Laboratorio 4 | Data Science

- **Mónica Salvatierra** - 22249
- **Derek Arreaga** - 22537

Los objetivos de este laboratorio son:

1. Comprender la diferencia entre una red neuronal multicapa simple (ANN) y una red convolucional (CNN) para clasificación de imágenes.
2. Implementar desde cero una CNN en Keras/TensorFlow para el conjunto de datos CIFAR-10.
3. Entrenar, evaluar y analizar los resultados obtenidos con respecto a un modelo ANN simple.
4. Reflexionar sobre las ventajas y limitaciones de las CNN en visión por computadora.
5. Aplicar "Data Augmentation" para mejorar la capacidad de generalización del modelo

Parte 1: Preparación del Conjunto de Datos (CIFAR-10)

In [ ]:
```
#%pip install numpy matplotlib tensorflow
#%pip install keras

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
```

```
Requirement already satisfied: keras in c:\users\ale\appdata\local\packages\pythonso
ftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site-
packages (3.11.3)
Requirement already satisfied: absl-py in c:\users\ale\appdata\local\packages\python
softwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\sit
e-packages (from keras) (2.3.1)
Requirement already satisfied: numpy in c:\users\ale\appdata\local\packages\pythonso
ftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site-
packages (from keras) (1.26.4)
Requirement already satisfied: rich in c:\users\ale\appdata\local\packages\pythonsof
twarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site-p
ackages (from keras) (14.1.0)
Requirement already satisfied: namex in c:\users\ale\appdata\local\packages\pythonso
ftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site-
packages (from keras) (0.1.0)
Requirement already satisfied: h5py in c:\users\ale\appdata\local\packages\pythonsof
twarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site-p
ackages (from keras) (3.14.0)
Requirement already satisfied: optree in c:\users\ale\appdata\local\packages\pythons
oftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\site
-packages (from keras) (0.17.0)
Requirement already satisfied: ml-dtypes in c:\users\ale\appdata\local\packages\pyth
onsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\s
ite-packages (from keras) (0.5.3)
Requirement already satisfied: packaging in c:\users\ale\appdata\local\packages\pyth
onsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310\s
ite-packages (from keras) (25.0)
Requirement already satisfied: typing-extensions>=4.6.0 in c:\users\ale\appdata\loca
l\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packa
ges\python310\site-packages (from optree->keras) (4.14.1)
Requirement already satisfied: markdown-it-py>=2.2.0 in c:\users\ale\appdata\local\p
ackages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages
\python310\site-packages (from rich->keras) (4.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in c:\users\ale\appdata\local
\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packag
es\python310\site-packages (from rich->keras) (2.19.2)
Requirement already satisfied: mdurl~=0.1 in c:\users\ale\appdata\local\packages\pyt
honsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-packages\python310
\site-packages (from markdown-it-py>=2.2.0->rich->keras) (0.1.2)
Note: you may need to restart the kernel to use updated packages.
```

1. Importar y cargar el conjunto de datos CIFAR-10

In [3]:
```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 36s 0us/step
```

In [4]:
```
print("Tamaño del conjunto de entrenamiento:", x_train.shape)
print("Tamaño del conjunto de prueba:", x_test.shape)
```

```
Tamaño del conjunto de entrenamiento: (50000, 32, 32, 3)
Tamaño del conjunto de prueba: (10000, 32, 32, 3)
```

In [5]:
```
min_val = np.min(x_train)
max_val = np.max(x_train)
```

```
print(f"Valor mínimo: {min_val}")
print(f"Valor máximo: {max_val}")
```

```
Valor mínimo: 0
Valor máximo: 255
```

2. Normalizar los datos: de rango [0,255] a [0,1]

Como se observó anteriormente, los valores de nuestro **dataset** son entre **0** y **255**, ahora los normalizamos para que sean entre **0.0** a **1.0**

In [6]:
```
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

print("Valores normalizados en rango:", x_train.min(), "a", x_train.max())
```

```
Valores normalizados en rango: 0.0 a 1.0
```

Ejemplos de Imágenes

Para desplegar los ejemplos, visitamos la documentación oficial del dataset **CIFAR-10**, ahí encontramos que las categorías de las imágenes corresponden a los siguientes números:

Categoría	Índice
avión	0
auto	1
pájaro	2
gato	3
ciervo	4
perro	5
rana	6
caballo	7
barco	8
camión	9

In [8]:
```
class_names = ["avión", "auto", "pájaro", "gato", "ciervo",
               "perro", "rana", "caballo", "barco", "camión"]

plt.figure(figsize=(10, 4))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_train[i])
    plt.title(class_names[y_train[i][0]])
    plt.axis("off")
plt.suptitle("Ejemplos de imágenes CIFAR-10", fontsize=14)
plt.show()
```

En esta sección se cargó el conjunto CIFAR-10 y se normalizaron los valores de píxel para que estuvieran entre 0 y 1. Esto se realizó dividiendo entre 255. La normalización estabiliza el entrenamiento porque mantiene las activaciones y gradientes en escalas manejables, lo que ayuda a los optimizadores a converger más rápido. Además, se muestra una cuadrícula de ejemplos con sus etiquetas para verificar visualmente que la carga y el preprocesamiento fueran correctos y para tener una intuición básica de la variabilidad de clases y fondos presente en las imágenes.

Parte 2: Modelo Base ANN

In [ ]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import matplotlib.pyplot as plt
import numpy as np
import time
```

1. Definición del modelo ANN

In [ ]:
```
model_ann = keras.Sequential([
    layers.Flatten(input_shape=(32,32,3)),
    layers.Dense(512, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

2. Compilación del modelo

In [27]:
```
model_ann.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
```

```
    metrics=['accuracy']
)
```

3. Entrenamiento

In [ ]:

```
start_time = time.time()

history_ann = model_ann.fit(
    x_train, y_train,
    epochs=20,
    batch_size=256,
    validation_split=0.2,
    verbose=1
)

end_time = time.time()
train_time = end_time - start_time
print(f" Tiempo total de entrenamiento ANN: {train_time:.2f} segundos")
```

```
Epoch 1/20
157/157 - 5s - 30ms/step - accuracy: 0.2907 - loss: 2.0028 - val_accuracy: 0.3504 -
val_loss: 1.8217
Epoch 2/20
157/157 - 3s - 22ms/step - accuracy: 0.3728 - loss: 1.7501 - val_accuracy: 0.3680 -
val_loss: 1.7948
Epoch 3/20
157/157 - 3s - 22ms/step - accuracy: 0.4083 - loss: 1.6558 - val_accuracy: 0.4075 -
val_loss: 1.6645
Epoch 4/20
157/157 - 4s - 23ms/step - accuracy: 0.4284 - loss: 1.5961 - val_accuracy: 0.4062 -
val_loss: 1.6771
Epoch 5/20
157/157 - 4s - 24ms/step - accuracy: 0.4489 - loss: 1.5527 - val_accuracy: 0.4214 -
val_loss: 1.6246
Epoch 6/20
157/157 - 4s - 23ms/step - accuracy: 0.4609 - loss: 1.5126 - val_accuracy: 0.4355 -
val_loss: 1.5962
Epoch 7/20
157/157 - 3s - 22ms/step - accuracy: 0.4722 - loss: 1.4874 - val_accuracy: 0.4545 -
val_loss: 1.5420
Epoch 8/20
157/157 - 4s - 23ms/step - accuracy: 0.4845 - loss: 1.4489 - val_accuracy: 0.4718 -
val_loss: 1.5074
Epoch 9/20
157/157 - 4s - 25ms/step - accuracy: 0.4935 - loss: 1.4238 - val_accuracy: 0.4584 -
val_loss: 1.5339
Epoch 10/20
157/157 - 4s - 24ms/step - accuracy: 0.5044 - loss: 1.4022 - val_accuracy: 0.4780 -
val_loss: 1.4769
Epoch 11/20
157/157 - 4s - 23ms/step - accuracy: 0.5100 - loss: 1.3783 - val_accuracy: 0.4731 -
val_loss: 1.4929
Epoch 12/20
157/157 - 4s - 24ms/step - accuracy: 0.5175 - loss: 1.3587 - val_accuracy: 0.4725 -
val_loss: 1.5150
Epoch 13/20
157/157 - 4s - 24ms/step - accuracy: 0.5289 - loss: 1.3366 - val_accuracy: 0.4839 -
val_loss: 1.4763
Epoch 14/20
157/157 - 4s - 25ms/step - accuracy: 0.5338 - loss: 1.3135 - val_accuracy: 0.4865 -
val_loss: 1.4714
Epoch 15/20
157/157 - 4s - 23ms/step - accuracy: 0.5372 - loss: 1.3025 - val_accuracy: 0.4955 -
val_loss: 1.4361
Epoch 16/20
157/157 - 4s - 23ms/step - accuracy: 0.5426 - loss: 1.2865 - val_accuracy: 0.5061 -
val_loss: 1.4157
Epoch 17/20
157/157 - 4s - 23ms/step - accuracy: 0.5515 - loss: 1.2625 - val_accuracy: 0.4901 -
val_loss: 1.4746
Epoch 18/20
157/157 - 3s - 21ms/step - accuracy: 0.5582 - loss: 1.2494 - val_accuracy: 0.5101 -
val_loss: 1.4166
Epoch 19/20
157/157 - 3s - 22ms/step - accuracy: 0.5679 - loss: 1.2192 - val_accuracy: 0.5025 -
```

```
val_loss: 1.4229
Epoch 20/20
157/157 - 4s - 25ms/step - accuracy: 0.5702 - loss: 1.2039 - val_accuracy: 0.5146 -
val_loss: 1.4046
 Tiempo total de entrenamiento ANN: 74.51 segundos
```

La red neuronal totalmente conectada multicapa partió con una exactitud de 29.1% en entrenamiento y 35.0% en validación (pérdidas 2.00 y 1.82, respectivamente) y fue mejorando de forma consistente hasta cerrar en 57.0% de exactitud de entrenamiento y 51.5% de validación, con pérdidas de 1.20 y 1.40 al final de las 20 épocas. La tendencia general fue descendente en la pérdida y ascendente en la exactitud, con el mejor val_accuracy en 51.46% (época 20). La brecha final entre entrenamiento y validación (5–6 puntos) podría mostrar un ligero sobreajuste, esperable en una ANN que aplana la imagen y no aprovecha la estructura espacial. El tiempo total de entrenamiento fue de 74.51 s, consistente con un entrenamiento rápido para un modelo base.

En general, el desempeño es coherente para una red totalmente conectada, puesto que aprende patrones globales pero su capacidad de generalización es limitada frente a arquitecturas convolucionales; por ello, se espera que la CNN mejore estos resultados al capturar rasgos locales (bordes y texturas) y reducir parámetros efectivos mediante pesos compartidos.

Parte 3: Implementación CNN

1. Definición del modelo

In [30]:
```
model_cnn = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=(32,32,
    layers.Conv2D(32, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),

    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.Conv2D(64, (3,3), activation='relu', padding='same'),
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),

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

```
C:\Users\Ale\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8
p0\LocalCache\local-packages\Python310\site-packages\keras\src\layers\convolutional
\base_conv.py:113: UserWarning: Do not pass an `input_shape`/`input_dim` argument to
a layer. When using Sequential models, prefer using an `Input(shape)` object as the
first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

2. Compilación del modelo

In [31]:
```
model_cnn.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

3. Callbacks

In [32]:
```
early = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

ckpt = ModelCheckpoint(
    filepath='cnn_best.keras',
    monitor='val_loss',
    save_best_only=True
)

start = time.time()
history_cnn = model_cnn.fit(
    x_train, y_train,
    epochs=40,
    batch_size=128,
    validation_split=0.2,
    callbacks=[early, ckpt],
    verbose=2
)
end = time.time()
cnn_train_time = end - start
print(f" Tiempo total de entrenamiento CNN: {cnn_train_time:.2f} s")
```

```
Epoch 1/40
313/313 - 28s - 88ms/step - accuracy: 0.3325 - loss: 1.8177 - val_accuracy: 0.4694 -
val_loss: 1.4595
Epoch 2/40
313/313 - 25s - 81ms/step - accuracy: 0.4843 - loss: 1.4244 - val_accuracy: 0.5756 -
val_loss: 1.1903
Epoch 3/40
313/313 - 28s - 90ms/step - accuracy: 0.5526 - loss: 1.2458 - val_accuracy: 0.6112 -
val_loss: 1.1090
Epoch 4/40
313/313 - 26s - 83ms/step - accuracy: 0.5913 - loss: 1.1441 - val_accuracy: 0.6503 -
val_loss: 0.9864
Epoch 5/40
313/313 - 26s - 83ms/step - accuracy: 0.6220 - loss: 1.0700 - val_accuracy: 0.6797 -
val_loss: 0.9051
Epoch 6/40
313/313 - 26s - 83ms/step - accuracy: 0.6462 - loss: 1.0010 - val_accuracy: 0.7037 -
val_loss: 0.8513
Epoch 7/40
313/313 - 26s - 84ms/step - accuracy: 0.6649 - loss: 0.9573 - val_accuracy: 0.7185 -
val_loss: 0.8040
Epoch 8/40
313/313 - 26s - 82ms/step - accuracy: 0.6815 - loss: 0.9120 - val_accuracy: 0.7261 -
val_loss: 0.7878
Epoch 9/40
313/313 - 25s - 81ms/step - accuracy: 0.6923 - loss: 0.8738 - val_accuracy: 0.7396 -
val_loss: 0.7533
Epoch 10/40
313/313 - 25s - 80ms/step - accuracy: 0.7031 - loss: 0.8378 - val_accuracy: 0.7281 -
val_loss: 0.7766
Epoch 11/40
313/313 - 24s - 78ms/step - accuracy: 0.7191 - loss: 0.8026 - val_accuracy: 0.7464 -
val_loss: 0.7289
Epoch 12/40
313/313 - 25s - 80ms/step - accuracy: 0.7262 - loss: 0.7805 - val_accuracy: 0.7435 -
val_loss: 0.7397
Epoch 13/40
313/313 - 26s - 84ms/step - accuracy: 0.7397 - loss: 0.7479 - val_accuracy: 0.7562 -
val_loss: 0.7167
Epoch 14/40
313/313 - 29s - 94ms/step - accuracy: 0.7465 - loss: 0.7289 - val_accuracy: 0.7446 -
val_loss: 0.7432
Epoch 15/40
313/313 - 27s - 86ms/step - accuracy: 0.7491 - loss: 0.7155 - val_accuracy: 0.7637 -
val_loss: 0.6948
Epoch 16/40
313/313 - 25s - 79ms/step - accuracy: 0.7603 - loss: 0.6821 - val_accuracy: 0.7651 -
val_loss: 0.6786
Epoch 17/40
313/313 - 27s - 86ms/step - accuracy: 0.7667 - loss: 0.6658 - val_accuracy: 0.7637 -
val_loss: 0.6880
Epoch 18/40
313/313 - 25s - 79ms/step - accuracy: 0.7710 - loss: 0.6529 - val_accuracy: 0.7715 -
val_loss: 0.6689
Epoch 19/40
313/313 - 25s - 81ms/step - accuracy: 0.7756 - loss: 0.6338 - val_accuracy: 0.7718 -
```

```
val_loss: 0.6711
Epoch 20/40
313/313 - 24s - 76ms/step - accuracy: 0.7823 - loss: 0.6193 - val_accuracy: 0.7728 -
val_loss: 0.6663
Epoch 21/40
313/313 - 24s - 78ms/step - accuracy: 0.7845 - loss: 0.6068 - val_accuracy: 0.7616 -
val_loss: 0.6996
Epoch 22/40
313/313 - 26s - 83ms/step - accuracy: 0.7883 - loss: 0.5900 - val_accuracy: 0.7777 -
val_loss: 0.6450
Epoch 23/40
313/313 - 25s - 79ms/step - accuracy: 0.7947 - loss: 0.5777 - val_accuracy: 0.7750 -
val_loss: 0.6653
Epoch 24/40
313/313 - 23s - 74ms/step - accuracy: 0.7997 - loss: 0.5635 - val_accuracy: 0.7822 -
val_loss: 0.6363
Epoch 25/40
313/313 - 24s - 75ms/step - accuracy: 0.8030 - loss: 0.5509 - val_accuracy: 0.7802 -
val_loss: 0.6526
Epoch 26/40
313/313 - 23s - 72ms/step - accuracy: 0.8028 - loss: 0.5517 - val_accuracy: 0.7779 -
val_loss: 0.6665
Epoch 27/40
313/313 - 23s - 74ms/step - accuracy: 0.8070 - loss: 0.5380 - val_accuracy: 0.7828 -
val_loss: 0.6501
Epoch 28/40
313/313 - 23s - 72ms/step - accuracy: 0.8106 - loss: 0.5266 - val_accuracy: 0.7813 -
val_loss: 0.6532
Epoch 29/40
313/313 - 23s - 74ms/step - accuracy: 0.8161 - loss: 0.5189 - val_accuracy: 0.7851 -
val_loss: 0.6538
 Tiempo total de entrenamiento CNN: 732.89 s
```

La CNN mostró una mejora clara frente a la ANN. Inició con 33.3% de exactitud en entrenamiento y 46.9% en validación (pérdidas 1.82 y 1.46). En las primeras 10 épocas la validación superó el 70% y luego fue estabilizándose alrededor de 76–78%. El mejor val_accuracy observado fue 78.51% (época 29), mientras que el mejor val_loss fue 0.6363 (época 24). Como se usó EarlyStopping con monitor='val_loss' y restore_best_weights=True, los pesos finales restaurados corresponden a la época 24 (val_acc ≈ 78.22%). La exactitud de entrenamiento llegó a 81.61% (época 29), dejando una brecha respecto a validación de alrededor del 3% (o 1.8% si consideramos los pesos restaurados), lo que indica buena generalización con un sobreajuste leve controlado por Dropout y EarlyStopping.

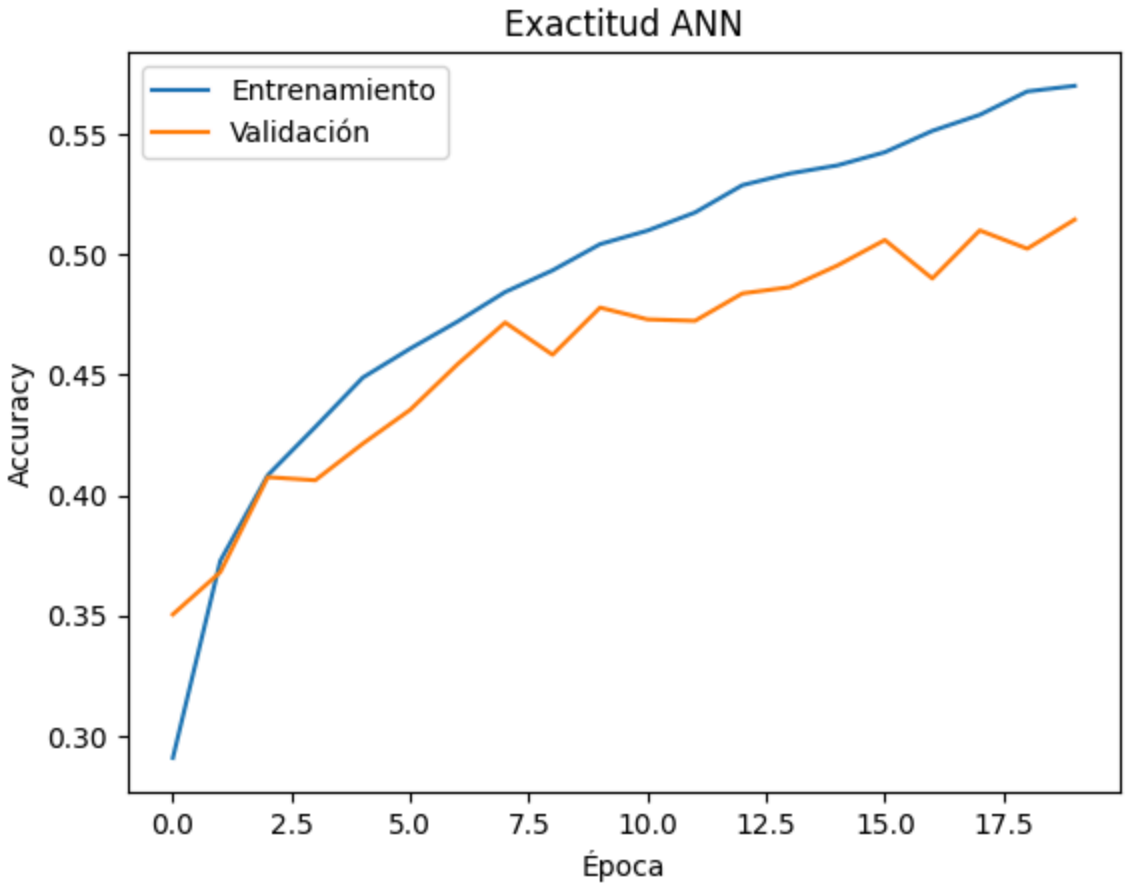
Parte 4: Evaluación y Comparación

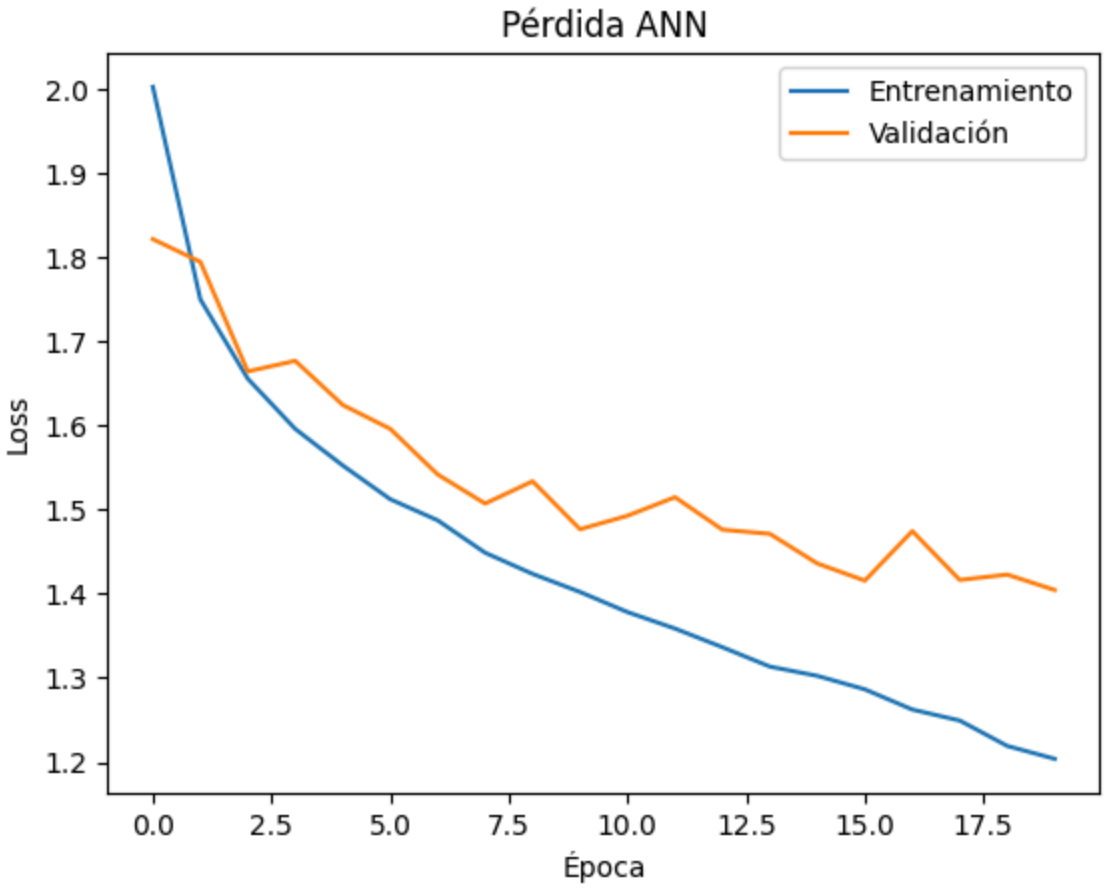
1. Gráficas de desempeño

In [34]:
```
# Exactitud
plt.plot(history_ann.history['accuracy'], label='Entrenamiento')
plt.plot(history_ann.history['val_accuracy'], label='Validación')
plt.title('Exactitud ANN')
```

```
plt.xlabel('Época')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Pérdida
plt.plot(history_ann.history['loss'], label='Entrenamiento')
plt.plot(history_ann.history['val_loss'], label='Validación')
plt.title('Pérdida ANN')
plt.xlabel('Época')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



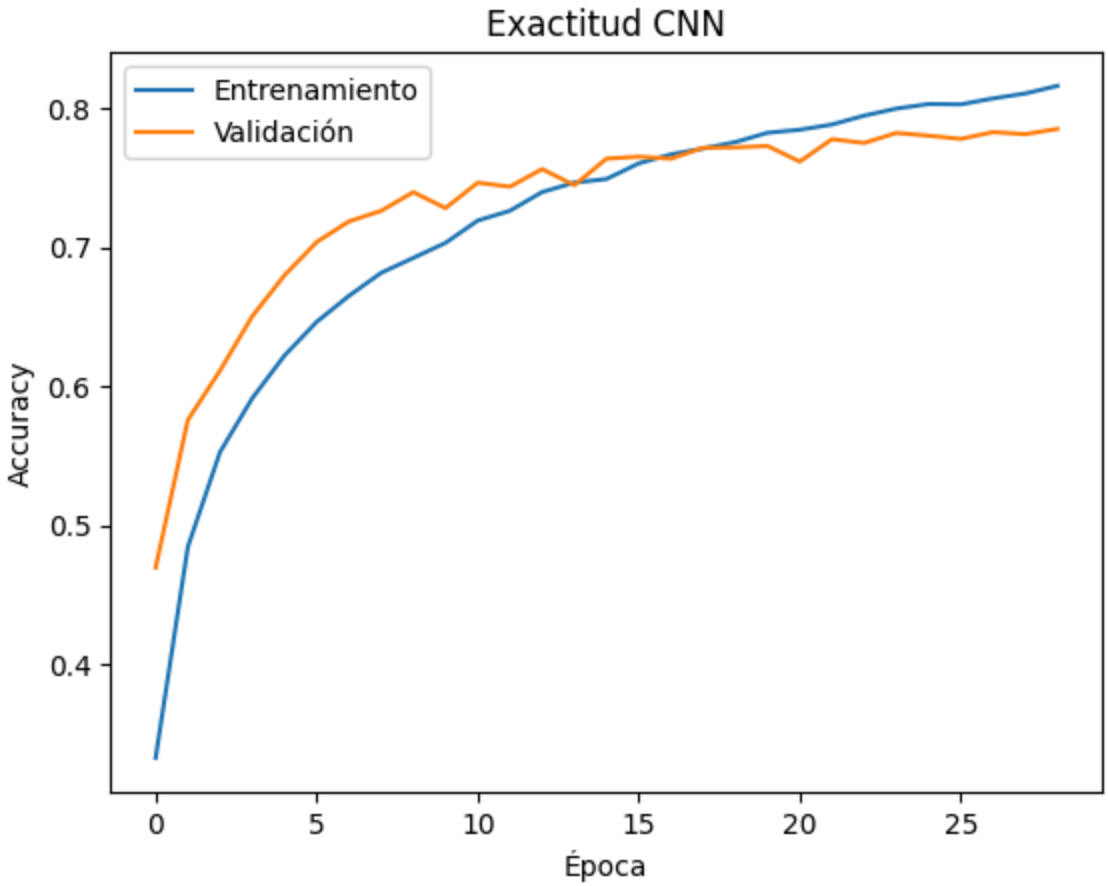


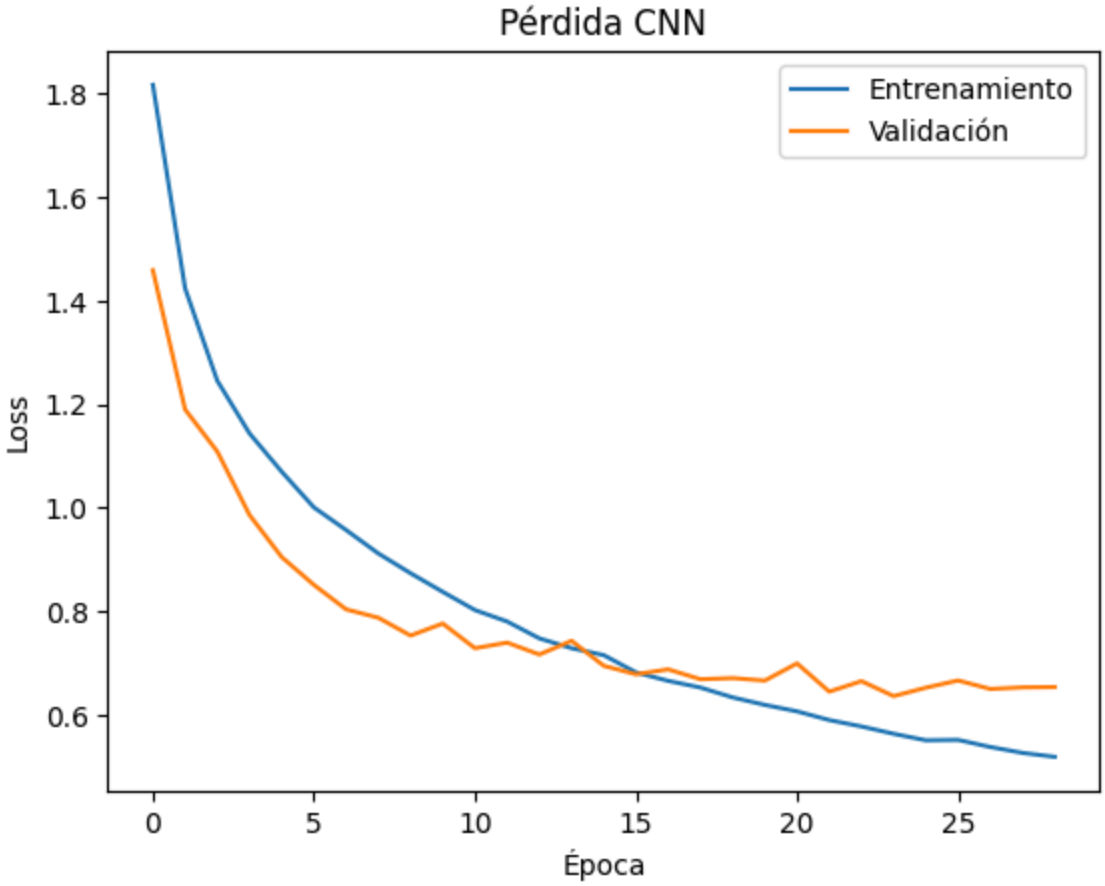
La exactitud de entrenamiento crece de ~29% a 57%, mientras que la de validación pasa de 35% a 51.5%. A partir de la época 7 la brecha entre ambas se estabiliza alrededor de 5–6 puntos porcentuales, con pequeñas oscilaciones (aproximadamente en las épocas 9–13 y 17). Esto indica aprendizaje real pero con capacidad de generalización limitada, ya que el modelo sigue mejorando en entrenamiento, aunque la validación progresa más lento. En general, el perceptrón aprende patrones globales, pero al aplanar la imagen no aprovecha la estructura espacial de CIFAR-10, por lo que se espera que la CNN (con filtros locales y pesos compartidos) alcance mejor exactitud y pérdidas de validación menores.

In [33]:
```
# Exactitud
plt.plot(history_cnn.history['accuracy'], label='Entrenamiento')
plt.plot(history_cnn.history['val_accuracy'], label='Validación')
plt.title('Exactitud CNN')
plt.xlabel('Época')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Pérdida
plt.plot(history_cnn.history['loss'], label='Entrenamiento')
plt.plot(history_cnn.history['val_loss'], label='Validación')
plt.title('Pérdida CNN')
plt.xlabel('Época')
plt.ylabel('Loss')
```

```
plt.legend()
plt.show()
```





Las curvas de la CNN muestran un aprendizaje rápido y estable; la exactitud de validación pasa de ~47% a >70% antes de la época 6 y luego se mantiene en 76–79%, con la de entrenamiento alcanzándola hacia las épocas 12–15; la brecha final es pequeña, señal de buena generalización. En pérdida, train desciende de ~1.8 a ~0.52 y validación de ~1.45 a ~0.64; al inicio val_loss < loss (efecto de Dropout en train), se cruzan cerca de la época 12–13 y aparecen oscilaciones leves. El mejor val_loss ronda 0.636 (alrededor de la época 24, pesos restaurados por early stopping). La CNN reduce claramente la pérdida y eleva la exactitud frente al ANN, coherente con que las convoluciones capturan patrones espaciales que el modelo totalmente conectado no aprovecha.

Característica	ANN (Red Neuronal Densa)	CNN (Red Convolucional)
Precisión inicial (época 1)	29% entrenamiento / 35% validación	33% entrenamiento / 47% validación
Precisión final	57% entrenamiento / 51% validación	81% entrenamiento / 78% validación
Pérdida final (val_loss)	1.40	0.65
Tiempo total de entrenamiento	74.5 segundos	732.9 segundos
Arquitectura	2 capas densas grandes (512 y 256 neuronas)	Capas convolucionales + pooling + dropout + capa densa

Característica	ANN (Red Neuronal Densa)	CNN (Red Convolucional)
Riesgo de overfitting	Moderado (pocas capas, pero se estanca en validación)	Bajo-moderado (uso de dropout y early stopping)
Capacidad de generalización	Limitada	Alta
Costo computacional	Bajo	Alto
Adecuación al problema de imágenes	Regular	Excelente

5. Reflexión Crítica

- ¿Por qué la `CNN` supera (o no) al modelo `ANN` en este problema?
    - En nuestro caso, la CNN superó al modelo `ANN` porque está diseñada para trabajar con datos de tipo imagen. La ANN, aunque logra aprender patrones generales, trata cada píxel de manera independiente después del aplanamiento, lo que hace que pierda la estructura espacial como los bordes o texturas. Por eso, su precisión se estancó alrededor del **51%** en validación, mientras que la `CNN` alcanzó aproximadamente un **78%**, demostrando su capacidad de extraer características jerárquicas. A pesar de que la CNN requiere un tiempo de entrenamiento mucho mayor, la ganancia en rendimiento justifica el costo computacional en este caso.
- ¿Qué papel juegan las capas de convolución y pooling?
    - Las capas de convolución son el núcleo de las `CNN`. Su función es aplicar filtros que detectan patrones locales como bordes, esquinas y texturas. Al apilar varias capas de convolución, la red logra construir representaciones progresivamente más abstractas, lo que facilita la clasificación de imágenes complejas.
    - El pooling, reduce la dimensionalidad conservando las características más relevantes. Esto no solo disminuye el costo computacional y la memoria requerida, sino que también ayuda a mejorar la invariancia a traslaciones y pequeñas deformaciones. La CNN implementada logra generalizar mejor a medida que se profundizan las capas convolucionales y se aplica pooling, por lo que se alcanzó una `val_accuracy` mucho más alta que la `ANN`.
- ¿Qué mejoras aplicaría?
    - Una primera mejora sería aplicar **data augmentation**, con transformaciones como rotaciones, traslaciones o espejados, lo que incrementaría la diversidad del dataset y reduciría el overfitting.
    - También sería útil probar **arquitecturas más profundas**, como **VGG**, **ResNet** o **DenseNet**, que están diseñadas específicamente para tareas de clasificación en imágenes.

    - Otra posible mejora es el **batch normalization**, que acelera el entrenamiento al estabilizar la distribución de activaciones, permitiendo entrenar redes más profundas con mayor estabilidad.

Referencias

- A [Ayşe Yılmaz]. (2024, 26 mayo). CNN:VGG, ResNet,DenseNet,MobileNet, EffecientNet,and YOLO. Medium. https://medium.com/@jaguuai/cnn-vgg-resnet-densenet-mobilenet-effecientnet-and-yolo-2329a9fa2d0f
- Sahota, H. (2025, 24 abril). An Intuitive Guide to Convolutional Neural Networks. Comet. https://www.comet.com/site/blog/an-intuitive-guide-to-convolutional-neural-networks/