deep-learning-ii-esp

November 15, 2024

1 Deep Learning II

1. Redes Neuronales
    - Redes Neuronales de varias capas (multilayer)
2. DeepLearning en Keras
3. Convolutional neural network (CNN)
4. Redes Neuronales Recurrentes

2 1. Redes Neuronales

Las redes neuronales son un método fundamental del aprendizaje automático, inspirado en la estructura y funcionamiento del cerebro humano. Se basan en una unidad básica conocida como la *unidad neuronal*, la cual es una función matemática que modela el comportamiento de una neurona biológica.

Una unidad neuronal (también conocida como neurona artificial o red neuronal de una capa) se define matemáticamente como:

$$\mathbf{y} = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b)$$

Donde: - $\mathbf{x} \in \mathbb{R}^n$: Vector de entrada con n características o variables. - $\mathbf{w} \in \mathbb{R}^n$: Vector de pesos asociado a las entradas, que representa la importancia relativa de cada característica. - $b \in \mathbb{R}$: Sesgo o bias, un valor escalar que ajusta la salida independientemente de las entradas. - σ: Función de activación, que introduce no linealidad en la salida. - $\mathbf{y} \in \mathbb{R}$: Salida de la neurona, conocida como *activación*, que puede representar una predicción, probabilidad o puntuación.

Es importante destacar que la fórmula correcta es:

$$\mathbf{y} = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b)$$

y no

$$\mathbf{y} = \sigma(\mathbf{x} \cdot \mathbf{w}^T + b).$$

2.0.1 ¿Por qué?

En álgebra lineal, para garantizar la compatibilidad de dimensiones: 1. $\mathbf{w}$ se transpone ($\mathbf{w}^T$) para que sea un vector columna ($n \times 1$), igual que $\mathbf{x}$. El producto escalar $\mathbf{w}^T \cdot \mathbf{x}$ da como resultado un escalar (1×1). 2. En cambio, $\mathbf{x} \cdot \mathbf{w}^T$ resultaría en una matriz de tamaño $n \times n$, incompatible con la función de activación σ, que opera sobre escalares o vectores.

Por estas razones, la convención estándar es usar $\mathbf{w}^T \cdot \mathbf{x}$. Los parámetros ajustables $(\mathbf{w}, b)$ se optimizan durante el entrenamiento para minimizar el error entre las predicciones de la red y los valores reales.

2.1 Funciones de Activación

La función de activación σ es esencial para capturar relaciones complejas entre las variables de entrada y salida. Sin la no linealidad que introduce σ, las redes neuronales serían equivalentes a modelos lineales.

2.1.1 1. Sigmoide:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Donde: - $\sigma(x) \in (0, 1)$: Salida normalizada entre 0 y 1.

Características: - Útil para problemas de clasificación binaria. - Puede sufrir de *vanishing gradients* (los gradientes se vuelven muy pequeños, ralentizando el aprendizaje).

2.1.2 2. ReLU (Rectified Linear Unit):

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

Donde: - $\sigma(x) = x$ si $x > 0$, de lo contrario $\sigma(x) = 0$.

Características: - Computacionalmente eficiente. - Mitiga el problema de *vanishing gradients*. - Puede causar *dying neurons* (neuronas que permanecen inactivas si reciben siempre valores negativos).

2.1.3 3. Leaky ReLU:

$$\sigma(x) = \begin{cases} x & \text{si } x > 0, \\ \alpha x & \text{si } x \leq 0. \end{cases}$$

Donde: - α es un pequeño valor positivo (e.g., $\alpha = 0.01$), que asegura una pequeña pendiente para $x \leq 0$.

Características: - Soluciona parcialmente el problema de las neuronas inactivas en ReLU.

2.1.4 4. Tangente Hipérbolica (TanH):

$$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Donde: - $\sigma(x) \in (-1, 1)$: Salida centrada en cero.

Características: - Similar a la Sigmoide, pero mejor para datos con distribuciones centradas en cero. - También puede sufrir de *vanishing gradients*.

2.2 Redes Neuronales de una Capa

Una red neuronal de una sola capa, conocida también como **perceptrón**, es capaz de resolver problemas de **clasificación lineal**. Esto significa que puede representar funciones de decisión de la forma:

$$\mathbf{w}^T \cdot \mathbf{x} + b = 0$$

Donde: - $\mathbf{w}^T \cdot \mathbf{x}$ es el producto interno entre el vector de pesos y las entradas. - b ajusta la posición del hiperplano de decisión.

Estas redes tienen limitaciones, ya que solo pueden clasificar correctamente datos que son linealmente separables. Para superar esta restricción, se utilizan redes neuronales con múltiples capas, que son capaces de representar funciones de decisión no lineales gracias a las combinaciones jerárquicas de sus unidades neuronales.

2.3 Redes Neuronales Multicapa

Las redes neuronales multicapa (también conocidas como redes profundas o **Deep Neural Networks**) son extensiones de las redes de una capa, donde las salidas de una capa se utilizan como entradas para la siguiente. Estas redes pueden aproximar funciones altamente no lineales y complejas, lo que las hace aptas para tareas como:

- Reconocimiento de voz.
- Visión por computadora.
- Traducción automática.
- Predicciones en series temporales.

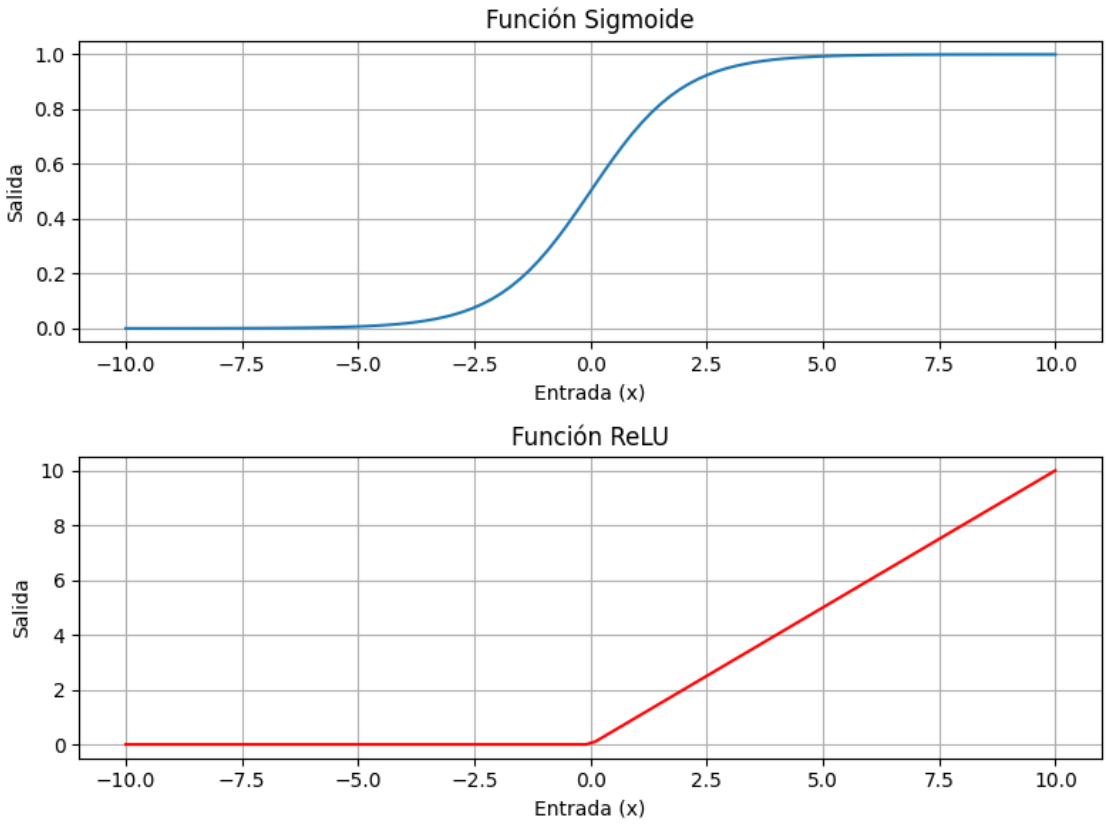
2.4 Importancia de las Redes Neuronales

Las redes neuronales han revolucionado diversos campos gracias a su capacidad para aprender patrones complejos en grandes volúmenes de datos. Su éxito en aplicaciones prácticas se debe a avances en: - Algoritmos de entrenamiento (e.g., retropropagación). - Hardware especializado (como GPUs y TPUs). - Técnicas de regularización para mejorar la generalización.

PREGUNTA: ¿Qué tipo de funciones de decisión están representadas por una red neuronal de una capa?
Respuesta: Una red neuronal de una capa representa funciones de decisión lineales, es decir, fronteras de decisión que pueden expresarse mediante una combinación lineal de las entradas.

[1]:
```
import numpy as np
import matplotlib.pylab as plt

# Función de activación sigmoide
def sigmoid(x):
    """
    La función sigmoide transforma los valores de entrada en un rango entre 0 y␣
 ↪1.
    Es útil para modelar probabilidades y se usa comúnmente en la capa de␣
 ↪salida de redes neuronales
    para problemas de clasificación binaria.
    """
    return 1 / (1 + np.exp(-x))

# Función de activación ReLU (Rectified Linear Unit)
def ReLU(x):
    """
    La función ReLU devuelve 0 para valores negativos y mantiene los valores␣
 ↪positivos sin cambios.
    Es ampliamente utilizada en redes neuronales profundas debido a su␣
 ↪simplicidad y capacidad para mitigar
    el problema del desvanecimiento del gradiente durante el entrenamiento.
    """
    return x * (x > 0)

# Creamos un rango de valores de entrada desde -10 a 10
x = np.linspace(-10.0, 10.0, 100)

# Calculamos la salida de la función sigmoide para el rango de valores de␣
 ↪entrada
y1 = sigmoid(x)

# Calculamos la salida de la función ReLU para el rango de valores de entrada
y2 = ReLU(x)

# Creamos la figura y los subgráficos
plt.figure(figsize=(8, 6))  # Ajustamos el tamaño de la figura

# Creamos el primer subgráfico (2 filas, 1 columna, primer gráfico)
plt.subplot(2, 1, 1)
plt.plot(x, y1)
```

```
plt.title('Función Sigmoide')  # Título del primer subgráfico
plt.xlabel('Entrada (x)')  # Etiqueta del eje x
plt.ylabel('Salida')  # Etiqueta del eje y
plt.grid(True)  # Agregamos una cuadrícula para mejor visualización

# Creamos el segundo subgráfico (2 filas, 1 columna, segundo gráfico)
plt.subplot(2, 1, 2)
plt.plot(x, y2, 'r')
plt.title('Función ReLU')  # Título del segundo subgráfico
plt.xlabel('Entrada (x)')  # Etiqueta del eje x
plt.ylabel('Salida')  # Etiqueta del eje y
plt.grid(True)  # Agregamos una cuadrícula para mejor visualización

# Ajustamos el diseño para evitar solapamientos
plt.tight_layout()

# Mostramos las gráficas
plt.show()
```



[2]:
```
# Vector de entrada
x = np.array([0.4, 1.2, 3.5])

# Vector de pesos
w = np.array([1.0, 2.0, 1.0])

# Sesgo
b = 1.3

# Calculamos el producto punto de x y w, sumamos el sesgo y aplicamos la␣
 ↪función sigmoide
y = sigmoid(np.dot(x, w) + b)

# Imprimimos el resultado
print(y)
```

```
0.9994997988929205
```

2.5 1.1. Redes Neuronales de Varias Capas (Multilayer)

Las neuronas individuales pueden organizarse en estructuras mayores aplicando diferentes conjuntos de pesos al mismo vector de datos de entrada. Esto crea una *capa* de neuronas, y al apilar múltiples capas (donde la salida de una capa se convierte en la entrada de la siguiente) obtenemos una red neuronal de varias capas o red neuronal profunda.

En una red neuronal multicapa, el modelo puede representarse como una composición de productos matriciales (las matrices de pesos) y funciones de activación no lineales. Por ejemplo, para una red de dos capas, el modelo puede expresarse como:

$$\mathbf{y} = \sigma\Big(W^1\sigma\Big(W^0\mathbf{x} + \mathbf{b}^0\Big) + \mathbf{b}^1\Big)$$

donde: - $\mathbf{x}$: Vector de entrada con las características del dato. - W^0 y W^1: Matrices de pesos de las capas, donde W^0 representa los pesos de la primera capa y W^1 los de la segunda capa. - $\mathbf{b}^0$ y $\mathbf{b}^1$: Vectores de sesgo (bias) para las capas correspondientes, que ajustan las salidas independientemente de las entradas. - σ: Función de activación aplicada de manera vectorial, introduciendo no linealidad y permitiendo que la red capture relaciones complejas. - $\mathbf{y}$: Salida de la red, también conocida como activación final.

2.5.1 Capacidad de Aproximación Universal

Las redes neuronales multicapa tienen una propiedad fundamental conocida como **capacidad de aproximación universal**: incluso una red con una sola capa oculta y un número finito de neuronas puede aproximar cualquier función continua definida en un espacio $\mathbb{R}^n$, con suficiente precisión. Esto hace que las redes neuronales sean extremadamente útiles para aprender patrones complejos a partir de datos.

La pregunta entonces es: ¿cómo encontrar los parámetros óptimos $(W^i, \mathbf{b}^i)$ para que la red aprenda a aproximar la función implícita en un conjunto de muestras $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$?

2.5.2 Optimización y Descenso de Gradiente

El proceso de entrenamiento de redes neuronales y otros modelos de aprendizaje automático a menudo implica la solución iterativa de un problema de optimización. Aunque en algunos casos existiría una solución analítica, en redes neuronales esto no es posible. Para resolver este problema se utiliza el método de **descenso de gradiente**, una técnica iterativa que ajusta los pesos para minimizar el error de la red con respecto a un conjunto de datos de entrenamiento.

Al aplicar descenso de gradiente a modelos como:

$$\mathbf{y} = \sigma\Big(W^1\sigma\Big(W^0\mathbf{x} + \mathbf{b}^0\Big) + \mathbf{b}^1\Big),$$

o redes de mayor complejidad, el modelo puede aprender adaptándose iterativamente a los datos mediante una **función de pérdida**.

2.5.3 Función de Pérdida

La función de pérdida mide la discrepancia entre las predicciones de la red y los valores reales, proporcionando una medida cuantitativa del error. Esta función depende del tipo de problema:

Regresión En problemas de regresión, donde la salida es continua, se utiliza comúnmente el **error cuadrático medio** (MSE, por sus siglas en inglés):

$$L = \frac{1}{n}\sum_{i=1}^{n}\Big(\mathbf{y}_i - \sigma\Big(W^1\sigma\Big(W^0\mathbf{x}_i + \mathbf{b}^0\Big) + \mathbf{b}^1\Big)\Big)^2$$

donde: - n: Número de ejemplos en el conjunto de entrenamiento. - $\mathbf{y}_i$: Valor real o esperado para el ejemplo i. - La expresión dentro del cuadrado es la diferencia entre la predicción de la red y el valor real.

Clasificación Binaria Para problemas de clasificación binaria (dos clases), es común usar la **función de pérdida logística** o **cross-entropy loss**:

$$L = \frac{1}{n}\sum_{i=1}^{n}\log\Big(1 + \exp\Big(-y_i\sigma\Big(W^1\sigma\Big(W^0\mathbf{x}_i + \mathbf{b}^0\Big) + \mathbf{b}^1\Big)\Big)\Big)$$

donde: - y_i: Etiqueta de clase para el ejemplo i, que toma valores en $\{+1, -1\}$. - Esta función de pérdida penaliza errores en las predicciones de manera que se maximice la probabilidad de predecir correctamente la clase.

2.5.4 Ventajas de las Redes Neuronales Profundas

Las redes neuronales profundas ofrecen varias ventajas gracias a su estructura jerárquica de múltiples capas:

- **Capacidad de Representación**: Pueden aproximar funciones complejas y modelar patrones intrincados en los datos.
- **Transferencia de Aprendizaje**: Una red entrenada para una tarea puede adaptarse a otras tareas similares mediante técnicas como el *fine-tuning*, donde los pesos entrenados se ajustan ligeramente para una nueva tarea.
- **Robustez**: Con grandes volúmenes de datos y técnicas de regularización, las redes profundas son menos propensas a problemas de *overfitting* y pueden generalizar bien.

En resumen, las redes neuronales de múltiples capas han revolucionado el campo del aprendizaje automático, permitiendo avances significativos en aplicaciones como el procesamiento de lenguaje natural, visión por computadora y sistemas de recomendación.

2.5.5 Jugando con redes neuronales.

- Clases concéntricas, 1 capa, Sigmoide.
- Clases concéntricas, 1 capa, ReLu.
- X-or, 0 capas.
- X-or, 1 capa.
- Datos en espiral.
- Regresión.

http://playground.tensorflow.org

3 2. Deep Learning en `keras`

> Keras es una biblioteca de redes neuronales de alto nivel, escrita en Python y capaz de ejecutarse sobre TensorFlow. Fue desarrollada con un enfoque en permitir la experimentación rápida y accesible.

3.1 Modelo Secuencial

El núcleo de la estructura de datos de Keras es un **modelo**, que organiza y gestiona las capas de la red neuronal. El principal tipo de modelo es el `Modelo secuencial`, una pila lineal de capas que se apilan una tras otra.

```
from tensorflow.keras.models import Sequential
model = Sequential()
```

3.2 Añadir Capas

Apilar capas es tan fácil como usar el método `.add()`. A continuación, se construye un ejemplo de red con dos capas densas (fully connected):

```
from tensorflow.keras.layers import Dense, Activation

model.add(Dense(units=64, input_dim=100))
```

```
model.add(Activation("relu"))
model.add(Dense(units=10))
model.add(Activation("softmax"))
```

En este ejemplo: - **`Dense`**: Capa totalmente conectada con un número específico de neuronas (`units`). - **`input_dim`**: Dimensionalidad de la entrada a la red. - **`Activation`**: Función de activación para cada capa, como "relu" o "softmax".

3.3 Configurar el Modelo

Antes de entrenar el modelo, es necesario configurar el proceso de aprendizaje usando el método `.compile()`:

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

Donde: - **`loss`**: Define la función de pérdida. Por ejemplo, `categorical_crossentropy` es común en problemas de clasificación. - **`optimizer`**: Define el algoritmo de optimización, como SGD (Stochastic Gradient Descent). - **`metrics`**: Métricas que se monitorean durante el entrenamiento, como la `accuracy`.

3.4 Configuración Avanzada del Optimizador

Es posible personalizar los hiperparámetros del optimizador. Por ejemplo:

```
from tensorflow.keras.optimizers import SGD
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(learning_rate=0.01, momentum=0.9, nesterov=True))
```

Aquí se ajustan: - **`learning_rate`**: Tasa de aprendizaje. - **`momentum`**: Factor para acelerar la convergencia. - **`nesterov`**: Si se usa el método de Nesterov para el gradiente.

3.5 Entrenamiento del Modelo

Ahora puedes entrenar tu modelo iterando sobre los datos de entrenamiento en lotes:

```
model.fit(X_train, Y_train, epochs=5, batch_size=32)
```

Donde: - **`X_train`**: Datos de entrada de entrenamiento. - **`Y_train`**: Etiquetas correspondientes. - **`epochs`**: Número de veces que el modelo verá los datos completos. - **`batch_size`**: Tamaño de los lotes en cada iteración.

3.6 Evaluación del Modelo

Evalúa el rendimiento de tu modelo en un conjunto de datos de prueba:

```
loss_and_metrics = model.evaluate(X_test, Y_test, batch_size=32)
```

El resultado incluye: - **Pérdida**: Valor de la función de pérdida en los datos de prueba. - **Métricas**: Valores de las métricas especificadas durante la compilación.

3.7 Generar Predicciones

Para usar el modelo entrenado en nuevos datos, puedes generar predicciones de la clase y probabilidades:

```
classes = model.predict_classes(X_test, batch_size=32)
proba = model.predict_proba(X_test, batch_size=32)
```

Donde: - **`predict_classes`**: Devuelve las clases predichas para cada muestra. - **`predict_proba`**: Devuelve las probabilidades predichas de cada clase.

3.8 Ventajas de `Keras`

- **Simplicidad**: API intuitiva y clara para definir y entrenar redes neuronales.
- **Flexibilidad**: Soporte para múltiples frameworks de backend como TensorFlow.
- **Escalabilidad**: Compatible con entrenamiento distribuido y hardware acelerado como GPUs y TPUs.
- **Comunidad**: Amplia documentación y soporte de una comunidad activa.
- **Interfaz de alto nivel**: Facilita la experimentación rápida sin tener que gestionar los detalles más bajos de los algoritmos.

3.9 Desventajas de `Keras`

- **Rendimiento**: Aunque `Keras` es muy fácil de usar, a veces no es tan eficiente como otros frameworks de bajo nivel como TensorFlow o PyTorch, especialmente para tareas que requieren mucha personalización.
- **Menos Control**: Ofrece un alto nivel de abstracción, lo que puede limitar el control total sobre el proceso de entrenamiento y la arquitectura de las redes neuronales.
- **Problemas con la optimización**: La implementación de algunos optimizadores o algoritmos avanzados podría no ser tan flexible o rápida como en otros entornos más personalizados.
- **Dependencia de TensorFlow**: Keras depende de un backend como TensorFlow, lo que significa que cualquier limitación de TensorFlow afecta también a Keras.

[3]:
```
#import os

#!pip3 install tensorflow
```

[4]:
```
import tensorflow as tf

# Verifica el nombre del dispositivo GPU disponible
# Si TensorFlow detecta una GPU, imprimirá el nombre del dispositivo.
# Si no se detecta ninguna GPU, la salida será una cadena vacía.
print(tf.test.gpu_device_name())
```

```
2024-11-15 09:20:42.511628: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2024-11-15 09:20:42.736532: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:32]
Could not find cuda drivers on your machine, GPU will not be used.
2024-11-15 09:20:42.993297: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:485] Unable to register
```

```
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-11-15 09:20:43.177833: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:8454] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-11-15 09:20:43.233808: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1452] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-11-15 09:20:43.551380: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
2024-11-15 09:20:46.009450: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

[5]:
```python
from tensorflow.python.client import device_lib

# Lista todos los dispositivos locales disponibles en TensorFlow
# Esto incluye CPUs y GPUs (si están disponibles y correctamente configuradas).
print(device_lib.list_local_devices())
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 15327594396104960511
xla_global_id: -1
]
```

Entrenar una NN profunda simple en el conjunto de datos MNIST. Alcanza una precisión de prueba del 98,40% tras 20 epochs. (hay *mucho* margen para ajustar los parámetros). 2 segundos por epoch en una GPU K520.

[6]:
```python
from __future__ import print_function

# Importación de TensorFlow y módulos específicos de Keras
import tensorflow.keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.optimizers import RMSprop
```

```
# Definición de parámetros de entrenamiento
batch_size = 64
num_classes = 10
epochs = 5

# Carga de datos MNIST
# MNIST es un conjunto de datos de dígitos escritos a mano
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocesamiento de datos
# Las imágenes son de 28x28 píxeles, por lo que las aplanamos a vectores de 784␣
 ↪elementos
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# Conversión de los datos a tipo flotante y normalización (valores entre 0 y 1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Imprimir el número de muestras de entrenamiento y prueba
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Conversión de los vectores de clase a matrices de clases binarias
# Esto es necesario para la función de pérdida 'categorical_crossentropy'
y_train = tensorflow.keras.utils.to_categorical(y_train, num_classes)
y_test = tensorflow.keras.utils.to_categorical(y_test, num_classes)

# Construcción del modelo
model = Sequential()
model.add(Input(shape=(784,)))  # Entrada del modelo con la forma especificada
model.add(Dense(16, activation='relu'))  # Capa densa con 16 neuronas y ReLU␣
 ↪como función de activación
model.add(Dropout(0.2))  # Capa de abandono para evitar el sobreajuste, con una␣
 ↪tasa de abandono del 20%
model.add(Dense(32, activation='relu'))  # Capa densa con 32 neuronas y ReLU␣
 ↪como función de activación
model.add(Dropout(0.2))  # Capa de abandono para evitar el sobreajuste, con una␣
 ↪tasa de abandono del 20%
model.add(Dense(num_classes, activation='softmax'))  # Capa de salida con␣
 ↪'num_classes' neuronas y softmax para clasificación multi-clase

# Mostrar un resumen del modelo (estructura de capas, número de parámetros, etc.
 ↪)
```

```
model.summary()

# Compilación del modelo
# Se especifica la función de pérdida, el optimizador y las métricas a seguir
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(),  # Optimizador RMSprop
              metrics=['accuracy'])  # Métrica de precisión para evaluar el␣
 ↪rendimiento

# Entrenamiento del modelo
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,  # Mostrar información sobre el progreso del␣
 ↪entrenamiento
                    validation_data=(x_test, y_test))  # Datos de validación␣
 ↪para monitorizar el rendimiento en datos no vistos

# Evaluación del modelo en el conjunto de prueba
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])  # Pérdida en el conjunto de prueba
print('Test accuracy:', score[1])  # Precisión en el conjunto de prueba
```

```
60000 train samples
10000 test samples
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16)	12,560
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 32)	544
dropout_1 (Dropout)	(None, 32)	0
dense_2 (Dense)	(None, 10)	330

Total params: 13,434 (52.48 KB)

Trainable params: 13,434 (52.48 KB)

```
 Non-trainable params: 0 (0.00 B)


Epoch 1/5
938/938             4s 4ms/step -
accuracy: 0.5968 - loss: 1.1877 - val_accuracy: 0.9045 - val_loss: 0.3308
Epoch 2/5
938/938             3s 4ms/step -
accuracy: 0.8274 - loss: 0.5587 - val_accuracy: 0.9170 - val_loss: 0.2885
Epoch 3/5
938/938             4s 4ms/step -
accuracy: 0.8499 - loss: 0.4884 - val_accuracy: 0.9179 - val_loss: 0.2709
Epoch 4/5
938/938             3s 3ms/step -
accuracy: 0.8570 - loss: 0.4606 - val_accuracy: 0.9218 - val_loss: 0.2613
Epoch 5/5
938/938             3s 4ms/step -
accuracy: 0.8613 - loss: 0.4549 - val_accuracy: 0.9216 - val_loss: 0.2505
Test loss: 0.25046399235725403
Test accuracy: 0.9215999841690063
```

[7]:
```python
# Establecer la semilla para la reproducibilidad
np.random.seed(42)

# Visualización de una pequeña parte de los datos de entrada y salida
# Seleccionamos un pequeño número de ejemplos para visualizar
num_examples = 5
indices = np.random.choice(x_test.shape[0], num_examples, replace=False)

# Configuración de la visualización
fig, axes = plt.subplots(1, num_examples, figsize=(15, 3))
fig.suptitle('Ejemplos de imágenes del conjunto de prueba')

for i, idx in enumerate(indices):
    ax = axes[i]
    # Redimensionamos el vector de 784 a una imagen de 28x28 para mostrar
    img = x_test[idx].reshape(28, 28)
    # Mostrar la imagen en escala de grises
    ax.imshow(img, cmap='gray')
    # Mostrar la etiqueta real en el título del subplot
    ax.set_title(f'Label: {np.argmax(y_test[idx])}')
    ax.axis('off')

plt.show()

# Predicciones del modelo para los ejemplos seleccionados
predictions = model.predict(x_test[indices])
```

```python
# Visualización de las predicciones
fig, axes = plt.subplots(1, num_examples, figsize=(15, 3))
fig.suptitle('Predicciones del modelo')

for i, idx in enumerate(indices):
    ax = axes[i]
    # Redimensionamos el vector de 784 a una imagen de 28x28 para mostrar
    img = x_test[idx].reshape(28, 28)
    # Mostrar la imagen en escala de grises
    ax.imshow(img, cmap='gray')
    # Mostrar la predicción en el título del subplot
    ax.set_title(f'Pred: {np.argmax(predictions[i])}')
    ax.axis('off')

plt.show()
```


```
1/1             0s 69ms/step
```

3.9.1 2.1. Dropout

El **dropout** es una técnica de regularización utilizada para evitar el sobreajuste (overfitting) en redes neuronales. Durante el entrenamiento, puede ocurrir que las neuronas de una capa concreta siempre se vean influenciadas únicamente por la salida de una neurona específica de la capa anterior. Este comportamiento puede llevar a que la red memorice los datos de entrenamiento en lugar de generalizar bien a nuevos datos, lo cual se conoce como sobreajuste.

Dropout ayuda a prevenir este problema **cortando aleatoriamente las conexiones** (también conocido como *dropping the connection*) entre neuronas de capas sucesivas durante el entrenamiento.

Durante cada paso del entrenamiento, algunas neuronas y sus conexiones son *desactivadas* de forma aleatoria, lo que obliga a la red a aprender representaciones más robustas y generalizables.

3.9.2 2.2. Keras Optimizers

Existen varias variantes del **descenso de gradiente** (gradient descent), que se diferencian en cómo se calcula el paso de optimización. Estas variantes incluyen técnicas que adaptan la tasa de aprendizaje de forma dinámica durante el proceso de entrenamiento.

Keras soporta siete optimizadores principales:

```
my_opt = tensorflow.keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
my_opt = tensorflow.keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
my_opt = tensorflow.keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
my_opt = tensorflow.keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
my_opt = tensorflow.keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, dec
my_opt = tensorflow.keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, d
my_opt = tensorflow.keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=None, sc
```

Momentum **Momentum** es una técnica que mejora el descenso de gradiente estocástico (SGD) añadiendo una fracción del vector de actualización del paso anterior al vector de actualización actual. Esto ayuda a **acelerar el SGD** en la dirección correcta y a **amortiguar las oscilaciones** en la dirección de descenso:

$$v_t = mv_{t-1} + \alpha \nabla_w f$$

$$w = w - v_t$$

Donde: - v_t es el vector de velocidad en el tiempo t. - m es el término de momentum (generalmente fijado en 0.9). - α es la tasa de aprendizaje. - $\nabla_w f$ es el gradiente de la función de pérdida con respecto a los pesos w.

Momentum ayuda a mejorar la eficiencia del entrenamiento, especialmente en áreas con curvaturas pronunciadas.

Adagrad **Adagrad** ajusta la tasa de aprendizaje de cada parámetro de forma adaptativa, realizando **actualizaciones mayores** para los parámetros menos frecuentes y **menores** para los parámetros más frecuentes. Esto se logra acumulando los cuadrados de los gradientes a lo largo del tiempo:

$$c = c + (\nabla_w f)^2$$

$$w = w - \frac{\alpha}{\sqrt{c}}$$

Donde: - c es la acumulación de los cuadrados de los gradientes anteriores. - α es la tasa de aprendizaje inicial. - $\nabla_w f$ es el gradiente de la función de pérdida.

Adagrad es útil para problemas donde los parámetros requieren ajustes variados en función de su frecuencia.

RMSprop **RMSprop** es una mejora sobre Adagrad que introduce un factor de decaimiento para la acumulación de los gradientes cuadrados. Esto permite que el optimizador se adapte a los cambios en la tasa de aprendizaje de manera más eficiente:

$$c = \rho c + (1 - \rho)(\nabla_w f)^2$$

$$w = w - \frac{\alpha}{\sqrt{c + \epsilon}}$$

Donde: - c es la acumulación de los cuadrados de los gradientes ponderada por ρ. - ρ es un parámetro de decaimiento. - ϵ es un valor pequeño para evitar divisiones por cero.

RMSprop es eficaz para problemas no estacionarios y es ampliamente utilizado en redes neuronales profundas.

Adadelta **Adadelta** es una extensión de Adagrad que también usa un término de decaimiento para la acumulación de los gradientes, evitando la disminución excesiva de la tasa de aprendizaje con el tiempo:

$$c = \rho c + (1 - \rho)(\nabla_w f)^2$$

$$w = w - \frac{\Delta w}{\sqrt{c + \epsilon}}$$

Donde: - Δw es el cambio en los pesos. - ρ es el factor de decaimiento. - c es la acumulación de los gradientes. - ϵ es un valor pequeño para estabilidad numérica.

Adadelta ajusta la tasa de aprendizaje de manera dinámica, sin la necesidad de una tasa de aprendizaje global.

Adam **Adam** (Adaptive Moment Estimation) combina las ideas de Momentum y RMSprop. Utiliza estimaciones de primer y segundo momento de los gradientes para ajustar la tasa de aprendizaje de cada parámetro:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_w f$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_w f)^2$$

$$w = w - \frac{\alpha m_t}{\sqrt{v_t} + \epsilon}$$

Donde: - m_t y v_t son las estimaciones del primer y segundo momento, respectivamente. - β_1 **y** β_2 son los factores de decaimiento. - ϵ es un valor pequeño para estabilidad numérica.

Adam es popular debido a su rendimiento confiable en una amplia variedad de problemas y es adecuado para grandes volúmenes de datos y redes profundas.

Adamax **Adamax** es una variante de Adam que utiliza la **norma infinita** (max) en lugar de la norma cuadrada para la actualización de los pesos:

$$v_t = \max(\beta_2 v_{t-1}, |\nabla_w f|)$$

$$w = w - \frac{\alpha m_t}{v_t + \epsilon}$$

Donde: - v_t es la norma infinita acumulada.

Adamax puede ser más robusto en algunos escenarios, especialmente cuando los gradientes son muy grandes.

Nadam **Nadam** combina las ideas de Adam y **Nesterov Accelerated Gradient** (NAG). Usa estimaciones de primer y segundo momento de los gradientes y aplica un ajuste Nesterov a la actualización:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_w f$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_w f)^2$$

$$w = w - \frac{\alpha(m_t + \frac{1-\beta_1}{1-\beta_1^t}\nabla_w f)}{\sqrt{v_t} + \epsilon}$$

Donde: - m_t y v_t son las estimaciones del primer y segundo momento. - ϵ es un valor pequeño para estabilidad numérica.

Nadam es útil para problemas donde la combinación de estos métodos puede ofrecer un rendimiento superior en términos de velocidad de convergencia y precisión.

3.10 2.3. Convolutional Neural Network (CNN)

Los perceptrones multicapa, mencionados anteriormente, representan el modelo de red neuronal de avance más general y potente posible. Estos modelos se organizan en capas, de forma que cada neurona en una capa recibe como entrada su propia copia de todas las salidas de la capa anterior. Este enfoque es efectivo para aprender a partir de datos no estructurados mediante la representación de un número fijo de parámetros.

3.10.1 Desafíos con Datos de Imagen

Sin embargo, al enfrentar datos de imagen sin procesar, los perceptrones multicapa pueden resultar menos eficientes debido al número extremadamente alto de parámetros requeridos. Por ejemplo, considere una imagen de tamaño (200 $\times$200) píxeles conectada a 1024 neuronas en la capa de entrada. En un modelo de perceptrón multicapa tradicional, cada uno de los 40,000 (200 $\times$ 200) píxeles tendría que estar conectado a cada una de las 1024 neuronas, lo que implica un número de parámetros de:

$$40{,}000 \times 1{,}024 = 40{,}960{,}000$$

Este gran número de parámetros puede hacer que el modelo sea difícil de entrenar y propenso a problemas de sobreajuste.

3.10.2 Introducción a las Redes Neuronales Convolucionales (CNN)

Las **Redes Neuronales Convolucionales (CNN)** están diseñadas específicamente para abordar este problema. En lugar de conectar cada neurona de una capa con todas las neuronas de la capa siguiente, las CNN utilizan **convoluciones** para explorar patrones locales en los datos de imagen.

Arquitectura de una CNN Una CNN generalmente consta de varias capas, que incluyen:

- **Capas Convolucionales**: Aplican filtros convolucionales (también llamados *kernels*) a las imágenes de entrada para extraer características locales. Cada filtro se aplica a una pequeña región de la imagen y produce un mapa de características (feature map). Los filtros detectan patrones locales como bordes, texturas y formas.

  $$\text{Feature Map} = \text{Convolution}(\text{Image}, \text{Filter})$$

  Convolución: Operación matemática que permite extraer características locales de la imagen mediante el deslizamiento de un filtro sobre la imagen, multiplicando elemento por elemento y sumando los resultados.

- **Capas de Pooling**: Reducen la dimensión espacial de los mapas de características, manteniendo la información importante y disminuyendo el número de parámetros. Las operaciones de pooling más comunes son el *max pooling* y el *average pooling*.

  $$\text{Pooling} = \text{Max}(\text{Feature Map})$$

  Pooling: Técnica utilizada para reducir las dimensiones de los mapas de características y mejorar la eficiencia del modelo. Ayuda a extraer las características más destacadas de cada región sin necesidad de mantener toda la información espacial.

- **Capas de Normalización**: Normalizan las activaciones para mejorar la estabilidad y velocidad del entrenamiento, como la *Batch Normalization*.

- **Capas Densas**: Después de varias capas convolucionales y de pooling, la CNN suele tener capas densas para realizar la clasificación final o regresión, donde cada neurona en una capa densa está conectada a todas las neuronas de la capa anterior.

Ventajas de las CNN

- **Reducción del Número de Parámetros**: Al compartir pesos entre neuronas en una capa convolucional, se reduce significativamente el número total de parámetros, lo que hace que el entrenamiento sea más eficiente.
- **Extracción de Características Locales**: Las CNN son muy efectivas para capturar patrones locales y jerárquicos en las imágenes.
- **Invariancia a Desplazamientos**: Las operaciones de pooling y convolución permiten que las redes sean invariantes a pequeños desplazamientos y distorsiones en las imágenes.

Desventajas de las CNN

- **Requiere gran cantidad de datos etiquetados**: Las CNN necesitan una cantidad significativa de datos etiquetados para un buen rendimiento, lo que puede ser costoso o poco práctico en algunos escenarios.
- **Complejidad Computacional**: Aunque las CNNs son más eficientes que los MLPs, pueden seguir siendo computacionalmente intensivas, especialmente para imágenes de alta resolución o redes profundas.
- **Dependencia del diseño adecuado**: La arquitectura de la CNN debe ser cuidadosamente diseñada para evitar sobreajuste o bajo rendimiento en tareas específicas.

En resumen, las CNN son altamente eficaces para tareas de visión por computadora y otras aplicaciones que involucran datos estructurados en forma de grillas o mallas, como imágenes y vídeos. Su capacidad para extraer y aprender características a diferentes niveles de complejidad las convierte en una herramienta poderosa en el aprendizaje automático y la inteligencia artificial.

```
200 * 200 * 1024 = 40960000
```

La situación rápidamente se vuelve inmanejable a medida que el tamaño de las imágenes crece, mucho antes de alcanzar el tipo de imágenes con las que la gente usualmente quiere trabajar en aplicaciones reales.

Una solución común es reducir el tamaño de las imágenes a un tamaño donde las redes neuronales de varias capas (MLP) puedan aplicarse de manera segura. Sin embargo, si reducimos directamente el tamaño de la imagen, potencialmente perdemos una gran cantidad de información; sería ideal si pudiéramos hacer algún procesamiento útil de la imagen (sin causar una explosión en el conteo de parámetros) antes de realizar la reducción de tamaño.

Resulta que hay una forma muy eficiente de lograr esto, y se aprovecha de la estructura de la información codificada dentro de una imagen: se asume que los píxeles que están espacialmente más cercanos colaborarán mucho más en la formación de una característica particular de interés que aquellos en esquinas opuestas de la imagen. Además, si se encuentra que una característica (más pequeña) es de gran importancia al definir la etiqueta de una imagen, será igualmente importante si esta característica se encontrara en cualquier lugar dentro de la imagen, independientemente de la ubicación.

Aquí entra el operador de **convolución**. Dada una imagen bidimensional, I, y una pequeña matriz, K, de tamaño $h \times w$ (conocida como kernel de convolución), que asumimos codifica una forma de extraer una característica interesante de la imagen, calculamos la imagen convolucionada, $I * K$, superponiendo el kernel sobre la imagen de todas las maneras posibles y registrando la suma de los productos elemento a elemento entre la imagen y el kernel:

$$\text{output}(x, y) = (I * K)(x, y) = \sum_{m=0}^{h-1} \sum_{n=0}^{w-1} K(m, n) I(x - n, y - m)$$

Donde: - $I(x, y)$ es el valor del píxel en la posición (x, y) de la imagen de entrada. - $K(m, n)$ es el valor del píxel en la posición (m, n) del kernel de convolución. - h **y** w son las dimensiones del kernel (alto y ancho, respectivamente). - $(x - n, y - m)$ representa la posición en la imagen de entrada que se está multiplicando por el valor del kernel en la posición (m, n). - **output**(x, y) es el valor del píxel en la posición (x, y) de la imagen resultante después de la convolución.

El operador de convolución forma la base fundamental de la capa convolucional de una CNN. La capa está completamente especificada por cierto número de kernels, K, y opera calculando la convolución de las imágenes de salida de una capa anterior con cada uno de esos kernels, luego añadiendo los sesgos (uno por cada imagen de salida). Finalmente, puede aplicarse una función de activación, σ, a todos los píxeles de las imágenes de salida.

Típicamente, la entrada a una capa convolucional tendrá d canales (por ejemplo, rojo/verde/azul en la capa de entrada), en cuyo caso los kernels se extienden para tener este número de canales también. Es decir, un kernel en una CNN con una entrada RGB de 3 canales tendrá dimensiones $h \times w \times 3$.

Los kernels, que son matrices pequeñas de pesos, se aprenden durante el proceso de entrenamiento mediante técnicas de optimización como el descenso de gradiente. En contraste con los MLPs, que requieren una gran cantidad de parámetros para conectarse entre capas completas, los CNNs utilizan kernels compartidos que se aplican a toda la imagen, lo que reduce significativamente el número de parámetros.

Aplicaciones comunes de los CNNs incluyen el reconocimiento de imágenes, la detección de objetos, y el análisis de video. En estas aplicaciones, los CNNs pueden detectar características como bordes, texturas y formas, y combinar estas características en niveles sucesivos para identificar patrones complejos.

3.10.3 Pooling

Después de aplicar la operación de convolución, es común aplicar técnicas adicionales para mejorar el rendimiento del modelo y reducir la complejidad computacional. Entre estas técnicas se encuentra el *pooling*, una operación que reduce la dimensión espacial de las imágenes mientras mantiene las características importantes.

Tipos de pooling más comunes:

1. **Max Pooling:** Esta técnica selecciona el valor máximo de cada ventana del filtro. Por ejemplo, un maxpool de 2×2 recorre la imagen con una ventana de 2×2 píxeles y toma el valor máximo de cada ventana para crear una nueva imagen de salida con dimensiones reducidas. La operación de max pooling es eficaz para captar las características más importantes y es menos sensible a pequeñas variaciones en la posición de las características dentro de la imagen.

$$\text{output}(x, y) = \max\left(I(x, y), I(x + 1, y), I(x, y + 1), I(x + 1, y + 1)\right)$$

2. **Average Pooling:** En lugar de tomar el valor máximo, el average pooling calcula el promedio de los valores en cada ventana del filtro. Esto puede ser útil en algunos casos para reducir la varianza y proporcionar una representación más suave de la imagen.

$$\text{output}(x, y) = \frac{1}{4} \left(I(x, y) + I(x+1, y) + I(x, y+1) + I(x+1, y+1) \right)$$

3. **Global Average Pooling:** En lugar de aplicar un filtro de pooling a pequeñas regiones de la imagen, el global average pooling toma el promedio de toda la imagen para cada canal. Esto se utiliza a menudo para reducir la dimensionalidad de la imagen a un solo valor por canal, lo cual es útil para la clasificación final.

$$\text{output}_c = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} I_c(i, j)$$

   Donde H y W son la altura y el ancho de la imagen, respectivamente, y I_c es el valor del canal c.

4. **Global Max Pooling:** Similar al global average pooling, el global max pooling toma el valor máximo de toda la imagen para cada canal, en lugar del promedio. Esto puede ayudar a capturar las características más prominentes en la imagen.

$$\text{output}_c = \max_{i=1}^{H} \max_{j=1}^{W} I_c(i, j)$$

Ventajas del pooling:

- **Reducción de Dimensionalidad:** El pooling ayuda a reducir el tamaño de los mapas de características, lo que disminuye la carga computacional y la cantidad de parámetros en la red.
- **Invariancia a la Escala y a la Posición:** Al tomar el máximo o el promedio en una ventana de píxeles, el pooling hace que la red sea menos sensible a pequeños cambios en la posición de las características.
- **Prevención del Sobreajuste:** Reducir la dimensionalidad de los datos puede ayudar a prevenir el sobreajuste al limitar la cantidad de información que la red tiene que aprender.

En resumen, las técnicas de pooling, al igual que la convolución, son fundamentales para la eficacia de las redes neuronales convolucionales (CNNs), ya que permiten manejar imágenes de alta resolución de manera eficiente y extraer características importantes de manera robusta.

Más información sobre CNNs

[8]:
```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

"""
## Prepare the data
"""
```

```
# Parámetros del modelo y de los datos
num_classes = 10  # Número de clases en el conjunto de datos (dígitos del 0 al␣
 ↪9)
input_shape = (28, 28, 1)  # Forma de entrada de cada imagen (28x28 píxeles en␣
 ↪escala de grises)

# Carga del conjunto de datos MNIST, dividido en conjuntos de entrenamiento y␣
 ↪prueba
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Escalar las imágenes al rango [0, 1] dividiendo por 255
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

# Asegurarse de que las imágenes tengan forma (28, 28, 1) para que se ajusten a␣
 ↪la entrada de la red
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

# Convertir los vectores de clase a matrices de clases binarias para usar en la␣
 ↪función de pérdida
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

"""
## Build the model
"""

# Construcción del modelo de red neuronal convolucional
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),  # Entrada del modelo con la forma␣
 ↪especificada
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  # Primera␣
 ↪capa convolucional
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  # Segunda␣
 ↪capa convolucional
        layers.MaxPooling2D(pool_size=(2, 2)),  # Capa de max pooling para␣
 ↪reducir la dimensión espacial
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  # Tercera␣
 ↪capa convolucional
```

```
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  # Cuarta␣
 ↪capa convolucional
        layers.MaxPooling2D(pool_size=(2, 2)),  # Segunda capa de max pooling
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  # Quinta␣
 ↪capa convolucional
        layers.MaxPooling2D(pool_size=(2, 2)),  # Tercera capa de max pooling
        layers.Flatten(),  # Aplana la salida para conectar con la capa densa
        layers.Dropout(0.5),  # Capa de dropout para prevenir sobreajuste
        layers.Dense(num_classes, activation="softmax"),  # Capa densa de␣
 ↪salida con función softmax para clasificación
    ]
)

# Mostrar un resumen del modelo, incluyendo la arquitectura y los parámetros
model.summary()

"""
## Train the model
"""

# Parámetros de entrenamiento
batch_size = 128  # Tamaño del lote
epochs = 5  # Número de épocas

# Compilación del modelo especificando la función de pérdida, el optimizador y␣
 ↪las métricas
model.compile(loss="categorical_crossentropy", optimizer="adam",␣
 ↪metrics=["accuracy"])

# Entrenamiento del modelo con el conjunto de entrenamiento y validación
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,␣
 ↪validation_split=0.1)

"""
## Evaluate the trained model
"""

# Evaluación del modelo en el conjunto de prueba
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])  # Pérdida en el conjunto de prueba
print("Test accuracy:", score[1])  # Exactitud en el conjunto de prueba
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
conv2d_1 (Conv2D)	(None, 24, 24, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_2 (Conv2D)	(None, 10, 10, 64)	18,496
conv2d_3 (Conv2D)	(None, 8, 8, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 4, 4, 64)	0
conv2d_4 (Conv2D)	(None, 2, 2, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 64)	0
flatten (Flatten)	(None, 64)	0
dropout_2 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 10)	650

Total params: 102,570 (400.66 KB)

Trainable params: 102,570 (400.66 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/5
422/422            51s 116ms/step -
accuracy: 0.7110 - loss: 0.8501 - val_accuracy: 0.9808 - val_loss: 0.0653
Epoch 2/5
422/422            48s 114ms/step -
accuracy: 0.9534 - loss: 0.1568 - val_accuracy: 0.9882 - val_loss: 0.0423
Epoch 3/5
422/422            47s 111ms/step -
accuracy: 0.9689 - loss: 0.1057 - val_accuracy: 0.9908 - val_loss: 0.0334
Epoch 4/5
422/422            45s 107ms/step -
accuracy: 0.9757 - loss: 0.0864 - val_accuracy: 0.9905 - val_loss: 0.0336
```

```
Epoch 5/5
422/422             49s 117ms/step -
accuracy: 0.9815 - loss: 0.0672 - val_accuracy: 0.9917 - val_loss: 0.0299
Test loss: 0.028934597969055176
Test accuracy: 0.9909999966621399
```

[9]:
```
"""
## Visualize predictions
"""

# Establecer la semilla para la reproducibilidad
np.random.seed(40)

# Visualización de una pequeña parte de los datos de entrada y salida
# Seleccionamos un pequeño número de ejemplos para visualizar
num_examples = 5
indices = np.random.choice(x_test.shape[0], num_examples, replace=False)

# Configuración de la visualización
fig, axes = plt.subplots(1, num_examples, figsize=(15, 3))
fig.suptitle('Ejemplos de imágenes del conjunto de prueba')

for i, idx in enumerate(indices):
    ax = axes[i]
    # Redimensionamos la imagen para mostrar
    img = x_test[idx].reshape(28, 28)
    # Mostrar la imagen en escala de grises
    ax.imshow(img, cmap='gray')
    # Mostrar la etiqueta real en el título del subplot
    ax.set_title(f'Label: {np.argmax(y_test[idx])}')
    ax.axis('off')

plt.show()

# Predicciones del modelo para los ejemplos seleccionados
predictions = model.predict(x_test[indices])

# Visualización de las predicciones
fig, axes = plt.subplots(1, num_examples, figsize=(15, 3))
fig.suptitle('Predicciones del modelo')

for i, idx in enumerate(indices):
    ax = axes[i]
    # Redimensionamos la imagen para mostrar
    img = x_test[idx].reshape(28, 28)
    # Mostrar la imagen en escala de grises
    ax.imshow(img, cmap='gray')
```

```
    # Mostrar la predicción en el título del subplot
    ax.set_title(f'Pred: {np.argmax(predictions[i])}')
    ax.axis('off')

plt.show()
```


```
1/1                 0s 100ms/step
```

3.11 3. Redes Neuronales Recurrentes

Las redes neuronales clásicas, incluidas las redes neuronales convolucionales (CNN), sufren de dos limitaciones importantes:

- Solo aceptan un vector de tamaño fijo como entrada y producen un vector de tamaño fijo como salida.
- No consideran la naturaleza secuencial de algunos datos (por ejemplo, lenguaje, cuadros de video, series temporales, etc.).

Las **Redes Neuronales Recurrentes (RNN)** superan estas limitaciones al permitir operar sobre secuencias de vectores (en la entrada, en la salida o en ambos). Las RNN se llaman "recurrentes" porque realizan la misma tarea para cada elemento de la secuencia, y su salida depende no solo de la entrada actual, sino también de los cálculos previos (es decir, la información que ha sido procesada anteriormente en la secuencia).

Las fórmulas básicas de una RNN simple son:

$$s_t = f_1(Ux_t + Ws_{t-1})$$
$$y_t = f_2(Vs_t)$$

Donde: - s_t es el **estado oculto** en el tiempo t. El estado oculto representa una memoria de los valores anteriores y se utiliza para almacenar información relevante sobre la secuencia que la red ha procesado hasta el momento. - x_t es el **vector de entrada** en el tiempo t, que representa la información actual de la secuencia. - s_{t-1} es el **estado oculto** en el tiempo $t-1$, que captura información del paso anterior de la secuencia. - y_t es la **salida** en el tiempo t, que es la predicción o decisión que la red produce en ese paso, basada en el estado oculto actual. - U**,** W**, y** V son **matrices de parámetros** que se aprenden durante el entrenamiento: - U transforma la entrada actual x_t. - W transforma el estado oculto anterior s_{t-1}. - V transforma el estado oculto actual s_t para producir la salida y_t. - f_1 **y** f_2 son **funciones de activación**, que introducen no linealidades en las redes neuronales. Normalmente, se utiliza la tangente hiperbólica (tanh) para f_1 y la función softmax para f_2.

Estas ecuaciones indican que el estado oculto actual, s_t, se calcula como una función f_1 del estado oculto anterior s_{t-1} y la entrada actual x_t, utilizando los parámetros U y W. La salida y_t se calcula como una función f_2 del estado oculto actual s_t utilizando el parámetro V.

Dada una secuencia de entrada, aplicamos las fórmulas de RNN de manera **recurrente** hasta que procesamos todos los elementos de la secuencia. La RNN **comparte los parámetros** U**,** V**,** W en todos los pasos recurrentes. Podemos pensar en el estado oculto como una memoria de la red que captura información sobre los pasos anteriores.

La novedad de este tipo de red es que hemos codificado en la **arquitectura misma de la red** un esquema de modelado de secuencias, utilizado en el pasado para predecir series temporales y modelar el lenguaje. A diferencia de las arquitecturas anteriores, donde las capas ocultas estaban indexadas solo por un índice espacial (en redes convolucionales), ahora las capas ocultas están indexadas tanto por un índice 'espacial' como 'temporal', lo que es necesario para secuencias.

Las entradas de una red recurrente siempre son vectores. Sin embargo, en el caso del lenguaje o texto, estos vectores representan símbolos o palabras codificados como vectores numéricos (como vectores de palabras, o **embeddings**).

Supongamos que queremos clasificar una frase o una serie de palabras. Sean $x^1, ..., x^C$ los vectores de palabras correspondientes a un corpus con C símbolos. La relación para calcular las características de salida de la capa oculta en cada paso de tiempo t es:

$$h_t = \sigma(W s_{t-1} + U x_t)$$

Donde: - $x_t \in \mathbb{R}^d$ es el **vector de palabra** de entrada en el tiempo t, representando una palabra o símbolo del corpus. - $U \in \mathbb{R}^{D_h \times d}$ es la **matriz de pesos** para la entrada x_t. - $W \in \mathbb{R}^{D_h \times D_h}$ es la **matriz de pesos** para el estado oculto anterior s_{t-1}. - $s_{t-1} \in \mathbb{R}^{D_h}$ es el **estado oculto** en el paso de tiempo anterior $t-1$. - $\sigma(\cdot)$ es la **función de activación** (normalmente, la función tangente hiperbólica, tanh), que introduce no linealidades en el modelo.

La salida de esta red es:

$$\hat{y}_t = \text{softmax}(V h_t)$$

Donde $\hat{y}_t$ representa la **distribución de probabilidad** de salida sobre el vocabulario en cada paso de tiempo t. Esencialmente, $\hat{y}_t$ es la próxima palabra predicha dado el contexto del documento hasta ese momento (es decir, h_{t-1}) y el último vector de palabra observado $x^{(t)}$.

La **función de pérdida** utilizada en las RNN es a menudo el **error de entropía cruzada**, que mide la diferencia entre las probabilidades predichas por la red y las etiquetas verdaderas:

$$L^{(t)}(W) = -\sum_{j=1}^{|V|} y_{t,j} \times \log(\hat{y}_{t,j})$$

Donde: - $y_{t,j}$ es el **valor verdadero** para la etiqueta j en el tiempo t. - $\hat{y}_{t,j}$ es la **probabilidad predicha** para la etiqueta j en el tiempo t.

El error de entropía cruzada sobre un corpus de tamaño C es:

$$L = \frac{1}{C}\sum_{c=1}^{C} L^{(c)}(W) = -\frac{1}{C}\sum_{c=1}^{C}\sum_{j=1}^{|V|} y_{c,j} \times \log(\hat{y}_{c,j})$$

Las **arquitecturas simples de RNN** han demostrado ser muy propensas a **olvidar** información cuando las secuencias son largas y también son **muy inestables** cuando se entrenan. Esto se debe a fenómenos como el *desvanecimiento* o *explosión del gradiente*, donde los gradientes pueden volverse demasiado pequeños o grandes durante el proceso de retropropagación, lo que hace el entrenamiento ineficaz.

Para resolver estos problemas, se han propuesto varias arquitecturas alternativas que mejoran el rendimiento y la estabilidad de las RNN. Estas alternativas están basadas en la presencia de **unidades con puertas** (*gated units*), que permiten controlar el flujo de información en la red de manera más efectiva. Las unidades con puertas están compuestas por una capa de red neuronal sigmoidea y una operación de multiplicación punto a punto. Las dos arquitecturas más importantes son:

1. **Redes de Memorias a Largo Corto Plazo (LSTM)**: Las **LSTM** introducen **células de memoria** que pueden mantener información durante períodos largos de tiempo. Utilizan puertas de entrada, olvido y salida para gestionar la información que entra, sale y se olvida en la célula de memoria.
2. **Unidades Recurrentes con Puertas (GRU)**: Las **GRU** simplifican el diseño de las LSTM al combinar las puertas de entrada y olvido en una sola puerta, lo que reduce la complejidad computacional mientras se mantienen muchas de las ventajas de las LSTM.

Script de ejemplo para generar texto a partir de los escritos de Nietzsche. Se necesitan al menos 20 epochs antes de que el texto generado empiece a sonar coherente. Se recomienda ejecutar este script en la GPU, ya que las redes son bastante intensivas computacionalmente. Si pruebas este script con datos nuevos, asegurate de que tu corpus tenga al menos ~100k caracteres. ~1M es mejor.

[10]:
```
from __future__ import print_function
from tensorflow.keras.callbacks import LambdaCallback
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, LSTM, Input  # Importar␣
↪Input
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.utils import get_file
import numpy as np
```

```
import random
import sys
import io

# Descargar el archivo de texto de Nietzsche
path = get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/
 ↪nietzsche.txt')
# Leer el archivo en minúsculas
with io.open(path, encoding='utf-8') as f:
    text = f.read().lower()
print('Longitud del corpus:', len(text))

# Crear un mapeo de caracteres a índices y viceversa
chars = sorted(list(set(text)))
print('Total de caracteres únicos:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

# Cortar el texto en secuencias semi-redundantes de longitud maxlen
maxlen = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('Número de secuencias:', len(sentences))

print('Vectorizando...')
# Crear arrays para las entradas y salidas
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=bool)  # Usar `bool`␣
 ↪en lugar de `np.bool`
y = np.zeros((len(sentences), len(chars)), dtype=bool)  # Usar `bool` en lugar␣
 ↪de `np.bool`
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1

# Construir el modelo: una sola capa LSTM
print('Construyendo el modelo...')
model = Sequential([
    Input(shape=(maxlen, len(chars))),  # Usar Input(shape) en lugar de␣
 ↪especificar input_shape en la primera capa
    LSTM(128),  # Capa LSTM con 128 unidades
    Dense(len(chars)),  # Capa densa con una salida para cada carácter
```

```
    Activation('softmax')  # Función de activación softmax para predicción de␣
 ↪probabilidades
])

# Compilar el modelo
optimizer = RMSprop(learning_rate=0.01)  # Optimizador RMSprop con tasa de␣
 ↪aprendizaje de 0.01
model.compile(loss='categorical_crossentropy', optimizer=optimizer)

# Función auxiliar para muestrear un índice de una distribución de probabilidad
def sample(preds, temperature=1.0):
    """
    Muestrea un índice de una distribución de probabilidad dada.

    Args:
    preds (array): Distribución de probabilidad para cada índice.
    temperature (float): Parámetro que ajusta la "aleatoriedad" de la muestra.
                         Valores bajos hacen que la distribución sea más␣
 ↪concentrada
                         y valores altos la hacen más uniforme.

    Returns:
    int: Índice muestreado basado en la distribución de probabilidad.
    """
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature  # Ajuste de temperatura para la muestra
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)  # Muestra de la distribución␣
 ↪de probabilidad
    return np.argmax(probas)

# Función que se llama al final de cada época para generar texto
def on_epoch_end(epoch, logs):
    """
    Función llamada al final de cada época de entrenamiento para generar y␣
 ↪mostrar texto.

    Args:
    epoch (int): Número de la época actual.
    logs (dict): Diccionario con información sobre el entrenamiento en esta␣
 ↪época.
    """
    print()
    print('----- Generando texto después de la época: %d' % epoch)
```

```
    start_index = random.randint(0, len(text) - maxlen - 1)  # Seleccionar un índice inicial aleatorio
    for diversity in [0.2, 0.5, 1.0, 1.2]:  # Diferentes valores de "diversidad" para generar texto
        print('----- diversidad:', diversity)

        generated = ''
        sentence = text[start_index: start_index + maxlen]  # Secuencia inicial para generación
        generated += sentence
        print('----- Generando con semilla: "' + sentence + '"')
        sys.stdout.write(generated)

        for i in range(400):
            x_pred = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(sentence):
                x_pred[0, t, char_indices[char]] = 1.0

            preds = model.predict(x_pred, verbose=0)[0]  # Predecir el siguiente carácter
            next_index = sample(preds, diversity)  # Seleccionar el índice del siguiente carácter
            next_char = indices_char[next_index]  # Obtener el carácter correspondiente al índice

            generated += next_char
            sentence = sentence[1:] + next_char  # Desplazar la ventana de entrada

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()

# Definir un callback para generar texto al final de cada época
print_callback = LambdaCallback(on_epoch_end=on_epoch_end)

# Entrenar el modelo con el conjunto de datos
model.fit(x, y,
          batch_size=128,
          epochs=2,
          callbacks=[print_callback])  # Llamar a la función de generación de texto al final de cada época
```

```
Longitud del corpus: 600893
Total de caracteres únicos: 57
Número de secuencias: 200285
Vectorizando…
```

```
Construyendo el modelo…
Epoch 1/2

2024-11-15 09:25:19.023253: W
external/local_tsl/tsl/framework/cpu_allocator_impl.cc:83] Allocation of
456649800 exceeds 10% of free system memory.

1565/1565            0s 87ms/step -
loss: 2.3212
----- Generando texto después de la época: 0
----- diversidad: 0.2
----- Generando con semilla: "edy, as if it were the bad which deserve"
edy, as if it were the bad which deserves and the relight of the still the sense
the sense the some himself the still the expersent of the seef the seet and the
south and it is the seeved the still the seet and a such and a the such and the
soul and the religious and the still the southes and the sente the still the
seet and a the still and the religious and the sense the still the seet and the
sense the still the sout the such and and
----- diversidad: 0.5
----- Generando con semilla: "edy, as if it were the bad which deserve"
edy, as if it were the bad which deserved that it is a so the distinge of where
when the sidications and
not the every conception and and accerting is was a re one canness it is realon
and the subferis and of his neight for upinies of the sout so the reconcertion
and the accentered the more disciscanded in the lanks in it a gerere there
which, he he concept and the sincession of the have all the seet of the relied,
a mand not in the sti
----- diversidad: 1.0
----- Generando con semilla: "edy, as if it were the bad which deserve"
edy, as if it were the bad which deserved the
lact runges apmined oneire dis wealnion. as she ronem,
in being of aboagholdance.,
is one daspetended have intimcleness of a proconsso.--ofter with there vinded to
the ercessing it in the
lives us
the body the light of boud
that
othe unflopacing this
the els, verpotrimy the suction, us in
as suct egre't must and for greaty even this a fanned all
of mand it
is is sclethion and muned that heil
----- diversidad: 1.2
----- Generando con semilla: "edy, as if it were the bad which deserve"
edy, as if it were the bad which deservestions and science over
and ofledire are. hovery by this herenoking and naccles a recamatious.
the sbomout is the p
come vold when wad eptramolers=sension oneo--broal, g this
```

```
chemptul hollo,
; we
: andwhiles the
amore whomen scrmranction at look
andives. in divincal orbined that frel" thauqublain
tham the not any expersupon is
with when the presi,tabilitulal iganiah most yoow-broundis" and ackin
1565/1565             304s 194ms/step
- loss: 2.3210
Epoch 2/2
1565/1565             0s 100ms/step -
loss: 1.6728
----- Generando texto después de la época: 1
----- diversidad: 0.2
----- Generando con semilla: " attaches more importance
to himself tha"
 attaches more importance
to himself that in the sensition of the soul of the seasing of the seasing and
and the searion of the sense of with the conscious of a still and in a some have
to the sensimed of the sear of the sear of the possible of the sential the
seasing and has the probless and in in in instinction of the servation of moral
in the sear of the sear and the sensation of sense of the sender and in individ
and in individion o
----- diversidad: 0.5
----- Generando con semilla: " attaches more importance
to himself tha"
 attaches more importance
to himself that that was a moral out of the termines of perhapsent the science,
in luture of
certain not to be in explanted to the soul of as a solite of propers of the to
say of and consequently far the sensal should person were modern and in which
a probably that we has he is in saint of must be will and him of wering is a the
modern of the life and morality and conscious of become and the sacters of in
its c
----- diversidad: 1.0
----- Generando con semilla: " attaches more importance
to himself tha"
 attaches more importance
to himself that it was to had his at as his tainse. which was hopwould mys.
obdection and backers far thing and m. eypecual modern of above in
the schole withound by more reelormly mapperstificems of modary in under
thrakure and men. thwose we how certains
of the tree. justice of poyscist." out absuls, and witked tyned in man in the
blame has condivioualed in whom
inersbour, men and origif is out,
dony in actua
----- diversidad: 1.2
----- Generando con semilla: " attaches more importance
```

```
to himself tha"
 attaches more importance
to himself that
bything
feetiuten otner why was sake tragably,s were peikoply--is teptrous in theirlen
have utnous morakes conscreitioual-prably spiratients certuened )ight
of
gnound to se.sing, 3ectingm how thanding ay upiles to , vards, how perili_,
starime" he master call and has comisting in isiuse, it is hite
agior and trone, who whoced (a wantef it wat?-

67
i! it eigh, equisive and mocrutent us
(readbmin
1565/1565             325s 208ms/step
- loss: 1.6728
```

[10]: <keras.src.callbacks.history.History at 0x7f0fe2baa740>