2025/2026 1Q

Lista de Problemas 3

APA

Javier Béjar

Departament de Ciències de la Computació

Grau en Enginyeria Informàtica - UPC

UPC FIB Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTAMENT DE CIÉNCIES DE LA COMPUTACIÓ

FACULTAT D'INFORMÁTICA DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Primera edición, septiembre 2021

Esta edición, Septiembre 2025

Instrucciones:

Para la entrega de grupo debéis elegir un problema del capítulo de problemas de grupo.

Para la entrega individual debéis elegir un problema del capítulo de problemas individuales. **Cada miembro del grupo debe elegir un problema diferente**.

Debéis hacer la entrega subiendo la solución al racó.

Evaluación:

La nota de esta entrega se calculará como 1/3 de la nota del problema de grupo más 2/3 de la nota del problema individual.

Puntuación:

En esta segunda lista, los errores leves se penalizarán con 2 puntos, los errores graves, sobre todo metodológicos, se penalizarán con 4 puntos. Explicaciones demasiado breves sobre lo que habéis hecho y los resultados también reducirán la nota. No hagáis una resolución mecánica de los problemas.

La penalización de los errores se doblará en la siguiente lista de problemas

Al realizar el informe correspondiente a los problemas explicad los resultados y las respuestas a las preguntas de la manera que os parezca necesaria. Se valorará más que uséis gráficas u otros elementos para ser más ilustrativos.

Entregad los resultados como un notebook (Colab/Jupyter). Podéis poner las respuestas a las preguntas en el notebook, este os permite insertar texto en markdown y en latex.

Aseguraos de que los notebooks mantienen la solución que habéis obtenido, no los entreguéis sin ejecutar.

Objetivos de aprendizaje:

1. Conocer las arquitecturas de perceptrón multicapa y redes convolucionales
2. Conocer las funciones de error según el tipo de tarea de un problema
3. Experimentar con diferentes configuraciones para una red y saberlas ajustar para diferentes tipos de problemas

CAPÍTULO 1

Problemas de Grupo

Al resolver el problema explicad bien los que hacéis, no hacer ningún comentario o hacer comentarios superficiales tendrán una nota más baja.

Tenéis que mostrar que habéis entendido los métodos que estáis aplicando, así que un corta y pega de problemas similares no es suficiente.

Estos problemas se han de resolver utilizando la librería PyTorch. Esto significa que deberéis hacer la experimentación usando Google Colab para poder hacerla en un tiempo razonable.

Aunque hagáis los experimentos en Colab necesitaréis cierto tiempo para obtener los resultados, así que no esperéis hasta el último momento para hacerlos.

1. **¿Qué número es este?**

   Parece haber una obsesión con conjuntos de datos que representan dígitos escritos a mano. Esto tiene una explicación sencilla. Uno de los primeros retos del OCR fue el reconocimiento de códigos postales en cartas. El clasificar cartas a mano consume demasiado tiempo y no es completamente fiable (además de ser muy aburrido), por lo que tener un sistema eficiente para clasificar el correo era un problema interesante[1]. Uno de los conjuntos de datos más conocido es MNIST digits. Vamos a utilizar una parte de este conjunto de datos, en concreto un subconjunto de los dígitos 3, 5, 6, 8 y 9, para experimentar con diferentes redes neuronales. Podéis obtenerlos mediante la función `load_MNIST` de la librería `apafib`. Esta función retornará cuatro matrices de datos, el conjunto de entrenamiento, el conjunto de test y las etiquetas de los dos. Resolved los siguientes apartados ilustrando los resultados de la manera que os parezca más adecuada.

   Para resolver los siguientes apartados tenéis un notebook con una serie de funciones que os servirán para resolverlos.

   a) Los datos corresponden a imágenes de 28×28 píxeles en niveles de gris. Estos ya están convertidos a vectores y normalizados a la escala [0,1]. Aplicad PCA y t-SNE a los datos

[1] Ahora ya nadie escribe cartas, pero siempre habrá texto escrito a mano que reconocer en otras aplicaciones.

de entrenamiento y represéntadlos en 2D. ¿Se puede ver separabilidad entre las clases? Comentad el resultado.

b) Comenzaremos por obtener un valor de acierto base. Asumiendo que los píxeles son independientes y se distribuyen de manera Multinomial (solo hay 16 valores distintos en los datos). Ajustad un modelo Naïve Bayes Multinomial. Evaluad la calidad del modelo y comentad los resultados.

c) Ahora aplicaremos un MLP a estos datos. En el código del notebook tenéis una función que permite generar modelos MLP variando el número de capas y la función de activación. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` cómo se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

Para pasarle los datos a un modelo de `torch` hay que definir una clase del tipo `Dataset`, tenéis una ya definida en el notebook a la que hay que pasarle la matriz de datos y el vector con las etiquetas. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), podéis ver un ejemplo en el notebook. Tendréis que separar una parte de datos de entrenamiento para validación (10 %), ya que no usaremos validación cruzada.

Para entrenar los modelos usaremos la función `train_loop` que tenéis en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, podéis ver en el notebook como generar el objeto del optimizador.

Entrenad diferentes redes de 2 y 3 capas con diferentes tamaños usando como función de activación `ReLU` y `Sigmoid`. Para los tamaños podéis elegir el tamaño de la primera capa oculta y después ir reduciéndolo a la mitad para las capas sucesivas. Para crear la red solo tenéis que crear un objeto de la clase `MLP` pasándole los parámetros adecuados. Al generar el objeto tendréis que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no lo hacéis torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Veréis que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usad esta información para comprobar el número de épocas que realmente se necesita para converger. Podéis comprobar si la red ha sobre especializado con el conjunto de test.

Evaluad los resultados. Para ello podéis usar la función `test_model` que tenéis en el notebook. Se le ha de pasar el modelo entrenado y un conjunto de datos y retornará las etiquetas de las predicciones y las reales, a partir de ahí podéis usar las funciones de `scikit learn` para obtener las medidas de evaluación.

d) Las redes MLP tienen bastantes parámetros y son menos eficientes que las redes convolucionales en tareas de imágenes. Tenéis en el notebook una clase que permite definir redes convolucionales para los datos del problema. Entrenad redes de dos capas que tengan diferentes números de kernels convolucionales, para ello, elegid el número de kernels para la primera capa y dobladlo para la segunda. Podéis evaluar el modelo de la misma manera que en el apartado anterior. Comparad los resultados.

e) Podemos visualizar el efecto tiene la función aprendida por las redes obteniendo el resultado antes de la capa de salida (la capa lineal final). Añadid un método a las clases que definen las redes que apliquen a la entrada las capas de la red excepto esa última capa. Obtened el resultado de esta transformación a partir de las mejores redes entrenadas para el MLP y para la convoluciona con ReLU y sigmoide. Aplicad un PCA a ese resultado y visualizadlo. Comentad los resultados.

2. **Clasificando texto**

Trabajar con texto es más complicado que tratar datos tabulares. Este se ha de procesar adecuadamente para poder aplicarle algoritmos de aprendizaje. En el área de recuperación de la información hay métodos clásicos que transforman los textos en vectores de números a partir de un vocabulario y cálculos sobre las frecuencias de las palabras en los documentos y su relación con sus frecuencias en el corpus de documentos. El mayor problema de estas transformaciones es que el número de variables en el conjunto de datos depende del tamaño del vocabulario y el espacio que se obtiene es esparso.

Las redes neuronales han cambiado el panorama de esta área permitiendo obtener modelos entrenados con grandes corpus de documentos que transformar textos en vectores densos de reducido tamaño.Los modelos están entrenados de manera que las relaciones entre las palabras queden reflejadas en la representación. El espacio que es aprendido por estas redes neuronales puede ser genérico o estar especializado en una tarea concreta, pero es un espacio denso.

Vamos a utilizar un corpus de texto que corresponde a resúmenes de documentos científicos extraídos de `arxiv.org` de diferentes disciplinas (física, astrofísica, altas energías, física cuántica, computación, matemáticas). Los datos los podéis obtener mediante la función `load_papers` de la librería `apafib`. Esta función retornará una lista con los textos y otra con las etiquetas que les corresponden. Resuelve los siguientes apartados ilustrando los resultados de la manera que te parezca más adecuada.

Para resolver los siguientes apartados tenéis un notebook con una serie de funciones que os servirán para resolverlos.

a) El trabajar con texto es algo diferente de los datos tabulares habituales. En este tipo de datos se obtiene la matriz de datos a partir de extraer un subconjunto de las palabras de los textos y hacer un cálculo sobre ellas. En este caso utilizaremos la frecuencia de las palabras en un documento (TF) y el inverso de las frecuencias de las palabras en el total de documentos (IDF). Esto lo podemos hacer con la función `TfidfVectorizer` del scikit-learn que es precisamente para este tipo de datos. Generaremos una matriz de datos con 1000 y 2000 palabras[2]. Emplea el parámetro `stop_words` que tiene esta función indicando que el idioma del texto es el inglés. Podemos también indicar el límite a la frecuencia máxima con la que aparecen las palabras en el corpus. Podemos asumir que palabras que aparecen frecuentemente en todos los documentos no son importantes para clasificarlos, usad un límite del 90 %.

Fijaos que esto **es un preproceso** como otros que hemos ido empleando, así que aplicadlo correctamente a los datos. Generad una partición de entrenamiento, validación y test (80 %/10 %/10 %) que sean estratificadas (fijad también el estado del generador de números aleatorios para la reproducibilidad).

Aplicad reducción de dimensionalidad a los conjuntos de datos usando t-SNE. ¿Se puede ver alguna separabilidad entre los datos en la proyección en 2D? Comentad los resultados.

b) Empezaremos aplicando un MLP a estos datos. En el código del notebook tenéis una función que permite generar modelos MLP variando el número de capas y la función de activación. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` cómo se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

En este caso no haremos validación cruzada y usaremos la partición de validación para calcular la calidad de modelo. Para pasarle los datos a un modelo de `torch` hay que definir

[2]Fijaos que este modelo tiene un parámetro `max_features` que te permite escoger el tamaño del vocabulario.

una clase del tipo `Dataset`, tenéis una definida en el notebook a la que hay que pasarle la matriz de datos y el vector con las etiquetas. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), podéis ver un ejemplo en el notebook.

´Para entrenar los modelos usaremos la función `train_loop` que tenéis en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación, la paciencia para la terminación temprana y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, podéis ver en el notebook como generar el objeto del optimizador.

Entrenad diferentes redes de una capa oculta de tamaños 8, 16, 32 y 64 usando como función de activación `Sigmoid`. ¿Cuál es el número de parámetros que tienen estas redes? Para crear la red solo tenéis que crear un objeto de la clase `MLP` pasándole los parámetros adecuados. Al generar el objeto tendréis que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no lo hacéis torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Veréis que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usad esta información para comprobar el número de épocas que realmente se realizan antes de parar. Podéis dejar la paciencia en su valor por defecto (20) y el número de épocas (1000). Veréis que el error en el conjunto de entrenamiento es menor que en validación, es normal, sin ninguna regularización la red neuronal se sobreajustará a los datos de entrenamiento. Comprobad el sobreajuste del modelo con los datos de test.

Evaluad los resultados. Para ello podéis usar la función `test_model` que tenéis en el notebook. Se le ha de pasar el modelo entrenado y un conjunto de datos y retornará las etiquetas de las predicciones y las reales, a partir de ahí podéis usar las funciones de `scikit learn` para obtener las medidas de evaluación. Observad también el comportamiendo de las redes durante el entrenamiento. ¿Tardan todas lo mismo? ¿Tienen el mismo valor final en la función de pérdida?

c) El modelo `ibm-granite/granite-embedding-30m-english`[3] es un modelo entrenado por IBM para hacer recuperación de información que es de un tamaño pequeño (30 millones de parámetros). Podemos transformar los documentos en vectores usando la librería `sentence-transformers`. En el capítulo 3 de esta lista de problemas tenéis todo el proceso para bajaros y usar el modelo con esta librería. Veréis que obtendréis matrices de datos con 384 dimensiones.

Aplicad reducción de dimensionalidad a los datos transformados usando t-SNE y comentad las diferencias con lo que habéis obtenido con TF-IDF.

Ajustad un modelo MLP de una capa igual que en el apartado anterior. Evaluad los resultados de estos modelos igual que antes y comparadlos con los del apartado anterior.

d) El modelo de IBM esta preparado para hacer recuperación de información en diálogos. El modelo `google/embeddinggemma-300m`[4] es un modelo multipropósito con diez veces más parámetros que tiene un comportamiento especializado dependiendo del prefijo que se le añada al texto. Para hacer clasificación de textos deberemos añadir a cada documento la cadena "`task: classification | query: `". En este modelo podemos transformar los datos en vectores de diferentes tamaños usando el parámetro `truncate_dim` en la función de transformación tal como se explica en el apéndice. Usad los tamaños 128 y 768.

Aplicad reducción de dimensionalidad a los datos transformados usando t-SNE y comentad las diferencias con lo que habéis obtenido con las transformaciones anteriores.

[3] https://huggingface.co/ibm-granite/granite-embedding-30m-english
[4] https://huggingface.co/google/embeddinggemma-300m

Ajustad un modelo MLP de una capa igual que en los apartados anteriores. Evaluad los resultados de estos modelos igual que antes y comparadlos con los de los apartados anteriores.

3. **El aire que respiramos**

   Barcelona es una ciudad que se preocupa por su aire, para ello tiene estaciones de medición en diferentes puntos de la ciudad. La estación de Palau Reial permite obtener datos horarios de concentraciones de diferentes contaminantes, entre ellos NO2, O3, SO2, PM2.5 y PM10. Utilizaremos datos que corresponden a los años 2022 y 2023 para obtener un modelo con redes neuronales que permita hacer predicción a una hora del NO2. Podéis obtener los datos mediante la función `load_BCN_air` de la librería `apafib`. Esta función retornará una matriz de datos con las series temporales para cada variable. Resolved los siguientes apartados ilustrando los resultados de la manera que os parezca más adecuada.

   Para resolver los siguientes apartados tenéis un notebook con una serie de funciones que os servirán para resolverlos.

   a) Primero haremos una limpieza de los datos. Veréis que existen datos perdidos en todas las variables. Dado que son series temporales que tienen una periodicidad horaria, los valores faltantes no deberían distar mucho de los más cercanos. Una opción para imputarlos es usar el método `interpolate` de los data frames de Pandas. Podemos usar interpolación lineal, de manera que cada dato faltante sea el valor de una interpolación lineal entre los extremos.

   Para generar los datos necesitaremos ventanas temporales de cierta longitud. `Pandas` permite generar una copia de una tabla de datos desplazada una serie de instantes temporales usando el método `shift`. Intentaremos hacer la predicción a partir de los datos de las últimas seis horas, así que tendréis que aplicar la operación para obtener seis matrices desplazadas y concatenarlas con los datos originales Eliminad todos los valores perdidos que ha generado esta transformación. Partid el conjunto de datos en entrenamiento, validación y test (70 %/15 %/15 %). Descartad las ventanas entre las particiones que tienen instantes compartidos para asegurar que las particiones sean independientes. Normalizad las particiones para que estén en el rango 0-1 de manera adecuada.

   Para poder tratar los datos con las redes neuronales para cada partición de datos tendréis que extraer la matriz de datos, separar las columnas del último instante temporal del resto, obtener solo la primera columna del último instante temporal (la variable NO2) y transformar la matriz de datos que usaremos para predecir de manera que tenga la forma `[ejemplos, variables, tiempo]` (tendréis que cambiar las dimensiones e intercambiar los dos últimos ejes entre si).

   Ahora ya tenéis tres conjuntos de datos que podéis usar para entrenar modelos (entrenamiento, validación y prueba).

   b) Tenemos un problema de regresión, podemos usar un MLP con varias capas ocultas que tenga una salida lineal para hacer la predicción y ajustadlo usando mínimos cuadráticos (mean squared error). En el código del notebook tenéis una función que permite generar modelos MLP variando el número de capas y la función de activación. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` cómo se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

   Para pasarle los datos a un modelo de `torch` hay que definir una clase del tipo `Dataset`, tenéis una definida en el notebook a la que hay que pasarle la matriz de datos y los valores a predecir. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar

el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), podéis ver un ejemplo en el notebook.

Para entrenar los modelos usaremos la función `train_loop` que tenéis en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación, la paciencia para la terminación temprana y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, podéis ver en el notebook como generar el objeto del optimizador.

Entrenad diferentes redes de 3 capas con diferentes tamaños usando como función de activación `Sigmoid` y `ReLU`. Para los tamaños podéis elegir el tamaño de la primera capa oculta y después ir reduciéndolo a la mitad para las capas sucesivas. Para crear la red solo tenéis que crear un objeto de la clase `MLP` pasándole los parámetros adecuados. Al generar el objeto tendréis que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no lo hacéis torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Veréis que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usad esta información para comprobar el número de épocas que realmente se realizan antes de parar. El sobreajuste lo podéis comprobar con la particion de test. Podéis dejar la paciencia en su valor por defecto.

Evaluad los resultados. Para ello podéis usar la función `test_model` que tenéis en el notebook. Se le ha de pasar el modelo entrenado y un conjunto de datos y retornará las etiquetas de las predicciones y las reales, a partir de ahí podéis usar las funciones de `scikit learn` para obtener las medidas de evaluación.

c) Las redes convolucionales también se pueden usar para predecir series temporales. Usaremos redes que usan convoluciones en una sola dimensión, de manera que las variables son los canales y la secuencia los valores. Tenéis en el notebook una clase que permite definir redes convolucionales para los datos del problema. Entrenad redes de dos capas que tengan kernels de tamaños 3 y 5 con diferentes números de kernels convolucionales, para ello, elegid el número de kernels para la primera capa y dobladlo para la segunda. Podéis evaluar la sobre especialización del modelo y evaluarlo de la misma manera que en el apartado anterior. Comparad los resultados.

d) A veces nos interesa predecir a más de una hora vista, pongamos que nos interesa predecir las próximas tres horas a partir de las seis anteriores. Modificad adecuadamente la clase que encapsula los datos (veréis que añade artificialmente una dimensión para predecir un vector de una columna). Modificad la clase que implementa el MLP para que pueda hacer tres predicciones en lugar de una. Entrenad redes de tres capas con diferentes tamaños usando la función de activación `ReLU`. Evaluad adecuadamente los resultados.

CAPÍTULO 2

Problemas Individuales

Al resolver el problema explicad bien los que hacéis, no hacer ningún comentario o hacer comentarios superficiales tendrán una nota más baja.

Tenéis que mostrar que habéis entendido los métodos que estáis aplicando, así que un corta y pega de problemas similares no es suficiente.

Para obtener los datos para algunos de estos problemas necesitaréis instalaros la última versión de la librería `apafib`. La podéis instalar localmente haciendo:

```
pip install -user -upgrade apafib
```

Para usar las funciones de carga de datos solo tenéis que añadir su importación desde la librería, en vuestro script o notebook, por ejemplo

```
from apafib import load_potability
```

La función os retornara un `DataFrame` de Pandas o arrays de numpy con los datos y la variable a predecir, cada problema os indicará que es lo que obtendréis.

Aunque hagáis los experimentos en Colab necesitaréis cierto tiempo para obtener los resultados, así que no esperéis hasta el último momento para hacerlos.

1. **De esta agua no beberé**

   El determinar si el agua que bebemos es potable es un problema bastante importante. Esto se puede comprobar a partir de un conjunto de características del agua que se pueden analizar fácilmente. Vamos a tratar un conjunto de datos que contiene los análisis de muestras de agua recolectados en diferentes puntos de un país para decidir si es potable o no.

   Puedes obtener estos datos mediante la función `load_potability` de la librería `apafib`. Resuelve los siguientes apartados ilustrando los resultados de la manera que te parezca más adecuada.

a) El conjunto de datos tiene una serie de valores perdidos que son muy difíciles de imputar, elimina del conjunto de datos esos ejemplos. Divide el conjunto de datos en entrenamiento y test (80 %/20 %). Haz una exploración del conjunto de datos de entrenamiento observando las relaciones con la variable objetivo. Estandariza las variables adecuadamente tanto para el conjunto de entrenamiento como para el de test.

Aplica Análisis de Componentes Principales (PCA) al conjunto de entrenamiento y visualízalo en 2D representando la variable objetivo. Observa la distribución de la variancia en los componentes. ¿Crees que puede haber una relación entre las variables del conjunto de datos y la variable objetivo? ¿Podría ser la relación entre las variables y la variable objetivo no lineal?

b) Para tener un valor base primero ajusta un modelo Naïve Bayes y una regresión logística a los datos. ¿Te parecen suficientemente buenos los resultados? Evalúa adecuadamente los modelos.

c) Las redes neuronales nos permiten obtener combinaciones no lineales de los datos que podrían capturar las interacciones entre las variables. Ajusta ahora un MLP con una capa oculta explorando sus hiperparámetros adecuadamente. Compara los resultados con los modelos anteriores.

d) Las redes neuronales no son buenas calculando las probabilidades y suelen mostrar excesiva confianza en la respuesta. En este problema parece más importante el no equivocarse en las fuentes de agua no potable, podemos obtener diferentes versiones del clasificador MLP manipulando como se toman las decisiones en el clasificador para ver que efecto tienen en el resultado.

Una primera opción es utilizar el modelo `TunedThresholdClassifierCV` que ajusta por validación cruzada la probabilidad que se usa para asignar los ejemplos a cada clase. Este recibe un clasificador preentrenado (el MLP con los hiperparámetros del apartado anterior) y decide esa probabilidad optimizando una medida de calidad. Usa como medidas de calidad `accuracy` y `roc_auc`.

Otra posibilidad es calibrar las probabilidades del MLP para que su confianza en las respuestas se distribuya más homogéneamente. Esto lo realiza el modelo `CalibratedClassifierCV` que hace validación cruzada para obtener ese calibrado. Usa para el calibrado el método `isotonic`.

Por último, podemos rizar el rizo ajustando el valor límite de probabilidad para tomar la decisión en el clasificador para el clasificador calibrado que acabamos de ajustar.

Compara adecuadamente los resultados de estos clasificadores y comenta sus ventajas e incovenientes.

2. **Sonríe**

En los inicios de las cámaras digitales de fotos[1] a un fabricante se le ocurrió que sería una buena idea el tener una cámara que detectara cuando se estaba sonriendo (podéis buscar la patente). Lo que entonces era un proceso más o menos algorítmico ahora se puede obtener una mejor precisión con una red neuronal. Vamos a utilizar un pequeño conjunto de imágenes (pequeñas) para entrenar un modelo que sea capaz de hacer esta detección.

Puedes obtener estos datos mediante la función `load_smile` de la librería `apafib`. Resuelve los siguientes apartados ilustrando los resultados de la manera que te parezca más adecuada.

Para resolver los siguientes apartados tienes un notebook con una serie de funciones que te servirán para resolverlos.

[1] Los teléfonos móviles acabaron con ellas.

a) Los datos corresponden a imágenes de 32×32 píxeles en colores RGB (cada color es una matriz 2D). Los datos ya estan normalizados a la escala [0,1]. Para poder entrenar el modelo primero partiremos el conjunto de datos en tres, el conjunto de entrenamiento propiamente dicho, un conjunto de validación y un conjunto de test (70 %/15 %/15 %), ya que no haremos validación cruzada. Aplica PCA y t-SNE a los datos de entrenamiento y represéntalos en 2D. ¿Se puede ver separabilidad entre las clases? Comenta el resultado.

b) Usaremos una red convolucional para entrenar el clasificador. En el código del notebook tienes una función que permite generar modelos de red convolucional variando diferentes parámetros. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` como se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

Para pasarle los datos a un modelo de `torch` hay que definir una clase del tipo `Dataset`, tienes una definida en el notebook a la que hay que pasarle la matriz de datos y el vector con las etiquetas. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), podéis ver un ejemplo en el notebook.

Para entrenar los modelos usaremos la función `train_loop` que tenéis en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación, la paciencia para la terminación temprana y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, puedes ver en el notebook como generar el objeto del optimizador.

Empezaremos explorando el número y el tamaño de las capas y el tamaño del kernel de convolución. Entrena diferentes redes de 2 y 3 capas con tamaños para la primera capa 2, 4 y 8. Para el resto ve doblando el tamaño de la primera capa para las capas sucesivas. Para los tamaños del kernel de convolución usa 3 y 5. Para crear la red solo tienes que crear un objeto de la clase `convolutional` pasándole los parámetros adecuados. Al generar el objeto tienes que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Verás que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usa esta información para comprobar el número de épocas que realmente se realizan antes de parar. La sobre especialización de la red la puedes comprobar con el conjunto de test. Puedes dejar la paciencia en su valor por defecto.

c) Verás en la implementación de la red que se usan una capa de pooling. Estas capas reducen el tamaño de la entrada haciendo una operación sobre grupos de valores vecinos. Por defecto usa `MaxPool2d` reduciendo el tamaño de la entrada a la cuarta parte (la mitad en dos dimensiones) usando el valor máximo de activación de cuatro valores contiguos. Otra posibilidad es usar la media de los valores con la capa `AvgPool2D`. Ajusta redes con el número de capas y el tamaño de kernel que te dio mejor resultado en el apartado anterior usando esta capa de pooling. Puedes evaluar la sobre especialización del modelo y evaluarlo de la misma manera que en el apartado anterior. Compara los resultados.

d) Habrás visto que las implementaciones de las redes tienen dos capas opcionales `BatchNorm2D` y `Dropout`. La primera utiliza estadísticas de las activaciones calculadas a partir de los datos para normalizarlas durante el entrenamiento, esto estabiliza el entrenamiento y actúa como una regularización. La segunda hace cero de manera aleatoria las activaciones que le llegan. Esto evita que la red dependa específicamente de alguna de las entradas que le llegan a las capas y actúa también como una regularización.

Repite el entrenamiento del apartado anterior escogiendo la capa de pooling que funcionó mejor activando `BatchNorm2D`. Haz lo mismo usando la capa de dropout usando como valores 0.05 y 0.01. Tendrás que aumentar la paciencia en el entrenamiento un poco para

estos experimentos para que no acaben enseguida. Ahora podemos ver el efecto de combinar las dos capas, ajusta las redes usando `BatchNorm2D` y el mejor valor de dropout. Compara los resultados y comenta las diferencias que hayas visto en como evoluciona la función de pérdida usando estas regularizaciones y el efecto en el número de épocas que hace la red.

3. **De esta nos hacemos ricos**

   Las finanzas son un área de aplicación del aprendizaje automático y las redes neuronales. Esta no es sencilla dados los múltiples factores que intervienen y lo difícil que es representarlos y estimarlos adecuadamente, pero a veces se pueden hacer estimaciones que dan una idea del comportamiento de los datos.

   En este problema trabajaremos con datos reales del NASDAQ sobre la cotización de las acciones de cinco empresas tecnológicas (Google, Microsoft, Apple, Intel y AMD) durante cinco años. Puedes obtener los datos mediante la función `load_Google` de la librería `apafib`. Esta retornará un dataframe que tiene tres columnas para cada acción, el valor final de la acción al final del día (P), el volumen de acciones que se intercambiaron (V) y la diferencia entre el mayor y el menor precio al que cotizaron en el día (GAP). En este problema intentaremos predecir la evolución del precio de la acción de Google (`GOOGLE-P`) usando sus valores y los de las otras acciones.

   Resuelve los siguientes apartados ilustrando los resultados de la manera que te parezca más adecuada.

   Para resolver los siguientes apartados tienes un notebook con una serie de funciones que te servirán para resolverlos.

   a) Haz un estudio de las características de los datos calculando sus estadísticas, la correlación entre los datos y representándolos de la manera que te parezca interesante. ¿Crees que puede ser posible predecir unas variables a partir de otras?

   Divide los datos en conjunto de entrenamiento (los 1000 primeros días), otro de validación (100 días) y test (el resto). Estandariza los datos.

   Para generar los datos para los modelos necesitaremos ventanas temporales de cierta longitud. `Pandas` permite generar una copia de una tabla de datos desplazada una serie de instantes temporales usando el método `shift` . Genera un conjunto de datos con longitud de ventana 8 de manera que puedas predecir un día a partir de las variables de los siete anteriores (fíjate que también tendremos los días anteriores de todas las variables). Elimina todos los valores perdidos que ha generado esta transformación y deja solo la variable GOOGLE-P del último día, que es la que tienes que predecir.

   Para pasarle los datos a un modelo de `torch` hay que definir una clase del tipo `Dataset`, tienes una definida en el notebook del problema a la que hay que pasarle la matriz de datos y el vector con los valores a predecir. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), puedes ver un ejemplo en el notebook.

   b) Tenemos un problema de regresión, podemos usar un MLP que tenga una salida lineal para hacer la predicción con varias capas ocultas y ajustarlo usando mínimos cuadráticos (mean squared error). En el código del notebook tienes una función que permite generar modelos MLP variando el número de capas y la función de activación. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` como se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

   Para entrenar los modelos usaremos la función `train_loop` que tienes en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación,

la paciencia para la terminación temprana y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, puedes ver en el notebook como generar el objeto del optimizador.

Entrena diferentes redes de 1 y 2 capas con diferentes tamaños usando como función de activación `Sigmoid` y `ReLU`. Para los tamaños de las dos capas puedes elegir el tamaño de la primera capa oculta y después reducirlo a la mitad para la segunda capas. Explora también la tasa de aprendizaje.

Para crear la red solo tienes que crear un objeto de la clase `MLP` pasándole los parámetros adecuados. Al generar el objeto tendrás que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Verás que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usa esta información para comprobar el número de épocas que realmente se realizan antes de parar. La sobre especialización del modelo la puedes comprobar con el conjunto de test.

Evalúa los resultados usando MAE y MSE. Para ello puedes usar la función `test_model` que tienes en el notebook. Se le ha de pasar el modelo entrenado y un conjunto de datos y retornará las predicciones y las reales, a partir de ahí puedes usar las funciones de `scikit learn` para obtener las medidas de evaluación.

c) Existen muchas funciones de activación más allá de las clásicas. Experimenta con las funciones de activación `LeakyReLU` y `SiLU` con la red con el número de capas que haya tenido mejores resultados explorando el tamaño de las capas. Compara la calidad de los modelos.

d) Una cosa que se nos ocurriría es que información sobre la fecha, como el día de la semana, la semana del año, o el mes podrían tener una influencia en las cotizaciones. Verás que el índice del `DataFrame` de los datos es la fecha. Extrae estos valores y vuelve a generar las ventanas para añadírselos a los datos (tendremos ventanas que tienen toda la secuencia de fechas). Ajusta el modelo de red con el número de capas y función de activación que mejor haya funcionado explorando diferentes tamaños en las capas. Compara los resultados.

e) Un modelo base que podemos utilizar para saber si realmente estamos prediciendo algo es el llamado modelo de persistencia. Para este problema sería usar el valor de la acción de Google del día anterior como predicción. Calcula el MAE y el MSE para este modelo y compáralo con los resultados que has obtenido con el MLP.

4. **Ajuste de salario**

En problemas de regresión habitualmente asumimos que hay una parte estocástica en el valor que estimamos que sigue una distribución que tiene una varianza fija. Esto es conocido como *homocedasticidad*. En la realidad esto no tiene por qué ser cierto y hay problemas en los que la variablidad del ruido depende del dato específico que predecimos, tenemos en este caso *heterocedasticidad*. Esto se puede capturar mejor si usamos modelos que pueden adaptarse a este ruido cambiante. Vamos a utilizar una versión de un conjunto de datos de salarios que se utiliza habitualmente en libros de econometría para ilustrar este fenómeno. Podéis obtenerlo mediante la librería `apafib` empleando la función `load_wages`. El objetivo es predecir la variable `wage` usando el resto, que corresponden a variables socioeconómicas.

Para resolver los siguientes apartados tienes un notebook con una serie de funciones que te servirán para resolverlos.

a) Vamos a dividir el conjunto de datos en tres partes, el conjunto de entrenamiento (80 %), el conjunto de validación (10 %) y el conjunto de test (10 %). En este caso el ajuste de hiperparámetros y la calidad del modelo se obtendrá mediante el conjunto de validación

y comprobaremos la sobre especialización con el conjunto de test (que no se ha usado en ningún momento para tomar decisiones sobre el modelo).

Comprueba la diferente variabilidad de la variable objetivo representándola respecto a las variables del conjunto de datos que no son discretas, ni ordenadas. Fíjate especialmente en las variables que deberían tener una relación creciente con el salario. Aplica un escalado min-max al conjunto de datos.

b) Tenemos un problema de regresión, podemos usar un MLP que tenga una salida lineal para hacer la predicción con varias capas ocultas y ajustarlo usando mínimos cuadráticos (mean squared error). En el código del notebook tienes una función que permite generar modelos MLP variando el número de capas y la función de activación. Se le puede indicar que no se modela la varianza pasando `False` en el parámetro `var`. La red se define como una clase python donde en el método `__init__` definimos los elementos de la red y en el `forward` como se hace la propagación hacia adelante en la red (la propagación hacia atrás se calcula automáticamente).

Para pasarle los datos a un modelo de `torch` hay que definir una clase del tipo `Dataset`, tienes una definida en el notebook a la que hay que pasarle la matriz de datos y el vector con los valores reales que les corresponden. Esta clase se ha de pasar a un objeto `Dataloader` que se encarga de organizar el uso de los datos durante el entrenamiento, entre otras cosas de partir los datos en grupos (`batch_size`), podéis ver un ejemplo en el notebook.

Para entrenar los modelos usaremos la función `train_loop` que tienes en el notebook. Esta función recibe el modelo, el optimizador a usar, los datos de entrenamiento y validación, la paciencia para la terminación temprana y el número de iteraciones que hará el entrenamiento. Para el optimizador usaremos `adam`, puedes ver en el notebook como generar el objeto del optimizador.

Entrena diferentes redes de 1 y 2 capas ocultas con diferentes tamaños usando como función de activación `nn.Sigmoid` y `nn.ReLU`. Para los tamaños de las dos capas puedes elegir el tamaño de la primera capa oculta y después reducirlo a la mitad para la segunda capas.

Para crear la red solo tienes que crear un objeto de la clase `MLP` pasándole los parámetros adecuados. Al generar el objeto tendras que subir el modelo a la GPU usando el método `modelo.to('cuda')` (si no torch se quejará de que los datos y el modelo no están en el mismo dispositivo). Verás que la función de entrenamiento retorna la historia de la función de pérdida para el entrenamiento y validación. Usa esta información para comprobar el número de épocas que realmente se realizan antes de parar. Puedes evaluar la sobre especialización de la red con el conjunto de test.

Evalúa los resultados usando MAE y MSE. Para ello puedes usar la función `test_model` que tienes en el notebook. Se le ha de pasar el modelo entrenado y un conjunto de datos y retornará las predicciones y los valores reales, a partir de ahí puedes usar las funciones de `scikit learn` para obtener las medidas de evaluación.

c) El ajuste por error cuadrático, como vimos en el tema de regresión lineal, asume que el ruido es gausiano y su varianza es la misma para todas las predicciones, por lo que solo es necesario aprender una función que prediga su media. El modelo que tienes en el notebook permite indicar que se quiere predecir también la varianza de los datos. En este caso tenemos dos salidas, una para la media y otra para la varianza.

Fijate que ahora la función de perdida no ha de ser el error cuadrático sino la función `RegressionLoss`. Esta función recibe las dos salidas de la red. Explica qué está prediciendo exactamente la red (qué valores de salida se calculan) y la función de pérdida, y como encajan con el cálculo de la log verosimilitud negativa de la gausiana cuando la varianza no es fija[2]. En otras palabras, mírate cuál es la fórmula de la log verosimilitud de la

[2]Puedes repasar el cálculo de la log verosimilitud negativa de la distribución gausiana en el material del primer tema

gausiana en este caso e identifica los cálculos en la función `RegressionLoss`.

Ajusta las mismas redes que has usado en el apartado anterior y selecciona la que dé el mejor resultado. Representa los residuos de las dos regresiones y las predicciones contra los valores reales. Compara los resultados que se han obtenido los dos modelos. Representa las predicciones que has obtenido con los dos modelos sobre los valores reales para las variables `exper` y `age`. Comenta las diferencias.

d) En el apartado anterior al ajustar la red hemos obtenido también la varianza de la predicción de la variable objetivo (el segundo valor que predice la red). Representa la **varianza** que se predice para el conjunto de test respecto al valor de la variable objetivo (fíjate en lo que predice la red en este segundo valor según lo que has respondido sobre el cálculo de la log verosimilitud). Comenta la relación entre los valores de la variable objetivo y la varianza predicha.

del curso.

CAPÍTULO 3

Uso de modelos de Hugging Face

Hugging Face es entre otras cosas un repositorio de modelos de aprendizaje profundo en donde se puede acceder a modelos preentrenados para muchas tareas diferentes.

La mayoría de los modelos se pueden descargar libremente, pero algunos necesitan estar dado de alta en Hugging Face y obtener un token de acceso. Este es el caso de los modelos del problema de grupo número 2.

Una vez estéis dados de alta, podéis encontrar la creación de tokens de acceso (*access token*) en el perfil de usuario. Tendréis también que acceder a la página de los modelos que queréis usar que no sean de libre acceso y aceptar sus condiciones.

Para acceder a los modelos desde un notebook tendréis que tener instalada la librería `huggingface_hub` y hacer en una celda del notebook:

```
!hf auth login --token TOKEN_DE_ACCESO
```

No os dejéis la exclamación del principio. **Cuando entreguéis el notebook borrad el token de acceso.**

3.1. Uso de los modelos para embeddings de texto

Los modelos de embedding de texto se pueden usar mediante la librería `sentence-transformers`. Esta se encarga de procesar el texto y pasarlo por el modelo.

Para cargar un modelo, por ejemplo `embeddinggemma`, tendréis que hacer:

```
model = SentenceTransformer("google/embeddinggemma-300m")
```

A partir de aquí podéis transformar una lista de documentos mediante:

```
docs_e = model.encode_query(docs, normalize_embeddings=True)
```

Donde `docs` es una lista con los documentos a transformar. El parámetro `normalize_embeddings` hace que todo vector este normalizado por su norma cuadrada (normalización l2).

Si el modelo permite obtener transformaciones de diferentes tamaños se le puede pasar a la función de codificación el parámetro `truncate_dim`, obteniendo así el tamaño deseado:

```
docs_e = model.encode_query(docs, truncate_dim=XXX, normalize_embeddings=True)
```

Esta matriz que obtenemos la podemos usar para crear el objeto `Dataset` correspondiente que luego usaremos en un `Dataloader`.