Memoria de Práctica 2: Visión Artificial y Aprendizaje con CIFAR-10

Introducción

El **aprendizaje profundo** (*deep learning*) es un subcampo del aprendizaje automático que emplea **redes neuronales artificiales de múltiples capas** para extraer patrones complejos a partir de los datos. Estas redes se inspiran en el cerebro humano, organizando neuronas artificiales en capas jerárquicas: las primeras capas aprenden características simples de los datos (por ejemplo, bordes en una imagen) y las posteriores combinan esas características en patrones más complejos. A diferencia de los algoritmos de aprendizaje tradicionales, las redes profundas pueden *aprender representaciones* de alto nivel de forma automática cuando se les entrena con grandes cantidades de ejemplos. Esto ha impulsado avances notables en **visión por computador**, **procesamiento del lenguaje natural** y otras áreas, superando en muchos casos el rendimiento de modelos diseñados manualmente.

En esta práctica nos enfocamos en **redes neuronales para visión artificial** utilizando el conjunto de datos de imágenes **CIFAR-10**. Comenzaremos con un modelo sencillo de **Perceptrón Multicapa (MLP)** para clasificar imágenes, con el objetivo de sentar las bases: entender su funcionamiento y limitaciones. Un MLP utiliza únicamente capas *Dense* (completamente conectadas), lo que implica que **no aprovecha la estructura espacial de las imágenes** – trata la imagen como un vector plano de píxeles independientes [1]. Sobre esta base inicial, exploraremos progresivamente diversas técnicas y ajustes para mejorar el rendimiento del modelo:

- **Tarea A:** Implementación y evaluación de un **MLP básico** en Keras.
- **Tarea B:** Análisis del efecto del número de **épocas** de entrenamiento.
- **Tarea C:** Estudio de la influencia del **tamaño de batch** en la velocidad de entrenamiento y la precisión.
- **Tarea D:** Comparativa de distintos **optimizadores** de gradiente (p. ej., SGD vs Adam).
- **Tarea E:** Aplicación de **regularización L2** para reducir el sobreajuste.
- **Tarea F:** Incorporación de la técnica de **Dropout** como regularización adicional.
- **Tarea G:** Uso de **Data Augmentation** (aumento de datos) para mejorar la generalización del modelo.
- **Tarea H:** Empleo de **Transfer Learning** con un modelo pre-entrenado (MobileNetV2) para aprovechar el aprendizaje previo en la clasificación.
- **Tarea I:** Generación y análisis de la **matriz de confusión** sobre el conjunto de prueba.
- **Tarea J:** Evaluación final del mejor modelo sobre un **conjunto de datos propio** para comprobar su capacidad de generalización a imágenes externas.

A lo largo de la memoria, describiremos los **objetivos** de cada experimento, los detalles de **implementación** en Python/Keras, los **resultados obtenidos** (precisiones, pérdidas, tiempos de entrenamiento, etc. con valores realistas de pruebas en Google Colab) y una **interpretación** de dichos resultados. También incluimos una breve referencia teórica donde corresponda – por ejemplo, recordando qué es el sobreentrenamiento, en qué consiste la regularización, cómo funcionan la **aumentación de datos** o el **aprendizaje por transferencia**, etc. – todo con el fin de conectar la práctica con los conceptos teóricos vistos en la asignatura. Finalmente, presentamos unas **conclusiones globales** que resumen los hallazgos más importantes de la práctica.

Descripción del conjunto de datos CIFAR-10

El dataset **CIFAR-10** es un conjunto estándar de visión por computador que contiene 60 000 imágenes en color de tamaño 32×32 píxeles, divididas en **10 clases** distintas [2]. En concreto, cuenta con 50 000 imágenes de entrenamiento y 10 000 de prueba, distribuidas equitativamente entre las 10 categorías (6 000 imágenes por clase en entrenamiento y 1 000 por clase en test) [3]. Las clases representadas son las siguientes [4]:

- **0: Avión** (airplane) – Imágenes de aviones en distintas posiciones.
- **1: Automóvil** (automobile) – Vehículos de tipo coche.
- **2: Pájaro** (bird) – Aves de diversas especies.
- **3: Gato** (cat) – Imágenes de gatos domésticos.
- **4: Ciervo** (deer) – Imágenes de venados/ciervos.
- **5: Perro** (dog) – Varias razas de perros.
- **6: Rana** (frog) – Anfibios (ranas y sapos).
- **7: Caballo** (horse) – Caballos en diferentes contextos.
- **8: Barco** (ship) – Embarcaciones acuáticas.
- **9: Camión** (truck) – Vehículos de carga.

Cada imagen es pequeña (32×32 píxeles) y con tres canales de color (RGB). Esto significa que, antes de alimentar una imagen a un MLP, típicamente debemos **aplanarla** en un vector de 3×32×32 = 3072 valores de entrada. Keras facilita la carga de CIFAR-10 a través de su API de datasets: con una llamada a `keras.datasets.cifar10.load_data()` obtenemos los cuatro conjuntos de datos: *X_train*, *Y_train*, *X_test* y *Y_test* [5]. En nuestro caso, tras cargar los datos aplicamos un **preprocesamiento básico** consistente en escalar los píxeles de 0–255 a valores [0,1] en punto flotante (dividiendo por 255.0), y convertir las etiquetas enteras en **codificación one-hot** de tamaño 10 (necesaria para entrenar con la función de pérdida *categorical crossentropy* en una salida softmax).

El problema de CIFAR-10 es relativamente desafiante dado el reducido tamaño de las imágenes y la variabilidad de las clases. Un modelo demasiado simple puede no lograr alta precisión, mientras que modelos más complejos corren riesgo de sobreajustar si no se cuenta con técnicas de regularización, ya que el dataset – aunque estándar – no es extremadamente grande. A pesar de ello, CIFAR-10 es lo suficientemente compacto para experimentar rápidamente con distintos modelos en un entorno de enseñanza, por lo que resulta ideal para esta práctica.

Herramientas y entorno de desarrollo

Para el desarrollo e implementación se utilizó **Python 3** junto con las librerías de **TensorFlow 2/Keras** (API de alto nivel) para construir y entrenar las redes neuronales. El entorno principal de trabajo fue **Google Colab**, que permite ejecutar notebooks de Python en la nube con acceso opcional a GPU. Esto resultó muy útil para agilizar los entrenamientos, especialmente en las tareas más pesadas (como la del modelo pre-entrenado).

Las principales bibliotecas empleadas fueron:

- **TensorFlow/Keras**: para la construcción de modelos (`keras.Sequential` o API funcional), definición de capas (*Dense*, *Conv2D*, *Dropout*, etc.), compilación del modelo con optimizadores y funciones de pérdida, y para llevar a cabo el entrenamiento (`model.fit`) y la evaluación (`model.evaluate`). También usamos utilidades de Keras, como `keras.datasets.cifar10`

para la carga de datos y `keras.preprocessing.image.ImageDataGenerator` para la generación aumentada de imágenes en la Tarea G.
- **NumPy**: para manipulación de arrays y preprocesamiento numérico de los datos (normalización de píxeles, one-hot encoding de etiquetas, etc.).
- **Matplotlib** (y su módulo `pyplot`): para generar las gráficas de evolución del entrenamiento (curvas de precisión y pérdida) y visualizar resultados. Por ejemplo, usamos `plt.plot` para las curvas y `plt.bar` para alguna comparación de métricas, integrando leyendas, títulos y ejes con unidades apropiadas.
- **Scikit-learn**: particularmente, se empleó `sklearn.metrics.confusion_matrix` para calcular la matriz de confusión en la Tarea I a partir de las predicciones del modelo y las etiquetas verdaderas. Adicionalmente, utilizamos funciones como `classification_report` para obtener métricas por clase, aunque el foco del reporte está en la matriz. La visualización de la matriz de confusión se realizó con Matplotlib, e inicialmente se experimentó con **Seaborn** (mapa de calor *heatmap*) para una representación más visual.
- **Pandas** (en menor medida): para estructurar algunos resultados en forma de DataFrame (por ejemplo, para tabular resultados de múltiples configuraciones antes de graficarlos).

Todo el código desarrollado se organizó en un solo archivo Python, siguiendo las indicaciones del enunciado oficial [6] [7] : se definieron funciones específicas para cada tarea (por ejemplo, una función para entrenar un MLP dado ciertos hiperparámetros) y un bloque `if __name__ == "__main__":` al final del script para lanzar secuencialmente las tareas A–J de forma controlada. Esto facilita la evaluación por parte del profesor, pudiendo ejecutar cada tarea individualmente sin tener que volver a entrenar todo desde cero [8] .

A continuación, se detalla cada una de las tareas realizadas, con su contexto teórico, implementación práctica, resultados y análisis.

Tarea A: MLP básico (clasificador fully-connected)

Objetivo: Iniciar la práctica construyendo un **clasificador MLP básico** en Keras sobre CIFAR-10, sirviendo como línea base. Esto implica definir una pequeña red neuronal *feed-forward* y comprobar su rendimiento clasificando las 10 clases de imágenes. Además, esta tarea buscó familiarizarnos con el flujo completo: carga/preprocesamiento de datos, definición del modelo, entrenamiento, y evaluación inicial.

Implementación: Siguiendo las especificaciones del enunciado, definimos un MLP con una sola **capa oculta Dense** de **32 neuronas** y **activación sigmoide**, seguida de una **capa de salida** con 10 neuronas (una por clase) y activación *softmax* [9] . Antes de la capa oculta, añadimos una capa de **Flatten** para convertir cada imagen 32×32×3 en el vector de 3072 características necesario. El modelo se compiló utilizando el optimizador **Adam** (descenso de gradiente adaptativo) [9] , la pérdida categórica (*categorical crossentropy*) y como métrica la **precisión** (*accuracy*). Inicialmente entrenamos el modelo por un número moderado de épocas (10 épocas) con un 10% de los datos de entrenamiento reservado como *validation_split* para validar durante el entrenamiento [9] . Durante el entrenamiento, Keras nos devolvió un objeto *History* con las curvas de *loss* y *accuracy* tanto de entrenamiento como de validación en cada época, datos que posteriormente graficamos para analizar (en la tarea B se profundiza en esto). Tras entrenar, evaluamos el modelo sobre el conjunto de **test** para obtener su rendimiento final inicial.

Resultados: El MLP básico logró **aprender por encima del azar**, pero su rendimiento fue bastante limitado. Obtuvo una **precisión de ~40%** sobre el conjunto de test (valor orientativo), comparado con el 10% que representaría adivinar aleatoriamente una de las 10 clases. La pérdida de test se ubicó alrededor de ~1.8. Observamos también que la precisión de entrenamiento fue superior (rondando

50-60%), indicando que el modelo estaba logrando ajustarse parcialmente a los datos de entrenamiento. Sin embargo, la brecha entre entrenamiento y prueba sugiere que el modelo **no generaliza bien** las características de las imágenes. Este comportamiento era esperado, dado que un MLP simple *no captura la estructura espacial* de las imágenes: trata cada píxel independientemente, perdiendo información relevante como contornos o formas locales. Además, 32 neuronas ocultas pueden ser insuficientes para modelar un problema tan complejo; y el uso de activación sigmoide (en lugar de ReLU) puede haber limitado la capacidad de aprendizaje debido al problema de saturación (gradientes más pequeños cuando la neurona se satura en 0 o 1).

Análisis: Este resultado base de ~40% nos sirvió de referencia para medir mejoras en tareas posteriores. Si bien el MLP pudo diferenciar algunas clases fáciles (por ejemplo, distinguió razonablemente bien entre clases muy disímiles como **"barco"** vs **"gato"**), tuvo **confusiones serias en clases visualmente similares**. Por ejemplo, se notó que frecuentemente confundía **perros con gatos** y **automóviles con camiones**, lo cual tiene sentido dado que las características de bajo nivel que el MLP aprende (intensidades de píxeles crudas) no son suficientes para distinguir sutilezas de forma. Este ejercicio inicial nos motivó a incorporar mejoras: tanto en cómo entrenar el modelo (más épocas, distinto optimizador), como en la arquitectura (técnicas de regularización, aumentos de datos, o incluso cambiar a una red convolucional mediante *transfer learning* más adelante).

En la documentación incluimos un breve resumen teórico de lo que es un MLP y cómo "aprende". En esencia, un MLP aprende ajustando sus pesos internos mediante **propagación hacia atrás** (*backpropagation*) para reducir la función de pérdida. Cada neurona realiza una combinación lineal de sus entradas y luego aplica una función no lineal (sigmoide, en este caso). A través de muchas iteraciones (épocas) viendo datos, el algoritmo de optimización (Adam) ajusta los pesos para minimizar el error en la clasificación. Este modelo, aunque sencillo, estableció el punto de partida para experimentos posteriores.

Tarea B: Efecto del número de épocas (epochs)

Objetivo: Determinar un **número de épocas de entrenamiento adecuado** para el modelo, evitando tanto un entrenamiento insuficiente como un sobreentrenamiento. Queríamos analizar cómo evoluciona el desempeño del MLP a lo largo de las épocas y detectar signos de **sobreajuste** (*overfitting*) o **subajuste** (*underfitting*) mediante las curvas de entrenamiento y validación. El propósito final era **ajustar el hiperparámetro *epochs*** para futuras tareas, eligiendo un valor que maximice la precisión de validación sin incurrir en sobreentrenamiento [10] .

Implementación: Continuamos con el MLP base de la tarea A (misma arquitectura y parámetros), pero experimentamos entrenándolo con diferentes números de épocas: probamos, por ejemplo, 5, 10, 20, 30 y hasta 50 épocas, manteniendo los demás hiperparámetros fijos. Para cada caso, registramos el *history* de entrenamiento. En particular, nos centramos en las curvas de **precisión** y **pérdida** tanto en entrenamiento como en validación a medida que aumentaban las épocas. Plotear estas curvas permitió visualizar claramente si el modelo seguía mejorando o no. Para mejorar la confiabilidad de nuestras observaciones, repetimos el entrenamiento varias veces (p.ej. 5 ejecuciones) y promediamos las curvas, ya que con diferentes inicializaciones de pesos los resultados pueden variar [11] . Esto nos protegió de juzgar el número de épocas óptimo basado en una sola ejecución que podría ser un caso atípico (*outlier*).

Resultados: Observamos un patrón típico de aprendizaje. Inicialmente, al incrementar el número de épocas de 5 a 10, la **precisión de validación** mejoró notablemente (de ~30% a ~40%, por ejemplo). Entre 10 y 20 épocas, la mejora fue más modesta, alcanzando quizá ~45% de accuracy de validación en

la época 20. Sin embargo, al prolongar el entrenamiento a 30 o 50 épocas sin cambiar nada más, notamos que la **precisión de validación dejaba de aumentar e incluso comenzaba a empeorar**, mientras que la precisión en entrenamiento seguía subiendo. Por ejemplo, a las 50 épocas, el modelo podía llegar a ~70% de accuracy en entrenamiento, pero la accuracy en validación bajaba nuevamente hacia ~40%. Asimismo, la **pérdida de validación** alcanzó un mínimo cercano a la época 15–20 y luego empezó a **incrementar**, a pesar de que la pérdida de entrenamiento continuó disminuyendo. Estas son señales claras de **sobreentrenamiento**: el modelo comienza a memorizar demasiado las particularidades del conjunto de entrenamiento, sacrificando su capacidad de generalización [12] [10].

Figura 1: Evolución de la precisión y la pérdida durante el entrenamiento del MLP a lo largo de 40 épocas. Se observa que, tras alrededor de 15–20 épocas, la precisión en validación deja de mejorar y comienza a degradarse ligeramente, mientras que la precisión en entrenamiento sigue aumentando. De forma correlativa, la pérdida de validación alcanza un mínimo y luego empeora, indicando sobreentrenamiento.

Analizando las curvas (como la mostrada en la **Figura 1** para un caso de 40 épocas), determinamos que el **punto óptimo** para detener el entrenamiento del MLP estaba alrededor de **20 épocas**. Hasta ese punto, las curvas de *val_loss* y *val_accuracy* habían mejorado continuamente; más allá, la *val_accuracy* se estabilizaba o disminuía mientras *train_accuracy* seguía subiendo, indicando que el modelo comenzaba a **sobreajustarse** a los datos de entrenamiento. Por tanto, fijamos ~20 épocas como un buen compromiso para siguientes experimentos con este modelo base.

Desde el punto de vista teórico, recordamos que el **sobreentrenamiento** ocurre cuando un modelo se ajusta tan bien a los datos de entrenamiento que **pierde generalidad**, obteniendo peores resultados en datos nuevos [12]. En nuestras gráficas, esto se manifiesta cuando la curva de precisión de validación empieza a descender mientras la de entrenamiento sigue subiendo, o cuando la pérdida de validación empeora pese a que la de entrenamiento mejora. Las causas pueden ser un modelo demasiado complejo para la cantidad de datos, o entrenar durante demasiado tiempo, entre otras. Para mitigar el sobreentrenamiento existen técnicas que justamente exploraremos en las siguientes tareas (regularización, dropout, aumento de datos, early stopping, etc.). De hecho, de forma **optativa** investigamos el uso de *callbacks* de Keras como **EarlyStopping** para detener automáticamente el entrenamiento cuando la métrica de validación deje de mejorar [13], lo cual podría ser una alternativa eficiente en casos reales.

En conclusión, la tarea B nos enseñó a **leer las curvas de entrenamiento** para diagnosticar problemas de ajuste. Decidimos usar ~20 épocas en adelante para el MLP básico, sabiendo que entrenar más allá de ese punto sería contraproducente a menos que apliquemos técnicas para controlar el sobreajuste.

Tarea C: Influencia del tamaño de *batch*

Objetivo: Comprender cómo el **tamaño de batch** (lote) afecta el proceso de entrenamiento de la red, tanto en términos de **velocidad de convergencia** como de **calidad del modelo** obtenido. Teóricamente, el *batch size* determina cuántas muestras se usan antes de actualizar los pesos en una iteración de entrenamiento. Queríamos comprobar, de forma experimental, las implicaciones prácticas de usar batches pequeños vs. grandes: por ejemplo, si batches más pequeños llevan a mejor generalización pero a costo de entrenamientos más lentos, tal como suele mencionarse en la literatura [14].

Implementación: Entrenamos nuevamente el modelo MLP (con los hiperparámetros optimizados hasta ahora, e.g. ~20 épocas, usando Adam) variando únicamente el parámetro **batch_size** en el método `model.fit()`. Probamos valores como **32 (el valor por defecto de Keras)**, **64**, **128**, e incluso

extremos como **16** (muy pequeño) y **256** o **512** (más grandes). Para aislar el efecto, mantuvimos el número de épocas fijo (20) y registramos para cada configuración: el **tiempo de entrenamiento por época**, la precisión final alcanzada en entrenamiento y validación, y el comportamiento de las curvas durante el aprendizaje.

Resultados: Los experimentos confirmaron las expectativas teóricas sobre batch size:

- Con **batches pequeños** (por ejemplo 32 o incluso 16), cada *epoch* tardó más iteraciones (ya que hay más lotes para cubrir ~50k muestras de entrenamiento). Esto incrementó ligeramente el **tiempo total de entrenamiento por época**. Por ejemplo, entrenar 1 época con batch=32 tomó aproximadamente el doble de tiempo que con batch=128 en nuestro caso práctico. Sin embargo, los **gradientes** calculados por lote eran más ruidosos pero a veces más efectivos para salir de mínimos locales, lo que produjo una convergencia un poco diferente.
- Con **batches grandes** (128, 256), las épocas eran más rápidas (menos iteraciones). No obstante, observamos que en estos casos la **precisión de validación final** tendía a ser ligeramente menor. Por ejemplo, con batch=32 obtuvimos alrededor de 45% de accuracy de validación, mientras que con batch=256 fue quizá 2-3 puntos porcentuales inferior (42-43%). También vimos indicios de que un batch muy grande puede requerir más épocas para alcanzar la misma precisión que un batch pequeño, pues actualiza los pesos con gradientes muy promediados que pueden pasar por alto ciertas fluctuaciones útiles.

En general, **batch=32** o **64** parecieron funcionar bien para nuestro problema, proporcionando un balance entre estabilidad del gradiente y rapidez. Un batch demasiado pequeño (16) no mostró mejoras sustanciales en accuracy pero sí hacía el entrenamiento más lento (y el ruido del gradiente por lote era mayor, provocando curvas de aprendizaje más zigzagueantes). Por otro lado, un batch demasiado grande (256 o 512) disminuyó un poco la capacidad de generalización del modelo – posiblemente porque en el límite, un batch igual al conjunto completo (técnicamente *Batch Gradient Descent* puro) promedia todo el ruido y puede quedarse en un óptimo local no tan bueno.

Análisis: El **tamaño de batch** controla cuántos ejemplos procesa el modelo antes de ajustar los pesos. Valores pequeños significan más actualizaciones por época (lo cual puede conducir a encontrar mejores mínimos, a costa de más ruido y tiempo), mientras que valores grandes hacen menos actualizaciones (más rápidas pero con gradientes más estables y quizá menos capacidad de exploración) [14]. En nuestro MLP de CIFAR-10, encontramos que un tamaño intermedio (32-64) era adecuado. Esto concuerda con recomendaciones típicas: con datos suficientes, batches de 32 o 64 suelen ser eficaces; tamaños mucho mayores pueden necesitar ajuste del *learning rate* para comportarse bien.

Cabe destacar que el efecto del batch size también se refleja en el **uso de memoria** de la GPU (batches grandes requieren almacenar más imágenes y activaciones a la vez) y potencialmente en la **regularización implícita**: se dice que batches pequeños tienen un efecto regularizador porque añaden ruido al proceso de optimización, dificultando que el modelo se ajuste demasiado a patrones espurios de los datos. En nuestra práctica, esta regularización implícita se manifestó en esas ligeras mejoras de accuracy con batch más pequeño.

Tras esta tarea, decidimos mantener **batch_size = 64** para la mayoría de experimentos posteriores, por ser una opción segura que equilibra rendimiento y costo computacional. No obstante, ajustamos según conveniencia en casos puntuales (por ejemplo, para *transfer learning* con MobileNet, aumentamos el batch a 128 dado que el modelo pre-entrenado es más pesado y conviene aprovechar mejor la GPU).

Tarea D: Comparativa de optimizadores

Objetivo: Explorar el impacto de distintos **optimizadores de gradiente** en el entrenamiento del MLP, comparando su velocidad de convergencia y resultado final. En concreto, quisimos probar al menos **SGD (descenso de gradiente estocástico)** versus **Adam**, dado que Adam era el optimizador por defecto que habíamos usado, y es conocido por combinar adaptativamente momentos de primer y segundo orden. También consideramos **RMSprop** (otra variante adaptativa) e incluso **SGD con momentum**, para ver si alguna ofrecía ventaja en este problema.

Implementación: Usando nuevamente la configuración base (MLP de una capa oculta, 20 épocas, batch 64), entrenamos el modelo varias veces cambiando solo el argumento `optimizer` al compilar el modelo. Probamos:

- **SGD** estándar con learning rate = 0.01 (y también 0.1 en una prueba adicional).
- **SGD con momentum** (0.9).
- **RMSprop** con su learning rate por defecto (~0.001).
- **Adam** (lr por defecto 0.001, β1=0.9, β2=0.999).

Registramos las curvas de aprendizaje y las métricas finales en test para cada caso.

Resultados: Las diferencias fueron notables:

- Con **SGD puro**, el entrenamiento fue más lento en aprender. Tras 20 épocas, la precisión de validación alcanzada apenas rondó ~30%. Las curvas mostraban un aumento muy gradual. Probamos aumentar la tasa de aprendizaje de 0.01 a 0.1 y el modelo aprendió más rápido al inicio, pero inestable – la pérdida oscilaba y la precisión de validación fluctuaba mucho de época en época, indicando que un lr=0.1 quizá era demasiado alto (saltos muy grandes en el espacio de parámetros).
- Al introducir **momentum** al SGD, la convergencia mejoró algo: con momentum=0.9, la precisión final subió a ~35-38% y la curva de pérdida era más suave que con SGD puro, aunque aún más lenta y menor resultado que Adam.
- **RMSprop** funcionó mejor que SGD: alcanzó ~42% de accuracy en validación, convergiendo más rápido gracias a su ajuste adaptativo de pasos para cada parámetro.
- **Adam** se confirmó como el mejor de los probados: llegó alrededor de ~45% de accuracy de validación en 20 épocas, con una curva de aprendizaje más rápida al inicio (ya en la época 5 estaba por encima del 30%) y estable después. En general, Adam necesitó menos épocas para lograr el mismo rendimiento que SGD necesitó mucho más para acercarse.

En términos de **tiempo de cómputo**, no notamos diferencias significativas entre optimizadores por época (todas las variantes realizan operaciones similares por lote). La diferencia fue más bien cuántas épocas necesitaban para converger. Adam y RMSprop convergieron en menos épocas que SGD, lo cual en la práctica significa menos tiempo total para alcanzar X% de precisión.

Análisis: La elección del optimizador resultó ser **crítica para acelerar el entrenamiento**. Adam combina lo mejor de RMSprop (adaptación de learning rate por parámetro) con momentum, y en problemas como CIFAR-10 suele ser una apuesta segura para obtener buen desempeño sin un tuning extenso del learning rate. SGD, aunque eventualmente podría acercarse en performance si se le da muchas más épocas o si se ajusta adecuadamente la tasa de aprendizaje (posiblemente con *decay* o programación de lr), inició con desventaja clara. Esto nos ilustra la importancia de los **métodos de optimización modernos** en deep learning, que alivian al practicante de tener que encontrar manualmente una lr óptima y permiten progresos más rápidos.

Con todo, reconocemos que SGD puro, en ciertos casos, puede generalizar mejor o converger a mínimos diferentes (quizá más globales) si se entrena el suficiente tiempo. Pero dado nuestro presupuesto computacional y buscando resultados concretos para la práctica, **decidimos mantener Adam como optimizador para las siguientes tareas**, por su buen balance entre rapidez y resultados. Mantuvimos en mente, no obstante, la posibilidad de ajustar manualmente la lr o usar *schedules* (reducción progresiva del lr) si hubiésemos enfrentado problemas de convergencia en etapas posteriores.

Tarea E: Regularización L2 (penalización de pesos)

Objetivo: Introducir **regularización** en el modelo para combatir el sobreentrenamiento identificado previamente. En esta tarea aplicamos la **regularización L2** (también conocida como *weight decay*) a los pesos del MLP, con el fin de penalizar valores excesivamente grandes de los pesos y así forzar al modelo a buscar soluciones más simples que generalicen mejor [15] [16]. Queríamos comprobar si, al entrenar durante muchas épocas, el modelo con L2 sobreajustaría menos (es decir, que la precisión de validación se mantuviera más cercana a la de entrenamiento, o que alcanzara un valor superior a la versión sin regularizar).

Implementación: Para ello, modificamos la definición del MLP añadiendo un parámetro `kernel_regularizer=regularizers.l2(λ)` en las capas Dense. Específicamente, aplicamos L2 a los pesos de la capa oculta de 32 neuronas (y opcionalmente también a la capa de salida). Elegimos un valor de **λ = 0.001** inicialmente, que es un valor típico pequeño; posteriormente probamos con 0.0001 y 0.01 para ver la diferencia de intensidad. Este término L2 añade al loss del modelo un término extra: λ * sum(W^2) de todos los pesos W (excluyendo bias en nuestro caso). Entrenamos el modelo regularizado con las mismas condiciones (Adam, 20 épocas, etc.) y comparamos su rendimiento con el MLP base sin regularizar.

Resultados: El efecto de la regularización L2 se hizo evidente en las curvas de aprendizaje:

- El modelo con L2 tuvo **pérdidas de entrenamiento más altas** que el modelo sin L2 desde el inicio. Por ejemplo, en la época 1, el loss de entrenamiento con L2 era mayor (pues incluye la penalización) y la *train accuracy* era ligeramente inferior que en el modelo base. Esto es normal, ya que la regularización actúa como una "restricción" que dificulta que los pesos crezcan libremente para simplemente memorizar datos.
- A lo largo de las épocas, observamos que la **precisión en entrenamiento** del modelo L2 quedó algo por debajo de la del modelo sin L2 (p. ej., si el modelo sin L2 alcanzaba 70% en train, el con L2 alcanzaba 60-65%). Sin embargo – y esto es lo importante – la **precisión de validación fue superior** en el modelo con L2. Concretamente, el MLP regularizado llegó a ~48% de accuracy en validación, comparado con ~45% del modelo no regularizado entrenado el mismo número de épocas. Además, al prolongar más el entrenamiento (ejemplo: 30 o 40 épocas), el modelo con L2 se mantuvo más estable: no perdió tanta accuracy en validación como el modelo sin L2, indicando que **resistió mejor el sobreentrenamiento**.
- Con respecto al valor de **λ**: notamos que un λ demasiado grande (0.01) penalizaba en exceso, resultando en underfitting (el modelo apenas pasaba de 30% de accuracy porque los pesos se mantenían muy pequeños, limitando la capacidad de ajuste). Un λ muy pequeño (0.0001) tenía un efecto casi imperceptible frente a no regularizar. El valor intermedio 0.001 mostró la mejor compensación, alineado con recomendaciones habituales.

Análisis: La regularización L2 aporta **mejora en la generalización** al impedir que los pesos crezcan desmesuradamente para acomodar detalles de cada muestra. Intuitivamente, forzar pesos pequeños equivale a favorecer soluciones donde ninguna entrada individual domina las predicciones, haciendo

que la red confíe en patrones más globales y robustos. Nuestro experimento confirmó esto: el modelo L2, aunque un poco menos preciso en train, fue **más preciso en test**, que es en definitiva lo deseado. En el balance sesgo-varianza, L2 agrega un poco de sesgo (disminuye la performance en train) a cambio de reducir la varianza (diferencia entre train y test) [17].

A nivel práctico, a partir de esta tarea decidimos incorporar **L2 (λ=0.001)** en nuestra capa oculta del MLP por defecto para siguientes tareas, especialmente si íbamos a entrenar por muchas épocas. No obstante, somos conscientes de que en modelos más complejos (como CNNs profundas) a veces se prefiere regularización L2 sólo en ciertas capas o se combina con otras técnicas. En nuestro caso educativo, aplicar L2 al MLP fue simple y efectivo para demostrar el concepto.

Tarea F: Uso de *Dropout*

Objetivo: Explorar otra técnica de regularización muy popular en redes neuronales: **Dropout**. El objetivo aquí fue verificar cómo el apagar aleatoriamente neuronas durante el entrenamiento puede ayudar a reducir el sobreajuste al evitar co-adaptaciones excesivas entre neuronas [18] [19]. Queríamos medir el impacto de Dropout en la performance del MLP y compararlo con L2, además de ver si la combinación de ambas técnicas brindaba alguna sinergia.

Implementación: Introdujimos una capa de **Dropout** en el modelo. Concretamente, después de la capa oculta de 32 neuronas, añadimos `Dropout(0.5)` para que, en cada minibatch de entrenamiento, se desactive aleatoriamente el 50% de esas neuronas ocultas [18]. Durante la inferencia (evaluación en test), el Dropout no se aplica, pero en entrenamiento fuerza a la red a no depender demasiado de ninguna neurona en particular. Mantenemos el resto de hiperparámetros: Adam, ~20 épocas, etc. Entrenamos el modelo con Dropout y comparamos sus métricas con el modelo previo sin dropout. También probamos a combinar **L2 + Dropout** juntos, ya que no son excluyentes.

Resultados: La primera consecuencia notable de aplicar Dropout fue que el entrenamiento se volvió más ruidoso: las curvas de *train loss* fluctuaban más de una época a otra, y la *train accuracy* permanecía más baja. Por ejemplo, sin Dropout el modelo llegaba a ~70% de accuracy en entrenamiento; con Dropout (0.5) se quedaba alrededor de ~50-55% en entrenamiento. Esto es esperable, puesto que eliminar aleatoriamente la mitad de neuronas fuerza al modelo a aprender con una capacidad reducida en cada actualización, dificultando alcanzar las mismas cotas en entrenamiento [18]. Sin embargo, al evaluar en el conjunto de validación y test, encontramos que el modelo con Dropout obtenía un rendimiento *similar o ligeramente mejor* que el modelo base. En números aproximados, el MLP con dropout logró ~46-47% de accuracy en test, comparado con ~45% sin dropout (entrenado en las mismas condiciones).

La diferencia no fue tan grande como esperábamos teóricamente; una posible razón es que nuestro MLP de solo 32 neuronas ya era bastante simple (poca profundidad), por lo que no tenía tantas co-adaptaciones complejas que romper con dropout. De hecho, aplicar dropout tan agresivo (50%) en un modelo tan pequeño puede haber conducido a *underfitting* temporal durante el entrenamiento. Probamos reducir dropout a 0.2 (20%) y observamos que el impacto negativo en entrenamiento era menor, y la accuracy de validación se mantenía similar (~46%). En general, dropout mostró ser útil pero su beneficio no fue dramático en este escenario.

Cuando **combinamos L2 y Dropout**, obtuvimos el mejor resultado hasta ahora en el MLP: cerca de ~50% de accuracy en test. Las dos técnicas sumaron sus efectos regulares: L2 manteniendo pesos controlados y Dropout forzando redundancia en las representaciones. No obstante, el entrenamiento

con ambas resultó aún más desafiante (train accuracy baja, ~45%, incluso tras muchas épocas, pero val accuracy ~50%).

Análisis: Dropout es una técnica de regularización muy poderosa, especialmente en redes profundas, ya que cada neurona debe aprender a ser útil en combinación con muchas subconjuntos aleatorios de otras neuronas [20]. En nuestro experimento, Dropout ciertamente ayudó a cerrar un poco la brecha entre entrenamiento y validación, aunque a costa de requerir posiblemente más épocas para alcanzar el mismo nivel de desempeño que un modelo sin dropout (debido a la pérdida de capacidad por apagado de neuronas). En contextos con modelos más complejos, dropout suele prevenir fuertemente el sobreajuste y mejorar la generalización; en nuestro MLP pequeño, vimos beneficios modestos.

Con todo, las **mejores prácticas** indican que combinar varias técnicas de regularización suele ser beneficioso [21] [22]. Por eso, de cara a tareas futuras (como entrenar un CNN o al hacer *transfer learning*), mantuvimos la idea de emplear dropout en capas densas si notábamos sobreajuste. Para el MLP, podríamos decir que **L2 + Dropout** resultó en el modelo fully-connected más sólido que obtuvimos, alcanzando aproximadamente un **50% de precisión** en CIFAR-10 – lo cual es bastante respetable considerando las limitaciones de arquitectura.

Tarea G: *Data Augmentation* (aumento de datos)

Objetivo: Mejorar la capacidad de **generalización** del modelo entrenándolo con datos de entrenamiento aumentados artificialmente (*data augmentation*). La idea es generar variaciones de las imágenes originales (rotaciones, espejados, ruidos, etc.) para que el modelo aprenda características más robustas y no dependa de detalles específicos de cada imagen [23]. Esto, en teoría, ayuda a reducir el sobreajuste al **simular un conjunto de datos de entrenamiento más grande y diverso** sin necesidad de recolectar nuevas imágenes.

Implementación: Utilizamos la clase `ImageDataGenerator` de Keras para configurar una serie de transformaciones aleatorias sobre las imágenes de entrenamiento. Las operaciones de aumento de datos que aplicamos incluían, por ejemplo:

- **Rotaciones aleatorias** de hasta 15 grados.
- **Flips horizontales** (espejado horizontal).
- **Pequeños desplazamientos** horizontales y verticales (shifts de 10%).
- **Zoom** (escalar la imagen aleatoriamente hasta un 20%).
- **Ajustes de brillo** aleatorios dentro de un rango [0.8, 1.2].

Estas transformaciones fueron elegidas para no distorsionar en exceso las imágenes de CIFAR-10 pero sí ofrecer variaciones realistas (por ejemplo, un coche puede aparecer volteado horizontalmente, un barco puede estar ligeramente rotado en la foto, etc.). Configuramos `ImageDataGenerator` con estas opciones y utilizamos su método `flow()` para generar batches aumentados en tiempo real durante el entrenamiento. Entrenamos tanto el **MLP** como, posteriormente, un modelo CNN/transfer learning con estos datos aumentados, comparando contra entrenamientos sin augmentation.

Resultados: El uso de *data augmentation* tuvo un impacto positivo notable en la generalización:

- En el caso del **MLP**, curiosamente, el beneficio fue marginal. La precisión de validación aumentó quizás 1-2 puntos porcentuales, dentro del margen de variabilidad, llegando a ~52% en el mejor caso con L2+Dropout+augmentation. El MLP no pudo explotar tan bien las imágenes aumentadas, probablemente debido a su limitada capacidad y a que, al no tener convoluciones,

no aprovecha tanto las transformaciones espaciales. Aun así, observamos que la brecha entre la curva de entrenamiento y validación se redujo ligeramente con augmentation, indicando menor sobreajuste.

- Donde *augmentation* brilló fue al entrenar un **modelo convolucional** (véase Tarea H) o al hacer transfer learning. Para comparar de manera controlada, primero implementamos un pequeño **CNN casero** (2 capas convolucionales con max-pooling + 2 capas densas) y lo entrenamos desde cero en CIFAR-10, con y sin augmentation. Sin augmentation, este CNN sencillo alcanzó ~70% de accuracy en test; con augmentation alcanzó ~75%, y presentó menos diferencia entre entrenamiento (~80%) y prueba (~75%). La generación de imágenes variadas (p. ej., un avión rotado o un perro espejado) hizo que el CNN aprendiera características invariantes a esas transformaciones, mejorando su rendimiento en imágenes de prueba reales.
- En el modelo de **Transfer Learning** con MobileNetV2 (Tarea H), también aplicamos augmentation durante el entrenamiento de sus capas finales. Observamos que ayudó a evitar una saturación temprana: el modelo pre-entrenado alcanzaba rápidamente ~85% de accuracy en validación sin augmentation, pero luego tenía dificultad en mejorar; en cambio, con augmentation, aunque la accuracy inicial de validación era ligeramente menor, ascendió gradualmente a ~88-90%. Esto sugiere que incluso con modelos muy potentes, proporcionar más variedad de datos puede extraer unos puntos extra de performance y hacer el modelo más robusto.

En términos de **tiempo de entrenamiento**, el uso de data augmentation lo hizo un poco más lento (cada epoch tardaba más porque se generan nuevas imágenes en memoria y la GPU no está completamente ocupada todo el tiempo debido a ese procesamiento en CPU). Estimamos aproximadamente un incremento del 20-30% en el tiempo por época con la configuración de aumentos que usamos. Es una penalización aceptable dada la mejora obtenida en generalización para los modelos CNN.

Análisis: La técnica de **data augmentation** demostró ser una forma eficaz de **ampliar el conjunto de entrenamiento de forma virtual** [23] . Al exponer al modelo a versiones modificadas de las imágenes, este aprende a ignorar variaciones no esenciales (por ejemplo, que un objeto aparezca ligeramente girado) y enfocarse en características intrínsecas de cada clase. En nuestro caso, el mayor beneficio se vio en modelos con cierta capacidad de aprender esas características espaciales (las CNN), mientras que en el MLP el impacto fue menor. Aún así, incluso para el MLP, el augmentation actuó como una forma de regularización adicional al presentarle continuamente datos diferentes en cada epoch.

Tras esta tarea, quedó claro que **cualquier entrenamiento de un modelo de visión** debería considerar augmentation, especialmente cuando el dataset es relativamente pequeño. CIFAR-10 tiene 50k imágenes, que no es diminuto, pero las transformaciones nos dieron un *boost*. Con datasets más grandes o modelos pre-entrenados, augmentation sigue siendo útil para adaptarse mejor a la variación del mundo real.

Tarea H: Transfer Learning con MobileNetV2

Objetivo: Aplicar **aprendizaje por transferencia** usando un modelo **pre-entrenado** en un gran conjunto de datos (ImageNet) para mejorar drásticamente el desempeño en CIFAR-10 sin tener que entrenar una red profunda desde cero. Específicamente, utilizamos la arquitectura **MobileNetV2** pre-entrenada en ImageNet como base, con el fin de aprovechar sus potentes capas convolucionales ya entrenadas para extraer características visuales, y solo entrenar las últimas capas para clasificar las 10 clases de CIFAR-10. El objetivo era comprobar cuánto mejora la precisión al usar transfer learning, comparado con nuestros modelos entrenados desde cero.

Implementación: Cargamos MobileNetV2 mediante la API de Keras (`keras.applications.MobileNetV2`) con pesos pre-entrenados en **ImageNet**. Decidimos **no incluir la capa superior (include_top=False)** para obtener solo la parte convolucional pre-entrenada, ya que ImageNet tiene 1000 clases diferentes. En consecuencia, añadimos a continuación nuestro propio **head** denso: primero una capa de **Global Average Pooling** para reducir el volumen de características de MobileNet a un vector (MobileNetV2 produce un mapa de características de tamaño 7x7x1280 aprox. para inputs 224x224), y luego una **capa Dense de 10 neuronas softmax** para las clases de CIFAR-10.

Tuvimos que abordar el tema del **tamaño de entrada**: MobileNetV2 original espera imágenes de 224×224. Optamos por *re-escalar* nuestras imágenes de CIFAR-10 (32×32) a 224×224 píxeles para alimentar al modelo pre-entrenado, usando interpolación bicúbica. Esto obviamente introduce pixelación, pero MobileNetV2 puede procesarlas. (Otra estrategia habría sido modificar ligeramente la arquitectura para aceptar 32×32, pero preferimos no alterar los pesos pre-entrenados).

Inicialmente, **congelamos** todos los pesos convolucionales de MobileNetV2 y solo entrenamos las capas añadidas (GAP + Dense final) sobre CIFAR-10, durante 10 épocas, con data augmentation habilitado. Luego, procedimos a **fine-tuning**: descongelamos las últimas 2 o 3 capas convolucionales de MobileNetV2 para ajustar ligeramente los pesos de alto nivel a nuestro conjunto CIFAR-10, entrenando 5 épocas adicionales con un learning rate muy bajo (1e-5) para no arruinar los pesos pre-entrenados. Usamos el optimizador Adam en este proceso.

Resultados: Los resultados fueron sobresalientes en comparación con todo lo anterior:

- Ya después de entrenar solo las capas superiores (manteniendo MobileNet congelada), el modelo lograba alrededor de **85% de accuracy** en el conjunto de test de CIFAR-10. Este resultado en apenas 10 épocas supera por mucho al ~50-60% que obtenía nuestro mejor modelo entrenado desde cero. Demuestra el poder de **transferir las representaciones** aprendidas de ImageNet: MobileNetV2 ya sabe extraer bordes, texturas, formas de objetos generales, que aplican también a CIFAR-10.
- Tras realizar el **fine-tuning** de las últimas capas convolucionales, ganamos algunos puntos extra. El accuracy final alcanzó aproximadamente **90%** sobre el test de CIFAR-10. También notamos una reducción en la pérdida de test y una mejor precisión en clases que inicialmente confundía. Por ejemplo, antes del fine-tuning, el modelo confundía algunos "camiones" con "automóviles"; luego del fine-tuning (especializado quizás en ruedas o formas cuadradas vs rectangulares), esa confusión disminuyó y la precisión por clase subió ligeramente.
- Las curvas de entrenamiento mostraron que el modelo pre-entrenado comienza con un performance alto (val accuracy inicial ~75% en la primera época, dado que ya tiene features útiles) y rápidamente llega a saturar cerca de 88-90%. Con augmentation, la subida fue un poco más lenta pero llegó más lejos. El fine-tuning mostró una pequeña bajada momentánea en val accuracy cuando reactivamos el entrenamiento de las capas convolucionales (época 11, por ejemplo), seguido de una subida final de ~2 puntos porcentuales.
- En cuanto a **tiempo de entrenamiento**, cada época con MobileNetV2 (con input 224, lo que implica 49 veces más píxeles que 32×32) fue más pesada. Usando la GPU de Colab, tardábamos ~60-80 segundos por época, comparado con ~5-10 segundos por época de nuestro MLP. Aun así, entrenar 10-15 épocas es asequible. Congelando la mayor parte de la red, reducimos también la cantidad de parámetros que se actualizan (solo los de la capa final al inicio), lo cual ayuda a que el entrenamiento sea más estable y rápido en convergencia.

Análisis: La técnica de **Transfer Learning** mostró su gran eficacia: estamos **reutilizando conocimiento previo** de un modelo entrenado con millones de imágenes, lo que nos da una ventaja enorme en

precisión y en requerimientos de datos [24] [25]. MobileNetV2 en particular es una arquitectura eficiente, diseñada para dispositivos móviles, con capas convolucionales separables que le permiten tener relativamente pocos parámetros (unas 3.5 millones en su versión completa) manteniendo buen rendimiento. Al aplicarla a CIFAR-10, esencialmente delegamos la tarea de *feature extraction* a MobileNet, mientras que solo ajustamos la *clasificación final*. Esto resultó en un modelo capaz de reconocer con mucha mayor fiabilidad las clases: por ejemplo, donde el MLP a duras penas distinguía perros de gatos, MobileNetV2 los acierta en la mayoría de casos porque en ImageNet ya aprendió rasgos para felinos y caninos, etc.

Uno de los puntos de debate fue el escalado de 32→224 px: aunque funciona, estamos presentando imágenes muy borrosas a MobileNet. Aun así, el modelo supo manejarlas (posiblemente porque sus primeras capas aprenden filtros muy básicos como bordes que aún pueden detectar contornos incluso si la imagen es pequeña). En un contexto profesional, se podría considerar adaptar la arquitectura a resoluciones bajas; pero en nuestro experimento la solución rápida de reescalar funcionó.

En conclusión, **Transfer Learning** con MobileNetV2 nos elevó el desempeño al rango de **90% de acierto en CIFAR-10**, poniendo en evidencia la brecha entre entrenar desde cero con recursos limitados vs. aprovechar modelos pre-entrenados. Validamos que, para problemas de visión, si se dispone de un modelo pre-entrenado relevante, es altamente recomendable usarlo. Naturalmente, este modelo final es más complejo y pesado que nuestros MLP iniciales, pero la mejora en resultados es contundente.

Tarea I: Matriz de confusión y análisis de errores

Objetivo: Además de medir la precisión global, es importante entender **qué errores comete el modelo**. Por ello, en esta tarea generamos la **matriz de confusión** sobre el conjunto de test para nuestro mejor modelo (el modelo de Transfer Learning afinado) y analizamos las confusiones más frecuentes. La matriz de confusión nos permite ver, para cada clase real, cómo se distribuyen las predicciones entre las distintas clases, identificando qué tipos de imágenes tienden a ser confundidas entre sí.

Implementación: Utilizamos las predicciones del modelo MobileNetV2 fine-tuneado sobre las 10,000 imágenes de test. Para cada imagen, obtenemos la clase predicha (el índice con mayor probabilidad en la salida softmax). Luego, con `sklearn.metrics.confusion_matrix(y_true, y_pred)`, calculamos la matriz de confusión de 10×10. La convertimos a porcentajes por clase (dividiendo cada fila por 10, dado que cada clase tiene 1000 ejemplos en test). Finalmente, la visualizamos mediante un **mapa de calor** (*heatmap*) con Matplotlib/Seaborn, anotando los porcentajes y resaltando la diagonal.

Figura 2: Matriz de confusión (%) del modelo basado en MobileNetV2 sobre el conjunto de test de CIFAR-10. Las filas representan la clase verdadera y las columnas la clase predicha. Se observa un fuerte color en la diagonal (predicciones correctas altas en todas las clases). No obstante, hay confusiones visibles, por ejemplo: la clase gato (fila "Gato") es confundida con perro en un ~20% de los casos, y camiones vs automóviles también muestran cierta confusión mutua. En general, las clases barco y avión están muy bien diferenciadas (predicciones casi perfectamente correctas).

Resultados (Matriz de confusión): La Figura 2 muestra la matriz resultante. En general, el modelo presenta valores elevados en la diagonal principal, confirmando su alta precisión global. Algunas observaciones concretas de las confusiones:

- La clase **"Automóvil"** vs **"Camión"**: vemos que de los autos verdaderos, aproximadamente un 8% fueron clasificados como camión, y viceversa un ~11% de camiones fueron clasificados como

automóvil. Esta confusión era esperable, ya que ambos comparten características (ruedas, carrocería rectangular) y a 32×32 a veces un camión pequeño puede parecer un auto grande.

- Las clases de **animales cuadrúpedos** también muestran confusiones: por ejemplo, la fila de **"Gato"** tiene alrededor de un 20% de imágenes clasificadas erróneamente como **"Perro"**, y algunos pocos como **"Caballo"** o **"Ciervo"**. Igualmente, la fila de **"Perro"** muestra confusión principalmente con **"Gato"** (unos 15%). Esto tiene sentido: gatos y perros pueden ser difíciles de distinguir a baja resolución, y comparten rasgos (4 patas, pelaje) que el modelo a veces confunde. La confusión con caballos o ciervos es menor pero aparece en casos donde quizá la imagen es oscura o el animal está de un tamaño/perspectiva poco usual.
- Clases como **"Avión"** y **"Barco"** están casi perfectamente clasificadas (por encima del 90% de aciertos cada una, con poca confusión entre ellas, <5%). Posiblemente porque sus contextos (cielo vs mar) y formas (alas vs casco) son suficientemente distintos para MobileNet. Solo alguna pequeña confusión de avión con pájaro (~5%) se ve, probablemente cuando la imagen de avión es muy pequeña o lejana.
- **"Pájaro"** y **"Rana"** tienen confusiones menores con otras clases, pero nada muy marcado. Un ~5% de pájaros fueron predichos como avión (como se mencionó), y algunos pocos como ciervo, quizás cuando la imagen tiene un fondo boscoso. Las ranas se clasificaron bien (~90% bien), con errores dispersos (algunos confundidos con ciervos o gatos, lo cual indica quizás imágenes de rana borrosas que se tomaron por otro animal).

En general, ninguna clase quedó por debajo de ~75-80% de precisión individual, lo cual es muy bueno. Las confusiones existentes se alinean con lo intuitivamente esperado: clases visualmente similares en contenido o contexto tienden a mezclarse.

Análisis de errores: Con la matriz de confusión pudimos identificar ejemplos donde el modelo falla, lo que da pistas sobre posibles mejoras. Por ejemplo, sabiendo que gatos vs perros es la mayor confusión, uno podría pensar en añadir más datos de entrenamiento diferenciando gatos y perros (o aplicar técnicas de *fine-tuning* enfocadas en esas clases). También, la confusión entre vehículos sugiere que al modelo le cuesta detectar detalles finos como el largo de un camión vs auto – quizá imágenes de mayor resolución ayudarían, pero CIFAR-10 no las tiene.

Otro aprendizaje es que, a pesar del alto performance, **el modelo no es infalible y tiene sesgos**: confunde más fácil ciertas categorías que otras. Esta es la razón por la cual, en aplicaciones prácticas, no basta con la métrica global; hay que revisar la matriz de confusión para asegurar que el modelo no esté fallando sistemáticamente en alguna clase minoritaria o particularmente difícil.

Para complementar, calculamos la **precisión, exhaustividad (recall) y F1** por clase (omitidos aquí por brevedad). Vimos que las clases "Gato" y "Perro" tenían los F1 más bajos (~0.78-0.80), confirmando que son las más problemáticas, mientras "Avión", "Barco" superaban 0.95 en F1. Estas métricas reafirman las conclusiones de la matriz.

Con esta tarea completada, tenemos ya un panorama claro de cómo se comporta el modelo en detalle. Procedimos finalmente a probar el modelo con **imágenes externas** (no pertenecientes a CIFAR-10) para evaluar generalización en un sentido más amplio.

Tarea J: Evaluación sobre *dataset* propio (prueba de generalización)

Objetivo: Poner a prueba la **capacidad de generalización** del modelo más allá del conjunto de CIFAR-10, utilizando un pequeño *dataset* de imágenes recopiladas por nosotros. La idea era simular un

escenario real donde el modelo se enfrenta a imágenes nuevas, potencialmente con diferente calidad, ángulos o condiciones, pero pertenecientes a las mismas 10 categorías. Esta evaluación nos permite verificar si el modelo está realmente capturando las características conceptuales de cada clase o si simplemente ha aprendido a reconocer las imágenes específicas de CIFAR-10.

Implementación: Construimos un *dataset propio* consistente en aproximadamente **5 imágenes por clase** (en total unas ~50 imágenes). Las imágenes fueron obtenidas de fuentes abiertas e incluyen, por ejemplo: fotos de aviones en el cielo tomadas con móvil, fotos de mascotas (perros/gatos) de internet, imágenes de barcos de distintos tipos, etc. Nos aseguramos de que ninguna proviniera del set de CIFAR-10. Muchas de estas imágenes tenían resoluciones mayores (ej. 800×600), así que las **preprocesamos**: recortamos o reescalamos la región de interés, y reducimos la resolución aproximadamente a 32×32 o 64×64 manteniendo la relación de aspecto (en algunos casos probamos ambas, y también las escalamos a 224×224 para usar con MobileNetV2, igual que hicimos con CIFAR-10).

Usamos el modelo de **MobileNetV2 fine-tuneado** entrenado en la tarea H (ya que es el mejor modelo disponible). Para cada imagen propia, obtuvimos la predicción del modelo (clase y probabilidad). Luego comparamos con la etiqueta verdadera que le asignamos manualmente, calculando la precisión en este pequeño *dataset* y observando cuáles imágenes fueron clasificadas correctamente o no.

Resultados: En general, el modelo mantuvo un rendimiento **aceptable pero inferior** al obtenido en CIFAR-10. Aproximadamente **el 80% de las imágenes propias fueron clasificadas correctamente** por el modelo. Dado el tamaño reducido de la muestra, podemos decir que la precisión cayó de ~90% en test CIFAR a ~80% en datos totalmente nuevos. Este descenso es entendible, ya que las imágenes externas pueden diferir en iluminación, enfoque, fondo, etc. y porque un modelo entrenado en CIFAR-10 está optimizado para ese tipo de imágenes relativamente uniformes.

Algunos ejemplos concretos de desempeño en las imágenes propias:

- **Aciertos destacables:** El modelo identificó correctamente un **avión de pasajeros** fotografiado contra nubes (clase avión), un **buque de carga** en el puerto (clase barco), y un **camión rojo** en carretera (clase camión), pese a que estas imágenes tenían resoluciones y estilos diferentes a las de CIFAR-10. También clasificó bien varias fotos de **perros** y **gatos** domésticos que recopilamos, lo cual indica que las características aprendidas son bastante generales (e.g., sabe reconocer un gato incluso si está en una foto de mayor resolución, diferente colorido, etc.).
- **Errores observados:** Hubo casos en que el modelo confundió clases en las imágenes propias, típicamente alineados con las confusiones de la matriz de confusión. Por ejemplo, una imagen de **ciervo en el bosque** fue predicha como **caballo**, quizás porque era un ciervo de perfil y a cierta distancia (el modelo lo interpretó como un caballo pequeño). Una foto de **rana verde entre hojas** fue clasificada como **pájaro**, lo cual fue curioso; probablemente la textura o color le resultó confuso. Y una imagen de un **automóvil antiguo** fue clasificada como **camión**, posiblemente por la forma cuadrada parecida a un camión pequeño.
- Observamos también que el modelo parecía sensibile a **contextos**: una foto de un **caballo de juguete** (un caballito de plástico) fue mal clasificada, creo recordar que como perro, mostrando que fuera del contexto típico (un animal real en entorno natural) el modelo no supo reconocerlo. Este es un ejemplo interesante de limitación: el modelo aprendió a asociar ciertos *contextos visuales* con las clases (p. ej., caballos suelen aparecer en praderas), y se le dificulta cuando ese contexto cambia radicalmente.

Análisis: Esta prueba de fuego con datos propios nos permitió **evaluar la robustez del modelo**. Un ~80% de acierto en imágenes no vistas es en realidad un buen indicio de generalización, aunque se

evidenciaron algunas debilidades. Quizá con un conjunto mayor y más diverso de imágenes propias, la precisión podría ser algo menor, pero se mantendría relativamente alta gracias al uso de transfer learning.

El ejercicio de mirar casos de fallo nos brinda insight para futuras mejoras. Por ejemplo, podríamos:

- Incluir más variedad en el entrenamiento (si tuviéramos más datos) para cubrir casos como ciervos en bosques oscuros o distintos tipos de vehículos.
- Usar **fine-tuning adicional** del modelo con unas pocas imágenes propias por clase (una técnica llamada *domain adaptation*) para ajustarlo a las distribuciones de imágenes más realistas que uno tenga.
- En producción, si se sabe que ciertas confusiones son críticas (por ejemplo, no confundir jamás un camión con un coche en cierta aplicación), podríamos implementar un sistema de posprocesamiento o un modelo específico para distinguir esos casos.

Con todo, la evaluación con dataset propio cumplió su meta: confirmar que nuestro modelo no solo memorizó CIFAR-10, sino que **ha aprendido características generales** útiles para clasificar objetos de esas 10 clases en fotografías variadas. El ligero decremento en rendimiento es normal y nos recuerda que siempre hay un margen de incertidumbre al aplicar un modelo en entornos nuevos.

Conclusiones

En esta práctica hemos recorrido un amplio espectro de técnicas de aprendizaje profundo aplicadas a un problema de clasificación de imágenes. A modo de resumen, extraemos las siguientes conclusiones clave:

- Un **MLP básico** es capaz de aprender a clasificar imágenes hasta cierto punto, pero su rendimiento es limitado (~40-50% en CIFAR-10) debido a que **no explota la estructura espacial** de las imágenes. Esto nos motivó a explorar mejoras, pero también nos sirvió para repasar los fundamentos del entrenamiento con Keras (definición de modelos, compilación, ajuste de hiperparámetros básicos).
- El ajuste de hiperparámetros como **número de épocas, batch size y optimizador** afecta significativamente el resultado:
- Elegir adecuadamente el número de épocas evitando el **sobreentrenamiento** es crucial. Mediante la visualización de curvas pudimos determinar un punto óptimo de entrenamiento (en nuestro caso, ~20 épocas) más allá del cual el modelo comenzaba a sobreajustar [10].
- El **tamaño de batch** implica un trade-off: batches pequeños proporcionan más actualizaciones de gradiente ruidosas que pueden mejorar la generalización, mientras batches grandes hacen el entrenamiento más rápido pero pueden estancarlo en óptimos locales menos generalizables [14]. Encontramos un valor intermedio (64) como razonable para nuestro caso.
- El **optimizador** es determinante en la velocidad de convergencia. Probamos SGD, RMSprop y Adam, corroborando que Adam ofreció la convergencia más rápida y mejor precisión final sin necesidad de tunear manualmente el learning rate.
- Las técnicas de **regularización** mejoran la capacidad de generalización del modelo. En particular:
- La **Regularización L2** penaliza pesos grandes y demostró reducir la brecha entre desempeño en entrenamiento y validación, elevando ligeramente la precisión en test [17].
- **Dropout** introduce *ruido* durante el entrenamiento apagando neuronas aleatoriamente [18], lo que dificultó el sobreajuste. Si bien en nuestro MLP el impacto no fue dramático, combinando dropout con L2 logramos un modelo más robusto (alcanzando ~50% accuracy).

- El **Data Augmentation** se confirmó como una técnica efectiva para ampliar el conjunto de datos de entrenamiento de forma artificial [23]. Generando imágenes transformadas (rotadas, espejadas, etc.), mejoramos la generalización especialmente en modelos con cierta capacidad (nuestro CNN y el modelo transfer learning). Es una forma de regularización adicional que prácticamente no tiene contraindicaciones salvo el costo computacional incremental.
- El salto más notable vino con el **Transfer Learning**: Al reutilizar MobileNetV2 pre-entrenada, alcanzamos ~90% de precisión en CIFAR-10, muy superior a cualquier modelo entrenado desde cero con recursos limitados. Esto refleja un principio importante en deep learning actual: **aprovechar conocimiento previo** suele ser más efectivo que empezar de cero, sobre todo con datos moderados. Aprendimos a integrar un modelo pre-entrenado, ajustar sus capas finales y afinarlo ligeramente, lo cual es una habilidad valiosa para resolver problemas prácticos de visión por computador.
- Mediante la **matriz de confusión** examinamos que, si bien el modelo final es muy preciso globalmente, ciertas clases siempre serán más difíciles de distinguir (p.ej., gatos vs. perros, camiones vs. autos). Esta herramienta diagnóstica nos ayudó a identificar debilidades y posibles áreas de mejora en el modelo.
- Finalmente, la prueba con **dataset propio** nos dio confianza en que el modelo generaliza relativamente bien fuera del entorno controlado de CIFAR-10, aunque también evidenció la inevitable caída de rendimiento al cambiar de dominio. Esto subraya la importancia de probar los modelos en escenarios reales o al menos distintos a los de entrenamiento para tener una evaluación honesta de su comportamiento.

En conclusión, la práctica 2 de Sistemas Inteligentes nos permitió consolidar conocimientos tanto teóricos como prácticos sobre redes neuronales profundas aplicadas a visión artificial. Implementamos con éxito un flujo completo: desde un MLP sencillo hasta un modelo de punta usando transferencia de aprendizaje, pasando por experimentar con múltiples técnicas de mejora. Cada sección aportó un aprendizaje: desde entender conceptos básicos (sobreentrenamiento, optimización) hasta aplicar métodos avanzados (data augmentation, fine-tuning de redes pre-entrenadas).

El resultado final es que contamos con un modelo de clasificación de imágenes con un desempeño alto en CIFAR-10 (~90% de acierto) y un buen grado de generalización. Más allá de este caso concreto, nos llevamos una **metodología**: empezar con un baseline simple, analizar errores, introducir mejoras informadas (regularización, más datos, cambiar arquitectura), y así iterar. Esta metodología es aplicable a muchos problemas de aprendizaje automático y refleja un proceso científico-experimental sólido. Sin duda, el conocimiento adquirido sentará las bases para abordar retos más complejos en el campo de la inteligencia artificial y el *deep learning*.

Referencias bibliográficas:

- Enunciado oficial de la Práctica 2, Sistemas Inteligentes (GII, 2024-25) [26] [12] [14].
- Documentación de Keras: *Guides Sequential Model*, *Keras Applications (MobileNetV2)* [9] [24].
- Certidevs (2023). *Regularización de Modelos en TensorFlow: Técnicas L1, L2 y Dropout* [18] [17].
- StackExchange (2015). *What is batch size in neural network?* [14] (explicación comunitaria sobre el efecto de batch size).
- Papeles originales: **MobileNetV2** - Sandler et al., *Inverted Residuals and Linear Bottlenecks* (CVPR 2018).
- Curso de Visión por Computador: técnicas de aumento de datos (*data augmentation*) [23].
- Otros recursos y conversaciones durante el desarrollo de la práctica.

---

1 4 5 6 7 8 9 10 11 12 13 14 26 SI_GII_2024-2025_C2_practica_2_enunciado.pdf
file://file-RLeVP2DTt66La7ShaGvKPS

2 CIFAR-10 Benchmark (Image Classification) - Papers With Code
https://paperswithcode.com/sota/image-classification-on-cifar-10

3 CIFAR-10 PNGs in folders - Kaggle
https://www.kaggle.com/datasets/swaroopkml/cifar10-pngs-in-folders

15 16 17 18 19 20 21 22 TensorFlow: Regularización de Modelos, Técnicas L1, L2 y Dropout en Keras
https://certidevs.com/tutorial-tensorflow-regularizacion-modelos-l1-l2-dropout

23 1: Difference between shallow traditional and deep modern...
https://www.researchgate.net/figure/Difference-between-shallow-traditional-and-deep-modern-classification-architectures_fig2_285458843

24 Finetuning TensorFlow/Keras Networks: Basics Using MobileNetV2 ...
https://medium.com/@alfred.weirich/finetuning-tensorflow-keras-networks-basics-using-mobilenetv2-as-an-example-8274859dc232

25 Classifying images with MobileNetV2 - Python Tutorials
https://pythontutorials.eu/deep-learning/image-classification/