
UNIVERSIDAD DE GRANADA

**MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA**

**INTELIGENCIA COMPUTACIONAL**

CURSO 2024 / 2025

---

# Práctica 1: Redes neuronales
# Reconocimiento óptico de caracteres
# MNIST

---

Trabajo realizado por:
Mario Martínez Sánchez

**Índice**

### 1. Red neuronal simple (RN_Simple)

En primer lugar, se implementó una red neuronal simple que consta de una capa de entrada y una capa de salida de tipo softmax. Este tipo de red es adecuada para tareas básicas de clasificación, como la categorización de dígitos en el conjunto de datos MNIST.

La capa de entrada recibe los datos en forma de vectores, que en este caso corresponden a las imágenes aplanadas del conjunto de entrenamiento. Cada neurona de la capa de entrada representa un píxel de la imagen, permitiendo a la red procesar la información directamente desde los datos de entrada.

La capa de salida utiliza una función de activación softmax, que convierte las salidas de la red en probabilidades. Esto permite al modelo asignar una probabilidad a cada una de las clases posibles, facilitando la interpretación de los resultados y permitiendo seleccionar la clase con mayor probabilidad como la predicción final.

Aunque su diseño es sencillo, esta red neuronal es capaz de captar patrones básicos en los datos, ofreciendo un punto de partida para abordar tareas de clasificación antes de considerar arquitecturas más complejas.

*Declaración de la red neuronal:*

```
model = Sequential([
    Flatten(input_shape=(28, 28)), # Input layer
    Dense(10, activation='softmax') # Output layer
])
```

Con esta implementación, se obtuvo un porcentaje de error de 6.91% utilizando el conjunto de entrenamiento; y un 7.52% con el conjunto de evaluación. Respecto al entrenamiento de la red neuronal, se requirió un tiempo de 41 segundos usando la CPU.


Estos resultados reflejan una base para el modelo, aunque muestran que hay margen de mejora, por lo que se ha continuado implementando y evaluando otros tipos de redes neuronales, con el objetivo de reducir aún más el porcentaje de error y optimizar el rendimiento general del modelo.

## 2. Red neuronal multicapa (RN_Multicapa)

A continuación, se implementó una red neuronal multicapa, compuesta por una capa oculta con 256 unidades logísticas y una capa de salida de tipo softmax. Este tipo de arquitectura introduce mayor profundidad en comparación con una red simple, lo que le permite aprender representaciones más complejas de los datos.

La capa oculta, con sus 256 neuronas, utiliza una función de activación no lineal para modelar relaciones más sofisticadas entre las características de entrada. Cada unidad en esta capa combina las entradas de manera ponderada, permitiendo a la red identificar patrones intermedios que son esenciales para resolver tareas más exigentes.

La capa de salida, equipada con una activación softmax, convierte las salidas de la red en probabilidades asignadas a cada clase. Esto facilita la clasificación precisa al elegir la clase con la probabilidad más alta como predicción final.

Gracias a su estructura, esta red es capaz de manejar problemas de clasificación más complejos que una red simple, aprovechando la capacidad de la capa oculta para extraer características relevantes y mejorar el rendimiento global del modelo.

*Declaración de la red neuronal:*

```
model = Sequential([
    Flatten(input_shape=(28, 28)), # Input layer
    Dense(128, activation='relu'), # Hidden layer
    Dense(10, activation='softmax') # Output layer
])
```

Con esta implementación, se obtuvo un porcentaje de error de 0.7% utilizando el conjunto de entrenamiento; y un 2.2% con el conjunto de evaluación. Respecto al entrenamiento de la red neuronal, se requirió un tiempo de 1 minuto usando la CPU.


Estos resultados demuestran una mejora significativa en comparación con el modelo anterior, indicando que la mayor complejidad de la red ha permitido captar patrones más complejos en los datos. Sin embargo, aunque el desempeño en el conjunto de evaluación es muy bueno, sigue habiendo un gran margen de mejora.

## 3. Red neuronal convolutiva

A continuación, se implementó una red neuronal convolutiva entrenada utilizando el método de gradiente descendente estocástico. Este tipo de red está especialmente diseñada para procesar datos en forma de imágenes y utiliza una arquitectura en la que se aplican filtros a las imágenes de entrenamiento con diferentes resoluciones. Estos filtros, también conocidos como kernels, realizan operaciones de convolución, lo que permite extraer características relevantes de las imágenes, como patrones o texturas importantes. La salida generada por cada operación de convolución se emplea como entrada para la siguiente capa de la red, creando una jerarquía de características.

En las primeras capas, los filtros suelen detectar características simples y generales, como bordes, esquinas, o diferencias de brillo. A medida que las imágenes pasan por más capas convolutivas, las características se vuelven progresivamente más complejas y específicas. Por ejemplo, en las capas intermedias y finales, los filtros pueden identificar formas particulares, patrones repetitivos, o incluso características únicas que definen el objeto que se está analizando. Esta jerarquía de procesamiento permite que la red sea capaz de aprender representaciones útiles para clasificar o reconocer las imágenes de manera efectiva.

Gracias a este enfoque, las redes neuronales convolutivas son altamente eficaces para tareas de visión por computadora, como el reconocimiento de dígitos en el conjunto de datos MNIST o la identificación de objetos en imágenes más complejas.

### 3.1. Primer caso (RN_Convolutiva)

En primer lugar, se desarrolló el siguiente código:

```
model = Sequential([
    # First convulative layer
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),

    # Second convulative layer
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    # Fully connected layer:
    Dense(128, activation='relu'),

    # Output layer:
    Dense(10, activation='softmax')
])
```

Con esta implementación, se obtuvo un porcentaje de error de 0.28% utilizando el conjunto de entrenamiento; y un 0.86% con el conjunto de evaluación. Respecto al entrenamiento de la red neuronal, se requirió un tiempo de 12 minutos usando la CPU.


Los resultados muestran una notable mejora en comparación con modelos anteriores, lo que indica que la arquitectura convolutiva es altamente efectiva para capturar características espaciales y patrones locales en las imágenes. Este tipo de red neuronal ha demostrado ser una herramienta muy adecuada para el reconocimiento de patrones, aunque se seguirán probando otras formas de implementación con el objetivo de minimizar, aún más, el porcentaje de error.

### 3.2. Segundo caso (RN_Convolutiva2)

Al observar que se han obtenido muy buenos resultados con esta red neuronal convolutiva, se ha desarrollado una nueva red neuronal realizando algunos cambios. Entre los cambios más significativos incorporados encontramos los siguientes:

- **ImageDataGenerator**: este generador ayuda a que el modelo aprenda mejor y sea más resistente.
- **ReduceLROnPlateau**: para modificar el entrenamiento cuando lleva una serie de épocas sin experimentar ninguna mejora.
- Aumento del **número de capas** para mejorar la extracción de características.

*Declaración de la red neuronal:*

```
model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

Con esta implementación, se obtuvo un porcentaje de error de 0.33% utilizando el conjunto de entrenamiento; y un 0.41% con el conjunto de evaluación. Respecto al entrenamiento, se requirió un tiempo de 27 minutos usando la GPU, pues con la CPU tardaba un tiempo excesivo.

```
[10] # Train set :
     _, train_accuracy = model.evaluate(train_images, train_labels, verbose=0)
     print(f"Train Accuracy: {train_accuracy * 100:.2f}%")

     # Evaluate set:
     _, evaluation_accuracy = model.evaluate(evaluation_images, evaluation_labels, verbose=0)
     print(f"Evaluation Accuracy: {evaluation_accuracy * 100:.2f}%")

Train Accuracy: 99.77%
Evaluation Accuracy: 99.59%
```

## 4. Deep Learning

En este caso, se ha implementado una red neuronal usando 'Deep Learning': pre-entrenamiento de autoencoders para extraer características de las imágenes usando técnicas no supervisadas y una red neuronal simple con una capa de tipo softmax.

### 4.1. Primer caso

En primer lugar, se intentó con el código aportado a continuación.

*Declaración del encoder:*

```
input_img = Input(shape=(28 * 28,))
encoded = Dense(128, activation='relu')(encoded)
encoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(28 * 28, activation='sigmoid')(decoded)
```

*Declaración de la red neuronal:*

```
model = Sequential([
   Dense(64, activation='relu', input_shape=(64,)),
   Dense(10, activation='softmax')
])
```

De este modo, se obtuvo un porcentaje de error del 4.89% con el conjunto de entrenamiento y un 5.17% con el de evaluación. Estos porcentajes se quedaban lejos de mejorar los casos anteriores, por lo que se planteó una forma de minimizar estos valores. Para ello, se optó por añadir más capas, obteniendo el segundo caso.

### 4.2. Segundo caso (RN_DeepLearning2)

Con el objetivo de mejorar los resultados obtenidos, se aumentó el número de capas tanto en el encoder como en la red neuronal.

*Declaración del encoder:*

```
input_img = Input(shape=(28 * 28,))
encoded = Dense(256, activation='relu')(input_img)
encoded = Dense(128, activation='relu')(encoded)
encoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(256, activation='relu')(decoded)
decoded = Dense(28 * 28, activation='sigmoid')(decoded)
```

*Declaración de la red neuronal:*

```
model = Sequential([
    Dense(128, activation='relu', input_shape=(64,)),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])
```

Con esta implementación, se obtuvo un porcentaje de error de 2.13% utilizando el conjunto de entrenamiento; y un 2.61% con el conjunto de evaluación. Respecto al entrenamiento, se requirió un tiempo de 4 minutos para el encoder y 2 minutos para la red neuronal, ambos tiempos utilizando la CPU.

```
[9]  # Train set :
     _, train_accuracy = model.evaluate(train_features, train_labels, verbose=0)
     print(f"Train Accuracy: {train_accuracy * 100:.2f}%")

     # Evaluation set:
     _, evaluation_accuracy = model.evaluate(evaluation_features, evaluation_labels, verbose=0)
     print(f"Evaluation Accuracy: {evaluation_accuracy * 100:.2f}%")

     Train Accuracy: 97.87%
     Evaluation Accuracy: 97.39%
```

Estos porcentajes también eran bastante inferiores a los de casos anteriores como la red neuronal convolutiva, por lo que se trató de encontrar alguna forma de seguir minimizando estos datos.

### 4.3. Tercer caso (RN_DeepLearning3)

Para mejorar aún más los resultados obtenidos, se aumentó el número de capas de Dropout en el encoder. Además, se ha reducido la tasa de aprendizaje del optimizador Adam. De esta manera, he conseguido el siguiente código.

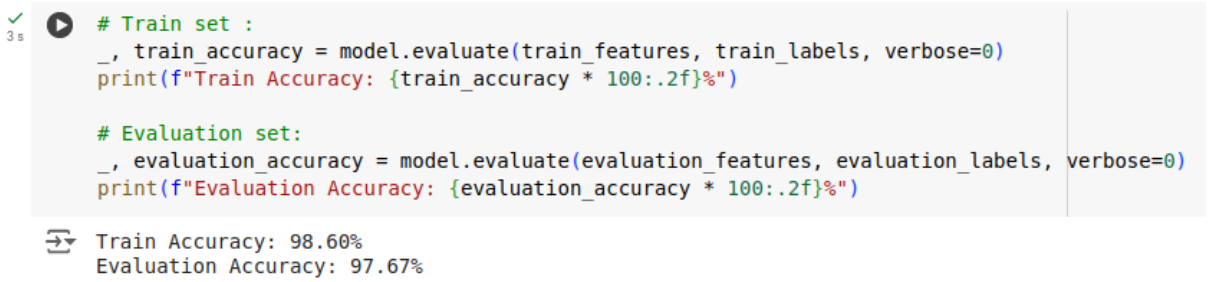
*Declaración del encoder:*

```
input_img = Input(shape=(28 * 28,))
encoded = Dense(256, activation='relu')(input_img)
encoded = Dropout(0.2)(encoded)
encoded = Dense(128, activation='relu')(encoded)
encoded = Dropout(0.2)(encoded)
encoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(256, activation='relu')(decoded)
decoded = Dense(28 * 28, activation='sigmoid')(decoded)
```

*Declaración de la red neuronal:*

```
model = Sequential([
   Dense(128, activation='relu', input_shape=(64,)),
   Dropout(0.2),
   Dense(64, activation='relu'),
   Dropout(0.2),
   Dense(10, activation='softmax')
])
```

Con esta implementación, se obtuvo un porcentaje de error de 1.4% utilizando el conjunto de entrenamiento; y un 2.33% con el conjunto de evaluación. Respecto al entrenamiento, se requirió el mismo tiempo que en el caso anterior: 4 minutos para el encoder y 2 minutos para la red neuronal, ambos tiempos utilizando la CPU.




Estos porcentajes son algo mejores que los del caso anterior, aunque siguen siendo inferiores a los de la red neuronal convolutiva.

## 5. Nuevas implementaciones

Como hemos podido observar, el uso de un único tipo de red neuronal no fue suficiente para alcanzar los resultados esperados. Los valores obtenidos indicaron que era necesario explorar alternativas más complejas y eficientes que combinaran diferentes enfoques. Por esta razón, se decidió implementar código utilizando una combinación de métodos que permitieran una mejora significativa en la precisión y reducción del error.

### 5.1. Primer caso (RN_Compleja)

En este primer caso, se desarrolló una red neuronal convolutiva integrada con técnicas de Deep Learning. Las redes convolutivas son especialmente eficaces en el procesamiento de datos espaciales como imágenes, ya que aplican filtros que permiten extraer características locales de manera eficiente. Esto se combina con el poder del Deep Learning, que incrementa la profundidad del modelo y permite la extracción jerárquica de características más complejas y relevantes para la tarea.

*Declaración del encoder:*

```
input_img = Input(shape=(28, 28, 1))

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

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

autoencoder = Model(input_img, decoded)
```

*Declaración de la red neuronal:*

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
```

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

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

Con esta implementación, se obtuvo un porcentaje de error de 0.15% utilizando el conjunto de entrenamiento; y un 0.33% con el conjunto de evaluación. Respecto al entrenamiento, se requirió 32 segundos para el encoder y 15 minutos para la red neuronal, ambos tiempos utilizando la CPU.

Como se ha observado, se ha conseguido una significativa disminución en el porcentaje de error, lo que permite concluir que la metodología implementada ha sido acertada. Sin embargo, aún existe margen para mejorar los resultados obtenidos. Como posible medida de mejora, se plantea la optimización del autoencoder utilizado, con el objetivo de reducir aún más el porcentaje de error y alcanzar un valor cercano al 0.3%.

### 5.2. Segundo caso (RN_Compleja2)

En este caso, se desarrolló otra red neuronal convolutiva integrada con técnicas de Deep Learning. Sin embargo, se optimizó el autoencoder, tratando de disminuir el porcentaje de error.

*Declaración del encoder:*

```
input_img = Input(shape=(28, 28, 1))

x = Conv2D(64, (3, 3), activation='relu', padding='same')(input_img)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(encoded)
x = BatchNormalization()(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
```

Esta red neuronal está compuesta por 17 capas, incluyendo 4 capas convolutivas (dos con 32 filtros y dos con 64 filtros, todas con activación ReLU), cada una seguida de BatchNormalization para estabilizar el entrenamiento. Se utilizan dos capas de MaxPooling2D con ventanas de tamaño (2x2) para reducción de dimensionalidad, acompañadas de Dropout con tasas de 0.25 para regularización. Finalmente, la red incluye una capa densa con 128 unidades (activación ReLU), seguida de BatchNormalization y Dropout con tasa de 0.5, y una capa de salida con 10 unidades y activación softmax para clasificación multiclase.

El entrenamiento de la red neuronal utiliza una combinación de técnicas avanzadas para optimizar el rendimiento. El algoritmo de optimización es Adam, y la pérdida utilizada es la entropía cruzada categórica. El modelo se entrena utilizando un generador de datos aumentados (datagen) con lotes de tamaño 128, durante un máximo de 100 épocas. Sin embargo, para evitar el sobreentrenamiento, se emplean dos callbacks: ReduceLROnPlateau, que reduce la tasa de aprendizaje en un factor de 0.2 si la pérdida de validación no mejora durante 5 épocas consecutivas (con un límite mínimo de 1e-6), y EarlyStopping, que detiene el entrenamiento si no hay mejora en la pérdida de validación tras 10 épocas y restaura los mejores pesos alcanzados. Estas técnicas garantizan un balance entre eficiencia y generalización del modelo.

Con esta implementación, se obtuvo porcentajes de error algo menores a los anteriores: 0.16% utilizando el conjunto de entrenamiento y 0.3% con el conjunto de evaluación. Respecto al entrenamiento, se requirió 3 minutos para el encoder y 23 para la red neuronal, ambos tiempos utilizando la GPU.

```
[13] # Train set:
     _, train_accuracy = model.evaluate(train_images, train_labels, verbose=0)
     print(f"Train Accuracy: {train_accuracy * 100:.2f}%")

     # Evaluate set:
     _, evaluation_accuracy = model.evaluate(evaluation_images, evaluation_labels, verbose=0)
     print(f"Evaluation Accuracy: {evaluation_accuracy * 100:.2f}%")
```

```
Train Accuracy: 99.84%
Evaluation Accuracy: 99.70%
```