Práctica 2: Visión artificial y aprendizaje

Objetivos

- Comprender el funcionamiento general de las redes neuronales artificiales como técnicas de aprendizaje supervisado, y en concreto los perceptrones multicapa y las redes neuronales convolucionales.
- Entender el concepto de imagen y píxel, y cómo podemos utilizar las técnicas de aprendizaje supervisado para distinguir lo que representa una imagen.
- Comprender cómo funciona el algoritmo de *backpropagation* y todo lo necesario para que una red neuronal pueda aprender a partir de un conjunto de datos.
- Conocer la librería Keras para implementar redes neuronales de forma rápida y efectiva.
- Utilizar redes neuronales para resolver un problema real de clasificación.
- Entender el papel de cada uno de los principales parámetros que configuran una red neuronal, y aprender a ajustarlos de forma efectiva y eficiente, para maximizar su tasa de acierto sin desperdiciar tiempo de entrenamiento.
- Utilizar Matplotlib (o similares) para mostrar resultados de forma visual.
- Acelerar el flujo de trabajo utilizando *scripts* que generen de forma automática las pruebas para el ajuste de parámetros y los contenidos para el informe (resultados de clasificación, tablas, gráficas).

Introducción

En esta práctica trabajaremos con técnicas de aprendizaje supervisado aplicadas al reconocimiento automático de imágenes. Esto nos permitirá también trabajar con imágenes, en problemas reales de dificultad moderada-alta, en nuestro caso, diez tipos de vehículos y animales de la base de datos Cifar10 (https://www.cs.toronto.edu/~kriz/cifar.html). Utilizaremos la librería Keras (https://keras.io/) de TensorFlow, que facilita configurar modelos de redes neuronales para problemas de clasificación y regresión.

La práctica se divide en dos partes. En la **primera parte** desarrollaremos modelos de **perceptrones multicapa** (**MLP** por sus siglas en inglés) para clasificar correctamente las imágenes de Cifar10. En la **segunda parte** utilizaremos **redes neuronales convolucionales** (**CNN** por sus siglas en inglés). Exploraremos los distintos parámetros de estos modelos y realizaremos las pruebas necesarias para validar la calidad de su entrenamiento.

Entorno de trabajo

En los laboratorios trabajaremos con **Python en Linux**. Python es un lenguaje interpretado multiplataforma

muy extendido, se puede usar en todos los sistemas operativos de uso general, y cada estudiante puede elegir el entorno de trabajo que prefiera (tanto sistema operativo como IDE), pero debes asegurarte de que tu código fuente funciona correctamente en Linux en los PC de los laboratorios usando Thonny (https://thonny.org/), porque ese es el entorno en que se evaluará la entrega.

Si no tienes experiencia con Python, se recomienda no usar Visual Studio y utilizar en su lugar el **IDE Thonny** (ya instalado en los laboratorios), puesto que da mejores ayudas a programadores de Python nóveles, y además tiene la ventaja de utilizar su propia instalación de Python, independiente de la del sistema operativo, de forma que no genera conflictos y facilita la instalación de paquetes como Tensorflow y Keras. Para instalar un paquete en Thonny ve a Herramientas → Gestionar paquetes, y ahí busca el paquete que desees instalar.

Además de Keras, en general siempre trabajaremos con Numpy (https://numpy.org/) para la manipulación eficiente de arrays, y alguna de las facilidades que nos provee Pandas (https://pandas.pydata.org/), y para el dibujado de imágenes y gráficas en pantalla utilizaremos Matplotlib (https://matplotlib.org/). También podrás utilizar cualquier librería de Python que se pueda instalar en vuestro Thonny o usando `pip --user install <librería>` en los laboratorios, pero deberás consultar primero con tu profesor/a. Como alternativa que no requiere instalación, tendrás la opción de utilizar Google Colab para realizar la práctica.

Si tu código no puede ejecutarse en los laboratorios o en Google Colab, no será evaluado, salvo acuerdo previo con tu profesor.

Evaluación

Cada parte de la práctica aporta hasta 5 puntos de la nota final. Dentro de cada parte hay apartados obligatorios y optativos (solo están marcados los apartados optativos, todos los demás son obligatorios). Para aprobar la práctica, todos los apartados obligatorios deben estar realizados. Los apartados marcados como optativos no son necesarios para aprobar (se puede sacar más de un 5 si se completan satisfactoriamente todos los apartados obligatorios), pero permiten compensar la nota de apartados obligatorios que no hayan salido bien. La implementación (código, experimentación) supone el 40% de la nota y la documentación (explicaciones, análisis) el 60%.

Documentación

La documentación es la parte más **importante** de la práctica, supone el **60% de la nota máxima**. Tu documentación incluirá una sección para cada apartado de la práctica, donde pondrás las respuestas a las preguntas que se te plantean y los resultados que se piden en cada apartado, además de cualquier apunte que consideres relevante sobre el desarrollo de la práctica, pruebas extra que hayas realizado, mejoras que hayas implementado, y conclusiones extraídas de esos experimentos. Es importante que concretes y sintetices (que vayas al grano y que resumas). Si no tienes clara la respuesta a una pregunta o de qué manera debes informar sobre tus experimentos, es mejor que le pidas ayuda tu profesor durante las horas de prácticas o por tutoría, y no que copies parrafadas de ChatGPT que no terminas de entender, o que pongas páginas y páginas de gráficas o tablas repetitivas que te llevó mucho tiempo completar y que no te va a subir la nota. Lo importante es que tu

documentación demuestre que has aprendido lo que se te pide, no que hayas hecho muchos experimentos repetitivos o que tu memoria tenga muchas páginas. Consulta a tu profesor si tienes dudas sobre esto.

La documentación **debe incluir una sección de bibliografía** con todas las referencias que utilices, o bien, estas referencias deben enlazarse en el texto donde se usen (entre paréntesis o como nota a pie). Las referencias a vídeos de plataformas como Youtube deben incluir un breve resumen de la información que se ha extraído de esa fuente. Las consultas a ChatGPT, DeepSeek, Bing y otros chatbots basados en grandes modelos de lenguaje son también parte de la bibliografía y deben ser referenciados. Si copias sus respuestas breves textualmente, debes marcarlas como una cita, entre comillas, dado que los resultados de un *prompt* no se pueden enlazar. Si los usas para entender conceptos o desarrollar explicaciones, en una conversación, no debes copiar la conversación entera en mitad de la documentación, pero puedes añadirla como un anexo al final (y citas la conversación desde el texto en que la usas), aunque será suficiente con que indiques en cada parte del texto en qué cuestiones te has apoyado en el chatbot. Si utilizas una sección de bibliografía, cada referencia en esa sección debe aparecer citada en el texto, en los apartados en que se ha usado. Una bibliografía mínima puede consistir solo en los apuntes de clase (aunque recomendamos que investigues más por tu cuenta). **La ausencia de bibliografía, o si no se referencia en el texto donde se use, podrá penalizar hasta 1 punto**. Este requerimiento no significa que debas buscar algo que citar para rellenar (una buena bibliografía no da puntos, solo te permite aprender más deprisa y hacer mejor el trabajo): si no has usado ninguna referencia externa, no debes perder el tiempo en inventarte una bibliografía, citar elementos que no utilizas también será motivo de penalización.

Fecha de entrega

Hasta el **27 de mayo de 2025 a las 23:55h**.

Formato de entrega

Las prácticas son individuales, no se pueden hacer en grupos y no se puede reusar código o texto de otros estudiantes o de otras fuentes sin la requerida referencia. La entrega se realizará **a través de Moodle**, y debe consistir en dos ficheros, **un fichero .pdf con la documentación y otro fichero .py con todo el código Python utilizado por el estudiante**, tanto las implementaciones de los algoritmos requeridos como cualquier código utilizado para producir cualquier resultado que se muestre en la documentación. No se puede entregar la práctica en varios ficheros de Python. Si deseas programar en varios archivos por el motivo que sea, asegúrate de juntarlo todo en un único archivo .py y comprobar que todo funciona correctamente antes de la entrega. Los dos archivos de la entrega se deben llamar con el nombre completo del estudiante, usando '_' como separador entre palabras (p. ej. Juan_Carlos_Sánchez-Arjona_de_los_Rios.py y .pdf). En el caso de que creas necesario adjuntar algún otro archivo (una hoja de cálculo con resultados, un archivo de log, etc.) consulta primero a tu profesor/a.

Para permitir la evaluación de tu entrega, deberás crear un módulo (una función) en tu archivo Python para cada tarea de la práctica, que llamará a todas las funciones implicadas en completar esa tarea. Estos módulos se podrán llamar individualmente en la última sección de tu archivo Python, utilizando el condicional `__name__`

== "__main__". Algo como lo que sigue:

```
if __name__ == "__main__":

    ## Funciones auxiliares relevantes para la evaluación, comentadas
    #test_MLP(…)

    X_train, Y_train, X_test, Y_test = cargar_y_preprocesar_cifar10()

    # tarea A
    mlpA = probar_MLP(X_train[0].shape, ocultas=[32], activ=["sigmoid"])

    # Tarea B
    mlpB = probar_MLP(X_train[0].shape, ocultas=[32], activ=["sigmoid"], ep=32)
```

Puede que algunas tareas requieran varios módulos, o que reusen módulos de otras tareas con distintos parámetros (en el código de ejemplo, el módulo probar_MLP se usa para declarar, compilar, entrenar y evaluar distintos modelos de MLP). Si los nombres de tus funciones son ambiguos, utiliza comentarios para dejar claro qué hace cada módulo. La idea es que los profesores podamos ejecutar solo la parte que queramos de tu código, sin tener que esperar a que se entrenen todos los clasificadores de todas las tareas.

Trabajando con las imágenes de Cifar10

Vamos a cargar la base de datos de Cifar10 y familiarizarnos con la manipulación de esas imágenes. Hay muchas maneras de cargar las imágenes de Cifar10 en un programa Python. Nosotros utilizaremos la función que nos facilita Keras (https://keras.io/api/datasets/cifar10/). Para ello primero hay que importar esa librería (e instalarla si no está instalada). Al ejecutar ese `import` algunos veréis una serie de mensajes de advertencia de TensorFlow, quejándose de que no encuentra soporte para usar CUDA en la tarjeta gráfica. Quienes tengan una tarjeta NVIDIA pueden instalar el soporte para CUDA y conseguir entrenamientos de modelos de aprendizaje mucho más rápidos gracias a la paralelización, pero no es necesario en esta práctica y no afecta a la calificación. Si queremos eliminar esos mensajes de error por comodidad, podemos utilizar el siguiente código al principio de nuestro archivo .py:

```
import logging, os
logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from tensorflow import keras
```

Ahora podemos cargar las imágenes de Cifar10 y ver en qué consisten:

```
(X_train, Y_train), (X_test, Y_test) = keras.datasets.cifar10.load_data()

print(X_train.shape, X_train.dtype)
print(Y_train.shape, Y_train.dtype)
print(X_test.shape, X_test.dtype)
print(Y_test.shape, Y_test.dtype)
```

Esos print nos dicen que `X_train` es un array de Numpy de 50 000 elementos (en este caso observaciones, o *samples* en inglés) que a su vez son arrays de 32x32x3 `uint8` (unsigned int de 8 bits, números de 0 a 255), es decir, un array de 50 000 imágenes en color de 32x32 píxeles. Cada uno de esos 32x32x3=3072 bytes de una imagen es una característica de una observación o muestra. `Y_train` es otro array de 50 000 elementos `uint8` que contiene los 50 000 números asociados a cada categoría (0: avión; 1: coche; 2: pájaro, etc.), y que son los

valores que querremos que nos devuelva nuestro MLP para cada imagen cuando esté bien entrenado. Y los arrays `X_test` e `Y_test` son iguales pero contienen 10 000 observaciones cada uno, y son los que usaremos para comprobar la calidad del entrenamiento. Las imágenes de test nunca se utilizan en el entrenamiento (`fit`), solo en la evaluación (`evaluate`).

Podemos mostrar en pantalla las imágenes que hay en `X_train` e `X_test` utilizando la función `imshow` de Matplotlib. El siguiente código mostrará tres imágenes al azar del conjunto de entrenamiento:

```
import matplotlib.pyplot as plt

def show_image(imagen, titulo):
    plt.figure()
    plt.suptitle(titulo)
    plt.imshow(imagen, cmap = “Greys”)
    plt.show()

from random import sample

for i in sample(list(range(len(X_train))), 3):
    titulo = "Mostrando imagen X_train[" + str(i) + "]"
    titulo = titulo + " -- Y_train[" + str(i) + "] = " + str(Y_train[i])
    show_image(X_train[i], titulo)
```

Puedes probar más cosas para adquirir destreza en Python con Numpy y Matplotlib, o sacar conclusiones acerca de los datos sobre los que trabajamos. Podrías contar la cantidad de imágenes de cada categoría en el conjunto de entrenamiento, o mostrar por pantalla los primeros tres perros. Por ejemplo, el siguiente código imprime una gráfica con los valores de las etiquetas de las 100 primeras imágenes del conjunto de test:

```
def plot_curva(Y, titulo, xscale = "linear", yscale = "linear"):
    plt.title(titulo)
    plt.plot(Y)
    plt.xscale(xscale)
    plt.yscale(yscale)
    plt.show()

plot_X(Y_test[:100], "Etiquetas de los primeros 1000 valores")
```

Para cada tarea de esta práctica deberás crear en tu código fuente una función que realice la carga de datos, el entrenamiento del modelo de MLP según los parámetros que tú le digas, y la evaluación de los resultados. Se recomienda crear una función `cargarCifar10()` que devuelva los cuatro conjuntos de datos de Cifar 10, con cualquier preprocesamiento que fuera necesario (ten en cuenta que MLPs y CNNs requieren de distintos preprocesamientos)), y explicarás en la memoria en qué consiste ese preprocesamiento y por qué es necesario o conveniente. También una o varias funciones más que muestren los siguientes resultados a partir de las predicciones del modelo de esa tarea y las clases verdaderas:

1. Las **gráficas de líneas que muestren la evolución de la pérdida y la tasa de acierto** para el conjunto de entrenamiento y para el conjunto de validación durante el entrenamiento de la red (en Keras, esta información la devuelve la función *fit*, que es la que realiza el entrenamiento).
2. La pérdida y la tasa de acierto finales con el conjunto de test, y el tiempo empleado en el entrenamiento (en Keras, esto lo conseguimos con la función *evaluate*, pasándole el conjunto de test), utilizando una **gráfica de barras para comparar resultados de varios modelos**.
3. La **matriz de confusión** para los resultados con el conjunto de test (investiga cómo puedes mostrar esa información de manera rápida y elegante, quizás tengas que instalar alguna otra librería de Python).

1. Definir, configurar y entrenar un MLP en Keras

Tarea A: Definir, utilizar y evaluar un MLP con Keras

En la primera tarea de esta práctica aprenderás a declarar, entrenar y emplear un MLP de Keras. En un MLP se utilizan capas Dense (es decir, cada neurona de una capa está conectada a todas las neuronas de la anterior capa), que no tienen en cuenta la organización espacial de las características de una observación y que esperarán que todas las características de una misma muestra aparezcan en un único vector.

El primer paso será crear un fichero .py con vuestro nombre y apellidos si no lo has hecho ya, que será el que usarás en la entrega, y programar la función para cargar los datos de Cifar10 y aplicarle cualquier preprocesamiento que sea necesario para que un MLP pueda utilizar esos datos, además de documentar la utilidad o necesidad de ese preprocesamiento.

El primer MLP que declararás ha de tener una sola capa oculta de tipo Dense con 32 neuronas y función de activación *sigmoid*, y una capa de salida con el número de neuronas que sea conveniente para nuestro problema de clasificación, función de activación *softmax*. Todo eso podemos hacerlo con este fragmento de código:

```
model = keras.Sequential(name="MLP_A")
model.add(layers.Flatten) # aplana las imágenes para que entren como un vector
model.add(layers.Dense(32))
model.add(layers.Activation("sigmoid"))
model.add(layers.Dense(?))
model.add(layers.Activation="softmax"))
```

Puedes encontrar más ejemplos en la documentación de keras.io (https://keras.io/guides/sequential_model/).

Una vez declarado el modelo, habrás de compilarlo con la función *compile*, indicando como mínimo el `optimizer`, la función de pérdida (`loss`), y las métricas que se mostrarán (`metrics`). En este caso queremos la función de pérdida *categorical crossentropy*, el optimizador *Adam* y la métrica *accuracy* (la tasa de acierto):

```
model.compile(loss="categorical_crossentropy",
              optimizer=keras.optimizers.Adam(),
              metrics=["accuracy"])
```

Una vez compilado el modelo, puedes mostrar un resumen de su estructura con `model.summary()`. Después tendrás que entrenarlo con el conjunto de entrenamiento utilizando la función `fit`, que devolverá un historial de los valores de tasa de acierto (*accuracy*) y de pérdida durante el entrenamiento, tanto del conjunto de entrenamiento como del de validación, que podrás usar para plotear la gráfica de evolución del entrenamiento (lo veremos en la siguiente tarea). Para indicar el tamaño del conjunto de validación utilizamos el parámetro `validation_split`, que por ahora fijaremos a 0.1. También debemos indicar el número de muestras que se pasan por la red antes de aplicar backpropagation, con el parámetro `batch_size`, que fijaremos por ahora a 32, y la cantidad de veces que se pasará el conjunto completo de muestras, `epochs`.

```
history = model.fit(x_train, y_train, batch_size=32, epochs=10,
                    validation_split=0.1, verbose=1)
```

Una vez entrenado el modelo, debes evaluarlo con el conjunto de test, para obtener una tasa de acierto y una pérdida con ese conjunto. Como veremos más adelante, comparando esa tasa de acierto con la que obtengas con

otros modelos podrás estimar qué modelo es mejor:

```
test_score = model.evaluate(x_test, y_test, verbose=1)
print("Test accuracy and loss:", test_score[1], test_score[0])
```

En la documentación has de explicar con tus palabras, de forma que se entienda el funcionamiento concreto, qué es un MLP y en qué fundamentos se basa para "aprender". Puedes apoyarte en cualquier fuente, incluido conversaciones con chatbots, pero debes demostrar que has entendido los conceptos que utilices.

Se valora que el código esté correctamente comentado y que sea autoexplicativo (evitar nombres de funciones o variables que no se entiendan). No incluyas fragmentos de código en la documentación salvo que sea imprescindible para explicar algo y que creas que no es suficiente con comentarlo en el propio código.

Tarea B: Ajustar el valor de los parámetros epochs y validation_split

En esta tarea analizaremos la evolución del entrenamiento de tu modelo para detectar si no se ha entrenado suficientemente o si se ha incurrido en sobreentrenamiento. El sobreentrenamiento ocurre cuando el clasificador se ajusta tan bien a las características del conjunto de entrenamiento que pierde generalidad y obtiene peores resultados en el mundo real, y para eso observamos lo que ocurre con el conjunto de validación durante el entrenamiento. El conjunto de validación es una parte del conjunto de entrenamiento que se aparta del resto (no se entrena con esos datos) y se utiliza como proxy de lo que sería el modelo en el mundo real. El conjunto de test es otro grupo de muestras que se mantienen fuera del estrenamiento (no se usan para validación) y que nos servirá como prueba final antes de sacar el modelo al mundo real. Para detectar el sobreentrenamiento necesitaremos ver las curvas de *train_accuracy*, *train_loss*, validation_accuracy y *validation_loss* (emplearemos Matplotlib): el momento adecuado para parar el entrenamiento, el número de epochs, es cuando validation accurary y loss se estabilizan, antes de que empiecen a descender. Para averiguar si nuestro conjunto de validación está sirviendo como proxy del mundo real o no (es decir, si nos está sirviendo para encontrar un buen número de épocas), compararemos entre sí las tasas de aciero que den los modelos finales tanto con el conjunto de validación como con el de test: si el conjunto de validación es demasiado grande le habrá restado opciones de entrenamiento al modelo y los resultados finales serán peores en general (otros modelos sacarán mejor tasa de acierto con el conjunto de test), y si es demasiado pequeño la estimación que de puede ser errónea y permitir sobreentrenamiento.

Para dibujar las gráficas, en la introducción vimos un ejemplo con una función `plot_curva` que dibuja una línea a partir de los puntos de una lista (Y), que nos puede servir para esta tarea. En la página oficial de Matplotlib tenemos un ejemplo más elaborado (https://matplotlib.org/stable/plot_types/basic/plot.html), para ploteados en el que X no es uniforme y que dibuja varias curvas en la misma gráfica. Idealmente, querremos plotear las curvas de *accuracy* sobre uno de los ejes verticales y las curvas de *loss* sobre el otro, para poder ajustar el rango independientemente y que se aprecie mejor la pendiente de las curvas. Esto lo veremos detenidamente en clase.

Una vez tengamos esa gráfica podemos comprobar rápidamente si el entrenamiento se detuvo prematuramente, cuando todas las curvas mostraban una mejora que podía continuar, o si se entrenó demasiado tiempo, cuando

las curvas de validación empezaron a empeorar. Con esa información, debes ajustar el valor del número de épocas para que el entrenamiento se detenga en los mejores valores de *val_accuracy* (y en caso de duda, los mejores valores de *val_loss*).

Y para comprobar si el `validation_split` es correcto deberás dibujar una gráfica de barras de modelos entrenados con diferentes validation_split, mostrando las tasas de acierto con los conjuntos de validación y de test. El modelo que haya conseguido mejor tasa de acierto de test final (que probablemente será similar a la tasa de acierto con el conjunto de validación) será el que tenga un `validation_split` mejor ajustado.

Ten en cuenta que el entrenamiento de estos modelos parte de estados iniciados al azar, que influyen en el éxito del entrenamiento, por lo que dos entrenamientos diferentes del mismo modelo (todos los parámetros iguales) darán resultados diferentes, a veces muy diferentes. Por ello **es necesario realizar varios entrenamientos independientes y analizar los resultados promedio**, e idealmente descartar resultados *outlier*. En esta práctica sobra con que hagas cinco repeticiones de cada entrenamiento (guardas cada history, los promedias al final y eso es lo que ploteas), pero ten en cuenta que para conseguir robustez estadística en un entorno profesional se necesitarían más (o muchas más) repeticiones, dependiendo de la complejidad del problema y de lo bien ajustado que estén el resto de parámetros para evitar quedar atrapado en mínimos locales demasiado pronto.

Explica en la documentación qué es el sobreentrenamiento, por qué puede ocurrir, por qué se debe evitar, qué es el número de épocas y el validation split, y cómo se pueden ajustar para maximizar la tasa de acierto.

Por último, utilizando *callbacks* de Keras (https://keras.io/api/callbacks/), haz que tu MLP detecte sobreentrenamiento de forma automática y corte el entrenamiento antes de alcanzar el máximo de épocas. Deberás configurar el detector adecuadamente para asegurarte de que no incurre en falsos positivos que detengan el entrenamiento demasiado pronto ni que tarde demasiado en decidir parar. Para conseguirlo, consulta la documentación de Keras y ajusta los parámetros de ese callback, haciendo pruebas con diferentes configuraciones y comprobando el resultado final en términos de la tasa de acierto con el conjunto de test.

Tarea C: Ajustar el valor del parámetro batch_size

En esta tarea debes documentarte (y explicar en la memoria) sobre qué controla el parámetro *batch_size* en el algoritmo de entrenamiento de un MLP (p.ej.: https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network) y comprobarás experimentalmente los efectos que tiene sobre la velocidad y el éxito del entrenamiento de un MLP.

Entrena y evalúa varios MLP con diferentes valores de batch_size. Discute si deberías ajustar un número de epochs diferente para cada MLP. Y utiliza Matplotlib para crear **una** gráfica de barras que muestre el *test accuracy* y el tiempo de entrenamiento de todos los modelos que hayas probado.

A partir de esa gráfica, has de seleccionar el valor de *batch_size* que consiga la mejor tasa de acierto sin emplear más tiempo del necesario.
Para comprender mejor los resultados de los modelos generados, se deberán incluir matrices de confusión que permitan visualizar las categorías sobre las que se confunde.

Tarea D: Probar diferentes funciones de activación

Al añadir una capa Dense a un MLP debemos indicar una función de activación (https://keras.io/api/layers/activations/). Documéntate sobre las recomendaciones de uso de cada función y el rango de valores que pueden devolver, para elegir las dos o tres que puedan ser mejores que sigmoid en el contexto de esta práctica. Después, entrena y evalúa varios MLP con esas funciones de activación, y plotea los resultados de tasa de acierto y tiempo de entrenamiento como en la tarea anterior, para elegir la mejor función de activación con este problema.

Tarea E: Ajustar el número de neuronas por capa

Entrena y evalúa varios MLP con diferente número de neuronas en su capa oculta, discute los resultados y selecciona el modelo que consiga la mejor test_acc sin desperdiciar tiempo. Explica qué efectos tiene incrementar o decrementar el número de neuronas de un MLP.

Tarea F: Optimizar un MLP de dos o más capas

Añade una o más capas Dense a tu modelo y vuelve a ajustar todos los parámetros, de la forma en que has aprendido en las tareas anteriores, para mejorar la tasa de acierto, o reducir el tiempo de entrenamiento sin empeorar la tasa de acierto. Para hacer una comparación justa entre diferentes números de capas, incluye configuraciones que tengan entre todas sus capas la misma cantidad de neuronas (p.ej. una capa de 96 vs dos capas de 48 vs tres capas de 32). También puedes comparar, para el mismo número de neuronas totales, el efecto de poner más neuronas al principio o al final (p.ej. 48+32+16 vs 32+32+32 vs 16+32+48). En la memoria, analiza los resultados, identifica los modelos que mejor rendimiento tienen y explica a qué crees que se debe.

2. CNN de Keras

En las siguientes tareas, muy parecidas a las anteriores, volverás a clasificar las imágenes de Cifar10 con redes neuronales, pero esta vez utilizando redes convolucionales, cuyas “neuronas” (en realidad unidades o nodos computacionales) aplican filtros de convolución a las imágenes de entrada, permitiendo extraer de características de más alto nivel que tienen en cuenta la información espacial, por lo que están mejor adaptadas a problemas de visión artificial. Otra diferencia importante con los MLP es que la salida de un filtro no es un único valor sino muchos, la matriz de productos internos del filtro pasado por toda la imagen, por lo que una CNN con el mismo número de “neuronas” por capa que un MLP tendrá en realidad muchos más parámetros (pesos) que entrenar y requerirá un mayor esfuerzo computacional.

Tarea G: Definir, entrenar y evaluar un CNN sencillo con Keras

Para completar esta tarea necesitarás documentarte sobre las CNN y buscar ejemplos de implementación utilizando Keras. Encontrarás varios en la propia web de Keras (p.ej. https://keras.io/examples/vision/mnist_convnet/). Ten en cuenta que las capas de una CNN utilizan la información espacial y de color, por lo que

las imágenes no se deben preprocesar de la misma manera que con las capas Dense de un MLP.

Para empezar a experimentar, declara una CNN con dos capas Conv2D de 16 y 32 filtros respectivamente, ambas con activación ReLu y filtros de 3x3, y como capa de salida una capa Dense con función de activación softmax. Deja el resto de parámetros en sus valores por defecto. Como hiciste en tareas anteriores, apóyate en *callbacks* de *Early Stopping*. Para parar el entrenamiento en el momento adecuado. Ten en cuenta que las CNN tienen un funcionamiento muy diferente del de los MLP y podrías neceitar reajustar los parámetros del callback.

Ahora añade una capa MaxPooling2D con tamaño de filtro 2x2 después de cada capa Conv2 (similar al código en el ejemplo enlazado al principio de esta página), ajusta el número de épocas y compara los resultados entre ambos modelos de CNN (con y sin capas MaxPooling2d). En la memoria, argumenta cuál es mejor, por qué crees que lo es, y explica también los fundamentos de funcionamiento de una CNN y en qué consisten las capas Conv2D y MaxPooling2D.

Tarea H: Optimizar el tamaño y el salto de los filtros de convolución.

Ahora deberás comparar entre sí CNNs con diferente tamaño de filtros y salto (stride), también añadiendo o no capas MaxPooling2D, con el objetivo de dar con el modelo que mejor resultados de en menos tiempo.

A mayor tamaño de filtro, más píxeles pueden utilizarse para detectar características relevantes en la imagen (por lo que podrá detectar patrones más grandes) pero también requiere más cálculos para obtener la matriz de productos internos, aunque al mismo tiempo producirá matrices ligeramente más pequeñas, que acelerará los cálculos de la siguiente capa. En cualquier caso, se recomienda usar números impares (3x3, 5x5, 7x7, etc.).

Por otro lado, a mayor stride, menos productos internos se han de calcular por lo que se acelera el procesamiento de la capa y también se reduce el número de inputs de la siguiente capa. Con la desventaja de que el modelo será más susceptible a trasformaciones como la traslación, ya que perderá capacidad de detección de patrones.

Tarea I (OPTATIVA): Optimizar la arquitectura del modelo

En esta tarea puedes incluir diferentes mejoras sobre el entrenamiento de las redes con el objetivo de aumentar su capacidad de generalización (su tasa de acierto con el conjunto de test) o reducir su tiempo de entrenamiento y/o su número de parámetros. Ten en cuenta que aunque el proceso individual más costoso en la explotación de un modelo neuronal es su entrenamiento, que cuesta muchísimo más que dar un resultado ante una entrada concreta, el número de parámetros del modelo sí afecta al coste de obtener una salida, y si el modelo se va a utilizar intensivamente por millones de usuarios el ahorro energético al optar por un modelo más sencillo puede ser muy relevante.

Algunas de las mejoras a incluir en esta sección pueden ser las siguientes:

- Métodos de agrupación (*pooling*): https://keras.io/api/layers/pooling_layers/
- Métodos de regularización: https://keras.io/api/layers/regularization_layers/

- Aumento de datos: https://keras.io/api/layers/preprocessing_layers/image_augmentation/
- Normalización y estandarización de los datos: https://keras.io/api/layers/normalization_layers/
- Técnicas de inicialización inteligente de los pesos de la red: https://keras.io/api/layers/initializers/

Las mejoras que propongas deberán estar fundamentadas, implementadas y evaluadas convenientemente para que puedan ser tenidas en consideración.

No olvides que también podrás (¿deberás?) reajustar el resto de parámetros con los que ya has estado trabajando en las tareas anteriores:

- Tamaño y profundidad de las capas ocultas (tanto Dense como Conv2D).
- Funciones de activación de las capas ocultas.
- Tamaño del batch y del conjunto de validación.
- Tamaño de los filtros y el stride en capas Conv2D.
- Tamaño de los filtros y el stride en capas MaxPooling2D

No es necesario que cambies cada uno de esos items. Recomiendo hacer pruebas pequeñas para estimar si valdría la pena repetir el ajuste o no. En cualquier caso, en la memoria comenta las pruebas que realices pero no es necesario incluyas los resultados de las pruebas que descartes.

Es recomendable que te documentes antes de iniciar la experimentación, para apoyarte en resultados previos de otros investigadores (todo bien referenciado en tu bibliografía) y poder dirigir mejor tu tiempo y esfuerzo. Busca artículos donde se presenten modelos de CNN para este o similares problemas y extrae de ellos las mejores ideas. Pero en cualquier caso tú debes repetir los experimentos e intentar encontrar alguna mejora. Si sencillamente citas un buen modelo, lo replicas en tu experimentación, y me comentas que no se podía mejorar, sin aportar experimentos que lo refrenden, no tendrás nota en este apartado.

AVISO IMPORTANTE

No cumplir cualquiera de las normas de entrega descritas al principio de este documento puede suponer un suspenso de la práctica.

Las prácticas son individuales. No se pueden hacer en pareja o grupos.

Cualquier código o texto copiado supondrá un suspenso de la práctica para todos los estudiantes implicados y **se informará a la dirección de la EPS para la toma de medidas oportunas** (Reglamento para la Evaluación de Aprendizajes de la Universidad de Alicante, BOUA 9/12/2015, y documento de Actuación ante copia en pruebas de evaluación de la EPS).