INSTITUTO TECNOLÓGICO DE COSTA RICA

ÁREA ACADÉMICA DE INGENIERÍA MECATRÓNICA

MT 8008
Inteligencia Artificial

Tarea 1 – Parte 2
Sistemas Conexionistas

Jose Fabio Navarro Naranjo – 2019049626

César Argüello Salas – 2019047699

Profesor: Juan Luis Crespo Mariño

Semestre II - 2022

Análisis del problema

Se brinda un conjunto de imágenes correspondientes a piezas manufacturadas por fundición, donde algunas de ellas están en buen estado y otras presentan defectos inherentes al proceso de manufactura. El objetivo de esta tarea es el de desarrollar un modelo de red neuronal convolucional que permita clasificar las imágenes en dos grupos, de acuerdo si la pieza cumple o si presenta algún defecto. Finalmente, se pretende realizar distintas pruebas al modelo para evaluar su desempeño como herramienta de clasificación.

El conjunto de imágenes empleado está disponible en [1]. Se trata de 7348 imágenes de 300x300 pixeles en escala de grises, repartidas en dos carpetas nombradas "train" y "test", correspondientes a los subconjuntos de entrenamiento y prueba, respectivamente. Dentro de ambos subconjuntos se encuentran imágenes de piezas en buen y en mal estado.

La naturaleza propia del problema implica el análisis de imágenes, las cuales se representan computacionalmente mediante matrices de intensidades de pixeles. La arquitectura de red neuronal más apta para esta clase de análisis corresponde a una red neuronal convolucional, ya que esta posee las capacidades de procesamiento necesarias para detectar "*features*" o las cualidades espaciales presentes en una imagen. Esto lo logra a través de una serie de capas que extraen sucesivamente las características de interés en una imagen y la reducen a una forma que es más sencilla de procesar por un computador [2].

A continuación, se presentan los conceptos teóricos claves para el desarrollo de esta tarea.

i. *Fundamento teórico*

En la figura 1 se muestra un ejemplo de la estructura básica de una red neuronal convolucional. Esta se compone de los siguientes tipos de capas:

- Capas convolucionales: su función es la de extraer las características visuales presentes en una imagen por medio de una operación de convolución. Emplean matrices especiales conocidas como filtros o *kernels*, que se desplazan sobre cada porción de la imagen ejecutando dicha operación, creando un mapa de activación donde se resalta una característica particular de la imagen original. Esta capa generalmente incluye la función de activación ReLU para filtrar el ruido resultante en el mapa de activación [3].
- Capas de agrupación: también conocidas como capas de *pooling*. Se agregan después de cada capa convolucional para condensar las características más importantes de la imagen en mapas de menor tamaño [3]. Es decir, su utilidad es la de reducir las dimensiones de los mapas de activación para disminuir el poder computacional requerido en el procesamiento posterior [2].
- Capas densas (completamente conectadas): este tipo de capas se colocan al final de la red para clasificar las imágenes según las características extraídas por las capas de convolución

y producir las salidas de la red. Previo a estas capas se debe realizar una operación de *flattening* para transformar los tensores de la imagen en un vector de entrada para la red completamente conectada [2].

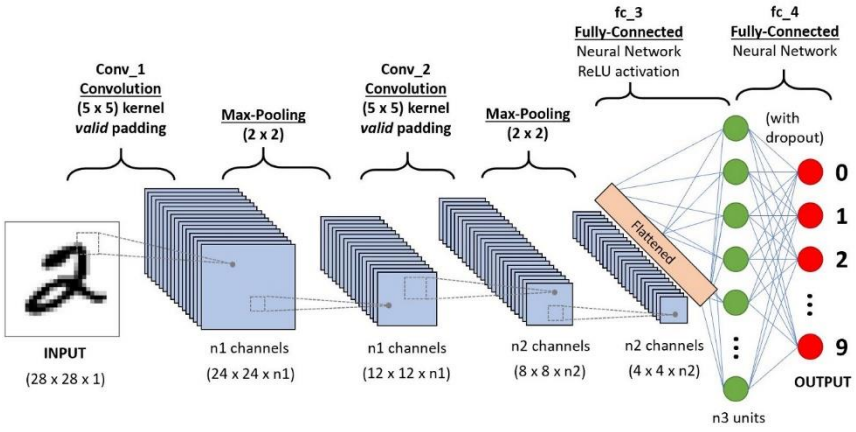

Figura 1. Diagrama de la estructura de una red neuronal convolucional [2].

Descripción de la solución

Para desarrollar la presente tarea se implementó el código de Python adjunto en el anexo A, basado en el modelo de red neuronal convolucional de Dabhi [4]. En la siguiente subsección se brinda una explicación detallada de los elementos del modelo desarrollado.

i. *Estructura del modelo desarrollado*

El modelo de red neuronal convolucional implementado para resolver el problema planteado posee las distintas capas descritas a continuación, en orden de profundidad desde la entrada hasta la salida:

1) Capa de preprocesamiento de imágenes: se trata de una capa que normaliza las entradas de la red. Dado que los valores de entrada son las intensidades de cada píxel, las cuales van de 0 hasta 255, se agregó esta capa para estandarizarlos a un rango de 0 a 1, de modo que coincidan con el rango activo de las funciones de activación utilizadas a lo largo del modelo [5].

2) Primera capa de convolución: realiza la extracción de las *features* de más bajo nivel. Posee varios hiperparámetros configurables, como el número de filtros, el tamaño de estos (de dimensión cuadrada) y el tamaño del *stride*. Asimismo, posee una función de activación

ReLU y un *padding* de tal manera que el mapa de activación que genere tenga el mismo tamaño que su entrada (para un *stride* igual a 1).

3) Primera capa de *pooling*: efectúa una operación de *max pooling* para resaltar las características más dominantes del mapa de activación. Se eligió usar *max pooling* ya que esta técnica tiene un desempeño mucho mayor que otras, como el *average pooling*, para eliminar ruido de los mapas de activación [2]. Los hiperparámetros definidos para esta capa son el tamaño de la ventana de *pooling* (de dimensión cuadrada) y el *stride*.

4) Segunda capa de convolución: actúa sobre cada uno de los mapas de activación generados en la etapa anterior para extraer características de mayor nivel de las imágenes. Es similar a la primera capa de convolución en términos de los hiperparámetros disponibles, de la función de activación y el *padding*.

5) Segunda capa de *pooling*: su función y configuración es la misma que la primera capa de agrupación.

6) Capa de *flattening*: para transformar los mapas de activación de la última etapa en un vector de entrada para las capas densas.

7) Primera capa densa: corresponde a una capa oculta completamente conectada para efectuar la clasificación de la imagen según la información extraída por las capas convolucionales. Emplea la sigmoide como función de activación. El número de neuronas de esta capa se puede configurar por medio de un hiperparámetro, al igual que el porcentaje de dropout que le es aplicado, a modo de prevención del sobreentrenamiento.

8) Capa de salida: como el problema planteado corresponde a uno de clasificación binaria de imágenes, la capa de salida corresponde a una única neurona cuya salida puede tomar un rango de valores entre 0 y 1 (ya que usa la función de activación sigmoide), donde una salida cercana a cero indica que la pieza está en buen estado; mientras que, si la salida es cercana a uno, el sistema predijo que está defectuosa.

El entrenamiento de la red se hará con el conjunto de datos definido por el autor [1] para este propósito. Sin embargo, primero se extraerá un 20% de las imágenes de este conjunto, de manera aleatoria, para crear un nuevo subconjunto de validación. Es decir, se usará un 80% de las imágenes dadas por el autor para modificar los pesos (subconjunto de entrenamiento) y un 20% para validar los resultados durante el entrenamiento (subconjunto de validación). Estos subconjuntos se ingresarán a la red en lotes de 32 imágenes.

Para entrenar la red se usará el optimizador Adam, el cual se caracteriza por usar una tasa de aprendizaje adaptada a cada parámetro, la cual se determina a partir del primer y segundo momento

del gradiente. Este optimizador es ampliamente usado en *deep learning* porque produce buenos resultados y converge rápidamente [6].

Asimismo, se usará la entropía cruzada como función de pérdida, debido a que se trata de un problema de clasificación binario. El entrenamiento se ejecutará durante 40 *epochs* (corridas completas del conjunto de imágenes de entrenamiento) como máximo, pero será detenido prematuramente si se detecta una tendencia de aumento en la pérdida de validación, con tal de prevenir el sobreentrenamiento de la red.

A continuación, se explicará la estrategia de validación para evaluar el desempeño de la red durante el entrenamiento, así como en ejecución en tiempo real.

ii. *Estrategia de validación*

Se efectuarán una serie de pruebas a la red, durante el proceso de entrenamiento y luego con el modelo final, para evaluar sus resultados. Luego de ingresar cada lote del subconjunto de entrenamiento a la red y modificar los pesos, se hará una prueba de validación, usando el subconjunto destinado para este fin. Se calcularán los porcentajes de pérdida y exactitud durante el entrenamiento y la validación, los cuales son indicadores del desempeño de la red durante el aprendizaje. Se graficarán estos porcentajes a través de las *epochs* en las que transcurre el entrenamiento, para poder observar su evolución. A partir de estos resultados se podrá deducir si el entrenamiento de la red es efectivo o si se tienen que modificar hiperparámetros para lograr una mejor convergencia.

Una vez que se haya definido un modelo final, este se someterá a varias pruebas para verificar su capacidad de predicción durante la ejecución en tiempo real. Para ello, primero se usará el conjunto de prueba definido por el autor de los datos [1]; y luego se harán pruebas con un nuevo conjunto, formado por 453 imágenes de piezas malas y 262 imágenes de piezas buenas, extraídas al azar de la totalidad de imágenes proporcionadas por el autor. En este caso, se obtendrá la matriz de confusión de cada prueba, la cual brinda información sobre la confiabilidad de la red para distinguir entre piezas con y sin defectos.

Seguidamente, se presenta el estudio de hiperparámetros realizado para definir el modelo en su totalidad.

iii. *Estudio de hiperparámetros*

Para ejecutar este estudio se modificaron los siguientes hiperparámetros:

- Número de filtros (*kernels*) por cada capa convolucional.
- Tamaño de los *kernels* de cada capa convolucional.

- Magnitud de los *strides* en las capas convolucionales y de agrupación (*pooling*).
- Tamaño de la ventana (*pool size*) de las capas de agrupación.
- Número de neuronas de la capa densa.
- Porcentaje de *dropout* aplicado a la capa densa.

Se realizaron varias iteraciones para definir la mejor combinación de hiperparámetros del modelo. A continuación, se presentan los resultados obtenidos. Además, en el anexo B se pueden consultar los porcentajes de pérdida y de exactitud obtenidos en cada iteración, los cuales fueron impresos por el programa realizado.

Primera iteración

Para la primera iteración se usarán los hiperparámetros descritos a continuación:

- Se usarán 8 filtros en ambas capas convolucionales. Los filtros serán de tamaño 3x3 y el *stride* será de 1.
- Ambas capas de *pooling* tendrán un *pool size* de 2x2 y un *stride* de 2.
- La capa densa oculta tendrá 32 neuronas y un porcentaje de *dropout* del 20%.

Se obtuvieron las curvas de pérdida y de exactitud de la figura 2.

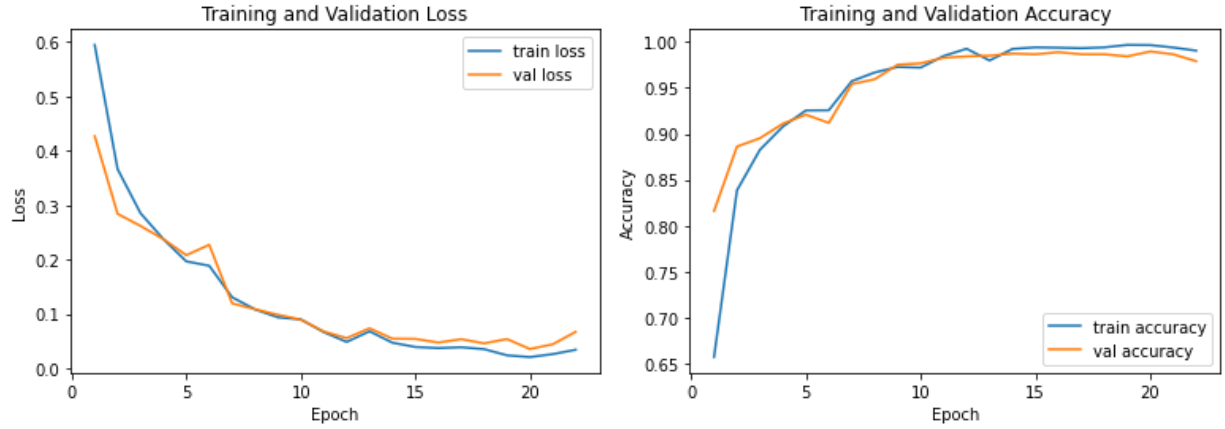

Figura 2. Curvas de pérdida y exactitud en la primera iteración de hiperparámetros.

De las curvas de la figura 2 y según los resultados presentados en el anexo B, se puede notar que, en general, el proceso de entrenamiento tuvo buenos resultados. La pérdida de validación fue de 3.54%, con una exactitud del 98.94%. Esto indica que la red tuvo un aprendizaje adecuado y es probable que tenga una buena capacidad de predicción en tiempo de ejecución. Sin embargo, los hiperparámetros de la red no están optimizados, por lo que un punto de mejora de la red es la capacidad computacional que requiere para procesar las imágenes.

Se realizará otra iteración donde se modifique el *stride* de la segunda capa de convolución. Se usará un *stride* de 3, igual que las dimensiones de los filtros, para reducir el tamaño de los mapas de activación que genera. De este modo se consigue disminuir el número de pesos que deben ser entrenados y, consecuentemente, se reduce la capacidad computacional requerida.

Segunda iteración

Las curvas obtenidas en este caso se muestran en la figura 3.

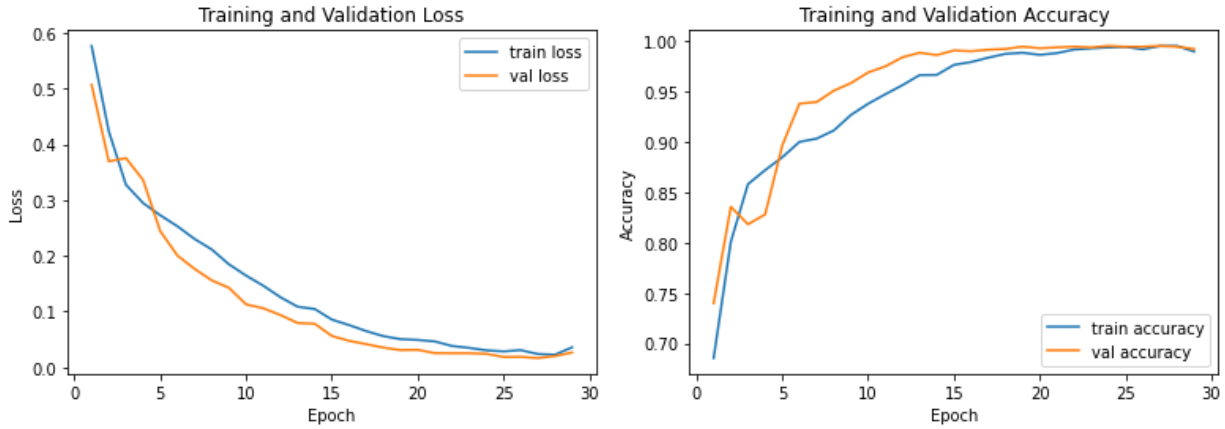

Figura 3. Curvas de pérdida y exactitud en la segunda iteración de hiperparámetros.

Se obtuvo una pérdida de validación del 1.66% y una exactitud de validación del 99.55%, las cuales son ligeramente mejores que en la primera iteración. Por lo tanto, la red sigue manteniendo un buen desempeño a pesar de haber reducido el número de pesos totales. Se puede continuar disminuyendo la demanda computacional de la red con tal de analizar si mantiene buenos resultados.

En la siguiente iteración se reducirá el número de filtros de la segunda capa convolucional: tendrá 4 en lugar que 8, para reducir el número de pesos involucrados y apreciar si esto afecta su desempeño.

Tercera iteración

Ahora se obtuvieron las curvas correspondientes a la figura 4.

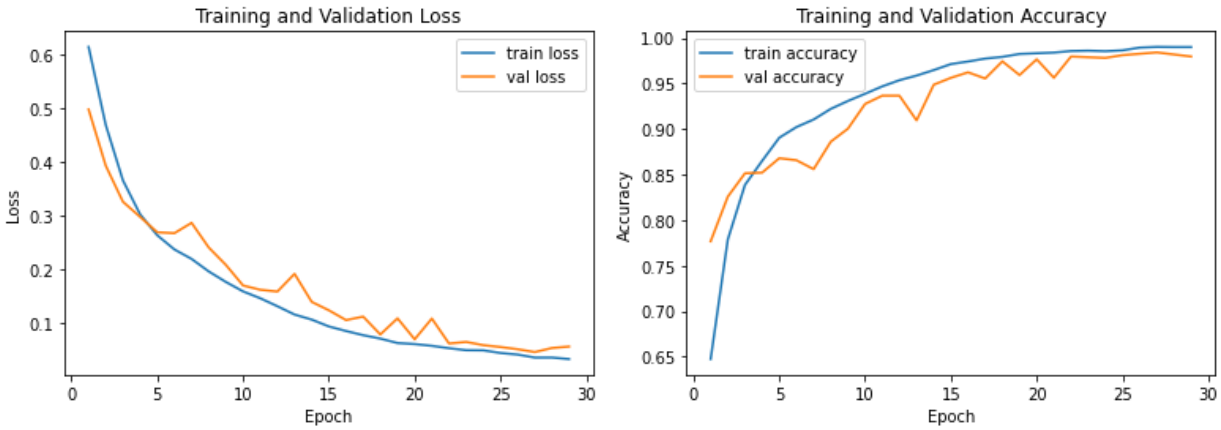

Figura 4. Curvas de pérdida y exactitud en la tercera iteración de hiperparámetros.

En este caso, la pérdida y la exactitud de validación fueron, respectivamente, de 4.47% y 98.42%. Estos porcentajes son ligeramente peores que en la primera iteración, pero de igual manera, la red mantiene un buen desempeño.

En la siguiente iteración se verificará si la red se beneficia de mantener el número de filtros de la segunda capa convolucional en 8, pero aumentando tamaño de la ventana de *pooling* en la segunda capa de agrupación y el *stride* en esta misma capa. Esto con tal de reducir el número de pesos de la red, pero sin modificar la cantidad de filtros con respecto a la primera iteración. Se usará un *pool size* de 5x5 y un *stride* de 5.

Cuarta iteración

En la figura 5 se aprecian las curvas de pérdida y de exactitud obtenidas en esta iteración.

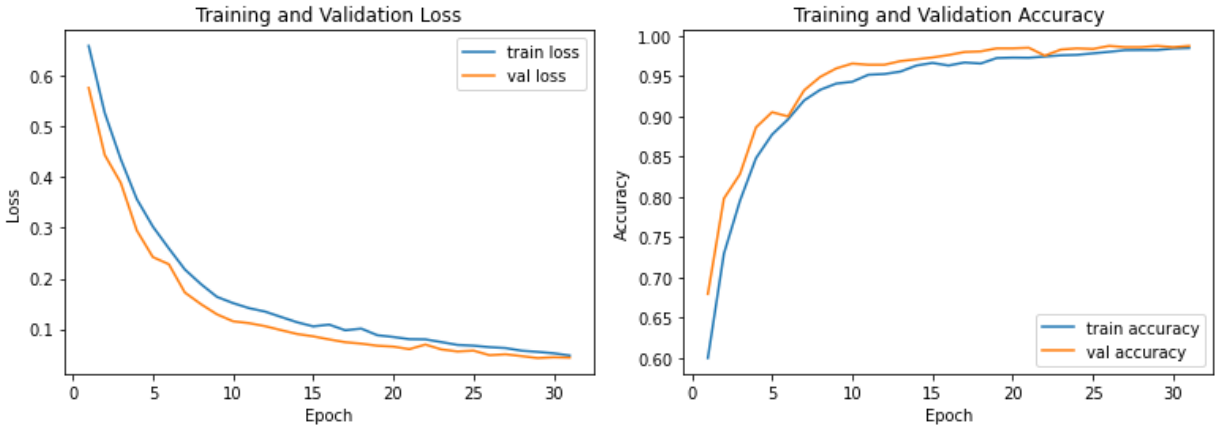

Figura 5. Curvas de pérdida y exactitud en la cuarta iteración de hiperparámetros.

Según los resultados presentados en el anexo B, la pérdida de validación fue de 4.30% y la exactitud fue de 98.72%, por lo que la red continúa con un buen desempeño, incluso ligeramente mayor que en la iteración número tres.

Se realizará otra iteración modificando la segunda capa de convolución, esta vez usando un *kernel* de tamaño 5x5, y manteniendo los *strides* en 3, para observar si existe un cambio que mejore o empeore la capacidad de la red.

Quinta iteración

Las curvas obtenidas en este caso se muestran en la figura 6.

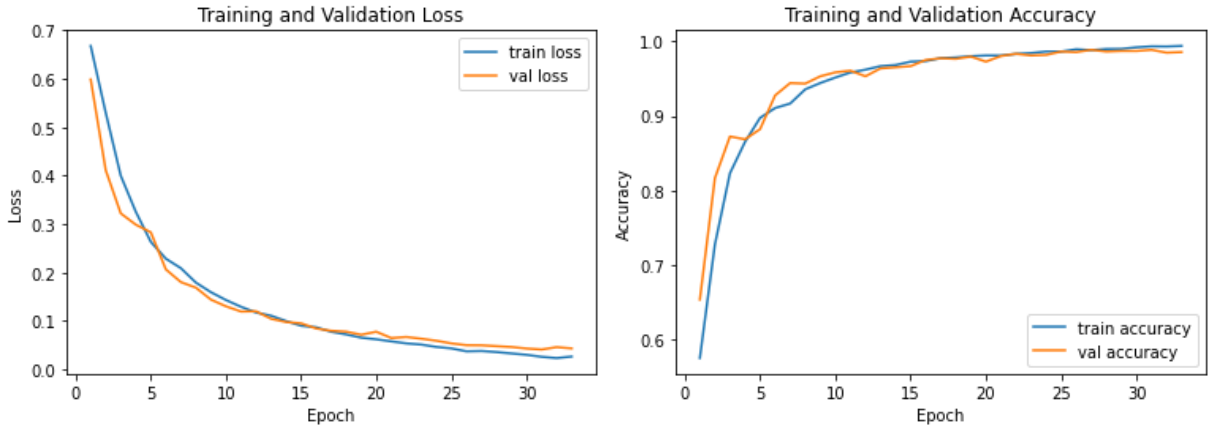

Figura 6. Curvas de pérdida y exactitud en la quinta iteración de hiperparámetros.

En esta iteración se obtuvieron porcentajes de 4.10% para la pérdida de validación y de 98.87% para la exactitud de validación, los cuales mejoraron ligeramente con respecto al caso anterior. Se obtuvieron buenos resultados de exactitud y a la vez se logró reducir el número de pesos que se tenían en relación con la primera iteración.

Ahora, se reducirá el número de neuronas en la capa densa, para observar qué efecto tiene sobre la red. Se usarán 25 neuronas en lugar de 32, lo cual disminuye la capacidad computacional requerida por la red.

Sexta iteración

Se obtuvieron las curvas mostradas en la figura 7.

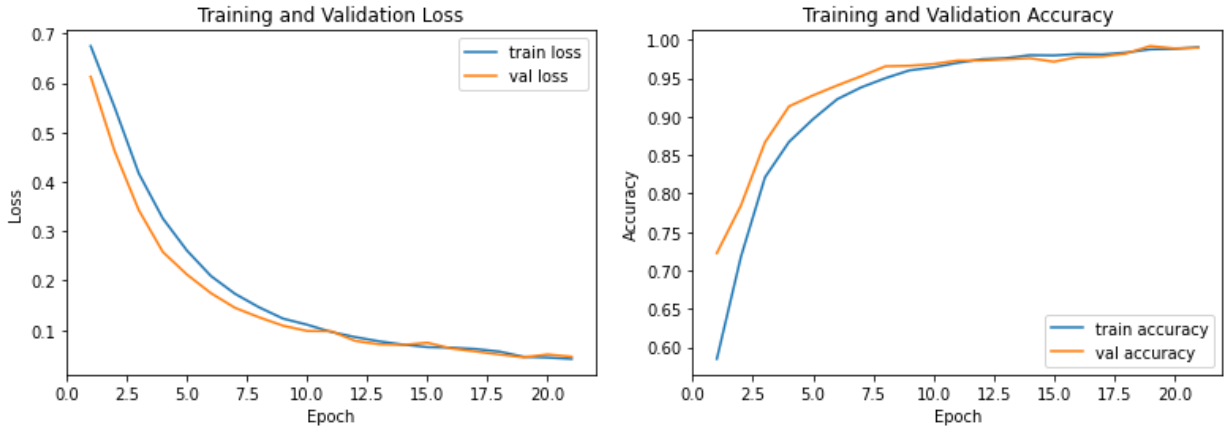

Figura 7. Curvas de pérdida y exactitud en la sexta iteración de hiperparámetros.

En este caso, la pérdida de validación fue de 4.48% y la exactitud del 99.17%, los cuales resultan ser bastante favorables y semejantes a los de la iteración anterior. Sin embargo, en este caso, se redujo el número de pesos de la red, logrando mantener su desempeño con una menor demanda de recursos computacionales.

Se realizará una última iteración para poder apreciar si la técnica de *dropout* tiene algún beneficio sobre la red. La iteración se realizará sin emplear dicha técnica, es decir, con el porcentaje de *dropout* definido en cero.

Sétima iteración

En la figura 8 se pueden observar las curvas obtenidas en este caso.

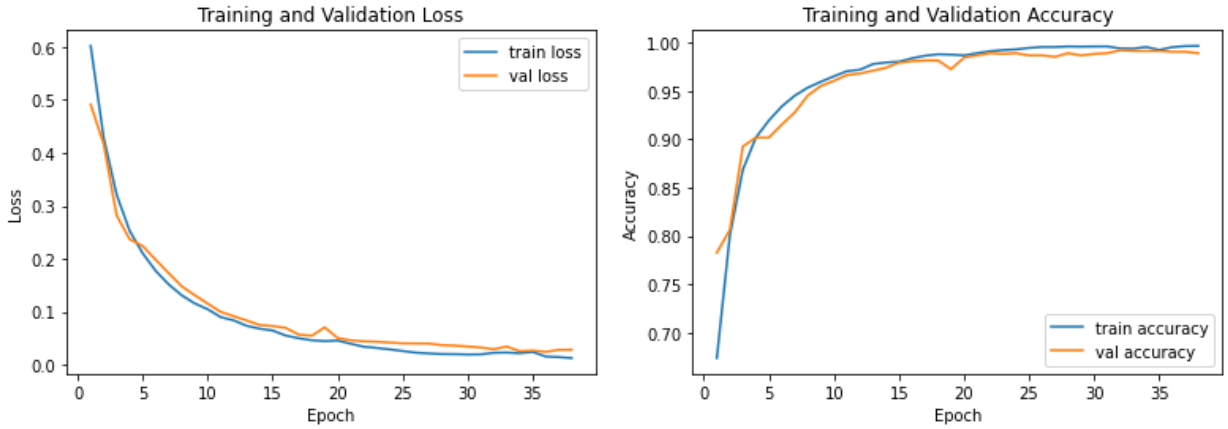

Figura 8. Curvas de pérdida y exactitud en la sétima iteración de hiperparámetros.

En esta última iteración se obtuvo un porcentaje de pérdida de validación de 2.41% y la exactitud de la validación fue del 99.10%. Se observa una mejora en estos porcentajes con respecto al caso anterior, por lo que se deduce que el modelo resultante tiene una mejor precisión si no se aplica la técnica de *dropout* en la capa oculta durante el entrenamiento.

iv. *Modelo final*

Luego de efectuar el estudio anterior, se obtuvo un modelo de red neuronal convolucional con los siguientes hiperparámetros:

- La primera capa convolucional tiene 8 filtros, de tamaño 3x3 y un *stride* de 1.
- La primera capa de *pooling* tiene un tamaño de ventana de 2x2 y un *stride* de 2.
- La segunda capa convolucional tiene 8 filtros, de tamaño 5x5 y un *stride* de 3.
- La segunda capa de *pooling* tiene un tamaño de ventana de 5x5 y un *stride* de 5.
- La capa densa oculta tiene 25 neuronas y sin *dropout*.

Resultados

Se realizaron dos procesos de entrenamiento y prueba sobre el modelo definido anteriormente. El primero de ellos fue empleando los subconjuntos de datos definidos por los autores [1]; y para el segundo proceso, se dividió la totalidad de los datos en subconjuntos de entrenamiento y prueba, de manera aleatoria y de modo que tuviesen la misma cantidad de imágenes de piezas defectuosas y en buen estado que los conjuntos originales. Se realizaron dos iteraciones de cada proceso para obtener un promedio entre los resultados de ambos.

En la figura 9 se presentan las matrices de confusión obtenidas para el primer proceso (usando los subconjuntos originales), y en la figura 10 se muestran las resultantes del segundo proceso descrito (con los subconjuntos personalizados).

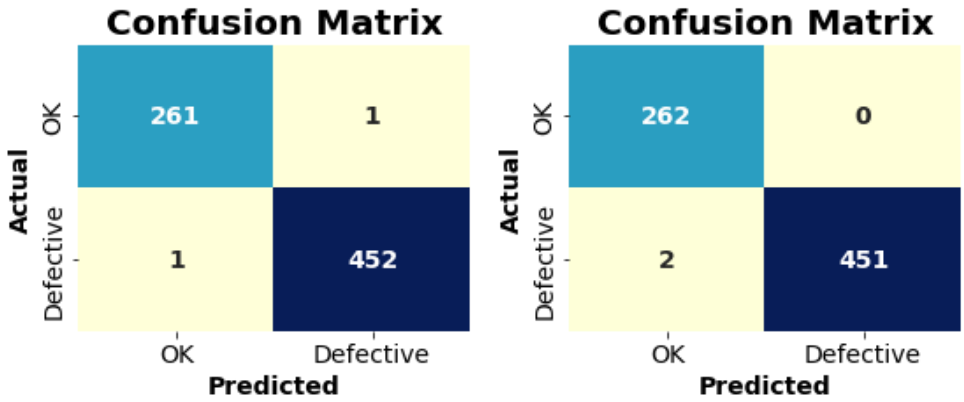

Figura 9. Matrices de confusión obtenidas usando los subconjuntos de datos originales.

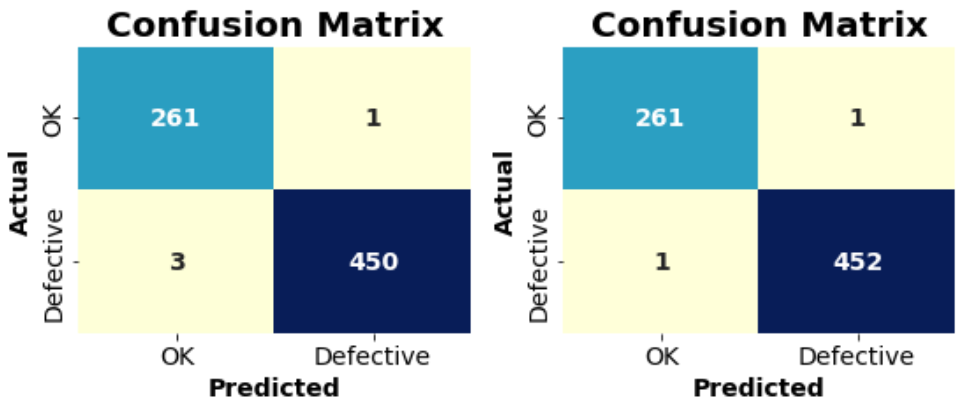

Figura 10. Matrices de confusión obtenidas usando los subconjuntos de datos personalizados.

En cuanto al proceso de entrenamiento y prueba con los subconjuntos de datos definidos por los autores, se puede notar que solo dos imágenes fueron clasificadas de manera incorrecta, en ambas iteraciones realizadas. Esto supone un 99.72% de exactitud en este caso.

Por otro lado, en relación con el proceso realizado con los subconjuntos de datos extraídos de manera aleatoria, se logra apreciar que en la primera iteración hubo un total de 4 imágenes clasificadas incorrectamente, y en la segunda, este número fue de 2. Esto se traduce en una precisión promedio del 99.58% entre ambas iteraciones.

Con respecto a estos resultados, es notorio que no hubo una diferencia significativa entre ambos procesos de entrenamiento y prueba. Esto se debe a que los subconjuntos de imágenes empleados en cada caso fueron prácticamente indistintos, en el sentido que tenían la misma cantidad de elementos de piezas buenas y malas. Además, todas las imágenes fueron tomadas en un entorno con iluminación controlada, con todas las piezas del mismo tamaño, en la misma posición y sobre un fondo de color blanco. Debido a estas características del conjunto de datos, no se presentaron sesgos que pudieran interferir con los resultados obtenidos en ambos casos.

Conclusiones y comentarios finales

Se logró desarrollar, entrenar y validar un modelo de red neuronal convolucional para clasificar imágenes, obteniendo una precisión mayor al 99% en todas las pruebas realizadas. Este modelo fue optimizado a través de un estudio de hiperparámetros, con lo que se consiguió reducir su demanda de recursos computacionales y a la vez manteniendo una buena capacidad de predicción.

El problema abordado en esta tarea es de suma interés ingenieril, debido a que permite automatizar procesos de control de calidad en industrias, a través de un sistema de visión y una red neuronal convolucional que detecte las piezas defectuosas y disponga de ellas, ahorrando el tiempo y dinero que implicaría efectuar este proceso de manera manual.

Referencias

[1] R. Dabhi, "Casting Product Image Data for Quality Inspection", *kaggle.com*. [Online]. Available: https://www.kaggle.com/datasets/ravirajsinh45/real-life-industrial-dataset-of-casting-product?resource=download. [Accessed Sept. 17, 2022].

[2] S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way", *towardsdatascience.com*. [Online] Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed Sept. 18, 2022].

[3] R. Holbrook, "Computer Vision", *kaggle.com*. [Online]. Available: https://www.kaggle.com/learn/computer-vision. [Accessed Sept. 18, 2022].

[4] R. Dabhi, "Simple Model for Casting Product Classification", *kaggle.com*. [Online]. Available: https://www.kaggle.com/code/ravirajsinh45/simple-model-for-casting-product-classification. [Accessed Sept. 17, 2022].

[5] A. P. Engelbretch, *Computational Intelligence. An Introduction*. West Sussex, England: John Wiley & Sons, Ltd, 2007.

[6] J. Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning", *machinelearningmastery.com*. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed Sept. 21, 2022].

Anexos

A. *Código de Python implementado*

```
#MT-8008 Inteligencia Artificial
#Tarea Evaluable: Sistemas Conexionistas parte II

#Elaborado por:
#César Argüello Salas
#Jose Fabio Navarro Naranjo


#-------------------------------------------------------------------------------
#Importación de librerías necesarias
import os
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import get_file
from tensorflow.keras.utils import image_dataset_from_directory
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

#No mostrar warnings en el output
warnings.filterwarnings('ignore')


#-------------------------------------------------------------------------------
#Descargar el conjunto de imágenes desde una carpeta de Dropbox personal

dataset_url = "https://www.dropbox.com/s/6v5v9275fntr18k/casting_data.zip?dl=1"
              #Enlace de descarga

#Función que realiza la descarga, extracción y guardado
#de las imágenes en la ruta /content/datasets
get_file(origin=dataset_url,
         fname='casting_data',
         cache_dir='/content',
```

```
          extract=True)


#-------------------------------------------------------------------------------
#Generar los datasets de entrenamiento, validación y prueba

#Hiperparámetros de esta sección
batch_size = 32           #Tamaño de lote (batch)
validation_split = 0.2    #Porcentaje de datos usados para validación


########### USANDO LOS SUBCONJUNTOS PREDETERMINADOS POR LOS AUTORES ###########
#                   (Comentar esta sección si no se utilizan)

#Los subconjuntos de entrenamiento/validación se extraen de la carpeta 'train'
#El subconjunto de prueba se extrae de la carpeta 'test'

seed = np.random.randint(1000) #Genera un entero aleatorio para emplear
       #como semilla para el split de datos de entrenamiento/validación


#Cargado de las imágenes en datasets mediante
#la función 'image_dataset_from_directory'

train_ds = image_dataset_from_directory(  #subconjunto de entrenamiento
  '/content/datasets/train',
  labels='inferred',
  label_mode='binary',
  class_names=['ok_front', 'def_front'],
  color_mode='grayscale',
  image_size=(300, 300),
  batch_size=batch_size,
  shuffle=True,
  seed=seed,
  validation_split=validation_split,
  subset='training')

val_ds = image_dataset_from_directory(    #subconjunto de validación
  '/content/datasets/train',
  labels='inferred',
  label_mode='binary',
  class_names=['ok_front', 'def_front'],
  color_mode='grayscale',
  image_size=(300, 300),
```

```
  batch_size=batch_size,
  shuffle=True,
  seed=seed,
  validation_split=validation_split,
  subset='validation')

test_ds = image_dataset_from_directory(   #subconjunto de prueba
  '/content/datasets/test',
  labels='inferred',
  label_mode='binary',
  class_names=['ok_front', 'def_front'],
  color_mode='grayscale',
  image_size=(300, 300),
  batch_size=batch_size,
  shuffle=True)


#################### USANDO SUBCONJUNTOS PERSONALIZADOS ######################
#                   (Comentar esta sección si no se utilizan)
"""
#El conjunto de imágenes completo se extrae de la carpeta 'custom'

#Se definen dos listas con los labels de cada grupo de imágenes
ok_labels = list(np.zeros(3137, dtype=int)) #lista de ceros para 'OK'
def_labels = list(np.ones(4211, dtype=int)) #lista de unos para 'Defective'


#Cargado del conjunto en dos datasets para cada tipo
ok_ds = image_dataset_from_directory(   #dataset de imágenes 'OK'
  '/content/datasets/custom/ok_front',
  labels=ok_labels,   #se etiquetan con 0
  label_mode='int',
  color_mode='grayscale',
  image_size=(300, 300),
  batch_size=None,
  shuffle=True)

def_ds = image_dataset_from_directory(   #dataset de imágenes 'Defective'
  '/content/datasets/custom/def_front',
  labels=def_labels,  #se etiquetan con 1
  label_mode='int',
  color_mode='grayscale',
  image_size=(300, 300),
  batch_size=None,
```

```
  shuffle=True)


#Se genera el subconjunto de entrenamiento con:
train_ds = ok_ds.skip(262)                          #2875 imágenes 'OK'
train_ds = train_ds.concatenate(def_ds.skip(453))   #3758 imágenes 'Defective'
train_ds = train_ds.shuffle(6633, reshuffle_each_iteration=False) #barajado

#Se genera el subconjunto de prueba con:

test_ds = ok_ds.take(262)                          #262 imágenes 'OK'
test_ds = test_ds.concatenate(def_ds.take(453))   #453 imágenes 'Defective'
test_ds = test_ds.shuffle(715, reshuffle_each_iteration=False)  #barajado

#Se extrae un subconjunto de 'train_ds' para validación
val_size = int(6633 * validation_split)
val_ds = train_ds.take(val_size)
train_ds = train_ds.skip(val_size)

#Se dividen los datasets en lotes (batches)
train_ds = train_ds.batch(batch_size)
val_ds = val_ds.batch(batch_size)
test_ds = test_ds.batch(batch_size)
"""
#############################################################################


#Configuración de los subconjuntos de datos para mejor rendimiento
AUTOTUNE = tf.data.AUTOTUNE

#Mantiene los datos en cache luego de cargarlos y superpone las tareas de
#preprocesamiento de datos y ejecución del modelo durante el entrenamiento
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)
test_ds = test_ds.cache().prefetch(buffer_size=AUTOTUNE)


#-----------------------------------------------------------------------------
#Definición del modelo de CNN

#Hiperparámetros de las capas convolucionales:
#filters            #número de filtros (kernels) por capa
#kernel_size        #tamaño del kernel (dimensión NxN)
#strides            #tamaño de paso entre deslizamientos
```

```
#Hiperparámetros de las capas de pooling:
#pool_size          #tamaño de la ventana de pooling (dimensión NxN)
#strides            #tamaño de paso entre deslizamientos

#Hiperparámetros de las capas densas:
#units              #número de neuronas por capa
dropout_rate = 0    #porcentaje de dropout


model = keras.Sequential([

  #Capa de preprocesamiento: normaliza los datos al rango [0, 1]
  layers.Rescaling(scale=1./255, input_shape=(300, 300, 1)),


  #Primera capa de convolución
  layers.Conv2D(filters=8,
                kernel_size=3,
                strides=1,
                padding='same',
                activation='relu'),

  #Primera capa de pooling
  layers.MaxPool2D(pool_size=2,
                   strides=2,
                   padding='same'),


  #Segunda capa de convolución
  layers.Conv2D(filters=8,
                kernel_size=5,
                strides=3,
                padding='same',
                activation='relu'),

  #Segunda capa de pooling
  layers.MaxPool2D(pool_size=5,
                   strides=5,
                   padding='same'),


  #Operación de flattening (convertir a vector)
  layers.Flatten(),

  #Primera capa densa
```

```
  layers.Dense(units=25, activation='sigmoid'),
  #Implementación de dropout a primera capa densa
  layers.Dropout(rate=dropout_rate),

  #Capa de salida
  layers.Dense(units=1, activation='sigmoid')
])

model.summary() #imprime un resumen del modelo creado


#-------------------------------------------------------------------------------
#Compilación y entrenamiento del modelo

model.compile(  #compilación: configura el entrenamiento del modelo
    optimizer='adam',           #se usa el optimizador Adam
    loss='binary_crossentropy', #función de pérdida empleada
    metrics=['accuracy'])       #métrica a ser evaluada: exactitud


#función para detener el entrenamiento de manera prematura
early_stopping = EarlyStopping( #se usa para evitar el sobreentrenamiento
    monitor='val_loss',         #variable a monitorear: pérdida de validación
    patience=2,                 #número máximo de epochs sin que haya mejora
    restore_best_weights=True,  #permite restaurar los pesos al mejor epoch
    verbose=1)  #muestra el resultado en el output


history = model.fit(  #ejecuta el entrenamiento del modelo
    train_ds,                   #dataset de entrenamiento
    validation_data=val_ds,     #dataset de validación
    epochs=40,                  #número máximo de epochs
    callbacks=[early_stopping], #llamada a la función de EarlyStopping
    verbose=1)  #muestra el resultado en el output


#-------------------------------------------------------------------------------
#Display de resultados del proceso de entrenamiento y validación

#conversión del historial de resultados a un dataframe
history_df = pd.DataFrame(history.history)
#se le suma 1 a cada índice para que coincidan con el número de epoch
history_df.index = map(lambda x : x+1, history_df.index)
```

```
#display de las curvas de pérdida de entrenamiento/validación
history_df.loc[:, ['loss', 'val_loss']].plot()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend(['train loss', 'val loss'])
plt.show()

#display de las curvas de exactitud de entrenamiento/validación
history_df.loc[:, ['accuracy', 'val_accuracy']].plot()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend(['train accuracy', 'val accuracy'])
plt.show()


#obtención de variables en la epoch con menor pérdida de validación
best_epoch = history_df['val_loss'].idxmin()
t_loss = history_df.loc[best_epoch, 'loss']
v_loss = history_df.loc[best_epoch, 'val_loss']
t_acc = history_df.loc[best_epoch, 'accuracy']
v_acc = history_df.loc[best_epoch, 'val_accuracy']

#impresión de resultados de la epoch con menor pérdida de validación
print("--------------------------------")
print("Best validation loss at epoch", best_epoch, "\n")
print("Training loss:            {:.2%}".format(t_loss))
print("Validation loss:          {:.2%}".format(v_loss))
print("Training accuracy:        {:.2%}".format(t_acc))
print("Validation accuracy:      {:.2%}".format(v_acc))
print("--------------------------------")


#-------------------------------------------------------------------------------
#Prueba del modelo mediante predicción con nuevos datos

#se ejecuta el modelo con el dataset de prueba
#y se guardan los resultados en un array
pred_probability = model.predict(test_ds)

#se evalúan los resultados de la clasificación
predictions = pred_probability > 0.5
#si la probabilidad de que esté defectuoso
#supera un 50% se clasifica como defectuoso (1)
```

```
#de lo contrario se clasifica como OK (0)

#se obtienen las etiquetas reales de las imágenes de prueba
#y se almacenan en un array
true_labels = np.concatenate([y for (x, y) in test_ds], axis=0)

#display de la matriz de confusión
plt.figure(figsize=(4,3))
plt.title('Confusion Matrix', size=20, weight='bold')
sns.heatmap(
    confusion_matrix(true_labels, predictions),
    annot=True,
    annot_kws={'size':14, 'weight':'bold'},
    fmt='d',
    cbar=False,
    cmap='YlGnBu',
    xticklabels=['OK', 'Defective'],
    yticklabels=['OK', 'Defective'])
plt.tick_params(axis='both', labelsize=14)
plt.ylabel('Actual', size=14, weight='bold')
plt.xlabel('Predicted', size=14, weight='bold')
plt.show()

#impresión del reporte de clasificación
print("\n-------------------------------------------------------")
print("Classification report:\n")
print(classification_report(true_labels, predictions, digits=4,
                            target_names=['OK', 'Defective']))
print("-------------------------------------------------------")
```

B. <u>Resultados del entrenamiento durante el estudio de hiperparámetros</u>

- **Primera iteración**

```
--------------------------------
Best validation loss at epoch 20

 Training loss:            2.06%
 Validation loss:          3.54%
Training accuracy:        99.64%
Validation accuracy:      98.94%
--------------------------------
```

- **Segunda iteración**

```
--------------------------------
Best validation loss at epoch 27

 Training loss:            2.39%
 Validation loss:          1.66%
Training accuracy:        99.53%
Validation accuracy:      99.55%
--------------------------------
```

- **Tercera iteración**

```
--------------------------------
Best validation loss at epoch 27

 Training loss:            3.44%
 Validation loss:          4.47%
Training accuracy:        99.02%
Validation accuracy:      98.42%
--------------------------------
```

- **Cuarta iteración**

```
--------------------------------
Best validation loss at epoch 29

 Training loss:            5.52%
 Validation loss:          4.30%
Training accuracy:        98.21%
Validation accuracy:      98.72%
--------------------------------
```

- **Quinta iteración**

```
--------------------------------
Best validation loss at epoch 31

 Training loss:            2.59%
 Validation loss:          4.10%
Training accuracy:        99.34%
Validation accuracy:      98.87%
--------------------------------
```

- **Sexta iteración**

```
--------------------------------
Best validation loss at epoch 19

 Training loss:            4.58%
 Validation loss:          4.48%
Training accuracy:        98.74%
Validation accuracy:      99.17%
--------------------------------
```

- **Sétima iteración**

```
--------------------------------
Best validation loss at epoch 36

 Training loss:            1.53%
 Validation loss:          2.41%
Training accuracy:        99.57%
Validation accuracy:      99.10%
--------------------------------
```