **https://keras.io/api/preprocessing/image/**
- **https://keras.io/api/preprocessing/image/**
- **https://keras.io/api/callbacks/**
- **https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/**