Práctica II: Caso práctico de uso de RNCs

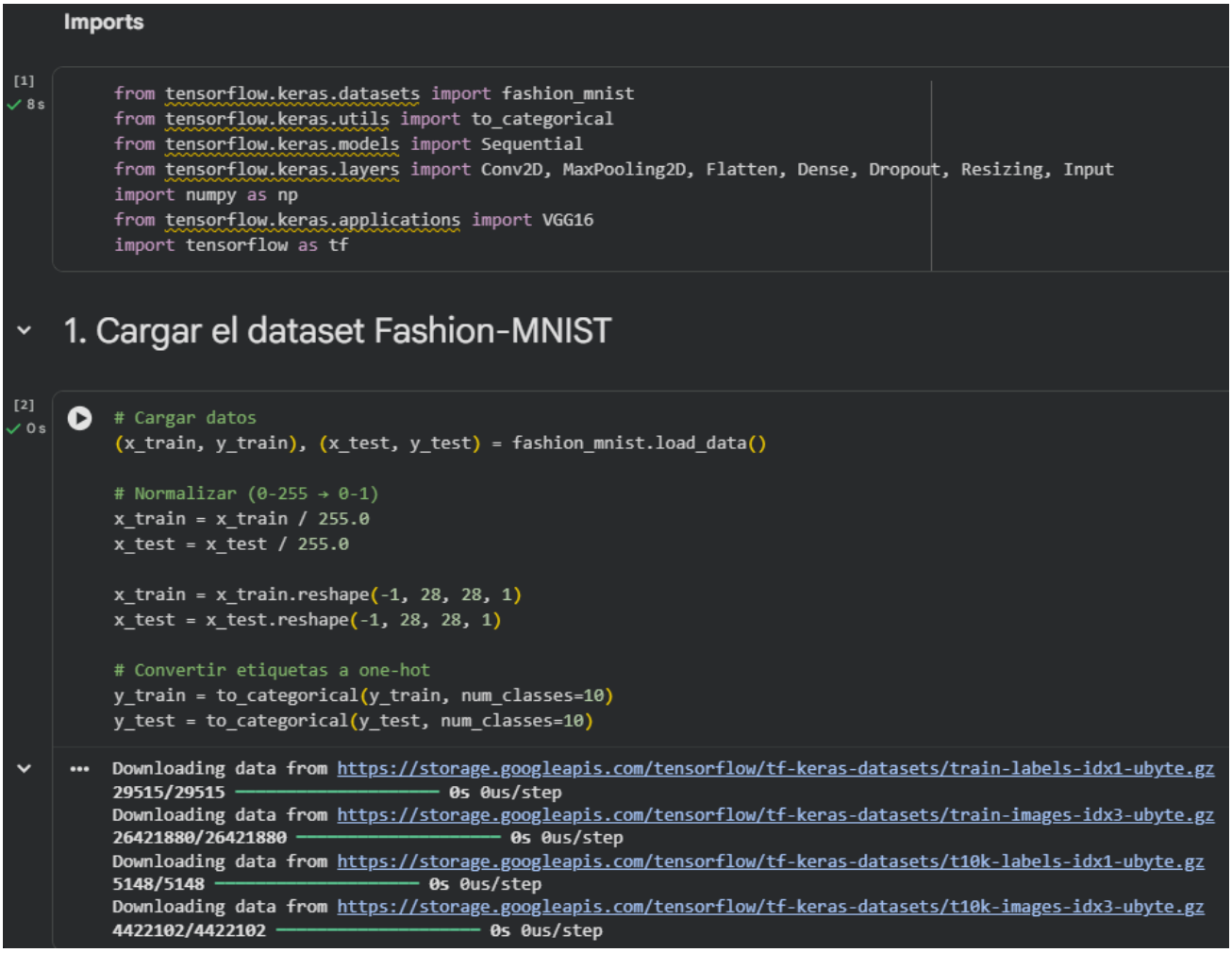
Opciones utilizadas

Entorno	Google Colab
Dataset	Fashion-MNIST precargado de Keras
Activación	ReLU
Optimizador	Adam (LR = 0,001)
Número de configuraciones	4

Tabla de configuraciones:

Modelo	Arquitectura / Capas	Filtros / Neuronas	Epochs	Batch size
1 (CNN Simple)	2 Bloques Conv + Salida	32, 64 (Filtros)	10	64
2 (CNN Profunda)	3 Bloques Conv + 1 Oculta	32-32, 64-64, 128 (Filtros) + 256 (Neuronas)	20	64
3 (VGG16 Scratch)	VGG16 Base + 1 Oculta	[64...512] (VGG) + 256 (Neuronas)	30	32
4 (VGG16 Transfer)	VGG16 (Congelada) + 1 Oculta	[64...512] (VGG) + 256 (Neuronas)	15	32

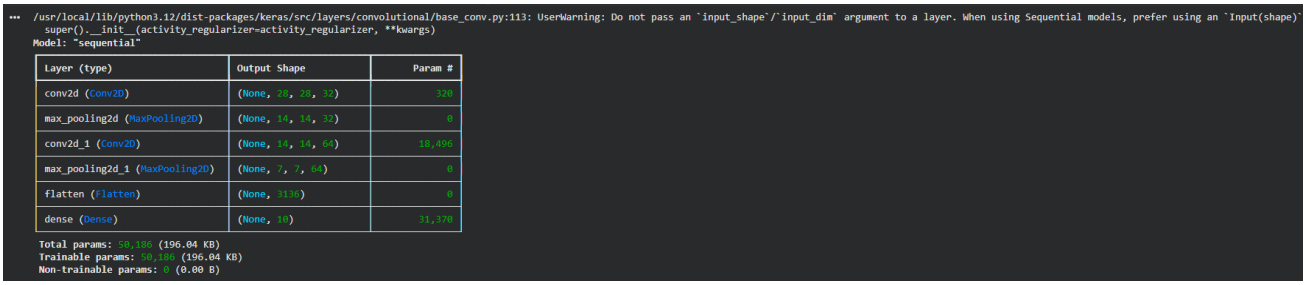
Proceso de desarrollo



En esta fase inicial del desarrollo, se prepara el entorno de ejecución en Google Colab. Como se aprecia en la captura, el proceso consta de dos bloques fundamentales:

1. **Importación de librerías**: Se cargan los módulos de TensorFlow y Keras. También se importa Sequential para la construcción de los modelos y las capas específicas que se utilizarán tanto en las CNN propias (Conv2D, MaxPooling2D) como en la adaptación de la red preentrenada (Resizing, Input, VGG16).
2. **Carga y Preprocesamiento de Datos**:
   - **Carga**: Se utiliza *fashion_mnist.load_data()* para descargar el conjunto de datos. En la salida de la celda se confirma que los archivos de entrenamiento y test se han descargado correctamente.
   - **Normalización**: La división de las imágenes entre *255.0* escala los valores de los píxeles al rango *[0, 1]*. Esto sirve para acelerar la convergencia del entrenamiento.

- **Redimensionamiento (*Reshape*)**: Se transforman las imágenes a la forma (-1, 28, 28, 1). Esto es vital ya que las capas Conv2D de Keras esperan una entrada de 4 dimensiones (batch, alto, ancho, canales), y el dataset original viene en escala de grises.
- **Codificación de etiquetas (*One-hot*)**: Finalmente, se aplica *to_categorical* para transformar las etiquetas numéricas en vectores binarios (one-hot encoding).



Aquí se realiza la construcción del primer modelo experimental (CNN ligera). Como se observa en el código y la tabla de resumen, se ha definido una **arquitectura *Sequential*** compuesto por dos etapas claramente diferenciadas:

- **Extracción de Características**: Consta de dos bloques convolucionales. El primero tiene 32 filtros y el segundo aumenta la profundidad a 64 filtros. La capa de MaxPooling2D (2x2) es la encargada de reducir la dimensionalidad a la mitad en cada paso (de 28x28 a 14x14, y finalmente a 7x7).

- **Etapa de Clasificación**: Tras aplanar los mapas de características con *Flatten*, se conecta directamente a una capa densa de salida con 10 neuronas (10 clases) y función de activación *softmax*, adecuada para la clasificación multiclase.

Entrenamiento

```
[13]
35
s
```

```
model1.compile(optimizer='adam',
               loss='categorical_crossentropy',
               metrics=['accuracy'])

model1.fit(x_train, y_train, epochs=10, batch_size=64)
```

```
Epoch 1/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7613 - loss: 0.6726
Epoch 2/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8846 - loss: 0.3286
Epoch 3/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9002 - loss: 0.2830
Epoch 4/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9101 - loss: 0.2532
Epoch 5/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9186 - loss: 0.2282
Epoch 6/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9234 - loss: 0.2146
Epoch 7/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9288 - loss: 0.1999
Epoch 8/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9360 - loss: 0.1797
Epoch 9/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9387 - loss: 0.1690
Epoch 10/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9431 - loss: 0.1562
<keras.src.callbacks.history.History at 0x7a6a1d3cd400>
```

En esta fase se ejecuta el proceso de aprendizaje supervisado del primer modelo. En la captura se puede ver:

1. **Compilación del Modelo**: Se define la estrategia de aprendizaje mediante el método *.compile()*.
2. **Ejecución del Entrenamiento (*.fit*)**: Se lanza el entrenamiento con los datos normalizados (*x_train*) durante 10 épocas y un tamaño de lote (*batch size*) de **64**.
   - **Análisis de Convergencia**: La traza de ejecución muestra una evolución muy positiva y estable. A medida que avanzan las iteraciones, la pérdida disminuye consistentemente y el *accuracy* aumenta.
   - **Eficiencia**: Se observa que cada época tarda aproximadamente 3-6 segundos, completando los 938 pasos (resultado de dividir 60.000 imágenes entre lotes de 64) de

forma rápida, lo que confirma que la arquitectura ligera diseñada es computacionalmente eficiente.

Modelo 2: CNN más profunda

[5]
✓ 0s

```
model2 = Sequential()

# Bloque 1
model2.add(Conv2D(32, (3,3), activation='relu', padding='same', input_shape=(28,28,1)))
model2.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model2.add(MaxPooling2D((2,2)))
model2.add(Dropout(0.25))

# Bloque 2
model2.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model2.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model2.add(MaxPooling2D((2,2)))
model2.add(Dropout(0.25))

# Bloque 3
model2.add(Conv2D(128, (3,3), activation='relu', padding='same'))
model2.add(MaxPooling2D((2,2)))
model2.add(Dropout(0.25))

# Parte densa
model2.add(Flatten())
model2.add(Dense(256, activation='relu'))
model2.add(Dropout(0.5))
model2.add(Dense(10, activation='softmax'))

model2.summary()
```

Model: "sequential_1"

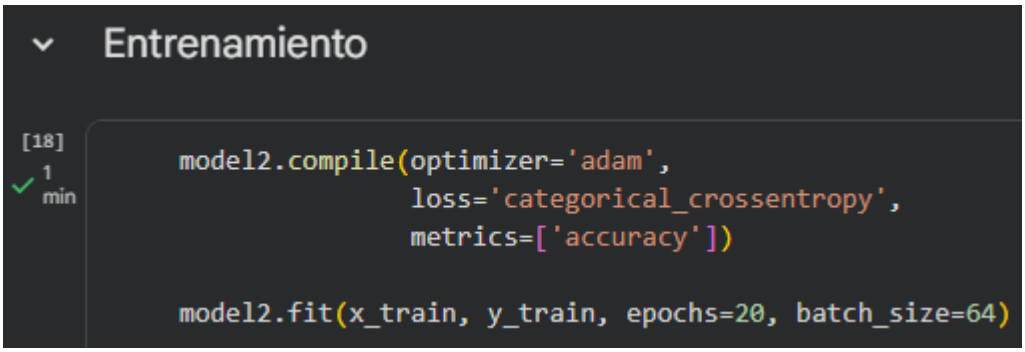
Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 28, 28, 32)	320
conv2d_3 (Conv2D)	(None, 28, 28, 32)	9,248
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 32)	0
dropout (Dropout)	(None, 14, 14, 32)	0
conv2d_4 (Conv2D)	(None, 14, 14, 64)	18,496
conv2d_5 (Conv2D)	(None, 14, 14, 64)	36,928
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_1 (Dropout)	(None, 7, 7, 64)	0
conv2d_6 (Conv2D)	(None, 7, 7, 128)	73,856
max_pooling2d_4 (MaxPooling2D)	(None, 3, 3, 128)	0
dropout_2 (Dropout)	(None, 3, 3, 128)	0
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 256)	295,168
dropout_3 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 10)	2,570

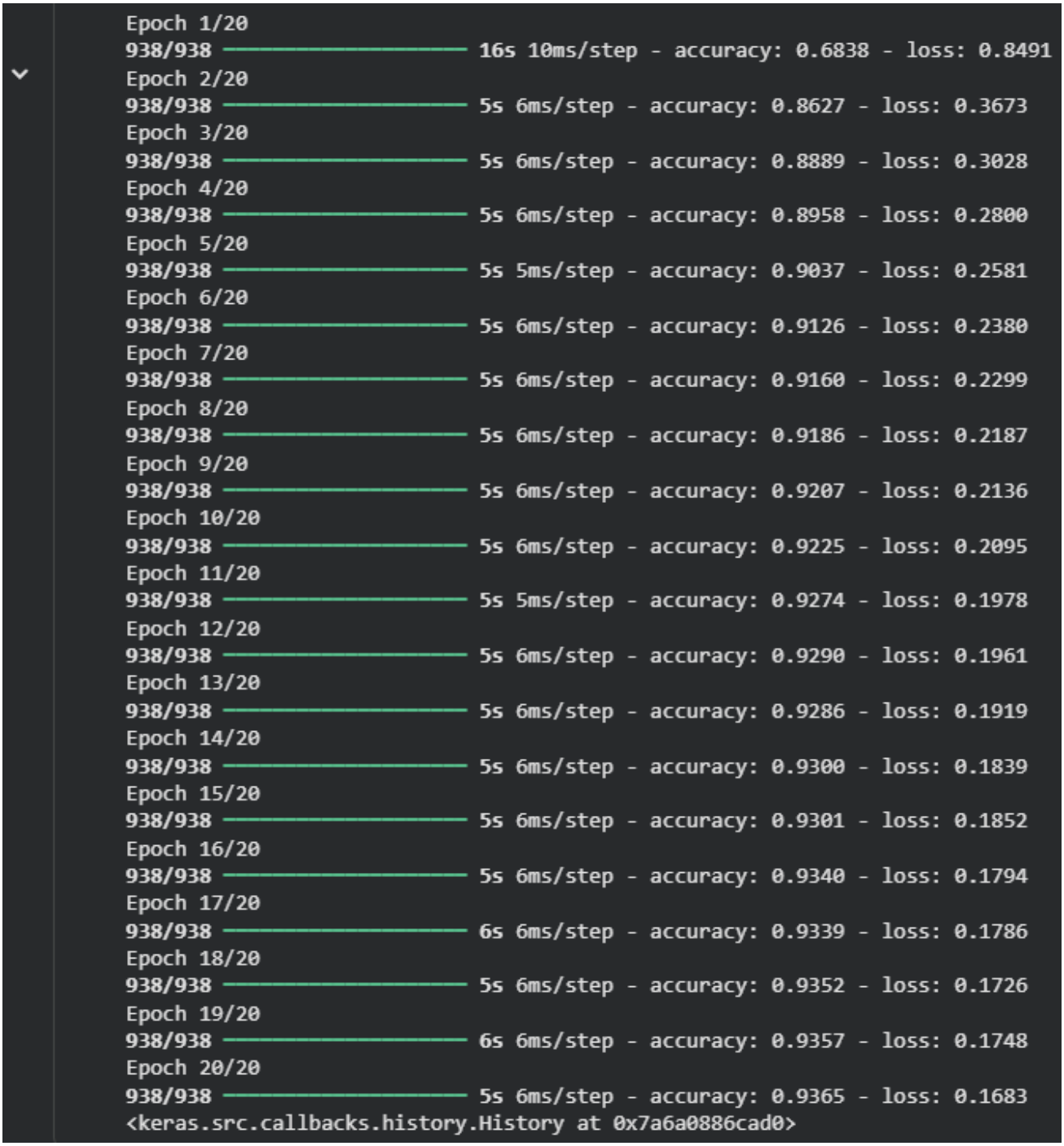
Total params: 436,586 (1.67 MB)
Trainable params: 436,586 (1.67 MB)
Non-trainable params: 0 (0.00 B)

En esta etapa, se evoluciona hacia una arquitectura más profunda y compleja que la anterior. En el código se pueden apreciar tres cambios estructurales fundamentales respecto al modelo anterior:

1. **Profundidad y Ensanchamiento Progresivo**: El modelo se estructura en **tres bloques de extracción de características**, siguiendo un patrón piramidal:
   - **Bloque 1**: Dos capas consecutivas *Conv2D* de 32 filtros.
   - **Bloque 2**: Dos capas consecutivas *Conv2D* de 64 filtros.
   - **Bloque 3**: Una capa *Conv2D* de 128 filtros.
2. **Estrategia de Regularización (Dropout)**: Debido al aumento de parámetros, se introduce el riesgo de sobreajuste. Para reducirlo, se añade la capa Dropout en cada bloque.
3. **Clasificador Denso Extendido**: A diferencia del modelo 1, tras el aplanado (*Flatten*), los datos pasan por una capa oculta densa de 256 neuronas antes de llegar a la salida. Esto da

al modelo una mayor capacidad de cómputo para relacionar las características extraídas antes de la predicción final.





La configuración se ajustó a **20 épocas** con un tamaño de lote de 64.

A diferencia del modelo 1 (que inició con >75% de *acuraccy*), el modelo 2 comienza la primera época con un *accuracy* menor (**68,38%**) y una pérdida más alta. Este inicio más lento se debe a la introducción de las capas **Dropout**. Al desactivar aleatoriamente neuronas durante el entrenamiento, se impide que la red memorice rápidamente los datos (sobreajuste), forzándola a aprender de forma progresiva.

9

A pesar de la mayor profundidad de la red, el tiempo por época se mantiene muy eficiente. El tiempo total de entrenamiento fue **<2 min**.

El modelo alcanza un *accuracy* en el conjunto de entrenamiento del **93,65%** con una pérdida de 0,1683. Este valor es ligeramente inferior al del Modelo 1 (94,3%) en entrenamiento. Esto no tiene por qué ser un comportamiento negativo, significa que el Dropout está funcionando correctamente, limitando el sobreajuste en los datos de entrenamiento con la expectativa de obtener un mejor rendimiento en los datos de test.

Modelo 3: VGG16 sin entrenamiento previo (desde cero)

[7] ✓ 0s

```
# Instanciamos la VGG16 base
# Le decimos que espere 96x96x3, porque nuestras capas previas se lo darán así.
vgg_base_scratch = VGG16(
    weights=None,          # sin pesos
    include_top=False,
    input_shape=(96, 96, 3)
)

model3 = Sequential()

# --- BLOQUE DE ADAPTACIÓN (para ahorrar RAM) ---
# 1. Entrada real de tus datos (pequeña)
model3.add(Input(shape=(28, 28, 1)))

# 2. El modelo se encarga de agrandar la imagen a 96x96
model3.add(Resizing(96, 96))

# 3. Convertir de 1 canal (Grayscale) a 3 canales (RGB)
# Usamos una convolución de 1x1 con 3 filtros. Es más eficiente que repetir y el modelo "aprende" la mejor forma de
# colorear la imagen.
model3.add(Conv2D(3, (1, 1), padding='same'))

model3.add(vgg_base_scratch)
model3.add(Flatten())
model3.add(Dense(256, activation='relu'))
model3.add(Dropout(0.5))
model3.add(Dense(10, activation='softmax'))

model3.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
resizing (Resizing)	(None, 96, 96, 1)	0
conv2d_7 (Conv2D)	(None, 96, 96, 3)	6
vgg16 (Functional)	(None, 3, 3, 512)	14,714,688
flatten_2 (Flatten)	(None, 4608)	0
dense_3 (Dense)	(None, 256)	1,179,904
dropout_4 (Dropout)	(None, 256)	0
dense_4 (Dense)	(None, 10)	2,570

Total params: 15,897,168 (60.64 MB)
Trainable params: 15,897,168 (60.64 MB)
Non-trainable params: 0 (0.00 B)

El tercer modelo está basado en la arquitectura estándar VGG16.

Dado que las imágenes de Fashion-MNIST son de escala de grises (28x28x1) y la arquitectura VGG16 fue diseñada para imágenes RGB de mayor resolución, se ha implementado un bloque inicial de adaptación:

- **Input & Resizing**: Se define la entrada original y se utiliza una capa *Resizing* para escalar las imágenes a **96x96** píxeles durante el entrenamiento. Esto optimiza el uso de la memoria RAM al evitar almacenar el dataset duplicado en alta resolución.
- **Expansión de Canales (Conv2D 1x1)**: Se aplica una convolución de 1x1 con 3 filtros. Esta técnica permite proyectar el canal único de entrada en los **3 canales** que requiere VGG16, permitiendo que la red aprenda la mejor combinación de características para 'colorear' la entrada artificialmente.

Para crear el modelo base desde cero, se instancia el bloque *VGG16* con el parámetro *weights=None* (no se utilizan pesos preentrenados). Además, se descarta la parte superior (*include_top=False*) para añadir un clasificador personalizado adaptado a las 10 clases del problema.

La tabla resumen evidencia un salto drástico en la complejidad computacional respecto a los modelos anteriores. El modelo cuenta con un total de **15.897.168 parámetros**, de los cuales el bloque *vgg16* aporta la gran mayoría (14.7M). Al entrenar desde cero, los **~15,9 millones de parámetros son entrenables**, lo que implica un coste computacional y una exigencia de memoria GPU significativamente mayores.

Entrenamiento

[8] ✓ 1h

```
model3.compile(optimizer='adam',
               loss='categorical_crossentropy',
               metrics=['accuracy'])

model3.fit(x_train, y_train, epochs=30, batch_size=32)
```

```
Epoch 1/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 237s 118ms/step - accuracy: 0.0974 - loss: 2.3033
Epoch 2/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 118ms/step - accuracy: 0.0986 - loss: 2.3028
Epoch 3/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 118ms/step - accuracy: 0.1003 - loss: 2.3028
Epoch 4/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1029 - loss: 2.3027
Epoch 5/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 224s 119ms/step - accuracy: 0.0993 - loss: 2.3027
Epoch 6/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0995 - loss: 2.3027
Epoch 7/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 119ms/step - accuracy: 0.1018 - loss: 2.3027
Epoch 8/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 119ms/step - accuracy: 0.1007 - loss: 2.3028
Epoch 9/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1001 - loss: 2.3027
Epoch 10/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0978 - loss: 2.3028
Epoch 11/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0960 - loss: 2.3028
Epoch 12/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 259s 117ms/step - accuracy: 0.0971 - loss: 2.3028
Epoch 13/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 118ms/step - accuracy: 0.1014 - loss: 2.3027
Epoch 14/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 119ms/step - accuracy: 0.0984 - loss: 2.3028
```

```
Epoch 15/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 119ms/step - accuracy: 0.0994 - loss: 2.3027
Epoch 16/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 118ms/step - accuracy: 0.0983 - loss: 2.3028
Epoch 17/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 118ms/step - accuracy: 0.0989 - loss: 2.3027
Epoch 18/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 222s 119ms/step - accuracy: 0.0975 - loss: 2.3028
Epoch 19/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0975 - loss: 2.3028
Epoch 20/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1013 - loss: 2.3027
Epoch 21/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0982 - loss: 2.3027
Epoch 22/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0981 - loss: 2.3028
Epoch 23/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1007 - loss: 2.3027
Epoch 24/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1007 - loss: 2.3027
Epoch 25/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0975 - loss: 2.3028
Epoch 26/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0991 - loss: 2.3027
Epoch 27/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0981 - loss: 2.3028
Epoch 28/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1004 - loss: 2.3027
Epoch 29/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.1021 - loss: 2.3027
Epoch 30/30
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 223s 119ms/step - accuracy: 0.0933 - loss: 2.3029
<keras.src.callbacks.history.History at 0x7a6a802c6030>
```

En el proceso de entrenamiento de la red VGG16 inicializada con pesos aleatorios, el impacto de procesar imágenes reescaladas a 96x96x3 a través de 16 millones de parámetros es evidente. El tiempo por época se dispara a un promedio de **223 segundos** (casi 4 minutos). Esto supone un tiempo total de entrenamiento cercano a las **2 horas** para completar las 30 épocas, demostrando lo que implica entrenar redes profundas masivas sin optimizaciones.

A nivel de aprendizaje, se observa un fenómeno de estancamiento total (subajuste). El **accuracy** oscila alrededor del **10%**. La función de pérdida se mantiene fija en **2,3**.

Debido a la inicialización aleatoria y la profundidad de la red, el modelo no ha logrado encontrar un gradiente descendente efectivo para minimizar el error, siendo incapaz de ajustar sus 16 millones de parámetros con el dataset actual. Este resultado negativo es crucial porque confirma la dificultad de entrenar redes profundas desde cero y **justifica la necesidad del siguiente paso**: utilizar Transfer Learning (modelo 4) para aprovechar pesos ya calibrados y solucionar este problema.

Modelo 4: VGG16 con entrenamiento previo (ImageNet)

[9] ✓ 0s

```python
# Cargar VGG16 preentrenado
vgg_base_pretrained = VGG16(
    weights='imagenet',     # preentrenado
    include_top=False,
    input_shape=(96, 96, 3)
)

# Congelar todas las capas del modelo base
vgg_base_pretrained.trainable = False

# Construir modelo final
model4 = Sequential()

# --- BLOQUE DE ADAPTACIÓN (Nuevo) ---
model4.add(Input(shape=(28, 28, 1)))       # Entrada: imágenes pequeñas (28x28)
model4.add(Resizing(96, 96))               # El modelo las agranda a 96x96
model4.add(Conv2D(3, (1, 1), padding='same')) # Convierte 1 canal a 3 canales (RGB)

# --- BLOQUE VGG Y CLASIFICACIÓN ---
model4.add(vgg_base_pretrained)
model4.add(Flatten())
model4.add(Dense(256, activation='relu'))
model4.add(Dropout(0.5))
model4.add(Dense(10, activation='softmax'))

model4.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
resizing_1 (Resizing)	(None, 96, 96, 1)	0
conv2d_8 (Conv2D)	(None, 96, 96, 3)	6
vgg16 (Functional)	(None, 3, 3, 512)	14,714,688
flatten_3 (Flatten)	(None, 4608)	0
dense_5 (Dense)	(None, 256)	1,179,904
dropout_5 (Dropout)	(None, 256)	0
dense_6 (Dense)	(None, 10)	2,570

```
Total params: 15,897,168 (60.64 MB)
Trainable params: 1,182,480 (4.51 MB)
Non-trainable params: 14,714,688 (56.13 MB)
```

Como solución a los problemas de convergencia observados en el modelo anterior, se implementa el modelo 4 utilizando la técnica de **Transfer Learning**. Las capturas muestran los pasos críticos que diferencian esta estrategia del entrenamiento desde cero:

La diferencia con el entrenamiento desde cero es que en la instanciación de la base VGG16 se utiliza el parámetro *weights='imagenet'*. Como se observa en la celda de salida, el sistema descarga

automáticamente el archivo de pesos (*vgg16_weights_tf...notop.h5*). Esto significa que la red ya no inicia con valores aleatorios, sino con filtros que ya saben detectar bordes, texturas y formas complejas aprendidas de millones de imágenes del dataset **ImageNet**.

Además, se aplica la instrucción *vgg_base_pretrained.trainable = False*, con la que se 'congelan' los pesos de la base VGG16 para que no se modifiquen durante el entrenamiento. El objetivo es utilizar la red como un extractor de características fijo y entrenar únicamente el clasificador final para adaptarlo a las clases de Fashion-MNIST.

La tabla resumen evidencia el impacto drástico de la congelación. Aunque el número de **parámetros** se mantiene en ~15,9 millones, la gran mayoría quedan estáticos (**no entrenables**) y el número de **parámetros entrenables** se reduce radicalmente a **1.182.480**. Esto reduce la carga computacional y facilita enormemente la convergencia.

Entrenamiento

[10] 35 min

```
model4.compile(optimizer='adam',
               loss='categorical_crossentropy',
               metrics=['accuracy'])

model4.fit(x_train, y_train, epochs=15, batch_size=32)
```

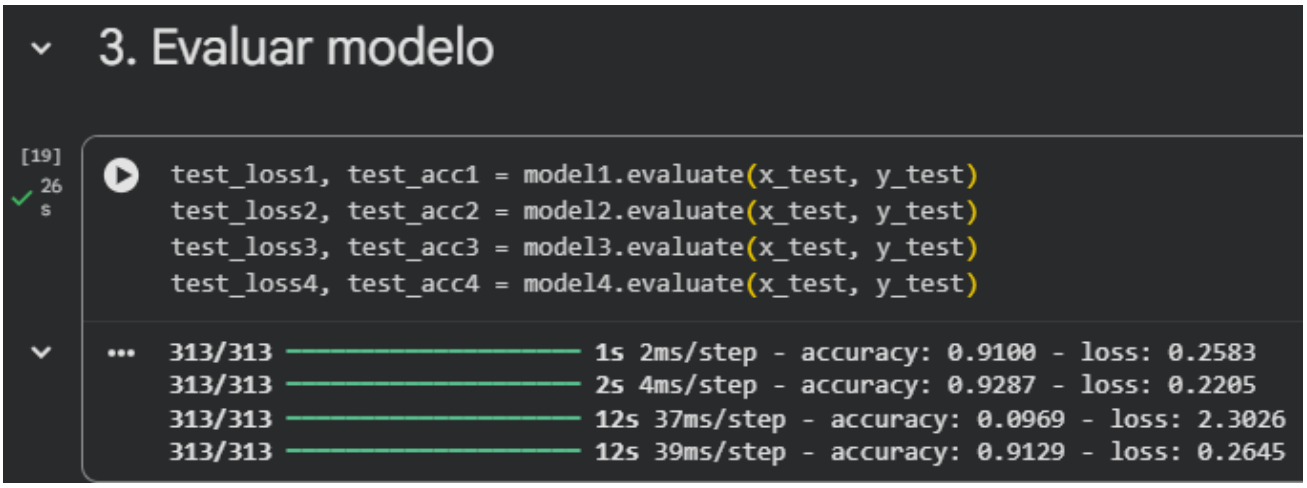
```
Epoch 1/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 142s 73ms/step - accuracy: 0.7687 - loss: 0.6551
Epoch 2/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.8680 - loss: 0.3601
Epoch 3/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.8810 - loss: 0.3218
Epoch 4/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.8842 - loss: 0.3138
Epoch 5/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.8917 - loss: 0.2895
Epoch 6/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.8974 - loss: 0.2772
Epoch 7/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9034 - loss: 0.2603
Epoch 8/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9037 - loss: 0.2613
Epoch 9/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9071 - loss: 0.2507
Epoch 10/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9103 - loss: 0.2408
Epoch 11/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9112 - loss: 0.2375
Epoch 12/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9137 - loss: 0.2272
Epoch 13/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9160 - loss: 0.2244
Epoch 14/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9172 - loss: 0.2216
Epoch 15/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 140s 75ms/step - accuracy: 0.9198 - loss: 0.2150
<keras.src.callbacks.history.History at 0x7a6a703b9eb0>
```

En el entrenamiento del modelo VGG16 preentrenado con pesos de ImageNet la ejecución se limitó a **15 épocas** debido a la rápida convergencia esperada.

El contraste con el modelo anterior es drástico. El modelo alcanza una exactitud de entrenamiento del 91,98% y una pérdida de 0,2150. Esto confirma que, aunque la red nunca había visto ropa de Fashion-MNIST, sus filtros internos (entrenados con **ImageNet**) ya eran capaces de extraer características visuales útiles desde el principio.

También se observa una reducción significativa en el tiempo por época, bajando de los ~223 segundos del modelo 3 a unos **140 segundos** estables en el modelo 4. Esto se debe a que el

algoritmo no tiene que actualizar los pesos de las capas profundas, solo trabaja sobre el clasificador final.



Finalmente, se someten los cuatro modelos entrenados a la fase de evaluación utilizando el conjunto de prueba (*x_test*), compuesto por 10.000 imágenes que la red nunca ha 'visto' durante el entrenamiento.

Tablas de resultados

La siguiente tabla resume los resultados obtenidos por cada una de las configuraciones evaluadas en esta práctica. Esto permite comparar fácilmente el efecto de las diferentes arquitectura sobre el rendimiento del modelo.

Modelo	Loss	Accuracy
1 (CNN Simple)	0,2583	0,91
2 (CNN Profunda)	0,2205	0,9287
3 (VGG16 Scratch)	2,3026	0,0969
4 (VGG16 Transfer)	0,2645	0,9129

Tipo de arquitectura	Modelo	Accuracy	Loss
Redes densas (Práctica I)	Modelo 1 (256-128)	0,8898	0,3135
	Modelo 2 (128-64)	0,8850	0,3328
	Modelo 3 (512-256-128)	0,8934	0,3580
CNNs (Práctica II)	Modelo 1 (CNN Simple)	0,91	0,2583
	Modelo 2 (CNN Profunda)	0,9287	0,2205
	Modelo 3 (VGG16 Scratch)	0,0969	2,3026
	Modelo 4 (VGG16 Transfer)	0,9129	0,2645

Análisis de las tablas de resultados

Al observar la primera tabla, se destacan tres conclusiones clave en el comportamiento de los modelos convolucionales:

1. **El ganador (Modelo 2 - CNN Profunda)**: El modelo alcanza una exactitud del **92,87%**, la más alta de todos los experimentos. Esto confirma que una arquitectura CNN personalizada, con profundidad moderada y regularización (Dropout), es la estrategia más efectiva para un dataset de complejidad media como Fashion-MNIST.
2. **La eficacia del transfer learning (modelo 4 vs modelo 3)**: Se ratifica el éxito del modelo 4 con un **91,29%** de acierto frente al desastroso **9,69%** del modelo 3. Pero, aunque el modelo 4 (VGG) es inmensamente más complejo, su rendimiento es muy similar al del Modelo 1 (91%). Esto sugiere que, para problemas sencillos de 28x28, una red

gigante preentrenada no ofrece una ventaja suficiente frente a una CNN ligera bien diseñada.

3. **Análisis del tiempo de inferencia**: Los **modelos 1 y 2** procesan todas las imágenes de test en **1 y 2 segundos** respectivamente. Son extremadamente rápidos y aptos para entornos de tiempo real. En contraste, los **modelos 3 y 4** tardan **12 segundos** en realizar la misma tarea. Con esto se puede ver que, aunque congelemos los pesos en el entrenamiento (modelo 4), durante la evaluación, la imagen debe atravesar las 16 capas profundas de la VGG para obtener una predicción.

En la segunda tabla podemos ver claramente la mejora que supone cambiar las redes densas de la primera práctica por las redes convolucionales de esta segunda.

Lo primero que se puede observar es la **ruptura de la barrera del 90%**. Ninguno de los modelos de redes densas logró superar el 90% de *accuracy*. En cambio, casi todas las CNNs (modelos 1, 2 y 4) superaron este umbral (91% - 92.8%).

También se da una **mejora** significativa en los valores de **la función de pérdida (*loss*)**. Las redes densas se estancaron en un error mínimo de **0,3135**. Sin embargo, las CNNs lograron reducirlo hasta **0,2205**. Esto indica que las redes convolucionales son capaces de clasificar mejor que las densas.

Esta superioridad se debe a la naturaleza de los datos. Las redes densas "aplanan" la imagen, destruyendo la estructura espacial (la relación entre un píxel y sus vecinos). Las CNNs, mediante el uso de filtros, preservan y explotan estas jerarquías espaciales (bordes, formas, texturas), lo que las convierte en la arquitectura óptima para problemas de visión artificial.

Concluimos que el **modelo 2 (CNN profunda personalizada)** es la solución óptima para el dataset Fashion-MNIST. Ofrece el mejor rendimiento (92.87%) y es computacionalmente más eficiente que adaptar grandes modelos preentrenados como VGG16.