Open in Colab

Clasificar FashionMNIST usando una Red Neuronal Convolucional

Lo que vamos a cubrir

Vamos a aplicar el flujo de trabajo de PyTorch a la visión artificial.

Específicamente, vamos a cubrir:

Tema	Contenido
0. Librerías de visión artificial en PyTorch	PyTorch tiene un conjunto de librerías integradas para la visión por computadora, vamos a verlas.
1. Configure el código agnóstico del dispositivo para modelos futuros	Es una buena práctica escribir código independiente del dispositivo, así que vamos a configurarlo.
2. Cargar datos	Para practicar la visión por computadora, comenzaremos con algunas imágenes de diferentes prendas de FashionMNIST.
3. Preparar datos	Tenemos algunas imágenes, carguémoslas con un PyTorch `DataLoader` para poder usarlas con nuestro ciclo de entrenamiento.
4. Modelo: Construcción de un modelo de Red Neuronal Convolucional (CNN)	Aquí crearemos un modelo de clasificación de varias clases para aprender patrones en los datos, también elegiremos una **función de pérdida**, un **optimizador** y construiremos un **ciclo de entrenamiento**.
5. Hacer predicciones y evaluar el modelo	Hagamos algunas predicciones sobre imágenes aleatorias y evaluemos nuestro mejor modelo.
6. Elaboración de una matriz de confusión	Una matriz de confusión es una excelente manera de evaluar un modelo de clasificación, veamos cómo podemos hacer uno.
7. Guardar y cargar el modelo	Como es posible que queramos usar nuestro modelo para más adelante, guardémoslo y asegurémonos de que se cargue correctamente.

0. Librerías de visión artificial en PyTorch

Antes de comenzar a escribir código, hablemos de algunas bibliotecas de visión por computadora de PyTorch que debe conocer.

Módulo PyTorch	¿Qué hace?
`torchvision`	Contiene conjuntos de datos, arquitecturas de modelos y transformaciones de imágenes que se utilizan a menudo para problemas de visión artificial.
`torchvision.datasets`	Aquí encontrará muchos ejemplos de conjuntos de datos de visión por computadora para una variedad de problemas, desde clasificación de imágenes, detección de objetos, subtítulos de imágenes, clasificación de videos y más. También contiene una serie de clases base para crear conjuntos de datos personalizados.
`torchvision.models`	Este módulo contiene arquitecturas de modelos de visión por computadora de buen rendimiento y de uso común implementadas en PyTorch, puede usarlas con sus propios problemas.
`torchvision.transforms`	A menudo, las imágenes deben transformarse (convertirse en números/procesarse/aumentarse) antes de usarse con un modelo; las transformaciones de imágenes comunes se encuentran aquí.
`torch.utils.data.Dataset`	Clase de conjunto de datos base para PyTorch.
`torch.utils.data.DataLoader`	Crea un iteralbe de Python sobre un conjunto de datos (creado con `torch.utils.data.Dataset`).

> **Nota:** Las clases `torch.utils.data.Dataset` y `torch.utils.data.DataLoader` no son solo para la visión por computadora en PyTorch, son capaces de manejar muchos tipos diferentes de datos.

Ahora que hemos cubierto algunas de las librerías de visión por computadora de PyTorch más importantes, importemos las dependencias relevantes.

Importar librerías

In [2]:
```
import torch, torchvision
from torchvision import datasets, transforms
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import DataLoader

import numpy as np
torchvision.__version__
```

Out[2]:
```
'0.13.1'
```

1.0 Fijar dispositivo GPU o CPU

In [3]:
```
# Fijar dispositivo GPU o CPU
if torch.backends.mps.is_available():
    dispositivo = 'mps'
elif torch.cuda.is_available():
    dispositivo = "cuda"
else: "cpu"
dispositivo
```

Out[3]:
```
'mps'
```

2. Cargar los datos

Para comenzar a trabajar en un problema de visión por computadora, obtengamos un conjunto de datos de visión por computadora.

Vamos a empezar con FashionMNIST.

El conjunto de datos original del MNIST contiene miles de ejemplos de dígitos escritos a mano (del 0 al 9) y se usó para construir modelos de visión por computadora para identificar números para los servicios postales.

FashionMNIST, hecho por Zalando Research, es una configuración similar.

Excepto que contiene imágenes en escala de grises de 10 tipos diferentes de ropa.

`PyTorch` tiene un montón de conjuntos de datos comunes de visión por computadora almacenados en `torchvision.datasets`.

Incluyendo FashionMNIST en `torchvision.datasets.FashionMNIST()`.

Para descargarlo, proporcionamos los siguientes parámetros:

- `root: str`: ¿a qué carpeta desea descargar los datos?
- `train: Bool`: ¿quieres la data de entrenamiento o la deprueba?
- `download: Bool`: ¿deberían descargarse los datos?
- `transform: torchvision.transforms` - ¿Qué transformaciones le gustaría hacer en los datos?
- `target_transform`: también puede transformar los objetivos (etiquetas) si lo desea.

Muchos otros conjuntos de datos en `torchvision` tienen estas opciones de parámetros.

In [4]:
```
# Bajar la data de entrenamiento
datos_entrenamiento = datasets.FashionMNIST(
    root='../datos/',
    download=True,
    train=True,
    transform=transforms.ToTensor())

# Bajar la data de validación
datos_validacion = datasets.FashionMNIST(
    root='../datos/',
    download=True,
    train=False,
    transform=transforms.ToTensor())
```

In [6]:
```
imagen, etiqueta = datos_entrenamiento[0]
imagen, etiqueta
```

Out[6]:
```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0039, 0.0000, 0.0000, 0.0510,
           0.2863, 0.0000, 0.0000, 0.0039, 0.0157, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0039, 0.0039, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0000, 0.1412, 0.5333,
           0.4980, 0.2431, 0.2118, 0.0000, 0.0000, 0.0000, 0.0039, 0.0118,
           0.0157, 0.0000, 0.0000, 0.0118],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0235, 0.0000, 0.4000, 0.8000,
           0.6902, 0.5255, 0.5647, 0.4824, 0.0902, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0471, 0.0392, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.6078, 0.9255,
           0.8118, 0.6980, 0.4196, 0.6118, 0.6314, 0.4275, 0.2510, 0.0902,
           0.3020, 0.5098, 0.2824, 0.0588],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0039, 0.0000, 0.2706, 0.8118, 0.8745,
           0.8549, 0.8471, 0.8471, 0.6392, 0.4980, 0.4745, 0.4784, 0.5725,
           0.5529, 0.3451, 0.6745, 0.2588],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0039, 0.0039, 0.0039, 0.0000, 0.7843, 0.9098, 0.9098,
           0.9137, 0.8980, 0.8745, 0.8745, 0.8431, 0.8353, 0.6431, 0.4980,
```

```
 0.4824, 0.7686, 0.8980, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.7176, 0.8824, 0.8471,
 0.8745, 0.8941, 0.9216, 0.8902, 0.8784, 0.8706, 0.8784, 0.8667,
 0.8745, 0.9608, 0.6784, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.7569, 0.8941, 0.8549,
 0.8353, 0.7765, 0.7059, 0.8314, 0.8235, 0.8275, 0.8353, 0.8745,
 0.8627, 0.9529, 0.7922, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0039, 0.0118, 0.0000, 0.0471, 0.8588, 0.8627, 0.8314,
 0.8549, 0.7529, 0.6627, 0.8902, 0.8157, 0.8549, 0.8784, 0.8314,
 0.8863, 0.7725, 0.8196, 0.2039],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0235, 0.0000, 0.3882, 0.9569, 0.8706, 0.8627,
 0.8549, 0.7961, 0.7765, 0.8667, 0.8431, 0.8353, 0.8706, 0.8627,
 0.9608, 0.4667, 0.6549, 0.2196],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0157, 0.0000, 0.0000, 0.2157, 0.9255, 0.8941, 0.9020,
 0.8941, 0.9412, 0.9098, 0.8353, 0.8549, 0.8745, 0.9176, 0.8510,
 0.8510, 0.8196, 0.3608, 0.0000],
[0.0000, 0.0000, 0.0039, 0.0157, 0.0235, 0.0275, 0.0078, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.9294, 0.8863, 0.8510, 0.8745,
 0.8706, 0.8588, 0.8706, 0.8667, 0.8471, 0.8745, 0.8980, 0.8431,
 0.8549, 1.0000, 0.3020, 0.0000],
[0.0000, 0.0118, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.2431, 0.5686, 0.8000, 0.8941, 0.8118, 0.8353, 0.8667,
 0.8549, 0.8157, 0.8275, 0.8549, 0.8784, 0.8745, 0.8588, 0.8431,
 0.8784, 0.9569, 0.6235, 0.0000],
[0.0000, 0.0000, 0.0000, 0.0000, 0.0706, 0.1725, 0.3216, 0.4196,
 0.7412, 0.8941, 0.8627, 0.8706, 0.8510, 0.8863, 0.7843, 0.8039,
 0.8275, 0.9020, 0.8784, 0.9176, 0.6902, 0.7373, 0.9804, 0.9725,
 0.9137, 0.9333, 0.8431, 0.0000],
[0.0000, 0.2235, 0.7333, 0.8157, 0.8784, 0.8667, 0.8784, 0.8157,
 0.8000, 0.8392, 0.8157, 0.8196, 0.7843, 0.6235, 0.9608, 0.7569,
 0.8078, 0.8745, 1.0000, 1.0000, 0.8667, 0.9176, 0.8667, 0.8275,
 0.8627, 0.9098, 0.9647, 0.0000],
[0.0118, 0.7922, 0.8941, 0.8784, 0.8667, 0.8275, 0.8275, 0.8392,
 0.8039, 0.8039, 0.8039, 0.8627, 0.9412, 0.3137, 0.5882, 1.0000,
 0.8980, 0.8667, 0.7373, 0.6039, 0.7490, 0.8235, 0.8000, 0.8196,
 0.8706, 0.8941, 0.8824, 0.0000],
[0.3843, 0.9137, 0.7765, 0.8235, 0.8706, 0.8980, 0.8980, 0.9176,
 0.9765, 0.8627, 0.7608, 0.8431, 0.8510, 0.9451, 0.2549, 0.2863,
 0.4157, 0.4588, 0.6588, 0.8588, 0.8667, 0.8431, 0.8510, 0.8745,
 0.8745, 0.8784, 0.8980, 0.1137],
[0.2941, 0.8000, 0.8314, 0.8000, 0.7569, 0.8039, 0.8275, 0.8824,
 0.8471, 0.7255, 0.7725, 0.8078, 0.7765, 0.8353, 0.9412, 0.7647,
 0.8902, 0.9608, 0.9373, 0.8745, 0.8549, 0.8314, 0.8196, 0.8706,
 0.8627, 0.8667, 0.9020, 0.2627],
[0.1882, 0.7961, 0.7176, 0.7608, 0.8353, 0.7725, 0.7255, 0.7451,
 0.7608, 0.7529, 0.7922, 0.8392, 0.8588, 0.8667, 0.8627, 0.9255,
 0.8824, 0.8471, 0.7804, 0.8078, 0.7294, 0.7098, 0.6941, 0.6745,
 0.7098, 0.8039, 0.8078, 0.4510],
```

```
          [0.0000, 0.4784, 0.8588, 0.7569, 0.7020, 0.6706, 0.7176, 0.7686,
           0.8000, 0.8235, 0.8353, 0.8118, 0.8275, 0.8235, 0.7843, 0.7686,
           0.7608, 0.7490, 0.7647, 0.7490, 0.7765, 0.7529, 0.6902, 0.6118,
           0.6549, 0.6941, 0.8235, 0.3608],
          [0.0000, 0.0000, 0.2902, 0.7412, 0.8314, 0.7490, 0.6863, 0.6745,
           0.6863, 0.7098, 0.7255, 0.7373, 0.7412, 0.7373, 0.7569, 0.7765,
           0.8000, 0.8196, 0.8235, 0.8235, 0.8275, 0.7373, 0.7373, 0.7608,
           0.7529, 0.8471, 0.6667, 0.0000],
          [0.0078, 0.0000, 0.0000, 0.0000, 0.2588, 0.7843, 0.8706, 0.9294,
           0.9373, 0.9490, 0.9647, 0.9529, 0.9569, 0.8667, 0.8627, 0.7569,
           0.7490, 0.7020, 0.7137, 0.7137, 0.7098, 0.6902, 0.6510, 0.6588,
           0.3882, 0.2275, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1569,
           0.2392, 0.1725, 0.2824, 0.1608, 0.1373, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000]]]),
 9)
```

2.1 Formas de entrada y salida de un modelo de visión artificial

Tenemos un gran tensor de valores (la imagen) que conduce a un solo valor para el objetivo (la etiqueta).

Veamos la forma de la imagen.

In [7]:
```
# Cuál es la forma de la imagen
imagen.shape
```

Out[7]:
```
torch.Size([1, 28, 28])
```

La forma del tensor de imagen es [1, 28, 28] o más específicamente:

```
[canales_color=1, alto=28, ancho=28]
```

Tener `canales_color=1` significa que la imagen está en escala de grises.

Si `canales_color=3`, la imagen viene en valores de píxeles para rojo, verde y azul (esto también se conoce como modelo de color RGB).

In [8]:
```
# Cuál es el número de ejemplos?
len(datos_entrenamiento), len(datos_entrenamiento.targets), len(datos_valida
```

Out[8]:
```
(60000, 60000, 10000, 10000)
```

Así que tenemos 60 000 muestras de entrenamiento y 10 000 muestras de prueba.

¿Qué clases hay?

Podemos encontrarlos a través del atributo `.classes`.

In [9]:
```
# Ver las clases
nombre_clases = datos_entrenamiento.classes
nombre_clases
```

Out[9]:
```
['T-shirt/top',
 'Trouser',
 'Pullover',
 'Dress',
 'Coat',
 'Sandal',
 'Shirt',
 'Sneaker',
 'Bag',
 'Ankle boot']
```

Parece que estamos tratando con 10 tipos diferentes de ropa.

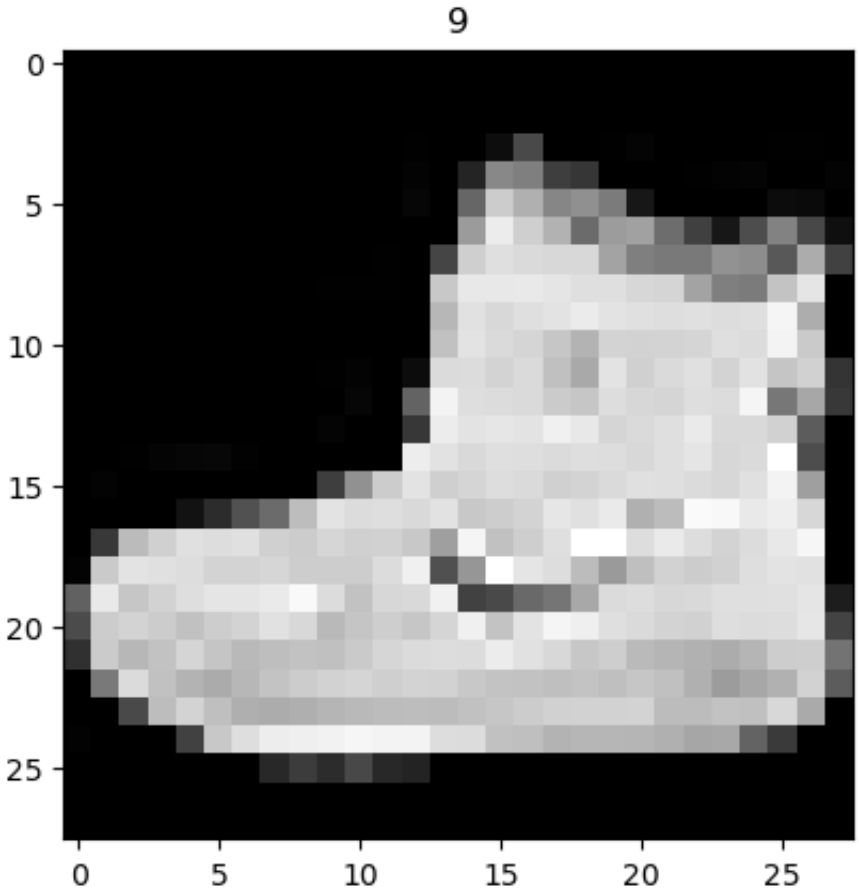
Debido a que estamos trabajando con 10 clases diferentes, significa que nuestro problema es **clasificación multiclase**.

Visualicemos.

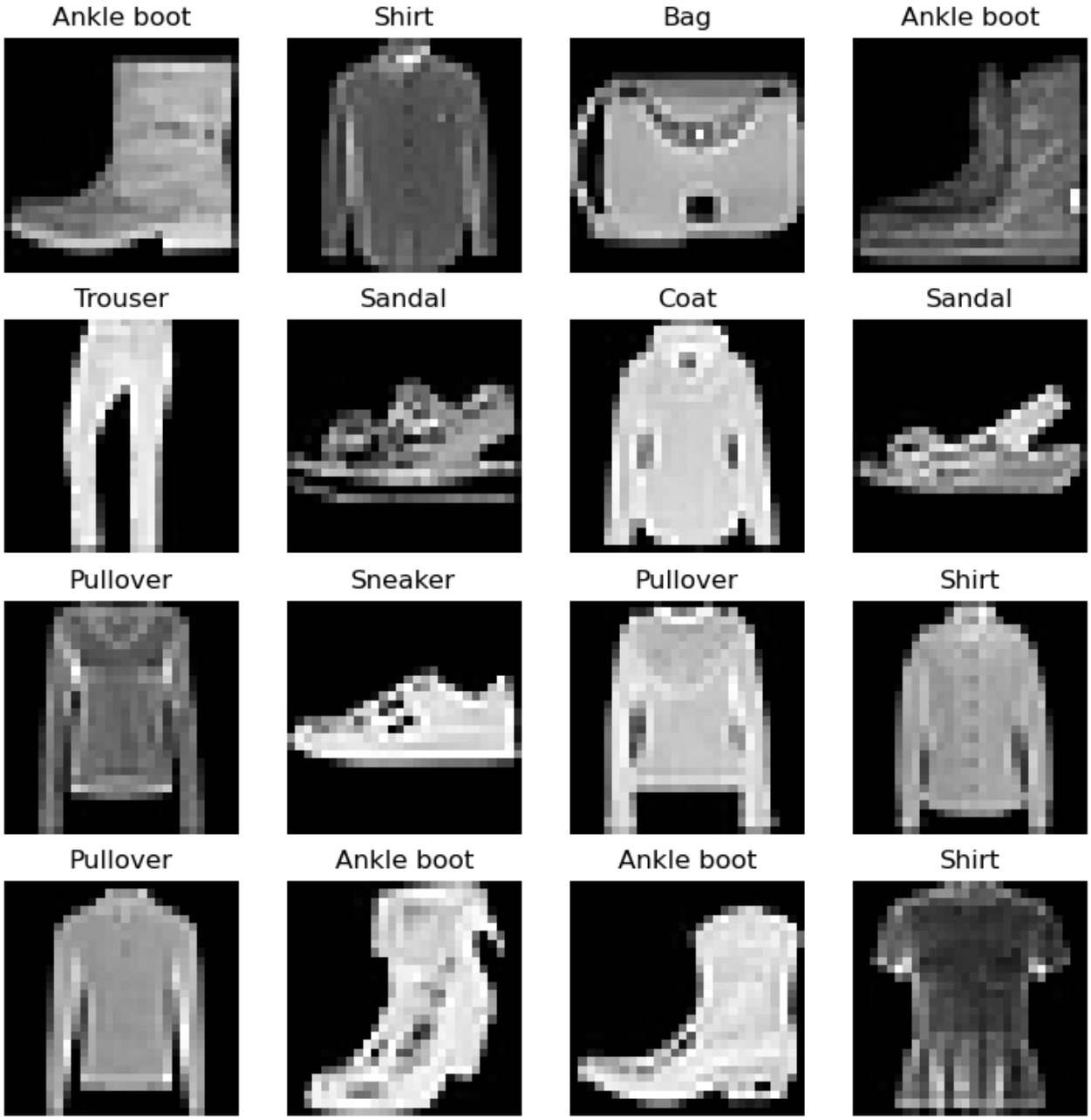
2.2 Visualizar los datos

In [16]:
```
import matplotlib.pyplot as plt
imagen, etiqueta = datos_entrenamiento[0]
print(f"Forma de la imagen: {imagen.shape}")
plt.imshow(imagen.squeeze(), cmap='gray')
plt.title(etiqueta);
```

```
Forma de la imagen: torch.Size([1, 28, 28])
```



In [17]:
```
# Graficar mas imagenes
torch.manual_seed(42)
fig = plt.figure(figsize=(9, 9))
filas, columnas = 4, 4
for i in range(1, filas * columnas + 1):
    index_aleatorio = torch.randint(0, len(datos_entrenamiento), size=[1]).i
    imgagen, etiqueta = datos_entrenamiento[index_aleatorio]
    fig.add_subplot(filas, columnas, i)
    plt.imshow(imgagen.squeeze(), cmap="gray")
    plt.title(nombre_clases[etiqueta])
    plt.axis(False);
```


Ankle boot
Shirt
Bag
Ankle boot
Trouser
Sandal
Coat
Sandal
Pullover
Sneaker
Pullover
Shirt
Pullover
Ankle boot
Ankle boot
Shirt

3. Preparar DataLoader

Ahora tenemos un conjunto de datos listo para funcionar.

El siguiente paso es prepararlo con un `torch.utils.data.DataLoader` o `DataLoader` para abreviar.

El `DataLoader` hace lo siguiente:

- Ayuda a cargar datos a un modelo. Para entrenamiento y para inferencia.
- Convierte un gran `Dataset` en un Python iterable de fragmentos más pequeños.
- Estos fragmentos más pequeños se denominan *lotes* o *mini lotes* y se pueden configurar mediante el parámetro `batch_size`.
    - ¿Por qué hacer esto?
    - Porque es más eficiente computacionalmente.
    - En un mundo ideal, podría hacer el pase hacia adelante y hacia atrás en todos sus datos a la vez.
    - Pero una vez que comienza a usar conjuntos de datos realmente grandes, a menos que tenga una potencia computacional infinita, es más fácil dividirlos en lotes.
    - También le da al modelo más oportunidades para mejorar.
    - Con minilotes (pequeñas porciones de datos), el descenso de gradiente se realiza más a menudo por época (una vez por minilote en lugar de una vez por época).
    - ¿Cuál es un buen tamaño de lote?
    - 32 es un buen lugar para comenzar con una buena cantidad de problemas.
    - Pero dado que este es un valor que puede establecer (un hiperparámetro), puede probar diferentes tipos de valores, aunque generalmente se usan potencias de 2 con mayor frecuencia (por ejemplo, 32, 64, 128, 256, 512).

A continuación se crearan los `DataLoader` para nuestros conjuntos de entrenamiento y validación

In [19]:
```
tamaño_lote = 32

# Crear cargadores para los datos de entrenamiento y validación
cargador_entrenamiento = DataLoader(datos_entrenamiento,
                                    batch_size=tamaño_lote,
                                    shuffle=True)
cargador_validacion = DataLoader(datos_validacion,
                                 batch_size=tamaño_lote,
                                 shuffle=False)

print(f"Dataloaders: {cargador_entrenamiento, cargador_validacion}")
print(f"Length of train dataloader: {len(cargador_entrenamiento)} batches of
print(f"Length of test dataloader: {len(cargador_validacion)} batches of {ta
```

```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x28f2d9a60>
, <torch.utils.data.dataloader.DataLoader object at 0x28f2e6f10>)
Length of train dataloader: 1875 batches of 32
Length of test dataloader: 313 batches of 32
```

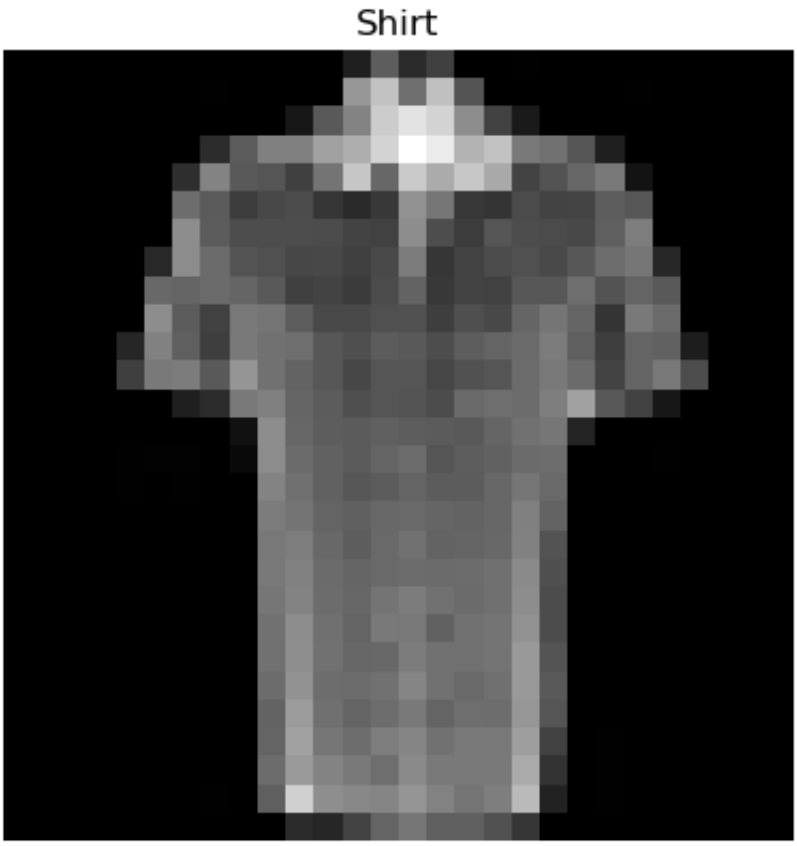
In [20]:
```
# Que hay dentro del cargador de entrenamiento
atributos_lote_entrenamiento, etiquetas_lote_entrenamiento = next(iter(carga
atributos_lote_entrenamiento.shape, etiquetas_lote_entrenamiento.shape
```

Out[20]:
```
(torch.Size([32, 1, 28, 28]), torch.Size([32]))
```

Y podemos ver que los datos permanecen sin cambios al verificar una sola muestra.

In [23]:
```
# Mostrar un ejemplo
torch.manual_seed(42)
index_aleatorio = torch.randint(0, len(atributos_lote_entrenamiento), size=[
imagen, etiqueta = atributos_lote_entrenamiento[index_aleatorio], etiquetas_
plt.imshow(imagen.squeeze(), cmap="gray")
plt.title(nombre_clases[etiqueta])
plt.axis("Off");
print(f"Tamaño de la imagen: {imagen.shape}")
print(f"Etiqueta: {etiqueta}, tamaño etiqueta: {etiqueta.shape}")
```

```
Tamaño de la imagen: torch.Size([1, 28, 28])
Etiqueta: 6, tamaño etiqueta: torch.Size([])
```



4. Construir el modelo como una Red Neuronal Convolucional en PyTorch

Vamos a crear una Red Neuronal Convolucional (CNN o ConvNet).

Las CNN son conocidas por sus capacidades para encontrar patrones en datos visuales.

El modelo de CNN que vamos a usar se conoce como TinyVGG del sitio web CNN Explainer.

Sigue la estructura típica de una red neuronal convolucional:

`Capa de entrada -> [Capa convolucional -> capa de activación -> capa de agrupación] -> Capa de salida`

Donde el contenido de `[Capa convolucional -> capa de activación -> capa de agrupación]` puede ampliarse y repetirse varias veces, según los requisitos.

Crearemos una CNN que replique el modelo del sitio web CNN Explainer.

In [37]:
```
class TinyVGG(nn.Module):
    """
    Arquitectura del modelo basada en TinyVGG:
    https://poloclub.github.io/cnn-explainer/
    """
    def __init__(self, forma_entrada: int, unidades_ocultas: int, forma_sali
        super().__init__()
        self.bloque_1 = nn.Sequential(
            nn.Conv2d(in_channels=forma_entrada,
                      out_channels=unidades_ocultas,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=unidades_ocultas,
                      out_channels=unidades_ocultas,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
        self.bloque_2 = nn.Sequential(
            nn.Conv2d(unidades_ocultas, unidades_ocultas, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(unidades_ocultas, unidades_ocultas, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.clasificador = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=unidades_ocultas*7*7,
                      out_features=forma_salida)
        )

    def forward(self, x: torch.Tensor):
        x = self.bloque_1(x)
        x = self.bloque_2(x)
        x = self.clasificador(x)
        return x
```

In [28]:
```
torch.manual_seed(42)
modelo = TinyVGG(forma_entrada=1,
    unidades_ocultas=10,
    forma_salida=len(nombre_clases)).to(dispositivo)
modelo
```

Out[28]:
```
FashionMNISTModelo(
  (bloque_1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
  )
  (bloque_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
  )
  (clasificador): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=490, out_features=10, bias=True)
  )
)
```

4.1 Definir funciones `paso_entrenamiento()` y `paso_validacion()`

In [34]:
```
def paso_entrenamiento(modelo,
                       dataloader,
                       perdida_fn,
                       optimizador):
    # Colocar modelo en modo entrenamiento
    modelo.train()

    # Inicializar valores de perdida y exactitud del entrenamiento
    perdida_entrenamiento, exactitud_entrenamiento = 0, 0

    # Iterar sobre los lotes del DataLoader
    for imagenes, etiquetas in dataloader:
        # Enviar datos al dispositivo
        imagenes, etiquetas = imagenes.to(dispositivo), etiquetas.to(disposi

        # 1. Propagar hacia adelante los datos de entrenamiento usando el mé
        salida = modelo(imagenes)

        # 2. Calcule la pérdida (qué tan diferentes son las predicciones de
        perdida = perdida_fn(salida, etiquetas)
        perdida_entrenamiento += perdida.item()

        # 3. Colocar a cero los gradientes del optimizador
        optimizador.zero_grad()

        # 4. Propagación hacia atrás
        perdida.backward()

        # 5. Realizar paso de optimización
        optimizador.step()

        # Calcular y acumular la exactitud sobre todos los lotes
        clase_predicha = torch.argmax(torch.softmax(salida, dim=1), dim=1)
        exactitud_entrenamiento += (clase_predicha == etiquetas).sum().item(

    # Ajustar métricas para obtener un promedio de la perdida y la exactituc
    perdida_entrenamiento /= len(dataloader)
    exactitud_entrenamiento /= len(dataloader)

    return perdida_entrenamiento, exactitud_entrenamiento
```

In [53]:
```python
def paso_validacion(modelo,
                    dataloader,
                    perdida_fn):

    # Colocar el modelo en modo evaluación
    modelo.eval()

    # Inicializar valores de perdida y exactitud de la validación
    perdida_validacion, exactitud_validacion = 0, 0

    # Iniciar el manejador de contexto para inferencia
    with torch.inference_mode():
        # Iterar sobre los lotes del DataLoader
        for imagenes, etiquetas in dataloader:
            # Enviar datos al dispositivo
            imagenes, etiquetas = imagenes.to(dispositivo), etiquetas.to(dis

            # 1. Propagar hacia adelante los datos de entrenamiento usando e
            salida = modelo(imagenes)

            # 2. Calcular y acumular la pérdida
            perdida = perdida_fn(salida, etiquetas)
            perdida_validacion += perdida.item()

            # 3. Calcular y acumular la exactitud
            clases_predicha = salida.argmax(dim=1)
            exactitud_validacion += ((clases_predicha == etiquetas).sum().it

    # Ajustar métricas para obtener un promedio de la perdida y la exactituc
    perdida_validacion /= len(dataloader)
    exactitud_validacion /= len(dataloader)

    return perdida_validacion, exactitud_validacion
```

4.2 Crear una función `entrenar()` para combinar `paso_entrenamiento()` y `paso_validacion()`

Ahora necesitamos una manera de juntar nuestras funciones `paso_entrenamiento()` y `paso_validacion()`.

Para hacerlo, los empaquetaremos en una función `entrenar()`.

Esta función entrenará el modelo y lo evaluará.

Específicamente, será:

1. Tome un modelo, un 'DataLoader' para conjuntos de entrenamiento y prueba, un optimizador, una función de pérdida y para cuántas épocas realizar cada paso de entrenamiento y prueba.
2. Cree un diccionario de resultados vacío para los valores `perdida_entrenamiento`, `exactitud_entrenamiento`, `perdida_validacion` y `exactitud_validacion` (podemos llenarlo a medida que avanza el entrenamiento).
3. Iterrar las funciones de paso de prueba y entrenamiento por varias épocas.
4. Imprime lo que sucede al final de cada época.
5. Actualiza el diccionario de resultados vacío con las métricas actualizadas cada época.
6. Devolver los resultados

Para realizar un seguimiento de la cantidad de épocas por las que hemos pasado, importemos `tqdm` desde `tqdm.auto` (`tqdm` es una de las librerías más populares de barra de progreso para Python y `tqdm.auto` decide automáticamente qué tipo de barra de progreso es mejor para su entorno informático, por ejemplo, Jupyter Notebook vs. Python script).

In [54]:
```
from tqdm.auto import tqdm

def entrenar(modelo: torch.nn.Module,
             cargador_entrenamiento: torch.utils.data.DataLoader,
             cargador_evaluacion: torch.utils.data.DataLoader,
             optimizador: torch.optim.Optimizer,
             perdida_fn: torch.nn.Module = nn.CrossEntropyLoss(),
             epocas: int = 5):

    # 2. Crear diccionario vacio para los resultados
    resultados = {"perdida_entrenamiento": [],
        "exactitud_entrenamiento": [],
        "perdida_evaluacion": [],
        "exactitud_evaluacion": []
    }

    # 3. Iterar sobre los pasos de entrenamiento y prueba por un número de e
    for epoca in tqdm(range(epocas)):
        perdida_entrenamiento, exactitud_entrenamiento = paso_entrenamiento(



        perdida_evaluacion, exactitud_evaluacion = paso_validacion(modelo=mc
                                                                   dataloade
                                                                   perdida_f

        # 4. Imprimir que esta pasando
        print(
            f"Epoca: {epoca+1} | "
            f"perdida_entrenamiento: {perdida_entrenamiento:.4f} | "
            f"exactitud_entrenamiento: {exactitud_entrenamiento:.4f} | "
            f"perdida_evaluación: {perdida_evaluacion:.4f} | "
            f"exactitud_evaluación: {exactitud_evaluacion:.4f}"
        )

        # 5. Actualizar el diccionario de resultados
        resultados["perdida_entrenamiento"].append(perdida_entrenamiento)
        resultados["exactitud_entrenamiento"].append(exactitud_entrenamiento
        resultados["perdida_evaluacion"].append(perdida_evaluacion)
        resultados["exactitud_evaluacion"].append(exactitud_evaluacion)

    # 6. Retornar el diccionario de resultadoss al final de cada epoca
    return resultados
```

4.3 Entrenar y evaluar el modelo

Bien, bien, bien, tenemos todos los ingredientes que necesitamos para entrenar y evaluar nuestro modelo.

Es hora de juntar nuestro modelo `TinyVGG`, las funciones `DataLoader` y `train()` para ver si podemos construir un modelo capaz de discernir entre pizza, bistec y sushi.

Vamos a recrear `modelo` (no es necesario, pero lo haremos para completar) y luego llamamos a nuestra función `entrenar()` pasando los parámetros necesarios.

Para que nuestros experimentos sean rápidos, entrenaremos nuestro modelo durante **10 épocas** (aunque puede aumentar esto si lo desea).

En cuanto al **optimizador** y la **función de pérdida**, usaremos `torch.nn.CrossEntropyLoss()` (dado que estamos trabajando con datos de clasificación de clases múltiples) y `torch.optim.Adam( )` con una tasa de aprendizaje de `1e-3` respectivamente.

Para ver cuánto tardan las cosas, importaremos el método `timeit.default_timer()` de Python para calcular el tiempo de entrenamiento. .

In [66]:
```
# Fijar semilla del generador de números aleatorios
torch.manual_seed(42)
torch.cuda.manual_seed(42)

# Fijar el número de epocas
NUMERO_EPOCAS = 20

# Recrear una instancia de TinyVGG
modelo = TinyVGG(forma_entrada=1,
                 unidades_ocultas=10,
                 forma_salida=len(datos_entrenamiento.classes)).to(dispositi

# Fijar la funcion de perdida y el optimizados
perdida_fn = nn.CrossEntropyLoss()
optimizador = torch.optim.Adam(params=modelo.parameters(), lr=0.001)

# Inicializar el temporizador
from timeit import default_timer as timer
tiempo_inicial = timer()

# Entrenar modelo
modelo_resultados = entrenar(modelo=modelo,
                             cargador_entrenamiento=cargador_entrenamiento,
                             cargador_evaluacion=cargador_validacion,
                             optimizador=optimizador,
                             perdida_fn=perdida_fn,
                             epocas=NUMERO_EPOCAS)

# Finalizar el temporizador e imprimir cuanto tardo el entrenamiento
tiempo_final = timer()
print(f"Tiempo total de entrenamiento: {tiempo_final-tiempo_inicial:.3f} seg
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
```

```
Epoca: 1 | perdida_entrenamiento: 0.5437 | exactitud_entrenamiento: 0.8031 |
perdida_evaluación: 0.4067 | exactitud_evaluación: 0.8527
Epoca: 2 | perdida_entrenamiento: 0.3592 | exactitud_entrenamiento: 0.8709 |
perdida_evaluación: 0.3526 | exactitud_evaluación: 0.8723
Epoca: 3 | perdida_entrenamiento: 0.3250 | exactitud_entrenamiento: 0.8819 |
perdida_evaluación: 0.3435 | exactitud_evaluación: 0.8751
Epoca: 4 | perdida_entrenamiento: 0.3008 | exactitud_entrenamiento: 0.8908 |
perdida_evaluación: 0.3236 | exactitud_evaluación: 0.8875
Epoca: 5 | perdida_entrenamiento: 0.2858 | exactitud_entrenamiento: 0.8972 |
perdida_evaluación: 0.3160 | exactitud_evaluación: 0.8888
Epoca: 6 | perdida_entrenamiento: 0.2729 | exactitud_entrenamiento: 0.9011 |
perdida_evaluación: 0.3079 | exactitud_evaluación: 0.8933
Epoca: 7 | perdida_entrenamiento: 0.2648 | exactitud_entrenamiento: 0.9042 |
perdida_evaluación: 0.2877 | exactitud_evaluación: 0.9024
Epoca: 8 | perdida_entrenamiento: 0.2544 | exactitud_entrenamiento: 0.9085 |
perdida_evaluación: 0.2994 | exactitud_evaluación: 0.8944
Epoca: 9 | perdida_entrenamiento: 0.2476 | exactitud_entrenamiento: 0.9103 |
perdida_evaluación: 0.2922 | exactitud_evaluación: 0.8956
Epoca: 10 | perdida_entrenamiento: 0.2387 | exactitud_entrenamiento: 0.9125
| perdida_evaluación: 0.2784 | exactitud_evaluación: 0.9024
Epoca: 11 | perdida_entrenamiento: 0.2333 | exactitud_entrenamiento: 0.9146
| perdida_evaluación: 0.2873 | exactitud_evaluación: 0.8949
Epoca: 12 | perdida_entrenamiento: 0.2281 | exactitud_entrenamiento: 0.9178
| perdida_evaluación: 0.2765 | exactitud_evaluación: 0.9067
Epoca: 13 | perdida_entrenamiento: 0.2218 | exactitud_entrenamiento: 0.9192
| perdida_evaluación: 0.2770 | exactitud_evaluación: 0.9036
Epoca: 14 | perdida_entrenamiento: 0.2164 | exactitud_entrenamiento: 0.9213
| perdida_evaluación: 0.2657 | exactitud_evaluación: 0.9080
Epoca: 15 | perdida_entrenamiento: 0.2135 | exactitud_entrenamiento: 0.9229
| perdida_evaluación: 0.2760 | exactitud_evaluación: 0.9006
Epoca: 16 | perdida_entrenamiento: 0.2100 | exactitud_entrenamiento: 0.9244
| perdida_evaluación: 0.2728 | exactitud_evaluación: 0.9071
Epoca: 17 | perdida_entrenamiento: 0.2055 | exactitud_entrenamiento: 0.9253
| perdida_evaluación: 0.2619 | exactitud_evaluación: 0.9062
Epoca: 18 | perdida_entrenamiento: 0.2030 | exactitud_entrenamiento: 0.9263
| perdida_evaluación: 0.2667 | exactitud_evaluación: 0.9054
Epoca: 19 | perdida_entrenamiento: 0.1994 | exactitud_entrenamiento: 0.9276
| perdida_evaluación: 0.2621 | exactitud_evaluación: 0.9061
Epoca: 20 | perdida_entrenamiento: 0.1959 | exactitud_entrenamiento: 0.9293
| perdida_evaluación: 0.2586 | exactitud_evaluación: 0.9083
Tiempo total de entrenamiento: 518.270 segundos
```

4.4 Graficar las curvas de pérdidas del Modelo

Según los resultados del entrenamiento del `modelo`, pareciera tener una buena exactitud alrededor del 91%.

Pero podemos evaluarlo aún más trazando las **curvas de pérdida** del modelo.

Las curvas de pérdida muestran los resultados del modelo a lo largo del tiempo.

Y son una excelente manera de ver cómo se desempeña su modelo en diferentes conjuntos de datos (por ejemplo, entrenamiento y prueba).

Vamos a crear una función para trazar los valores en nuestro diccionario `modelo_resultados`.

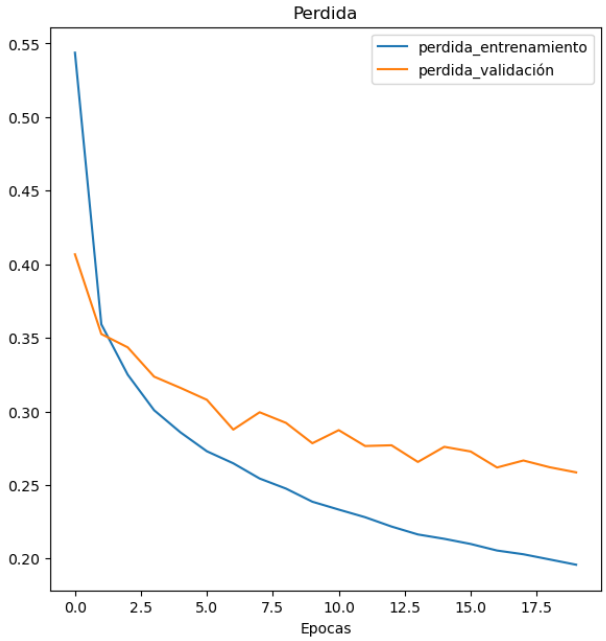
In [67]:
```
# Ver las claves del diccionario modelo_resultados
modelo_resultados.keys()
```

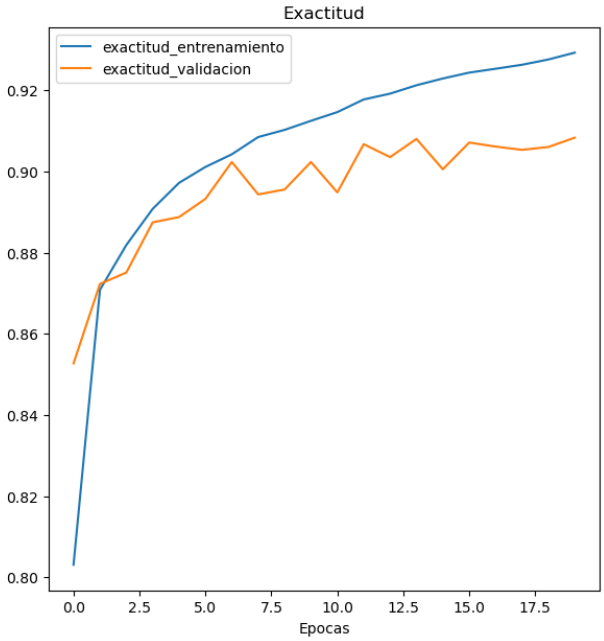
Out[67]:
```
dict_keys(['perdida_entrenamiento', 'exactitud_entrenamiento', 'perdida_eval
uacion', 'exactitud_evaluacion'])
```

Tendremos que extraer cada una de estas claves y convertirlas en un gráfico.

In [68]:
```
def graficar_curvas_perdida(resultados):
    """Graficar las curvas de entrenamiento del diccionario de resultados.

    Argumentoss:
        resultadoss (dict): diccionario conteniendo una lista de valores con
            {"perdida_entrenamiento": [...],
             "exactitud_entrenamiento": [...],
             "perdida_evaluacion": [...],
             "exactitud_evaluacion": [...]}
    """

    # Obtener los valores de la perdida del diccionario de resultados (entre
    perdida = resultados['perdida_entrenamiento']
    perdida_validacion = resultados['perdida_evaluacion']

    # Obtener los valores de la exactitud del diccionario de resultados (ent
    exactitud = resultados['exactitud_entrenamiento']
    exactitud_validacion = resultados['exactitud_evaluacion']

    # Determinar el número de epocas
    epocas = range(len(resultados['perdida_entrenamiento']))

    # Configurar la gráfica
    plt.figure(figsize=(15, 7))

    # Graficar la perdida
    plt.subplot(1, 2, 1)
    plt.plot(epocas, perdida, label='perdida_entrenamiento')
    plt.plot(epocas, perdida_validacion, label='perdida_validación')
    plt.title('Perdida')
    plt.xlabel('Epocas')
    plt.legend()

    # Graficar la exactitud
    plt.subplot(1, 2, 2)
    plt.plot(epocas, exactitud, label='exactitud_entrenamiento')
    plt.plot(epocas, exactitud_validacion, label='exactitud_validacion')
    plt.title('Exactitud')
    plt.xlabel('Epocas')
    plt.legend();
```

In [69]:
```
graficar_curvas_perdida(modelo_resultados)
```





5. Hacer y evaluar predicciones aleatorias

Para hacerlo, creemos una función `hacer_predicciones()` donde podemos pasar el modelo y algunos datos para que haga predicciones.

In [70]:
```
def hacer_predicciones(modelo: torch.nn.Module, data: list, dispositivo: tor
    probabilidad_predicciones = []
    modelo.eval()
    with torch.inference_mode():
        for ejemplo in data:
            # Preparar ejemplo
            ejemplo = torch.unsqueeze(ejemplo, dim=0).to(dispositivo) # Agre

            # propagar hacia adelante
            salida = modelo(ejemplo)

            # Obtener la probabilidda de la predicción (salida -> probabilic
            probabilidad_prediccion = torch.softmax(salida.squeeze(), dim=0)

            # Sacar probabilidad_prediccion del GPU para siguientes calculos
            probabilidad_predicciones.append(probabilidad_prediccion.cpu())

    # Apilar las probabilidad_predicciones para convertir la lista en un ten
    return torch.stack(probabilidad_predicciones)
```

In [71]:
```
import random
random.seed(42)
ejemplos_validacion = []
etiquetas_validacion = []
for ejemplo, etiqueta in random.sample(list(datos_validacion), k=9):
    ejemplos_validacion.append(ejemplo)
    etiquetas_validacion.append(etiqueta)

# Ver la forma del primer ejemplo y la etiqueta
print(f"Forma imagen ejemplo: {ejemplos_validacion[0].shape}\nEtiqueta image
```

```
Forma imagen ejemplo: torch.Size([1, 28, 28])
Etiqueta imagen ejemplo: 5 (Sandal)
```

Ahora podemos usar la función `hacer_predicciones()` para predecir sobre los `ejemplos_validacion`

In [72]:
```
# Make predictions on test samples with model 2
probabilidades_prediccion = hacer_predicciones(modelo=modelo,
                                               data=ejemplos_validacion)

# Ver las probabilidades de las dos primeras predicciones
probabilidades_prediccion[:2]
```

Out[72]:
```
tensor([[1.2353e-11, 2.7154e-19, 2.8920e-14, 1.5371e-13, 6.1322e-15, 1.0000e
+00,
         1.6095e-11, 4.5483e-09, 1.9945e-08, 2.9644e-06],
        [4.2404e-03, 9.8495e-01, 9.6983e-05, 8.1875e-03, 1.6630e-04, 2.1481e
-08,
         2.3564e-03, 9.8863e-11, 1.4311e-06, 5.6663e-09]])
```

¡Excelente!

Y ahora podemos pasar de las probabilidades de predicción a las etiquetas de predicción tomando `torch.argmax()` de la salida de la función de activación `torch.softmax()`.

In [73]:
```
# Convertir las probabilides de la predicción en las etiquetas usando argmax
clases_predicha = probabilidades_prediccion.argmax(dim=1)
clases_predicha
```

Out[73]:
```
tensor([5, 1, 7, 4, 3, 0, 4, 7, 1])
```

In [74]:
```
# Are our predictions in the same form as our test labels?
etiquetas_validacion, clases_predicha
```

Out[74]:
```
([5, 1, 7, 4, 3, 0, 4, 7, 1], tensor([5, 1, 7, 4, 3, 0, 4, 7, 1]))
```

Ahora nuestras clases pronosticadas están en el mismo formato que nuestras etiquetas de prueba, podemos comparar. Ya que estamos tratando con datos de imágenes, vamos a visualizar las predicciones.

In [75]:
```
# Graficar predicciones
plt.figure(figsize=(9, 9))
filas = 3
columnas = 3
for i, ejemplo in enumerate(ejemplos_validacion):
  # Crear un subplot
  plt.subplot(filas, columnas, i+1)

  # Graficar la imagen objetivo
  plt.imshow(ejemplo.squeeze(), cmap="gray")

  # Encontrar la etiqueta de la predicción (en forma de texto, p.e. "Sandal"
  etiqueta_predicha = nombre_clases[clases_predicha[i]]

  # Encontrar la etiqueta real (en forma de texto, p.e. "T-shirt")
  etiqueta_real = nombre_clases[etiquetas_validacion[i]]

  # Crear el titulo de la gráfica
  texto_titulo = f"Pred: {etiqueta_predicha} | Real: {etiqueta_real}"

  # Chequear si son iguales y cambiar color
  if etiqueta_predicha == etiqueta_real:
      plt.title(texto_titulo, fontsize=10, c="g") # texto verde si correcto
  else:
      plt.title(texto_titulo, fontsize=10, c="r") # texto rojo si incorrecto
  plt.axis(False);
```


Pred: Sandal | Real: Sandal
Pred: Trouser | Real: Trouser
Pred: Sneaker | Real: Sneaker
Pred: Coat | Real: Coat
Pred: Dress | Real: Dress
Pred: T-shirt/top | Real: T-shirt/top
Pred: Coat | Real: Coat
Pred: Sneaker | Real: Sneaker
Pred: Trouser | Real: Trouser

6. Elaboración de una matriz de confusión para una mejor evaluación de la predicción

Hay muchas diferentes métricas de evaluación que podemos usar para problemas de clasificación.

Uno de los más visuales es una matriz de confusión.

Una matriz de confusión le muestra dónde se confundió su modelo de clasificación entre predicciones y etiquetas verdaderas.

Para hacer una matriz de confusión, seguiremos tres pasos:

1. Hacer predicciones con el modelo entrenado, `modelo` (una matriz de confusión compara las predicciones con las etiquetas verdaderas).
2. Hacer una matriz de confusión usando [`torch.ConfusionMatrix` (https://torchmetrics.readthedocs.io/en/latest/references/
highlight=confusion#confusionmatrix).
3. Grafica la matriz de confusión usando [`mlxtend.plotting.plot_confusion_matrix()` (http://rasbt.github.io/mlxtend/user_guide/plotting/plot_confusion_matrix/).

Comencemos por hacer predicciones con nuestro modelo entrenado.

In [77]:
```
# 1. Hacer predicciones con el modelo entrenado
predicciones = []
modelo.eval()
with torch.inference_mode():
  for imagen, etiqueta in tqdm(cargador_validacion, desc="Haciendo prediccio
    # Enviar datos al dispositivo
    imagen, etiqueta = imagen.to(dispositivo), etiqueta.to(dispositivo)
    # Realizar la propagación hacia adelante
    salida = modelo(imagen)
    # Convertir la salida a predicciones salida -> probabilidades predicción
    prediccion = torch.softmax(salida.squeeze(), dim=0).argmax(dim=1)
    # Colocar las predicciones en el CPU para evaluación
    predicciones.append(prediccion.cpu())
# Concatenar lista de predicciones en un tensor
tensor_predicciones = torch.cat(predicciones)
```

```
Haciendo predicciones:   0%|          | 0/313 [00:00<?, ?it/s]
```

Ahora que tenemos predicciones, sigamos los pasos 2 y 3:

1. Crear una matriz de confusión usando `torchmetrics.ConfusionMatrix`.
2. Graficar la matriz de confusión usando `mlxtend.plotting.plot_confusion_matrix()`.

Primero necesitaremos asegurarnos de tener `torchmetrics` y `mlxtend` instalados (estas dos bibliotecas nos ayudarán a crear y visualizar una matriz de confusión).

> **Nota:** Si está utilizando Google Colab, la versión predeterminada de `mlxtend` instalada es 0.14.0 (a partir de marzo de 2022), sin embargo, para los parámetros de la función `plot_confusion_matrix()`, necesitamos 0.19.0 o superior.

In [78]:
```
# Ver si existe torchmetrics, si no, instalarla.
try:
    import torchmetrics, mlxtend
    print(f"versión mlxtend: {mlxtend.__version__}")
    assert int(mlxtend.__version__.split(".")[1]) >= 19, "mlxtend versión de
except:
    !pip install -q torchmetrics -U mlxtend # <- Nota: si está utilizando Gc
    import torchmetrics, mlxtend
    print(f"versión mlxtend: {mlxtend.__version__}")
```

```
versión mlxtend: 0.21.0
```

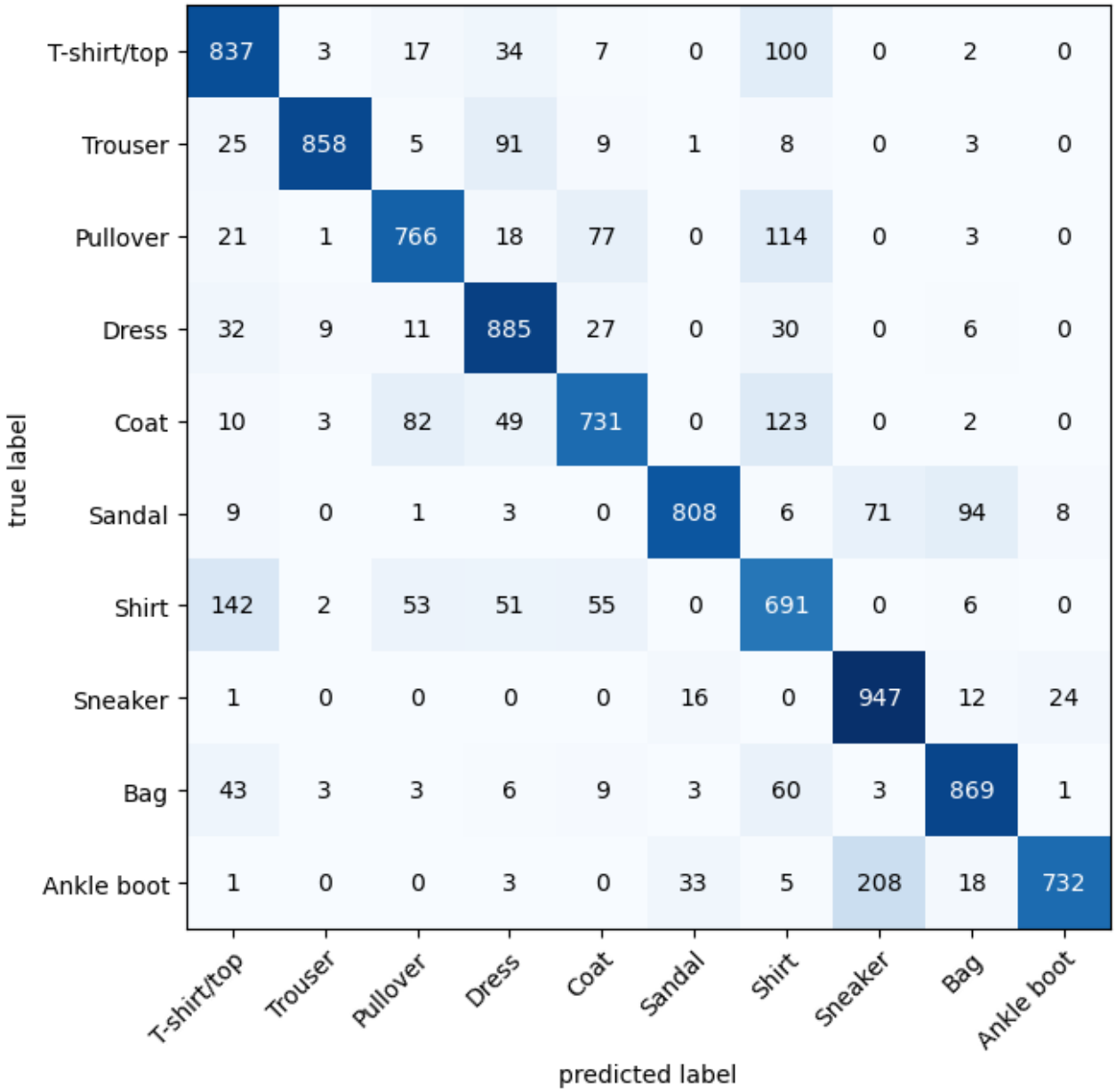
Con `torchmetrics` y `mlxtend` instalados, ¡hagamos una matriz de confusión!

Primero crearemos una instancia `torchmetrics.ConfusionMatrix` diciéndole con cuántas clases estamos tratando configurando `num_classes=len(nombre_clases)`.

Luego, crearemos una matriz de confusión (en formato de tensor) pasando a nuestra instancia las predicciones de nuestro modelo (`preds=tensor_predicciones`) y objetivos (`target=datos_validacion.targets`).

Finalmente, podemos graficar nuestra matriz de confusión usando la función `plot_confusion_matrix()` de `mlxtend.plotting`.

In [79]:
```
from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

# 2. Configurar una instancia de la matriz de confusión matrix y comparar pr
matriz_confusion = ConfusionMatrix(num_classes=len(nombre_clases))
tensor_matriz_confusion = matriz_confusion(preds=tensor_predicciones,
                                           target=datos_validacion.targets)

# 3. Graficar la matriz de the confusión
fig, ax = plot_confusion_matrix(
    conf_mat=tensor_matriz_confusion.numpy(), # matplotlib trabaja con NumPy
    class_names=nombre_clases, # convertir etiquetas de fila y columna en nc
    figsize=(10, 7)
);
```



Podemos ver que nuestro modelo funciona bastante bien ya que la mayoría de los cuadrados oscuros están en la diagonal desde la parte superior izquierda hasta la parte inferior derecha (y el modelo ideal tendrá solo valores en estos cuadrados y 0 en todos los demás). El modelo se "confunde" más en las clases que son similares, por ejemplo, prediciendo "Pullover" para imágenes que en realidad están etiquetadas como "Shirt". Y lo mismo para predecir "Shirt" para las clases que en realidad están etiquetadas como "T-shirt/top". Este tipo de información a menudo es más útil que una sola métrica de precisión porque indica dónde un modelo se está equivocando. También sugiere por qué el modelo puede estar equivocado en ciertas cosas. Es comprensible que el modelo a veces prediga "Shirt" para imágenes etiquetadas como "T-shirt/top". Podemos usar este tipo de información para inspeccionar más a fondo nuestros modelos y datos para ver cómo se podrían mejorar.

7. Guardar el modelo

Estamos contentos con las predicciones de nuestros modelos, así que guardémoslo en un archivo para que pueda usarse más tarde.

In [80]:
```
from pathlib import Path

# 1. Crear un directorio para los modelos
MODELOS_PATH = Path('../modelos')
MODELOS_PATH.mkdir(parents=True, exist_ok=True)

# 2. Crear path para el modelo
MODELO_NOMBRE = "23-RNC_modelo.pth"
MODELO_DIRECCION = MODELOS_PATH / MODELO_NOMBRE

# 3. Guardar el state_dict del modelo
print(f"Guardando modelo en: {MODELO_DIRECCION}")
torch.save(obj=modelo.state_dict(), # guardando state_dict() solo guarda los
           f=MODELO_DIRECCION)
```

```
Guardando modelo en: ../modelos/23-RNC_modelo.pth
```

7.1 Cargar modelo y continuar entrenamiento

In [84]:
```
# Instanciar una nueva instancia del modelo
modelo_cargado = TinyVGG(forma_entrada=1,
                         unidades_ocultas=10,
                         forma_salida=10)

# Cargar el state dict del modelo
modelo_cargado.load_state_dict(torch.load(MODELO_DIRECCION))

# Colocar el modelo en el dispositivo destino (si los datos estan en el GPU,
modelo_cargado.to(dispositivo)

print(f"Modelo cargado:\n{modelo}")
```

```
Modelo cargado:
TinyVGG(
  (bloque_1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
  )
  (bloque_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
  )
  (clasificador): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=490, out_features=10, bias=True)
  )
)
```

Formulas para calcular los tamaños de salida de las capas de convolución y *pooling* (agregación o reducción)

La formula que relaciona el tamaño de salida de la convolución con el tamaño de la entrada es:

$$ W = \lfloor \frac{W_{prev} - f + 2 \times relleno}{paso} \rfloor +1 $$$$f = \text{número de filtros usados en la convolución}$$

La formula que relaciona el tamaño de salida del *pooling* con el tamaño de la entrada es:

$$ W = \lfloor \frac{W_{prev} - f}{paso} \rfloor +1 $$$$f = \text{tamaño de la ventana de pooling}$$