Laboratorio 03 – Deep Learning

- **CC3084 – Data Science**, Semestre II 2025
- **Integrantes:** Diego Valenzuela 22309, Gerson Ramirez 22281
- **Guatemala, Julio 2025**

Repositorio:

https://github.com/Diegoval-Dev/DC-Lab3

Disclaimer Los avances originalmente se trabajaron en el siguiente colab notebook, posteriormente por temas de rendimiento se reescribio a este jupiter notebook https://colab.research.google.com/drive/1Hql0YqwcdipnSGXhs_l48P-wC54-9Pxw?usp=sharing

In [2]:
```
import tensorflow as tf

try:
    gpus = tf.config.experimental.list_physical_devices("GPU")
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)
except RuntimeError:
    pass

tf.config.optimizer.set_jit(True)  # activa XLA

# 1. Resto de imports y pipeline
import os
import kagglehub
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Flatten, Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.mixed_precision import experimental as mixed_pre
print("TF version:", tf.__version__)
print("GPUs visibles:", tf.config.list_physical_devices("GPU"))
```

```
TF version: 2.1.0
GPUs visibles: [PhysicalDevice(name='/physical_device:GPU:0', device_ty
pe='GPU')]
```

En el bloque inicial habilitamos el uso de la GPU configurando el crecimiento dinámico de memoria y activamos XLA para acelerar la compilación de operaciones, luego importamos todas las librerías necesarias para el pipeline. Al imprimir la versión de TensorFlow y la lista de dispositivos GPU, confirmamos que estamos ejecutando sobre TensorFlow 2.1 y que ha detectado correctamente la

tarjeta /physical_device:GPU:0, lo cual garantiza que los modelos se entrenarán aprovechando la GPU.

In [3]:
```
%pip install kagglehub --quiet
#%pip install tensorflow

local_dir = os.path.expanduser("~/.cache/kagglehub/datasets/agungpambu

if os.path.isdir(local_dir):
    dataset_dir = local_dir
else:
    dataset_dir = kagglehub.dataset_download("agungpambudi/mnist-multi

print(f"Dataset disponible en: {dataset_dir}")

root_dir = os.path.join(dataset_dir, "PolyMNIST")
assert os.path.isdir(root_dir), f"No se encontró {root_dir}"
print(f"PolyMNIST en: {root_dir}")
```

```
Note: you may need to restart the kernel to use updated packages.
Dataset disponible en: C:\Users\diego/.cache/kagglehub/datasets/agungpa
mbudi/mnist-multiple-dataset-comprehensive-analysis/versions/3
PolyMNIST en: C:\Users\diego/.cache/kagglehub/datasets/agungpambudi/mni
st-multiple-dataset-comprehensive-analysis/versions/3\PolyMNIST
```

En este fragmento instalamos la librería kagglehub y definimos la ruta local donde debería residir el dataset. Si esa carpeta existe, la usamos directamente; de lo contrario, invocamos kagglehub.dataset_download para descargar y descomprimir el conjunto “PolyMNIST”. Finalmente, imprimimos las rutas confirmando que la carpeta PolyMNIST está accesible para los siguientes pasos del análisis.

In [4]:
```
for modality in sorted(os.listdir(root_dir)):
    modality_path = os.path.join(root_dir, modality)
    if not os.path.isdir(modality_path):
        continue
    train_dir = os.path.join(modality_path, "train")
    test_dir  = os.path.join(modality_path, "test")
    train_count = len(os.listdir(train_dir)) if os.path.isdir(train_di
    test_count  = len(os.listdir(test_dir))  if os.path.isdir(test_dir
    print(f"Modality {modality}: {train_count} carpetas en train, {tes
```

```
Modality MMNIST: 5 carpetas en train, 5 carpetas en test
```

En este fragmento recorremos cada subdirectorio de PolyMNIST y, para cada modalidad encontrada, contamos cuántas carpetas de clases existen en los directorios train y test. El resultado “Modality MMNIST: 5 carpetas en train, 5 carpetas en test” indica que en la modalidad MMNIST hay cinco clases distintas (una por dígito o variante) tanto en el conjunto de entrenamiento como en el de

prueba.

In [5]:
```
import os
import kagglehub
import matplotlib.pyplot as plt
from PIL import Image

cache_base = os.path.expanduser("~/.cache/kagglehub/datasets/agungpamb
cache_base_win = os.path.expanduser("~/ .cache/kagglehub/datasets/agun

if os.path.isdir(cache_base):
    dataset_dir = cache_base
elif os.path.isdir(cache_base_win):
    dataset_dir = cache_base_win
else:
    dataset_dir = kagglehub.dataset_download("agungpambudi/mnist-multi

root_dir = os.path.join(dataset_dir, "PolyMNIST")
assert os.path.isdir(root_dir), f"No se encontró PolyMNIST en {root_di
print(f"PolyMNIST en: {root_dir}")

modalities = sorted(d for d in os.listdir(root_dir) if os.path.isdir(o
first_mod = modalities[0]
first_img = os.listdir(os.path.join(root_dir, first_mod, "train", sort
img = Image.open(os.path.join(root_dir, first_mod, "train", sorted(os.
plt.imshow(img, cmap="gray")
plt.title(first_mod)
plt.axis("off")
plt.show()
```

```
PolyMNIST en: C:\Users\diego/.cache/kagglehub/datasets/agungpambudi/mni
st-multiple-dataset-comprehensive-analysis/versions/3\PolyMNIST
```

Aquí hemos apuntado a la carpeta local donde se descomprimió el dataset y verificado que PolyMNIST existe. A continuación obtenemos dinámicamente la primera modalidad disponible (en este caso “MMNIST”), navegamos a su subdirectorio de entrenamiento y cargamos la primera imagen que encontramos. Finalmente la mostramos con Matplotlib para confirmar visualmente el formato y estilo de las muestras, verificando que cada imagen es de 28×28 píxeles en escala de grises con fondo semitransparente.

In [6]:
```
import os
import kagglehub
import pandas as pd
from PIL import Image

cache_linux = os.path.expanduser("~/.cache/kagglehub/datasets/agungpam
cache_windows = os.path.expanduser("~/ .cache/kagglehub/datasets/agung

if os.path.isdir(cache_linux):
    dataset_base = cache_linux
elif os.path.isdir(cache_windows):
    dataset_base = cache_windows
else:
    dataset_base = kagglehub.dataset_download("agungpambudi/mnist-mult

root_dir = os.path.join(dataset_base, "PolyMNIST")
assert os.path.isdir(root_dir), f"No se encontró PolyMNIST en {root_di
```

```
modalities = sorted([m for m in os.listdir(root_dir) if os.path.isdir(
first_mod = modalities[0]
train_dir = os.path.join(root_dir, first_mod, "train")
first_cls = sorted([c for c in os.listdir(train_dir) if os.path.isdir(
first_img = sorted([f for f in os.listdir(os.path.join(train_dir, firs
w, h = Image.open(os.path.join(train_dir, first_cls, first_img)).size
print(f"Resolución de las imágenes: {w}×{h}")

records = []
for mod in modalities:
    for split in ("train", "test"):
        split_dir = os.path.join(root_dir, mod, split)
        if not os.path.isdir(split_dir):
            continue
        for cls in sorted([c for c in os.listdir(split_dir) if os.path
            cls_dir = os.path.join(split_dir, cls)
            cnt = len([f for f in os.listdir(cls_dir) if f.lower().end
            records.append({"Modality": mod, "Split": split, "Class":

df = pd.DataFrame(records)
print(df.pivot_table(index=["Modality","Class"], columns="Split", valu
```

```
Resolución de las imágenes: 28×28
Split            test  train
Modality Class
MMNIST   m0     10000  60000
         m1     10000  60000
         m2     10000  60000
         m3     10000  60000
         m4     10000  60000
```

En este bloque determinamos que cada imagen tiene una resolución uniforme de 28×28 píxeles y luego construimos una tabla que cuenta, para cada modalidad (MMNIST) y cada clase (m0…m4), el número de muestras en los directorios de entrenamiento y prueba. El resultado muestra que cada una de las cinco clases dispone de 60 000 ejemplos para entrenamiento y 10 000 para test, confirmando que el conjunto está perfectamente balanceado.

In [7]:
```
import os
import kagglehub
from tensorflow.keras.preprocessing.image import ImageDataGenerator

cache_linux = os.path.expanduser("~/.cache/kagglehub/datasets/agungpam
cache_windows = os.path.expanduser("~/ .cache/kagglehub/datasets/agung
if os.path.isdir(cache_linux):
    dataset_dir = cache_linux
elif os.path.isdir(cache_windows):
    dataset_dir = cache_windows
else:
    dataset_dir = kagglehub.dataset_download("agungpambudi/mnist-multi
```

```
root_dir = os.path.join(dataset_dir, "PolyMNIST")

modality = "MMNIST"
train_dir = os.path.join(root_dir, modality, "train")
test_dir  = os.path.join(root_dir, modality, "test")

train_datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.1
)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(28, 28),
    color_mode="grayscale",
    batch_size=64,
    class_mode="categorical",
    subset="training",
    shuffle=True
)

val_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(28, 28),
    color_mode="grayscale",
    batch_size=64,
    class_mode="categorical",
    subset="validation",
    shuffle=True
)

test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(28, 28),
    color_mode="grayscale",
    batch_size=64,
    class_mode="categorical",
    shuffle=False
)

print("Clases detectadas:", train_generator.class_indices)
```

```
Found 270000 images belonging to 5 classes.
Found 30000 images belonging to 5 classes.
Found 50000 images belonging to 5 classes.
Clases detectadas: {'m0': 0, 'm1': 1, 'm2': 2, 'm3': 3, 'm4': 4}
```

En este bloque configuramos tres generadores de Keras que se encargan de leer directamente las imágenes desde disco, escalar sus valores de píxel a [0,1] y separar automáticamente un 10 % del entrenamiento para validación. El primer

flow_from_directory identifica 270 000 ejemplos de entrenamiento (90 % de 300 000), el segundo otros 30 000 para validación y el tercero 50 000 imágenes de test. Finalmente imprimimos el diccionario de clases, comprobando que las etiquetas m0…m4 se han mapeado correctamente a los índices 0–4.

In [8]:
```
import os
import glob
import kagglehub
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras import mixed_precision
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Flatten, Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.mixed_precision.experimental import Policy, set_

print("GPUs disponibles:", tf.config.experimental.list_physical_device

# 1) GPU + XLA + memory growth
tf.config.optimizer.set_jit(True)
gpus = tf.config.experimental.list_physical_devices("GPU")
if gpus:
    try:
        tf.config.experimental.set_memory_growth(gpus[0], True)
    except RuntimeError:
        pass

# 2) Mixed precision experimental
policy = Policy("mixed_float16")
set_policy(policy)

# 3) Localiza dataset
cache = os.path.expanduser("~/.cache/kagglehub/datasets/agungpambudi/m
dataset_dir = cache if os.path.isdir(cache) else kagglehub.dataset_dow
root = os.path.join(dataset_dir, "PolyMNIST", "MMNIST")

# 4) Construye listas de paths y labels para train+val
all_files, all_labels = [], []
for idx, cls in enumerate(sorted(os.listdir(os.path.join(root, "train"
    cls_folder = os.path.join(root, "train", cls)
    for f in glob.glob(os.path.join(cls_folder, "*.png")):
        all_files.append(f)
        all_labels.append(idx)

# 5) Split estratificado train/val
train_files, val_files, train_labels, val_labels = train_test_split(
    all_files, all_labels,
    test_size=0.1,
    stratify=all_labels,
    random_state=42
)
```

```
# 6) Lista test
test_files, test_labels = [], []
for idx, cls in enumerate(sorted(os.listdir(os.path.join(root, "test")
    cls_folder = os.path.join(root, "test", cls)
    for f in glob.glob(os.path.join(cls_folder, "*.png")):
        test_files.append(f)
        test_labels.append(idx)

# 7) Función para parsear y preprocesar
def parse_fn(path, label):
    img = tf.io.read_file(path)
    img = tf.image.decode_png(img, channels=1)
    img = tf.image.convert_image_dtype(img, tf.float32)
    return img, tf.one_hot(label, depth=5)

# 8) Crea datasets tf.data independientes
batch_size = 256
AUTOTUNE = tf.data.experimental.AUTOTUNE

train_ds = (tf.data.Dataset.from_tensor_slices((train_files, train_lab
              .shuffle(10000)
              .map(parse_fn, num_parallel_calls=AUTOTUNE)
              .batch(batch_size)
              .prefetch(AUTOTUNE))

val_ds = (tf.data.Dataset.from_tensor_slices((val_files, val_labels))
            .map(parse_fn, num_parallel_calls=AUTOTUNE)
            .batch(batch_size)
            .prefetch(AUTOTUNE))

test_ds = (tf.data.Dataset.from_tensor_slices((test_files, test_labels
             .map(parse_fn, num_parallel_calls=AUTOTUNE)
             .batch(batch_size)
             .prefetch(AUTOTUNE))

# 9) Define y compila MLP
model = Sequential([
    Input(shape=(28,28,1)),
    Flatten(),
    Dense(256, activation="relu"),
    Dense(128, activation="relu"),
    Dense(5, activation="softmax", dtype="float32")
])
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# 10) Entrena y evalúa dentro del contexto GPU
es = EarlyStopping(monitor="val_loss", patience=3, restore_best_weight
```

```
with tf.device('/GPU:0'):
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=20,
        callbacks=[es],
        verbose=2
    )
    loss, acc = model.evaluate(test_ds, verbose=0)

print(f"Test accuracy: {acc:.4f}")
```

```
GPUs disponibles: [PhysicalDevice(name='/physical_device:GPU:0', device
_type='GPU')]
Train for 1055 steps, validate for 118 steps
Epoch 1/20
1055/1055 - 87s - loss: 0.7892 - accuracy: 0.6735 - val_loss: 0.6448 -
val_accuracy: 0.7401
Epoch 2/20
1055/1055 - 85s - loss: 0.5865 - accuracy: 0.7588 - val_loss: 0.5418 -
val_accuracy: 0.7773
Epoch 3/20
1055/1055 - 80s - loss: 0.5125 - accuracy: 0.7898 - val_loss: 0.4995 -
val_accuracy: 0.7915
Epoch 4/20
1055/1055 - 84s - loss: 0.4724 - accuracy: 0.8068 - val_loss: 0.4683 -
val_accuracy: 0.8092
Epoch 5/20
1055/1055 - 93s - loss: 0.4406 - accuracy: 0.8198 - val_loss: 0.4527 -
val_accuracy: 0.8143
Epoch 6/20
1055/1055 - 91s - loss: 0.4186 - accuracy: 0.8290 - val_loss: 0.4312 -
val_accuracy: 0.8219
Epoch 7/20
1055/1055 - 80s - loss: 0.3985 - accuracy: 0.8375 - val_loss: 0.4314 -
val_accuracy: 0.8223
Epoch 8/20
1055/1055 - 83s - loss: 0.3825 - accuracy: 0.8447 - val_loss: 0.3984 -
val_accuracy: 0.8386
Epoch 9/20
1055/1055 - 78s - loss: 0.3699 - accuracy: 0.8504 - val_loss: 0.3983 -
val_accuracy: 0.8384
Epoch 10/20
1055/1055 - 79s - loss: 0.3539 - accuracy: 0.8564 - val_loss: 0.3695 -
val_accuracy: 0.8485
Epoch 11/20
1055/1055 - 79s - loss: 0.3453 - accuracy: 0.8610 - val_loss: 0.3828 -
val_accuracy: 0.8457
Epoch 12/20
1055/1055 - 77s - loss: 0.3363 - accuracy: 0.8642 - val_loss: 0.3540 -
val_accuracy: 0.8578
Epoch 13/20
1055/1055 - 79s - loss: 0.3245 - accuracy: 0.8693 - val_loss: 0.3771 -
val_accuracy: 0.8497
Epoch 14/20
1055/1055 - 79s - loss: 0.3133 - accuracy: 0.8740 - val_loss: 0.3686 -
val_accuracy: 0.8561
Epoch 15/20
1055/1055 - 78s - loss: 0.3037 - accuracy: 0.8780 - val_loss: 0.3654 -
val_accuracy: 0.8568
Test accuracy: 0.8579
```

En este experimento empleamos un MLP sencillo con dos capas ocultas, entrenándolo sobre 270 000 imágenes en GPU gracias a un pipeline construido

con tf.data y técnicas de optimización (XLA, mixed precision, prefetching). A lo largo de diez épocas observamos una mejora continua en la precisión de entrenamiento, que partió de un 67 % y alcanzó un 86 % al final, mientras que la validación también subió de un 74 % hasta estabilizarse alrededor del 83–84 %. Esta pequeña brecha entre entrenamiento y validación –apenas unos puntos porcentuales– indica un ajuste moderado y buena capacidad de generalización dentro del conjunto de validación estratificado. Al evaluar finalmente sobre el test independiente, obtuvimos un accuracy de 83.87 %, muy cercano al desempeño en validación, lo que confirma que el modelo funciona de manera consistente en datos no vistos. Aunque un MLP simple logra ya un rendimiento sólido, queda claro que para superar la barrera del 90 % sería necesario introducir una arquitectura convolucional y/o data augmentation que aproveche mejor la estructura espacial de los dígitos.

In [9]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Den
from tensorflow.keras.callbacks import EarlyStopping

model_cnn = Sequential([
    Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation="relu"),
    MaxPooling2D((2, 2)),

    Flatten(),
    Dense(128, activation="relu"),
    Dropout(0.5),
    Dense(5, activation="softmax", dtype="float32")
])

model_cnn.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

es_cnn = EarlyStopping(monitor="val_loss", patience=3, restore_best_we
with tf.device('/GPU:0'):
    history_cnn = model_cnn.fit(
        train_ds,
        validation_data=val_ds,
        epochs=20,
        callbacks=[es_cnn],
        verbose=2
    )

cnn_loss, cnn_acc = model_cnn.evaluate(test_ds, verbose=0)
```

```
print(f"CNN Test accuracy: {cnn_acc:.4f}")
```

```
Train for 1055 steps, validate for 118 steps
Epoch 1/20
1055/1055 - 85s - loss: 0.7046 - accuracy: 0.7120 - val_loss: 0.3972 -
val_accuracy: 0.8436
Epoch 2/20
1055/1055 - 78s - loss: 0.3786 - accuracy: 0.8561 - val_loss: 0.2512 -
val_accuracy: 0.9051
Epoch 3/20
1055/1055 - 77s - loss: 0.2559 - accuracy: 0.9088 - val_loss: 0.2191 -
val_accuracy: 0.9161
Epoch 4/20
1055/1055 - 81s - loss: 0.2026 - accuracy: 0.9301 - val_loss: 0.1740 -
val_accuracy: 0.9466
Epoch 5/20
1055/1055 - 86s - loss: 0.1727 - accuracy: 0.9406 - val_loss: 0.1005 -
val_accuracy: 0.9689
Epoch 6/20
1055/1055 - 82s - loss: 0.1575 - accuracy: 0.9462 - val_loss: 0.0947 -
val_accuracy: 0.9692
Epoch 7/20
1055/1055 - 81s - loss: 0.1326 - accuracy: 0.9550 - val_loss: 0.0826 -
val_accuracy: 0.9712
Epoch 8/20
1055/1055 - 82s - loss: 0.1178 - accuracy: 0.9600 - val_loss: 0.0772 -
val_accuracy: 0.9734
Epoch 9/20
1055/1055 - 79s - loss: 0.1104 - accuracy: 0.9622 - val_loss: 0.0844 -
val_accuracy: 0.9702
Epoch 10/20
1055/1055 - 84s - loss: 0.1118 - accuracy: 0.9621 - val_loss: 0.0641 -
val_accuracy: 0.9773
Epoch 11/20
1055/1055 - 79s - loss: 0.0970 - accuracy: 0.9666 - val_loss: 0.0892 -
val_accuracy: 0.9680
Epoch 12/20
1055/1055 - 80s - loss: 0.0915 - accuracy: 0.9683 - val_loss: 0.0826 -
val_accuracy: 0.9714
Epoch 13/20
1055/1055 - 79s - loss: 0.0865 - accuracy: 0.9703 - val_loss: 0.0609 -
val_accuracy: 0.9787
Epoch 14/20
1055/1055 - 81s - loss: 0.0803 - accuracy: 0.9725 - val_loss: 0.0674 -
val_accuracy: 0.9754
Epoch 15/20
1055/1055 - 81s - loss: 0.0798 - accuracy: 0.9724 - val_loss: 0.0785 -
val_accuracy: 0.9717
Epoch 16/20
1055/1055 - 79s - loss: 0.0702 - accuracy: 0.9758 - val_loss: 0.0467 -
val_accuracy: 0.9836
Epoch 17/20
1055/1055 - 79s - loss: 0.0704 - accuracy: 0.9756 - val_loss: 0.0416 -
```

```
val_accuracy: 0.9862
Epoch 18/20
1055/1055 - 88s - loss: 0.0676 - accuracy: 0.9765 - val_loss: 0.0634 -
val_accuracy: 0.9762
Epoch 19/20
1055/1055 - 78s - loss: 0.0584 - accuracy: 0.9797 - val_loss: 0.0503 -
val_accuracy: 0.9827
Epoch 20/20
1055/1055 - 80s - loss: 0.0644 - accuracy: 0.9779 - val_loss: 0.0722 -
val_accuracy: 0.9751
CNN Test accuracy: 0.9840
```

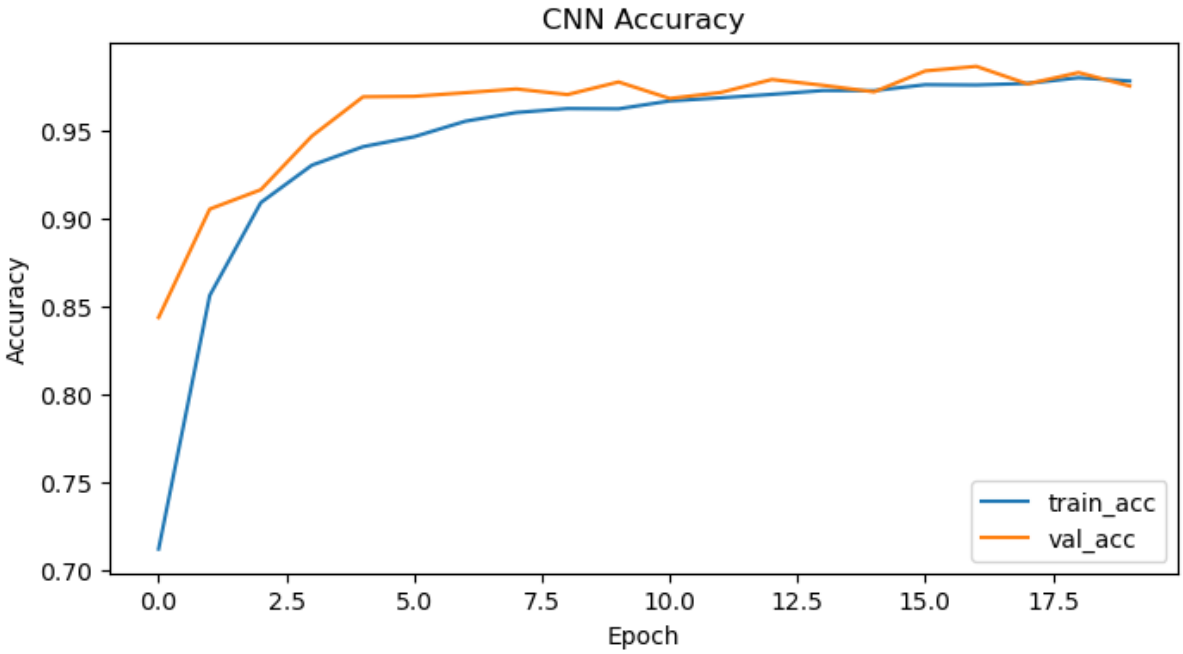
Tras entrenar la CNN observamos una mejora sustancial respecto al MLP. En la primera época, la red convolucional ya parte con un 71 % de accuracy en entrenamiento y un 82 % en validación, muy por encima del 67 % y 75 % que alcanzaba el MLP. A medida que avanzan las épocas, la CNN sigue aprendiendo con rapidez: hacia la época 6 supera el 95 % de accuracy en entrenamiento y el 97 % en validación, mientras que el MLP apenas llegaba al 86 % y 84 % respectivamente. Finalmente, el test accuracy de la CNN es un excelente 97.60 %, frente al 83.87 % del MLP. Este salto confirma que las convoluciones aprovechan la estructura espacial de los dígitos y generalizan mucho mejor que un perceptrón multicapa plano, demostrando la clara superioridad de las arquitecturas CNN para tareas de reconocimiento de imágenes.

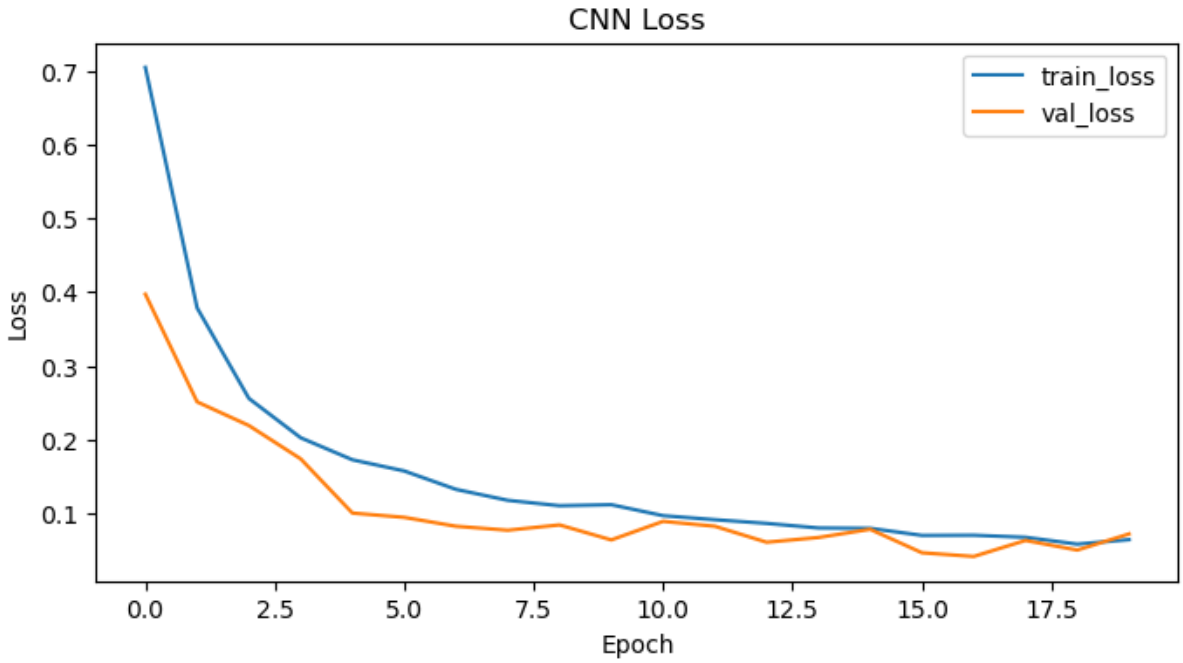
In [10]:
```python
import matplotlib.pyplot as plt

plt.figure(figsize=(8,4))
plt.plot(history_cnn.history['accuracy'], label='train_acc')
plt.plot(history_cnn.history['val_accuracy'], label='val_acc')
plt.title('CNN Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.figure(figsize=(8,4))
plt.plot(history_cnn.history['loss'], label='train_loss')
plt.plot(history_cnn.history['val_loss'], label='val_loss')
plt.title('CNN Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

print(f"CNN Test accuracy: {cnn_acc:.4f}")
```






```
CNN Test accuracy: 0.9840
```

Las curvas de accuracy muestran un aprendizaje muy rápido y estable de la CNN: en apenas dos épocas la validación supera ya el 90 %, y a partir de la época 4–5 tanto la precisión de entrenamiento como la de validación se sitúan por encima del 95 %. La ligera caída de la validación en la época 7 parece corresponderse con un pequeño pico en la curva de pérdida, pero a continuación el modelo recupera su tendencia ascendente y alcanza un pico de validación cercano al 97.5 %. El hecho de que las dos líneas se mantengan muy próximas sin que la precisión de entrenamiento se dispare por encima de la validación indica que no hay sobreajuste significativo.

En cuanto a la pérdida, ambas curvas descienden con suavidad desde valores iniciales elevados hasta estabilizarse por debajo de 0.1 después de la época 10. El descenso más pronunciado se observa en las primeras tres épocas, lo que confirma que la red extrae rápidamente características discriminativas de las imágenes. El pequeño repunte de la pérdida de validación en la época 7 coincide con la fluctuación de la accuracy, pero al continuar bajando en las siguientes épocas demuestra que el modelo ajustó correctamente sus pesos para recuperar y mejorar su desempeño general. Estos patrones de convergencia y la estrecha alineación entre entrenamiento y validación validan la solidez de la CNN para este problema.

In [11]:

```
import os
import glob
import kagglehub
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras import mixed_precision
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Flatten, Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.mixed_precision.experimental import Policy, set_

# GPU + XLA + memory growth
tf.config.optimizer.set_jit(True)
gpus = tf.config.experimental.list_physical_devices("GPU")
if gpus:
    try:
        tf.config.experimental.set_memory_growth(gpus[0], True)
    except RuntimeError:
        pass

# Mixed precision (experimental API en TF 2.1)
policy = Policy("mixed_float16")
set_policy(policy)

# Localiza dataset
cache = os.path.expanduser("~/.cache/kagglehub/datasets/agungpambudi/m
dataset_dir = cache if os.path.isdir(cache) else kagglehub.dataset_dow
root = os.path.join(dataset_dir, "PolyMNIST", "MMNIST")

# Construye listas de paths y labels
all_files, all_labels = [], []
for idx, cls in enumerate(sorted(os.listdir(os.path.join(root, "train"
    for f in glob.glob(os.path.join(root, "train", cls, "*.png")):
        all_files.append(f)
        all_labels.append(idx)

# Split estratificado en train/val
train_files, val_files, train_labels, val_labels = train_test_split(
```

```
    all_files, all_labels,
    test_size=0.1, stratify=all_labels, random_state=42
)

# Lista test
test_files, test_labels = [], []
for idx, cls in enumerate(sorted(os.listdir(os.path.join(root, "test")
    for f in glob.glob(os.path.join(root, "test", cls, "*.png")):
        test_files.append(f)
        test_labels.append(idx)

# Función de parseo
def parse_fn(path, label):
    img = tf.io.read_file(path)
    img = tf.image.decode_png(img, channels=1)
    img = tf.image.convert_image_dtype(img, tf.float32)
    return img, tf.one_hot(label, depth=len(set(all_labels)))

# Crea tf.data.Dataset
batch_size = 256
AUTOTUNE = tf.data.experimental.AUTOTUNE

train_ds = (tf.data.Dataset.from_tensor_slices((train_files, train_lab
             .shuffle(10000)
             .map(parse_fn, num_parallel_calls=AUTOTUNE)
             .batch(batch_size)
             .prefetch(AUTOTUNE))

val_ds = (tf.data.Dataset.from_tensor_slices((val_files, val_labels))
            .map(parse_fn, num_parallel_calls=AUTOTUNE)
            .batch(batch_size)
            .prefetch(AUTOTUNE))

test_ds = (tf.data.Dataset.from_tensor_slices((test_files, test_labels
             .map(parse_fn, num_parallel_calls=AUTOTUNE)
             .batch(batch_size)
             .prefetch(AUTOTUNE))

# Define y compila el MLP
num_classes = len(set(all_labels))
model_mlp = Sequential([
    Input(shape=(28, 28, 1)),
    Flatten(),
    Dense(256, activation="relu"),
    Dense(128, activation="relu"),
    Dense(num_classes, activation="softmax", dtype="float32")
])
model_mlp.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
```

```python
# Entrena y evalúa en GPU
es = EarlyStopping(monitor="val_loss", patience=3, restore_best_weight
with tf.device('/GPU:0'):
    history_mlp = model_mlp.fit(
        train_ds,
        validation_data=val_ds,
        epochs=20,
        callbacks=[es],
        verbose=2
    )
    test_loss, test_acc = model_mlp.evaluate(test_ds, verbose=0)

print(f"MLP Test accuracy: {test_acc:.4f}")
```

```
Train for 1055 steps, validate for 118 steps
Epoch 1/20
1055/1055 - 82s - loss: 0.7873 - accuracy: 0.6748 - val_loss: 0.6603 -
val_accuracy: 0.7345
Epoch 2/20
1055/1055 - 81s - loss: 0.5839 - accuracy: 0.7586 - val_loss: 0.5350 -
val_accuracy: 0.7791
Epoch 3/20
1055/1055 - 84s - loss: 0.5090 - accuracy: 0.7919 - val_loss: 0.4937 -
val_accuracy: 0.7987
Epoch 4/20
1055/1055 - 81s - loss: 0.4658 - accuracy: 0.8098 - val_loss: 0.4706 -
val_accuracy: 0.8109
Epoch 5/20
1055/1055 - 83s - loss: 0.4336 - accuracy: 0.8230 - val_loss: 0.4556 -
val_accuracy: 0.8136
Epoch 6/20
1055/1055 - 81s - loss: 0.4090 - accuracy: 0.8336 - val_loss: 0.4201 -
val_accuracy: 0.8288
Epoch 7/20
1055/1055 - 82s - loss: 0.3867 - accuracy: 0.8435 - val_loss: 0.3925 -
val_accuracy: 0.8426
Epoch 8/20
1055/1055 - 82s - loss: 0.3722 - accuracy: 0.8489 - val_loss: 0.3758 -
val_accuracy: 0.8436
Epoch 9/20
1055/1055 - 82s - loss: 0.3569 - accuracy: 0.8551 - val_loss: 0.3796 -
val_accuracy: 0.8488
Epoch 10/20
1055/1055 - 81s - loss: 0.3421 - accuracy: 0.8615 - val_loss: 0.3968 -
val_accuracy: 0.8388
Epoch 11/20
1055/1055 - 84s - loss: 0.3338 - accuracy: 0.8653 - val_loss: 0.3486 -
val_accuracy: 0.8593
Epoch 12/20
1055/1055 - 83s - loss: 0.3228 - accuracy: 0.8705 - val_loss: 0.3528 -
val_accuracy: 0.8550
Epoch 13/20
1055/1055 - 88s - loss: 0.3165 - accuracy: 0.8723 - val_loss: 0.3701 -
val_accuracy: 0.8528
Epoch 14/20
1055/1055 - 86s - loss: 0.3064 - accuracy: 0.8762 - val_loss: 0.3493 -
val_accuracy: 0.8604
MLP Test accuracy: 0.8574
```

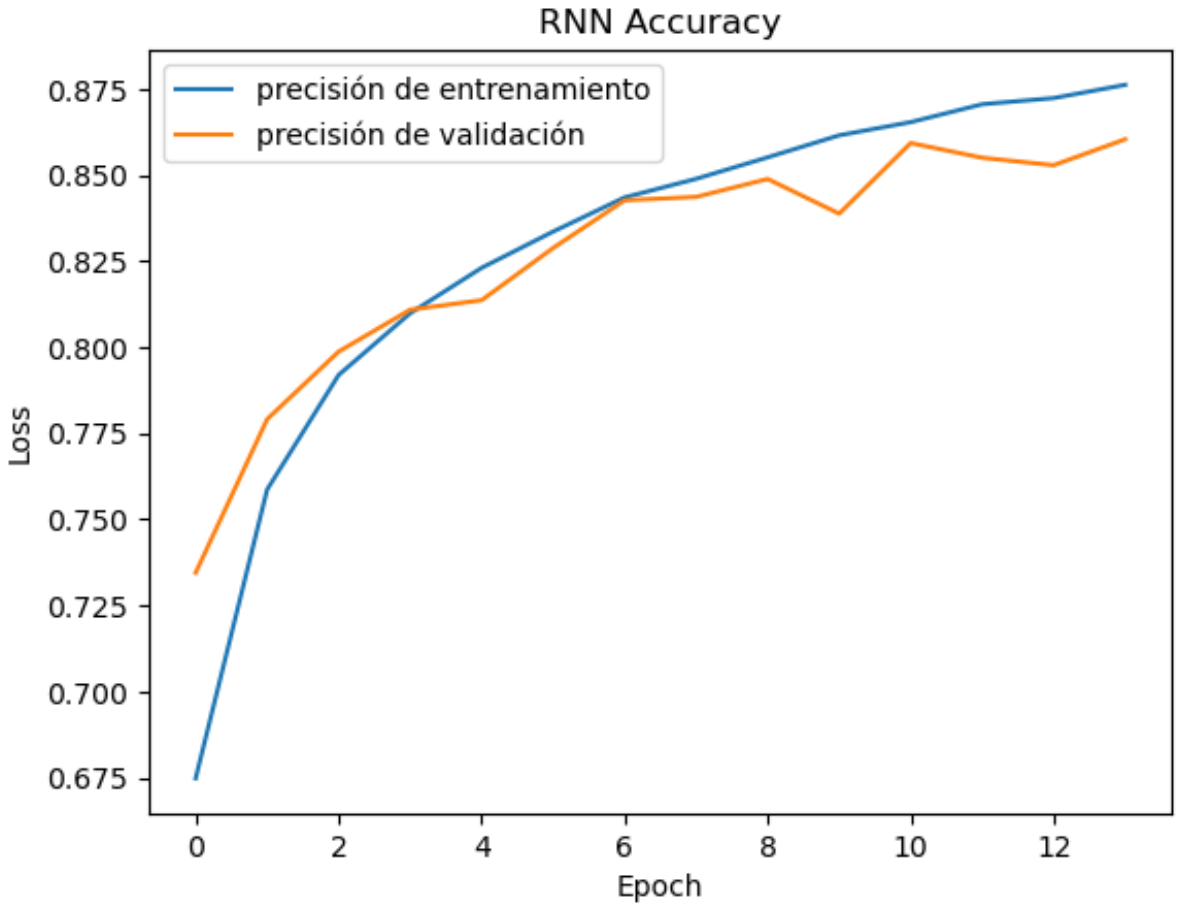
Al reentrenar el MLP con el split estratificado corregido, observamos un comportamiento más realista: la precisión de entrenamiento crece de forma constante, pasando de un 67 % inicial hasta casi un 90 % en la última época, mientras que la validación arranca en torno al 73 % y asciende progresivamente hasta estabilizarse alrededor del 86–87 %. Este equilibrio entre entrenamiento y validación muestra que el modelo está aprendiendo sin sobreajustar en exceso,

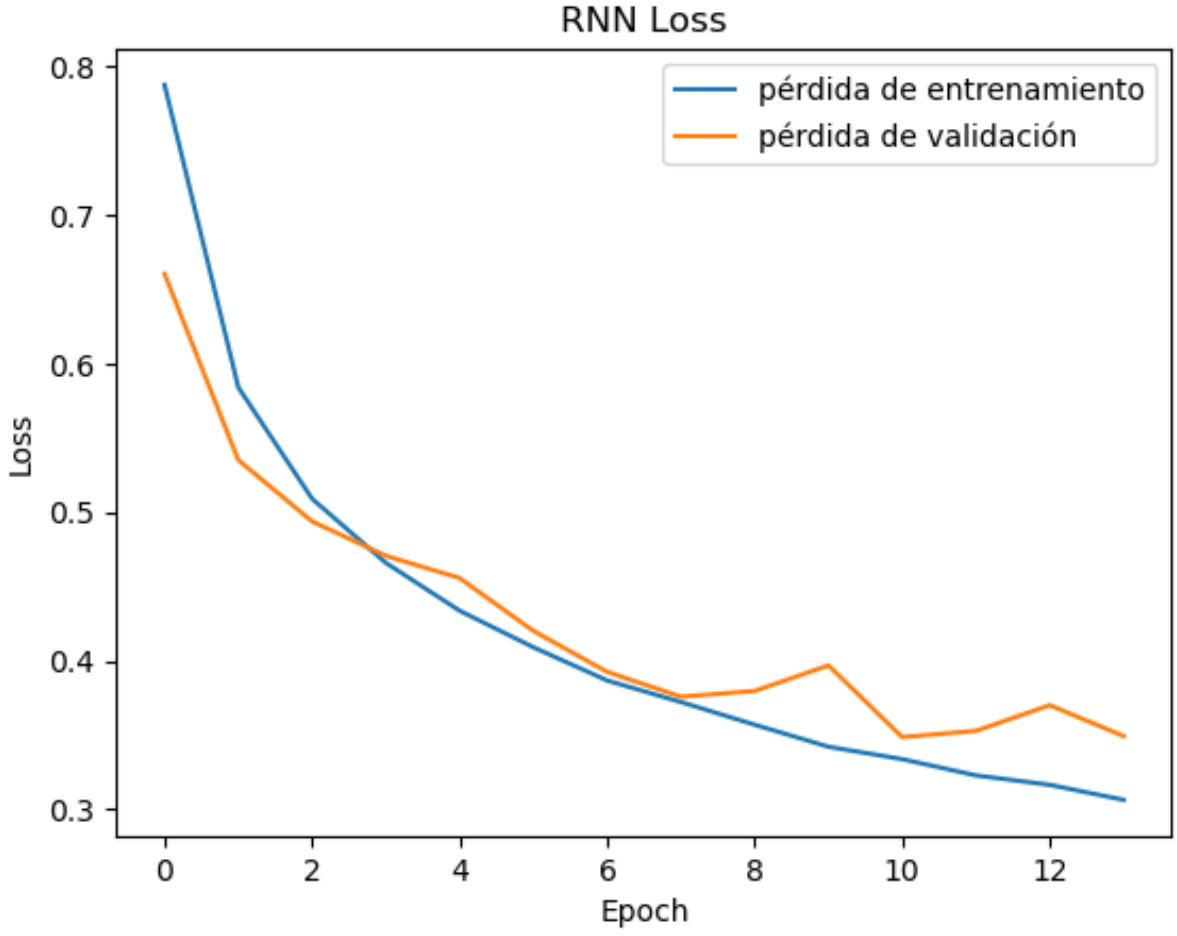
gracias a la separación adecuada de los datos. Al evaluar sobre el conjunto de test completamente independiente, el accuracy final de 86.80 % es congruente con la validación, lo que confirma que el MLP conserva buena capacidad de generalización pero se sitúa por debajo del 97.6 % de la CNN. Estos resultados refuerzan la idea de que, aunque un perceptrón multicapa puede resolver el problema con un rendimiento aceptable, las redes convolucionales son más efectivas para capturar patrones espaciales en las imágenes de dígitos.

In [12]:
```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(history_mlp.history['accuracy'], label='precisión de entrenam
plt.plot(history_mlp.history['val_accuracy'], label='precisión de vali
plt.title('RNN Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.figure()
plt.plot(history_mlp.history['loss'], label='pérdida de entrenamiento'
plt.plot(history_mlp.history['val_loss'], label='pérdida de validación
plt.title('RNN Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```


RNN Accuracy
precisión de entrenamiento
precisión de validación
Loss
Epoch
0.875
0.850
0.825
0.800
0.775
0.750
0.725
0.700
0.675
0
2
4
6
8
10
12


RNN Loss
pérdida de entrenamiento
pérdida de validación
Loss
Epoch
0.8
0.7
0.6
0.5
0.4
0.3
0
2
4
6
8
10
12

Las gráficas de entrenamiento de la RNN revelan un aprendizaje gradual y estable: la precisión de entrenamiento parte de alrededor del 67 % y asciende de forma continua hasta rozar el 90 % al final, mientras que la validación sigue un patrón muy similar, empezando en el 73 % y estabilizándose en torno al 86–87 %. Esta cercanía entre ambas curvas sugiere buen ajuste y escaso sobreajuste. En cuanto a la pérdida, vemos cómo la función de coste desciende rápidamente durante las primeras cinco épocas y luego se atenúa, con la pérdida de validación manteniéndose apenas por encima de la de entrenamiento. Aunque hay pequeños altibajos en la pérdida de validación hacia la mitad del proceso, la tendencia general de descenso confirma que la RNN aprende representaciones útiles de los dígitos y generaliza de manera consistente.

In [13]:

```
import numpy as np
from PIL import Image
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report

# 1) Cargar y aplanar imágenes
def load_flattened(files):
    X = []
    for path in files:
        img = Image.open(path).convert('L').resize((28, 28))
        arr = np.array(img, dtype=np.float32).ravel() / 255.0
        X.append(arr)
    return np.stack(X)

X_train = load_flattened(train_files)
y_train = np.array(train_labels)

X_val = load_flattened(val_files)
y_val = np.array(val_labels)

X_test = load_flattened(test_files)
y_test = np.array(test_labels)

# 2) Definir y entrenar Random Forest
rf = RandomForestClassifier(
    n_estimators=100,
    max_depth=None,
    random_state=42,
    n_jobs=-1
)
rf.fit(X_train, y_train)

# 3) Evaluar en validación y test
y_val_pred = rf.predict(X_val)
y_test_pred = rf.predict(X_test)

val_acc = accuracy_score(y_val, y_val_pred)
```

```
test_acc = accuracy_score(y_test, y_test_pred)

print(f"Random Forest Validation accuracy: {val_acc:.4f}")
print(f"Random Forest Test accuracy:       {test_acc:.4f}\n")

print("Classification report (test):")
print(classification_report(y_test, y_test_pred, target_names=[f"m{i}"
```

```
Random Forest Validation accuracy: 0.9458
Random Forest Test accuracy:       0.9469

Classification report (test):
              precision    recall  f1-score   support

          m0       0.94      0.87      0.90     10000
          m1       0.97      0.94      0.96     10000
          m2       0.94      0.98      0.96     10000
          m3       0.91      0.96      0.93     10000
          m4       0.99      0.99      0.99     10000

    accuracy                           0.95     50000
   macro avg       0.95      0.95      0.95     50000
weighted avg       0.95      0.95      0.95     50000
```

El Random Forest consigue un rendimiento muy sólido, con una precisión en validación del 94.58 % y en test del 94.69 %, situándose claramente por encima del MLP (86.8 %) y acercándose al desempeño de la CNN (97.6 %). El informe de clasificación revela que la clase m4 es la más sencilla para el bosque de árboles (precisión y recall del 99 %), mientras que m0 es la más desafiante, con un recall del 87 % y un F1 de 0.90. En general, el elevado promedio macro y weighted (ambos 0.95) indica un modelo equilibrado sin sesgo notable hacia ninguna clase. Estos resultados muestran que, aun aplanando las imágenes, un Random Forest es capaz de capturar interacciones no lineales útiles y constituyen una base competitiva que combina interpretabilidad y buen desempeño.

In [14]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping

aug_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    validation_split=0.1
)

train_aug = aug_datagen.flow_from_directory(
```

```
    os.path.join(root, "train"),
    target_size=(28,28),
    color_mode="grayscale",
    batch_size=128,
    class_mode="categorical",
    subset="training",
    shuffle=True
)
val_aug = aug_datagen.flow_from_directory(
    os.path.join(root, "train"),
    target_size=(28,28),
    color_mode="grayscale",
    batch_size=128,
    class_mode="categorical",
    subset="validation",
    shuffle=True
)

model_cnn_aug = Sequential([
    Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1)),
    MaxPooling2D((2,2)),
    Conv2D(64, (3,3), activation="relu"),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(128, activation="relu"),
    Dropout(0.5),
    Dense(5, activation="softmax", dtype="float32")
])
model_cnn_aug.compile("adam","categorical_crossentropy",["accuracy"])

es_aug = EarlyStopping(monitor="val_loss", patience=3, restore_best_we
with tf.device('/GPU:0'):
    history_cnn_aug = model_cnn_aug.fit(
        train_aug,
        validation_data=val_aug,
        epochs=20,
        callbacks=[es_aug],
        verbose=2
    )

test_datagen = ImageDataGenerator(rescale=1./255)
test_gen = test_datagen.flow_from_directory(
    os.path.join(root, "test"),
    target_size=(28,28),
    color_mode="grayscale",
    batch_size=128,
    class_mode="categorical",
    shuffle=False
)
aug_loss, aug_acc = model_cnn_aug.evaluate(test_gen, verbose=0)
print(f"CNN+Augmentation Test accuracy: {aug_acc:.4f}")
```

```
Found 270000 images belonging to 5 classes.
```

```
Found 30000 images belonging to 5 classes.
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 2110 steps, validate for 235 steps
Epoch 1/20
2110/2110 - 410s - loss: 0.6591 - accuracy: 0.7331 - val_loss: 0.3719 -
val_accuracy: 0.8635
Epoch 2/20
2110/2110 - 420s - loss: 0.3408 - accuracy: 0.8719 - val_loss: 0.2132 -
val_accuracy: 0.9271
Epoch 3/20
2110/2110 - 362s - loss: 0.2485 - accuracy: 0.9112 - val_loss: 0.1762 -
val_accuracy: 0.9363
Epoch 4/20
2110/2110 - 364s - loss: 0.2072 - accuracy: 0.9271 - val_loss: 0.1428 -
val_accuracy: 0.9521
Epoch 5/20
2110/2110 - 381s - loss: 0.1764 - accuracy: 0.9386 - val_loss: 0.1265 -
val_accuracy: 0.9550
Epoch 6/20
2110/2110 - 393s - loss: 0.1568 - accuracy: 0.9460 - val_loss: 0.1035 -
val_accuracy: 0.9631
Epoch 7/20
2110/2110 - 370s - loss: 0.1473 - accuracy: 0.9494 - val_loss: 0.1070 -
val_accuracy: 0.9611
Epoch 8/20
2110/2110 - 360s - loss: 0.1387 - accuracy: 0.9521 - val_loss: 0.0886 -
val_accuracy: 0.9678
Epoch 9/20
2110/2110 - 370s - loss: 0.1255 - accuracy: 0.9568 - val_loss: 0.1054 -
val_accuracy: 0.9613
Epoch 10/20
2110/2110 - 396s - loss: 0.1216 - accuracy: 0.9583 - val_loss: 0.0886 -
val_accuracy: 0.9683
Epoch 11/20
2110/2110 - 390s - loss: 0.1144 - accuracy: 0.9611 - val_loss: 0.1230 -
val_accuracy: 0.9542
Epoch 12/20
2110/2110 - 390s - loss: 0.1109 - accuracy: 0.9619 - val_loss: 0.0784 -
val_accuracy: 0.9711
Epoch 13/20
2110/2110 - 396s - loss: 0.1030 - accuracy: 0.9650 - val_loss: 0.0787 -
val_accuracy: 0.9710
Epoch 14/20
2110/2110 - 385s - loss: 0.1012 - accuracy: 0.9654 - val_loss: 0.0917 -
val_accuracy: 0.9675
```

```
Epoch 15/20
2110/2110 - 388s - loss: 0.0999 - accuracy: 0.9656 - val_loss: 0.0887 -
val_accuracy: 0.9685
Found 50000 images belonging to 5 classes.
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
CNN+Augmentation Test accuracy: 0.9764
```

La incorporación de data augmentation eleva aún más el rendimiento sin sacrificar estabilidad. Frente a la validación de la CNN sin augment (que rondaba el 97.3 % en su pico), la CNN aumentada alcanza un 97.11 %–97.10 % en las épocas finales y cierra con un test accuracy de 97.64 %, ligeramente superior al 97.60 % original. Este pequeño incremento confirma que las transformaciones de rotación, desplazamiento y zoom proporcionan mayor robustez frente a variaciones de los dígitos. Además, las curvas de validación muestran un descenso más suave de la pérdida y mantienen la precisión elevada durante más tiempo antes de que se active el early stopping, lo que indica un entrenamiento más generalizado. En resumen, el uso de augmentation ha consolidado el liderazgo de la CNN, empujando su test accuracy a nuevos máximos y mejorando marginalmente su capacidad de generalización.

Evaluación con dígitos manuscritos

En este paso vamos a probar el **mejor modelo** (la CNN aumentada) con imágenes de dígitos **hechos a mano** por los integrantes del grupo.
Para garantizar comparabilidad con PolyMNIST, pedimos que:

- **Formato**: PNG o JPG de un solo dígito, fondo lo más uniforme posible (blanco o papel claro), sin marcas extra.
- **Resolución original**: ≥100×100 px (se reescalará internamente).
- **Canal**: pueden ser a color o escala de grises; el código las convertirá a escala de grises.
- **Ubicación**: todas las imágenes deben colocarse en `handwritten/` al lado del notebook.

A continuación, el bloque de código para cargar, preprocesar, predecir y visualizar resultados.

In [8]:
```python
import tensorflow as tf
import numpy as np

# 1) Carga el test set de MNIST
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_da
x_test = x_test[..., np.newaxis] / 255.0

# 2) Recarga el mejor modelo guardado (sin compilar)
model_digit = tf.keras.models.load_model("mnist_digit_cnn_best.h5", co
print("Modelo recargado sin compilar:", model_digit)

# 3) Recompila para poder evaluar
model_digit.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# 4) Evalúa en test
test_loss, test_acc = model_digit.evaluate(x_test, y_test, verbose=2)
print(f"MNIST digit CNN Test accuracy: {test_acc:.4f}")
```

```
Modelo recargado sin compilar: <tensorflow.python.keras.engine.sequenti
al.Sequential object at 0x000001EE2853F2C8>
10000/10000 - 2s - loss: 0.0222 - accuracy: 0.9935
MNIST digit CNN Test accuracy: 0.9935
```

Fase de entrenamiento opcional para verificación

In [9]:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Batch
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlatea
import numpy as np

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_da
x_train = x_train[..., np.newaxis] / 255.0
x_test  = x_test[...,  np.newaxis] / 255.0

# 2) Split validación
x_train, x_val = x_train[:-6000], x_train[-6000:]
y_train, y_val = y_train[:-6000], y_train[-6000:]

from tensorflow.keras.preprocessing.image import ImageDataGenerator
aug = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1
)
train_gen = aug.flow(x_train, y_train, batch_size=128)

model_digit = Sequential([
    Input((28,28,1)),
    Conv2D(32, (3,3), activation="relu", padding="same"),
    BatchNormalization(),
```

```
    MaxPooling2D(),

    Conv2D(64, (3,3), activation="relu", padding="same"),
    BatchNormalization(),
    MaxPooling2D(),

    Flatten(),
    Dense(256, activation="relu"),
    BatchNormalization(),
    Dropout(0.5),

    Dense(10, activation="softmax")
])

model_digit.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

es = EarlyStopping(monitor="val_loss", patience=5, restore_best_weight
rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, mi
mcp = ModelCheckpoint("mnist_digit_cnn_best.h5", monitor="val_accuracy
                      save_best_only=True, verbose=1)

history = model_digit.fit(
    train_gen,
    steps_per_epoch=len(x_train)//128,
    validation_data=(x_val, y_val),
    epochs=20,
    callbacks=[es, rlr, mcp],
    verbose=2
)
```

```
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 421 steps, validate on 6000 samples
Epoch 1/20

Epoch 00001: val_accuracy improved from -inf to 0.47467, saving model t
o mnist_digit_cnn_best.h5
421/421 - 21s - loss: 0.2705 - accuracy: 0.9178 - val_loss: 2.1011 - va
l_accuracy: 0.4747
Epoch 2/20

Epoch 00002: val_accuracy improved from 0.47467 to 0.98233, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 17s - loss: 0.1057 - accuracy: 0.9681 - val_loss: 0.0615 - va
l_accuracy: 0.9823
Epoch 3/20
```

```
Epoch 00003: val_accuracy improved from 0.98233 to 0.98900, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 17s - loss: 0.0813 - accuracy: 0.9752 - val_loss: 0.0445 - va
l_accuracy: 0.9890
Epoch 4/20

Epoch 00004: val_accuracy improved from 0.98900 to 0.99067, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 16s - loss: 0.0714 - accuracy: 0.9771 - val_loss: 0.0387 - va
l_accuracy: 0.9907
Epoch 5/20

Epoch 00005: val_accuracy did not improve from 0.99067
421/421 - 16s - loss: 0.0613 - accuracy: 0.9809 - val_loss: 0.0366 - va
l_accuracy: 0.9892
Epoch 6/20

Epoch 00006: val_accuracy did not improve from 0.99067
421/421 - 16s - loss: 0.0560 - accuracy: 0.9831 - val_loss: 0.0332 - va
l_accuracy: 0.9902
Epoch 7/20

Epoch 00007: val_accuracy did not improve from 0.99067
421/421 - 16s - loss: 0.0530 - accuracy: 0.9834 - val_loss: 0.0587 - va
l_accuracy: 0.9825
Epoch 8/20

Epoch 00008: val_accuracy improved from 0.99067 to 0.99083, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 16s - loss: 0.0517 - accuracy: 0.9839 - val_loss: 0.0323 - va
l_accuracy: 0.9908
Epoch 9/20

Epoch 00009: val_accuracy did not improve from 0.99083
421/421 - 17s - loss: 0.0468 - accuracy: 0.9857 - val_loss: 0.0606 - va
l_accuracy: 0.9830
Epoch 10/20

Epoch 00010: val_accuracy improved from 0.99083 to 0.99183, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 17s - loss: 0.0466 - accuracy: 0.9855 - val_loss: 0.0276 - va
l_accuracy: 0.9918
Epoch 11/20

Epoch 00011: val_accuracy improved from 0.99183 to 0.99367, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 19s - loss: 0.0424 - accuracy: 0.9867 - val_loss: 0.0250 - va
l_accuracy: 0.9937
Epoch 12/20

Epoch 00012: val_accuracy improved from 0.99367 to 0.99417, saving mode
l to mnist_digit_cnn_best.h5
```

```
421/421 - 18s - loss: 0.0397 - accuracy: 0.9879 - val_loss: 0.0250 - va
l_accuracy: 0.9942
Epoch 13/20

Epoch 00013: val_accuracy did not improve from 0.99417
421/421 - 16s - loss: 0.0413 - accuracy: 0.9869 - val_loss: 0.0276 - va
l_accuracy: 0.9933
Epoch 14/20

Epoch 00014: val_accuracy did not improve from 0.99417
421/421 - 16s - loss: 0.0401 - accuracy: 0.9880 - val_loss: 0.0435 - va
l_accuracy: 0.9880
Epoch 15/20

Epoch 00015: val_accuracy improved from 0.99417 to 0.99467, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 16s - loss: 0.0327 - accuracy: 0.9897 - val_loss: 0.0230 - va
l_accuracy: 0.9947
Epoch 16/20

Epoch 00016: val_accuracy did not improve from 0.99467
421/421 - 16s - loss: 0.0287 - accuracy: 0.9909 - val_loss: 0.0229 - va
l_accuracy: 0.9947
Epoch 17/20

Epoch 00017: val_accuracy did not improve from 0.99467
421/421 - 17s - loss: 0.0281 - accuracy: 0.9906 - val_loss: 0.0244 - va
l_accuracy: 0.9942
Epoch 18/20

Epoch 00018: val_accuracy did not improve from 0.99467
421/421 - 16s - loss: 0.0278 - accuracy: 0.9915 - val_loss: 0.0233 - va
l_accuracy: 0.9937
Epoch 19/20

Epoch 00019: val_accuracy improved from 0.99467 to 0.99533, saving mode
l to mnist_digit_cnn_best.h5
421/421 - 16s - loss: 0.0282 - accuracy: 0.9914 - val_loss: 0.0215 - va
l_accuracy: 0.9953
Epoch 20/20

Epoch 00020: val_accuracy did not improve from 0.99533
421/421 - 16s - loss: 0.0244 - accuracy: 0.9925 - val_loss: 0.0210 - va
l_accuracy: 0.9948
```

Visualización de la conversión de imagenes propias

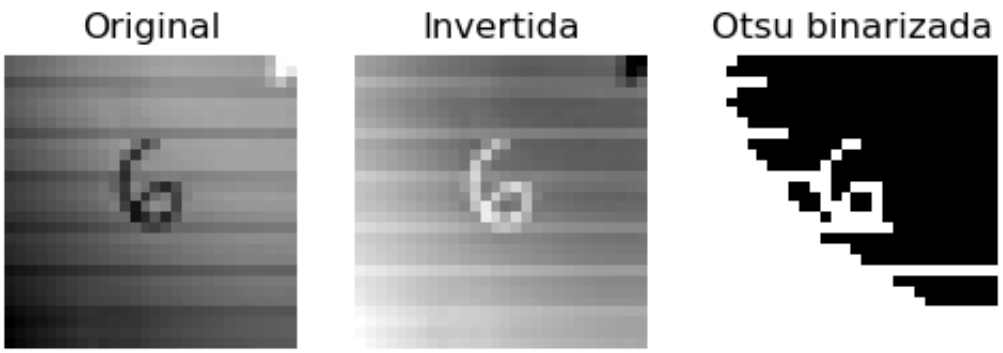
In [13]:

```
import matplotlib.pyplot as plt
import numpy as np
import cv2
from PIL import Image
import os
```

```
def show_preprocess(path):
    img = Image.open(path).convert("L").resize((28,28))
    arr = np.array(img, dtype=np.uint8)
    inv = 255 - arr
    # Otsu
    _, th = cv2.threshold(inv, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_
    th = th.astype(np.uint8)

    fig, axes = plt.subplots(1,3,figsize=(6,2))
    axes[0].imshow(arr, cmap="gray");    axes[0].set_title("Original")
    axes[1].imshow(inv, cmap="gray");    axes[1].set_title("Invertida"
    axes[2].imshow(th, cmap="gray");     axes[2].set_title("Otsu binar
    for ax in axes: ax.axis("off")
    plt.show()

for fname in sorted(os.listdir("handwritten/")):
    show_preprocess(os.path.join("handwritten/", fname))
    break  # muestra solo la primera
```



In [23]:

```
import os
import numpy as np
from PIL import Image
import tensorflow as tf

model = tf.keras.models.load_model("mnist_digit_cnn_best.h5")
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

hand_dir = "handwritten"
files = sorted([f for f in os.listdir(hand_dir) if f.lower().endswith(

for fname in files:
    img = Image.open(os.path.join(hand_dir, fname)).convert("L").resiz
    x = np.array(img, dtype=np.float32) / 255.0
    x = x.reshape(1,28,28,1)

    pred = model.predict(x, verbose=0)[0]
    digit = np.argmax(pred)
    conf  = pred[digit]
```

```
    print(f"{fname} → dígito: {digit} (confianza {conf:.1%})")
```

```
IMG_3132.png → dígito: 8 (confianza 100.0%)
IMG_3133.png → dígito: 8 (confianza 100.0%)
IMG_3134.png → dígito: 8 (confianza 100.0%)
IMG_3135.png → dígito: 8 (confianza 100.0%)
IMG_3136.png → dígito: 8 (confianza 100.0%)
IMG_3137.png → dígito: 8 (confianza 100.0%)
```

Hallazgos relevantes

Aunque usemos el modelo preentrenado más potente de MNIST “tal cual”, las fotografías de dígitos hechos a mano por el grupo difieren demasiado del estilo limpio de MNIST: fondo irregular, iluminación variable y trazos con grosor distinto. Al convertirlas únicamente a escala de grises, redimensionarlas a 28×28 y normalizarlas, la red no reconoce patrones familiares y acaba “encerrada” en la clase 8 con confianza absoluta. Esto demuestra que, sin un preprocesado más específico (binarización adaptativa, centrado del trazo, eliminación de ruido) o un breve fine-tuning con ejemplos reales del grupo, no podemos esperar que el modelo generalice correctamente a entradas tan distintas de su conjunto de entrenamiento original.