2023/2024 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 2023

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.

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.

La parte que no es de programación la podéis hacer a mano y escanearla a un archivo **PDF**. Comprobad que **sea legible**.

Para la parte de programación podéis entregar los resultados como un notebook (Colab/Jupyter). Alternativamente, podéis hacer un documento explicando los resultados como un PDF y un archivo python con el código

También, si queréis, podéis poner las respuestas a las preguntas en el notebook, este os permite insertar texto en markdown y en latex.

Objetivos:

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

Estos problemas se han de resolver utilizando la librería Keras que permite utilizar la librería de redes neuronales TensorFlow de una manera más amigable. Esto significa que deberéis hacer la experimentación usando Google Colab para poder hacerla en un tiempo razonable.

En el capítulo 3 tenéis una guía mínima que explica el funcionamiento de esta librería para la definición y el entrenamiento de redes.

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

1. **Predecirme a mí mismo**

   Una de las cosas interesantes de las redes neuronales es que se pueden usar fácilmente para muchas tareas además de la clasificación y la regresión clásicas, dado que pueden representar cualquier tipo de función. Una posibilidad es implementar la función de identidad, básicamente una red que es capaz de predecir su entrada.

   Obviamente, la idea no es solo copiar la entrada, sino aprender una representación que pueda usarse para otros fines. Este tipo de red se llama *auto-codificador* (*autoencoder*). La arquitectura de dicha red crea un cuello de botella en las capas ocultas, reduciendo el número de neuronas en cada capa hasta alcanzar un mínimo que corresponde a la dimensionalidad del espacio de representación de destino y luego va haciendo crecer el tamaño de las capas en orden inverso hasta tener una capa de salida que tiene la dimensionalidad de la entrada

   El objetivo de este problema es implementar un auto-codificador y probarlo con el conjunto de datos `dígits` que forma parte de los conjuntos de datos de scikit-learn (*load_digits*). Estos son dígitos escritos a mano transformados en imágenes de tamaño 8×8 representados como vectores de 64 elementos con valores en el rango [0,15] (escala de grises).

   El siguiente código implementa un auto-codificador usando keras/tensorflow:

```
act = 'linear'
entrada = keras.Input(shape=(64,))
```

```
codificar = keras.layers.Dense(d1, activation=act)(entrada)
codificar = keras.layers.Dense(d2, activation=actuar)(codificar)
decodificar = keras.layers.Dense(d1, activation=actuar)(codificar)
decodificar = keras.layers.Dense(64, activation=actuar)(decodificar)

autocodificador = keras.Model(entrada, decodificar)
codificador = keras.Model(entrada, codificar)
```

La red tiene capas de entrada y salida de tamaño 64 (el tamaño de los dígitos de ejemplo) y tres capas ocultas densas (MLP), dos capas gemelas con un tamaño más pequeño que la entrada y una capa intermedia con un tamaño de la dimensionalidad del espacio de codificación. Por ejemplo (16,2,16) será una red que primero reduce la dimensionalidad a un 25 % de la entrada, obtiene un espacio bidimensional y luego aumenta la dimensionalidad hasta volver a la dimensionalidad de los datos.

Para entrenar la red solo tenéis que definir el optimizador y la funcion de pérdida y luego ajustar los datos de la siguiente manera:

```
autocodificador.compile(optimizer=keras.optimizers.Adam(),
                        loss=keras.losses.MeanSquaredError())
autocodificador.fit(X_train, X_train,batch_size=BATCH_SIZE,
                            epochs=EPOCHS, verbose=False)
```

La pérdida será el error al cuadrático, el optimizador adam. Podéis ver que al ajustar el auto-codificador, la entrada y la salida son los mismos datos. El tamaño de `BATCH` y el número de `EPOCHS` son hiperparámetros. Una vez entrenamos el `autocodificador`, la red `codificador` (que es solo la primera parte del auto-codificador) se puede usar para transformar los datos al espacio de menor dimensionalidad.

Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

a) Cargad el conjunto de datos de dígitos, dividid los datos en entrenamiento y test (70 %/30 %) y normalizad los datos a la escala [0,1] adecuadamente (**pensad** que son imágenes).

b) Fijad el tamaño de la capa intermedia a 2 y experimentad con algunos tamaños para las otras capas ocultas (siempre una potencia de 2) usando como funciones de activación `linear`, `sigmoid` y `relu`. Para mantener bajo el tiempo de entrenamiento, el tamaño del lote puede ser grande (alrededor de 100), el número de épocas debe ser del orden de miles, usad los experimentos con la función de activación lineal para fijar estos valores.

Calculad el error cuadrático (MSE) final después de entrenar las redes para los datos de entrenamiento y los datos de test como la media de las diferencias cuadráticas entre los datos y las predicciones.

- Para cada función de activación, para la red con el mejor error cuadrático, representad las predicciones 2D para los datos de entrenamiento usando la red `codificador`[1]. **Fijaos** en que la red ya esta entrenada, ya que forma parte de la red completa. ¿Alguna de las funciones de activación ayuda a la separabilidad de las clases en 2D? Calculad el PCA de los datos y comparad los resultados de los primeros 2 componentes con los resultados del codificador. ¿Alguna de las redes produce una salida similar? Buscad

[1] La clase `keras.Model` tiene un método `predict`.

en Google la relación entre PCA y auto-codificadores y explicad brevemente lo que encontréis.

- Calculad la codificación para algunos ejemplos de los datos de test utilizando la mejor red y representad los resultados. ¿La salida es similar a la entrada?[2]

c) Aprender a reproducir la entrada también tiene otras aplicaciones como la corrección de errores. Un auto-codificador de eliminación de ruido (*denoising autoencoder*) se entrena con **una entrada corrupta para reproducir la entrada original**. Utilizad la mejor red de la pregunta anterior. Generad un nuevo conjunto de datos como una copia del **conjunto de entrenamiento** original y corromped estos datos cambiando posiciones aleatorias en los ejemplos a valores de una distribución uniforme [0-1]. Generad un conjunto de datos con 10 % y 30 % de valores corruptos.

- Entrenad el auto-codificador y calculad el error cuadrático de las predicciones de los datos **originales** para el entrenamiento y el test. ¿Es el error ahora mejor o peor? ¿Por qué?
- Concatenad dos copias de los datos de entrenamiento (duplicad los datos) y cread dos conjuntos de datos con 10 % y 30 % de corrupción respectivamente. Entrenad el auto-codificador con estos datos. ¿Es el error ahora mejor?
- Utilizad el codificador entrenado con los conjuntos de datos de entrenamiento corrupto doblado que tiene el mejor error cuadrático y transformad los datos de entrenamiento originales. Comparad la representación 2D con la del codificador original (el que no tiene ruido). ¿Son los datos más separables ahora?

2. **Redes neuronales con sentido de la moda**

El conjunto de datos Fashion MNIST es un conjunto de datos de referencia creado por Zalando[3]. Fue generado para tener un conjunto de datos no tan fácil como otros conjuntos de datos de referencia similares basados en dígitos escritos a mano. En este algunas clases son más difíciles de identificar que otras. El conjunto de datos contiene imágenes de 28×28 en escala de grises de diferentes tipos de ropa, zapatos y bolsos.

Vamos a trabajar con un subconjunto de los datos, específicamente con las clases camiseta (0), jersey (2) y camisa (6) y experimentar con clasificación y reducción de dimensionalidad.

Utilizaremos keras para implementar las redes neuronales.

Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

a) Podéis descargar el conjunto de entrenamiento y test de este conjunto de datos usando lo siguiente:

```
(x_train, y_train), (x_test, y_test) =
            keras.datasets.fashion_mnist.load_data()
```

Los resultados serán un conjunto de matrices `numpy`. Ahora deberéis seleccionar las clases 0, 2 y 6 para los datos de entrenamiento y test. Tendréis que cambiar las etiquetas para que correspondan a 0, 1 y 2. Normalizad los datos para que estén en el rango [0-1], solo necesitaréis dividir los datos entre 255. Como podéis ver, la división en entrenamiento y test ya está hecha.

[2] Para representar las predicciones y los datos, podéis transformar los datos a una matriz de 8×8 y usar la función `matshow` de `matplotlib`.

[3] Podéis encontrar una descripción completa en https://github.com/zalandoresearch/fashion-mnist.

b) Primero vamos a usar un MLP para clasificar los datos. El siguiente código define un MLP de 2 capas ocultas con tres salidas:

```
act = 'linear'
input = keras.Input(shape=(28,28))
model = keras.layers.Flatten()(input)
model = keras.layers.Dense(NN1, activation=act)(model)
model = keras.layers.Dense(NN2, activation=act)(model)
model = keras.layers.Dense(3, activation='softmax')(model)
nn = keras.Model(input, model)
```

Tendréis que experimentar con el número de neuronas para las dos capas ocultas (con NN2<NN1). No es necesario que sean un valor grande. Experimentad con estas funciones de activación `linear`, `sigmoid` y `relu`. Para entrenar la red podéis usar el siguiente código:

```
nn.compile(optimizer='adam',
           loss='sparse_categorical_crossentropy',
           metrics=['accuracy'])
error =  nn.fit(x_train, y_train,
                epochs=EPOCHS, verbose=False,
                validation_data=(x_test, y_test))
```

Esto usa el optimizador adam, como función de pérdida usa la entropía cruzada categórica (como en la regresión logística)[4] y mide la función de pérdida durante el entrenamiento junto al acierto. Agregar unos datos de test en el parámetro `validation_data` hace que también se mida función de pérdida con esos datos. Dado que el entrenamiento es bastante costoso, no vamos a hacer validación cruzada y usaremos datos de test para decidir los mejores parámetros.

En cuanto al número de épocas, el entrenamiento devuelve un objeto que recoge la pérdida y el acierto de cada época. Podéis tomarla del diccionario que está almacenado en el atributo `history` de este objeto. Representad el acierto del entrenamiento y el test y decidid qué número de épocas es el más adecuado para entrenar la red. ¿Qué sucede cuando el número de épocas es grande?

Usad el `classification_report` de scikit learn para verificar qué tan buenos son los mejores modelos para cada función de activación usando los datos de test. La red tiene un método `predict` que devuelve las probabilidades de los ejemplos, podéis transformarlos en etiquetas utilizando la función numpy `argmax`. Comentad los resultados.

c) Cuando explicamos los métodos de reducción de dimensionalidad, todos menos uno fueron sin supervisión. El único que es supervisado (el discriminante de Fisher) es lineal. Explicamos que funcionaba proyectando los datos a unas pocas dimensiones optimizando su separabilidad. Las redes neuronales son lo suficientemente flexibles como para ayudarnos a obtener una representación de baja dimensionalidad de los datos casi gratis con solo un ajuste de la red. Vamos a insertar una capa densa con dos neuronas entre las dos capas ocultas de esta manera:

[4] La pérdida `sparse_categorical_crossentropy` solo le dice a Keras que tiene el número de la clase en lugar de una codificación one-hot, por lo que tiene que encargarse de la conversión.

```
model = keras.layers.Dense(NN1, activation=act)(model)
rdim = keras.layers.Dense(2, activation=act)(model)
model = keras.layers.Dense(NN2, activation=act)(rdim)
```

Cada capa de una red neuronal es básicamente una proyección a un espacio con dimensionalidad el número de neuronas de la entrada de la capa, estamos proyectando la primera capa oculta a 2D y luego proyectando nuevamente a la dimensionalidad de la última capa oculta. Para poder obtener la proyección necesitamos definir otro modelo como este:

```
ndim = keras.Model(input, rdim)
```

Ahora podéis entrenar de nuevo **la red completa** usando los mejores parámetros para cada una de las funciones de activación. Comprobad que el acierto de los modelos no hayan cambiado mucho. Podemos utilizar el modelo adicional (que se entrena al mismo tiempo) para predecir las coordenadas de los datos del test. Representad los datos proyectados empleando las clases como colores. ¿Cuál parece mejor? Calculad el PCA y comparad los resultados (tendréis que cambiar la forma de la matriz de datos para esto). Mirad la implementación de scikit-learn de los métodos de reducción de dimensionalidad no lineal que explicamos en clase. ¿Cuál permite usar la transformación que obtenemos de los datos de entrenamiento para aplicarla a los datos de test? Ajustad una transformación 2D con ese método empleando una muestra del 20 % de los datos de entrenamiento y aplicadla a los datos de test. Tendréis que explorar los hiperparametros de esta transformación para encontrar una transformación que parezca adecuada. Representad el resultado. ¿Es mejor esta transformación? ¿Por qué?

3. **Es un pájaro, es un avión...**

   CIFAR-10[5] es uno de los muchos conjuntos de imágenes que existen para poder hacer experimentos. Este conjunto tiene imágenes de 32×32 de 10 clases diferentes. Dos de esas clases son pájaro y avión. Uno pensaría que debería ser fácil el poder distinguir ejemplos de estas dos clases, el detalle es que muchas veces son imágenes en la que su mayor parte corresponde al cielo y el objeto puede ocupar una parte relativamente pequeña de la imagen. El objetivo de este problema es experimentar con diferentes redes que usan capas convolucionales para ver cual es el nivel de acierto al que se puede llegar.

   Utilizaremos keras para implementar las redes neuronales.

   Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

   a) Podéis descargar el conjunto de entrenamiento y test de este conjunto de datos usando lo siguiente:

   ```
   (X_train, y_train), (X_test, y_test) =
                          keras.datasets.cifar10.load_data()
   y_train = y_train.squeeze()
   y_test = y_test.squeeze()
   ```

   Los resultados serán un conjunto de matrices `numpy`. Ahora deberéis seleccionar las clases 0 (avión), 2 (pájaro) para los datos de entrenamiento y test. Tendréis que cambiar las etiquetas para que correspondan a 0 y 1. Normalizad los datos para que estén en el rango [0-1],

[5] CIFAR es el acrónimo del Canadian Institute For Advanced Research.

solo necesitaréis dividir los datos entre 255. Como podéis ver, la división en entrenamiento y test ya está hecha.

b) En redes neuronales la regularización es clave para que no se sobre especialicen. Existen muchas maneras de regularizar una red, una de ellas es usar capas de **dropout**, como os cuenta el capítulo 3 de este documento estas capas hacer que con cierta probabilidad algunas de las conexiones entre dos capas se hagan cero descartando la información que llega a través de esa conexión.

Esta es una red con una capa convolucional de filtros de tamaño 3×3 que tiene una capa de dropout antes de un MLP con dos capas ocultas. La capa de salida tiene una neurona con activación sigmoide que nos servirá para clasificación binaria.

```
model = keras.models.Sequential()
model.add(keras.Input(shape=(32, 32, 3)))
model.add(keras.layer.Conv2D(filters, (3, 3),
                             activation='relu'))
model.add(keras.layer.Flatten())
model.add(keras.layer.Dropout(drop))
model.add(keras.layer.Dense(64, activation='relu'))
model.add(keras.layer.Dense(32, activation='relu'))
model.add(keras.layer.Dense(1, activation='sigmoid'))
```

Podéis entrenar esta red con el siguiente código:

```
model.compile(optimizer='adam', loss='binary_crossentropy')
res = model.fit(X_train, y_train, epochs=100, batch_size=256,
                verbose=0, validation_data=(X_test, y_test),
                 callbacks=[early, ckpt])
```

Se está entrenando el modelo durante 100 épocas usando el optimizador adam con la función de pérdida entropía cruzada binaria. Los parámetros del callback corresponden a los objetos para hacer la terminación temprana y el guardado del modelo que tendréis que definir, podéis ver como se hace al final del capítulo 3.

Entrenad este modelo con una capa convolucional con 8 filtros y un valor de dropout de 0 a 0.8 en pasos de 0.2 (5 modelos). Tendréis que ir salvando cada modelo durante el entrenamiento y usar terminación temprana con una paciencia de 20.

Representad la evolución de la función de pérdida de la red en cada época para los datos de entrenamiento y de test, podéis obtener esa información dentro del campo `history` del objeto que retorna el entrenamiento (`loss`, `val_loss`).

Comentad la diferencia que hay en la evolución de la función de pérdida para los diferentes valores de dropout. ¿Cuál os parece que es mejor? ¿Por qué?

También se puede usar regularización en el MLP. Entrenad la red con un dropout de 0 y añadiendo regularización L_2 a las dos capas densas ocultas de esta manera:

```
model.add(keras.layer.Dense(XX, activation='relu',
                            kernel_regularizer='l2' ))
```

Haced lo mismo combinando un dropout de 0.8 y la regularización L_2. Comparad todas las evoluciones de la función de pérdida que habéis obtenido y calculad también los aciertos de las redes para el conjunto de entrenamiento y el de test (**tendréis que cargar el último modelo grabado de cada red para calcularlos**). Comentad los resultados.

c) La profundidad de una red da ciertas ventajas en problemas complejos como comentamos en teoría. Añadid sucesivamente a la red inicial después de la primera capa convolucional más capas convolucionales con número filtros de 16 a 128 siguiendo las potencias de 2 y entrenad las redes para cada tamaño (tendréis 4 redes más de 2 a 5 capas convolucionales). Entrenad de la misma manera que el apartado anterior usando un dropout de 0.8 y regularización L_2.

Calculad el acierto sobre los datos de entrenamiento y el conjunto de test y representad la evolución de la función de pérdida para los datos de entrenamiento y test de las redes. ¿Son mejores redes más profundas? No solo os fijéis en el acierto, mirad que está pasando con la función de pérdida y su evolución durante el entrenamiento.

d) El ir añadiendo más y más capas aumenta bastante el número de parámetros que tiene el modelo. Una manera de reducirlos es ir reduciendo poco a poco la salida de cada capa. En redes convolucionales se utilizan capas de *pooling* que reducen las dimensiones espaciales, por ejemplo haciendo un cálculo con las posiciones vecinas (media/máximo). Vamos a usar capas de max pooling interpuestas entre las capas convolucionales de la siguiente manera.

```
model.add(keras.layer.Conv2D(filters, (3, 3),
                             activation='relu'))
model.add(keras.layer.MaxPooling2D(pool_size=(2,2),
                                   strides=(1, 1),
                                   padding='valid'))
```

Estas capas calculan el máximo de posiciones adyacentes colocadas en cuadrículas 2×2 reduciendo a una cuarta parte los valores que se pasan a la siguiente capa. Entrenad redes como las del apartado anterior intercalando capas de max pooling (una por cada capa convolucional) igual que habéis hecho en el apartado anterior. Comparad la evolución de la función de pérdida y el acierto para los datos de entrenamiento y test.

Se puede calcular el número de parámetros que tiene una red mediante la funcion `summary` del modelo. Obtened los tamaños de las diferentes redes que habéis entrenado. ¿Con qué modelo final os quedaríais? ¿Por qué?

CAPÍTULO 2

Problemas Individuales

Para obtener los datos para algunos de estos problemas necesitaréis instalaros la última versión de la libreria `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_wages
```

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. **Me quedo con tu cara**

   Scikit learn simplifica mucho el uso de redes neuronales ocultando las cosas feas pero reduciendo las opciones que se pueden usar para definir arquitecturas más allá del MLP. Para obtener la flexibilidad necesaria para aplicaciones más complejas, debemos pasarnos a cualquiera de las librerías utilizadas para el aprendizaje profundo. La idea de este problema es probar cómo usar una de estas librerías para un problema de clasificación de imágenes.

   Las *Olivetti faces* son un conjunto de datos clásico de reconocimiento de imágenes. Es una colección de 400 imágenes de 40 personas diferentes de tamaño 64×64. El objetivo es obtener un clasificador capaz de etiquetar correctamente las imágenes.

   Vamos a utilizar la librería Tensorflow de Google a través de la API Keras, que simplifica la definición de arquitecturas de redes neuronales, pero no es tan simple como scikit learn.

   Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

   a) La primera tarea es obtener los datos. Esto se puede hacer usando la función de scikit-learn `fetch_olivetti_faces(return_X_y=True)` que devolverá dos matrices de datos, una

para los datos de entrada y otra para las etiquetas. Deberás dividir los datos en un conjunto de entrenamiento y test (70 %/30 %) y normalizar los datos para que estén en el rango [0-1] (simplemente divide los datos por el valor máximo de los píxeles).

Keras no distingue entre clasificación y regresión como lo hace scikit learn, dado que la tarea se define por la función de pérdida que se utiliza. Este es un problema multiclase por lo que la función de pérdida que corresponde es la entropía cruzada categórica. Los problemas multiclase, como hemos visto varias veces, se resuelven mediante una regresión multisalida que genera la probabilidad para cada clase mediante la función softmax.

Transforma las etiquetas del problema en una matriz con codificación one-hot utilizando la clase scikit learn `OneHotEncoder` (sin salida dispersa). Ahora tenemos el conjunto de datos listo para entrenar.

b) El perceptrón multicapa utiliza capas totalmente conectadas (densas) para realizar cálculos. El tamaño de un perceptrón para imágenes puede ser muy grande. Se puede usar PCA para reducir el número de dimensiones a algo más razonable. Aplica el PCA a las imágenes y genera dos conjuntos de datos utilizando los primeros 10 y 20 componentes (no hace falta que estandarices los datos).

El siguiente código define un perceptrón con una capa oculta con función de activación ReLU. Fíjate en que la capa de salida es un softmax de tamaño 40 que permite predecir las probabilidades de cada clase.

```
model = keras.Sequential()
model.add(keras.Input(shape=(INPUT_SIZE,)))
model.add(keras.layers.Dense(NEURONS, activation="relu"))
model.add(keras.layers.Dense(40, activation="softmax"))
```

Se puede entrenar este modelo usando la entropía cruzada categórica como función de pérdida mediante descenso de gradiente estocástico usando este código.

```
model.compile(optimizer=keras.optimizers.SGD(),
              loss=tf.keras.losses.CategoricalCrossentropy())
model.fit(X_train,y_train,batch_size=BATCH_SIZE,
          epochs=EPOCHS,verbose=False)
```

Entrena el perceptrón con los dos conjuntos de datos explorando tamaños para la capa oculta de 25 a 100 neuronas en pasos de 25 en 25 usando lotes de 16 ejemplos y 200 épocas. Calcula el acierto de los modelos. Para ello puedes utilizar el método `predict` del objeto `modelo`. Tendrás que transformar el vector de predicciones del softmax a etiquetas empleando la función `np.argmax`. Dado que no hacemos validación cruzada nos fijaremos en el acierto sobre los datos de entrenamiento y de test. Escoge el modelo que tenga el mejor resultado. ¿Un número mayor de neuronas en la red corresponde con un mejor modelo? ¿Por qué?

c) Una alternativa a los MLP son las capas convolucionales. Como hemos visto en clase, se trata de redes neuronales inspiradas en el funcionamiento de la corteza visual y especializadas en problemas de visión. Repasa lo que vimos en teoría para entender como funcionan.

Vamos a utilizar una capa convolucional para clasificar el conjunto de datos **original**. Como la entrada debe ser una matriz cuadrada, tendremos que transformar la forma de la matriz entrada. Además, las capas convolucionales se usan para procesar imágenes en color, por lo que asumen que cada imagen es una matriz 3D, la tercera dimensión es para los canales de color. En este caso, las imágenes son en escala de grises, por lo que tendremos

que simular que tenemos una dimensión adicional. Podemos usar `numpy` para transformar los datos de entrenamiento y test usando la función reshape (`X.reshape(-1,64,64,1)`), esto dará como resultado una matriz 4D, la primera dimensión son los ejemplos, las otras tres son la imagen de cada ejemplo.

En este caso vamos a explorar el número de neuronas (filters) de la capa convolucional fijando el paso de las convoluciones (stride) a 1 y el tamaño del kernel de las convoluciones a 3. Experimenta con un número de neuronas de 1 a 10.

```
model = keras.Sequential()
model.add(keras.Input(shape=(64,64,1,)))
model.add(keras.layers.Conv2D(filters=NEURONS, kernel_size=3,
                              strides=1, activation="relu"))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(40, activation="softmax"))
```

Calcula el acierto igual que en el apartado anterior y escoge el mejor modelo. Escoge cuál de los dos modelos (este y el del apartado anterior) te parece mejor.

d) El método `count_params()` de la clase `Model` calcula cuántos parámetros tiene la red y el método `summary` calcula los parámetros por capa. Dado cómo se definen las capas convolucionales y las densas, explica cómo se calculan los tamaños y la cantidad de parámetros de las capas en las redes. Compara y comenta los resultados que se obtienen en los dos tipos de redes que has definido y las ventajas/inconvenientes que tienen respecto a su número de parámetros. ¿Cuáles habrían sido los tamaños de las capas de la red MLP si hubiéramos usado los datos en su forma original?

2. **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 no lineales 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.

Vamos a utilizar la librería Tensorflow de Google a través de la API Keras, que simplifica la definición de arquitecturas de redes neuronales, pero no es tan simple como scikit learn.

Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

a) Vamos a dividir el conjunto de datos en dos partes, el conjunto de entrenamiento (70 %) y el conjunto de test (30 %), que usaremos para obtener la calidad final del modelo. Comprueba la diferente variabilidad de la variable objetivo representándola respecto a las variables del conjunto de datos que no son discretas. Aplica un escalado minmax al conjunto de datos.

b) El código que tienes a continuación corresponde a una regresión lineal por mínimos cuadrados sobre los datos usando un perceptrón monocapa entrenado mediante el optimizador Adam (un algoritmo de descenso de gradiente adaptativo) con regularización L_2 (Ridge regression).

```
model = keras.Sequential()
model.add(keras.Input(shape=(X_train.shape[1])))
model.add(keras.layers.Dense(1, activation='linear',
                             kernel_regularizer='l2'))
model.compile(loss='mean_squared_error', optimizer=Adam(),
              metrics='mean_squared_error')
model.fit(X_train, y_train, batch_size=64, epochs=150, verbose=0)
```

Entrena este modelo y calcula las predicciones sobre el conjunto de test usando el método `predict` que tiene el objeto model. Calcula el error cuadrático sobre el test.

El siguiente código define un perceptrón multicapa con dos capas con función de activación ReLU, la segunda con la mitad de neuronas que la primera.

```
model = keras.Sequential()
model.add(keras.Input(shape=(X_train.shape[1])))
model.add(keras.layers.Dense(neurons, activation='relu',
                             kernel_regularizer='l2'))
model.add(keras.layers.Dense(neurons//2, activation='relu',
                             kernel_regularizer='l2'))
model.add(keras.layers.Dense(1, activation='linear',
                             kernel_regularizer='l2'))
```

Ajusta los parámetros de esta red usando como número de neuronas potencias de 2 desde 32 a 1024 usando el mismo entrenamiento que para el modelo anterior. Calcula el error sobre el conjunto de test y escoge el mejor modelo. ¿Se obtiene un mejor resultado que con el perceptrón monocapa? ¿Aumentar el número de neuronas mejora el resultado? Representa las predicciones de la mejor red para el conjunto del test y sus errores (residuos) contra los valores reales. Representa la distribución del error. Comenta los resultados.

c) El ajuste por error cuadrático, como vimos en el tema de regresión lineal, asume que el ruido es gausiano y su la varianza es la misma para todas las predicciones, por lo que solo es necesario aprender una función que prediga su media. El siguiente código es una red que predice como salida la media y la varianza y optimiza el negativo de la verosimilitud de la distribución gausiana[1].

```
def nllike(y,y_hat):
    mean, logvar = tf.split(y_hat,2, -1)
    square = tf.square(mean - y)
    sigsq = tf.add(tf.exp(logvar),1e-3)
    ms = tf.add(tf.divide(square,sigsq), logvar)
    return tf.reduce_mean(ms)

input = keras.Input(shape=(X_train.shape[1]))
net = keras.layers.Dense(neurons, activation='relu',
                kernel_regularizer='l2')(input)
output1 = keras.layers.Dense(neurons//2, activation='relu',
                kernel_regularizer='l2')(net)
output1 = keras.layers.Dense(1, activation="linear",
```

[1]Es igual que en la regresión lineal por mínimos cuadrados, pero ahora la varianza también es un parámetro de la función de pérdida dado que no es constante.

```
                kernel_regularizer='l2')(output1)
output2 = keras.layers.Dense(neurons//2, activation='relu',
                kernel_regularizer='l2')(net)
output2 = keras.layers.Dense(1, activation='linear',
                kernel_regularizer='l2')(output2)
cout = keras.layers.Concatenate(axis=-1)([output1, output2])
model = keras.Model(inputs=input, outputs=cout)

model.compile(loss=nllike, optimizer=Adam(), metrics=nllike)
```

La red tiene dos ramas separadas, una para la media y otra para la varianza,. El código define una red con una capa oculta común y dos capas ocultas independientes, una que se usa para predecir la media y otra para la varianza.

La función `nllike` recibe las dos salidas de la red. **Explica qué está prediciendo exactamente la red** (qué representan los valores retorna) y como encaja 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 gausiana en este caso e identifica los cálculos en la función `nllike`.

Ajusta los parámetros de esta red usando como número de neuronas potencias de 2 desde 32 a 1024 usando el mismo entrenamiento que para el modelo anterior. Fijate que ahora la loss es la función `nllike`. Calcula el error cuadrático en el conjunto de test. Fíjate que la red predice dos valores, el primer valor es el que corresponde a la variable objetivo. Representa las predicciones de la mejor red para el conjunto del test contra las predicciones de la mejor red del apartado anterior. ¿Hay una gran diferencia entre las predicciones?

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.

3. **Perder hasta la camisa**

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. A veces lo que nos puede parecer una aplicación obvia y directa de métodos a los datos no lo es tanto y eso nos puede hacer perder hasta la camisa.

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. Podéis obtener los datos mediante la función `load_NASDAQ` de la librería `apafib`. Esta os 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). Resuelve los siguientes apartados ilustrando los resultados de la manera que te parezca más adecuada.

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?

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

Divide los datos en conjunto de entrenamiento (los 1000 primeros días) y test (el resto) y normalízalos a la escala [0-1]. Para generar los datos, tendrás que obtener ventanas de una cierta longitud w. La función de numpy `sliding_window_view` permite obtener una vista de una matriz que corresponde a lo que necesitas[3]. Tendrás que generar conjuntos de datos usando **todas** las variables para una longitud de ventana de 3. Fíjate en que trabajas con matrices con 3 dimensiones, adapta los datos para poder aplicar un MLP que trabaja con matrices de 2 dimensiones (usa la función `reshape`). Vamos a intentar resolver un problema más sencillo que predecir la cotización de las acciones con las ventanas pasadas. A partir de la variable del precio de las acciones de Google genera una variable que indique si el valor subió o bajó respecto al día anterior. Fíjate que en que el primer día que tendrás que predecir es el que corresponde al w+1. Esa es la variable que intentaremos predecir con las ventanas de todas las variables, resolveremos un problema de clasificación.

b) Usaremos como modelo base la regresión logística. Ajusta este modelo explorando sus hiperparámetros adecuadamente. Utiliza la exploración bayesiana como hemos hecho en otros ejemplos para poder explorar mejor el espacio de hiperparámetros. Obtén el acierto de validación cruzada, el acierto en el test, la matriz de confusión, la curva ROC y el informe de clasificación.

c) Ajusta ahora un MLP explorando sus hiperparámetros adecuadamente. Usa también la exploración bayesiana. Obtén el acierto de validación cruzada, el acierto en el test, la matriz de confusión, la curva ROC y el informe de clasificación. Comenta las diferencias entre este modelo y la regresión logística. Te parecen adecuados los modelos que se han obtenido ¿Por qué?

d) Quizás el resultado sea diferente si predecimos otra cosa más fácil. Calcula como etiquetas si el volumen de negociación de las acciones de Google sube o no respecto al día anterior y ajusta los mismos modelos que en los apartados anteriores. Compara los resultados de los modelos para este nuevo problema. ¿Hay alguna diferencia con la otra variable? ¿Se te ocurre alguna explicación?

4. **Nuestras líneas aéreas le aseguran el máximo confort, a veces**

El análisis de sentimientos es una herramienta utilizada en márquetin para poder determinar el sentido de la opinión de los consumidores de un producto o servicio. Es bastante caro el ir haciendo encuestas periódicas para obtener esa información, por lo que se puede intentar obtener de manera indirecta a partir de los mensajes que la gente deja en las redes sociales. El elemento clave es tener un clasificador para esos mensajes que nos dé el sentido de esa opinión. El conjunto de datos `airlines`[4] recoge una colección de tweets sobre la opinión de personas sobre líneas aéreas etiquetadas como positivas, neutras o negativas. En este caso para simplificar se han unido las categorías positiva y neutra. Lo podéis obtener mediante la función `load_sentiment` de la librería `apafib`. Esta función recibe un parámetro que corresponde al número de palabras más frecuentes que se usarán para extraer las características del conjunto de datos (`nwords`) y el porcentaje de datos que corresponde al conjunto de entrenamiento (`train_size`) que por defecto es 0.8. La función retornará seis matrices que corresponden al conjunto de entrenamiento y sus etiquetas, el conjunto de validación y sus etiquetas y el conjunto de test y sus etiquetas. El tamaño de los conjuntos de validación y test corresponde a los que no entren en el conjunto de entrenamiento dividido a partes iguales.

En las matrices de datos aparece la secuencia de palabras donde cada palabra ha sido substituida por un número entero en [1..nwords]. El número de columnas de la matriz corresponde a la

[3] Puedes ver como se usa en la sesión de laboratorio de Knn y MLP.

[4] Podéis encontrar una descripcion en https://www.kaggle.com/datasets/crowdflower/twitter-airline-sentiment.

longitud del tweet con más palabras. Si un tweet no tiene suficientes palabras el resto de posiciones se ha rellenado con ceros. Esta es una codificación alternativa al bag-of-words que hemos visto en alguna sesión de laboratorio. Esta codificación preserva el orden en el que aparecen las palabras en el texto.

El objetivo del problema es experimentar con diferentes tamaños de vocabulario y arquitecturas de redes para obtener una clasificación de los tweets en dos clases. Para todos los entrenamientos **has de utilizar terminación temprana** y **registrar el modelo con mejor resultado** en el entrenamiento. Al calcular el acierto de los modelos deberás cargar el modelo que se ha grabado, que se corresponderá con el mejor del entrenamiento. Tendrás que consultar el capítulo 3 para ver como se hace.

Para resolver este ejercicio necesitaréis Colab, llevará demasiado tiempo ejecutarlo sin GPU.

a) Genera conjuntos de datos con 500, 1500 y 3000 palabras. Los datos no necesitan ningún preprocesamiento. Utiliza t-SNE (inicializado con PCA) para visualizar los conjunto de datos de entrenamiento, representa las clases sobre la visualización y comenta los resultados.

b) Como habrás podido ver al leer el capítulo sobre Keras en este documento, existe una capa que transforma vectores compuestos por números enteros a vectores de reales (`Embedding`). Esto permite ajustar la representación de datos categóricos para que la red pueda tratarlos mejor y es una alternativa para texto a otros métodos de transformación a representación en vectores como los que hemos visto en algún laboratorio, básicamente la representación se adapta al problema a resolver en lugar de usar una transformación fija.

Vamos a entrenar una red que tendrá la siguiente estructura:

```
model = keras.Sequential()
model.add(keras.layers.Embedding(numwords + 1,
                    embedding_size,
                    input_length=train_x.shape[1]))
model.add(keras.layers.Dense(nneurons, activation='relu'))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

Esta red es un MLP con una capa oculta que aprende un *embedding* del texto a la vez que aprende a clasificar el texto. En este caso el problema es binario, así que podemos usar la entropía cruzada binaria como función de pérdida. El siguiente código permite entrenar un modelo con esta función usando el optimizador Adam con lotes de 512 ejemplos por época.

```
model.compile(loss='binary_crossentropy', optimizer=Adam())
res = model.fit(train_x, train_y,
            batch_size=512,
            epochs=epochs,
            validation_data=(val_x, val_y),
            verbose=verbose, callbacks=[early, ckpt])
```

Comprueba con los tres conjuntos de datos si aumentando el tamaño del *embedding* de 5 a 30 dimensiones en pasos de 5 tiene un efecto positivo en el acierto sobre el conjunto de validación. Comprueba si es consistente también el acierto en el conjunto de test. Usa un máximo de 50 épocas para el entrenamiento, una paciencia para la terminación temprana

de 10 (mira como funciona la terminación temprana en el capítulo 3) y un tamaño para la capa oculta del MLP de 128.

c) Selecciona el tamaño del *embedding* a partir de las conclusiones que has sacado del apartado anterior. Ahora probaremos el efecto de la terminación temprana, usa una terminación temprana con paciencia de 5 a 20 en pasos de 5 optimizando la misma red del apartado anterior para los tres conjuntos de datos. Observa cuantos pasos de optimización se realizan dependiendo de la paciencia[5]. ¿Ha merecido la pena cambiar ese parámetro? Elige la paciencia que te parezca más adecuada según los resultados con el conjunto de validación y de test.

d) Con el tamaño del `embedding` y la paciencia que has seleccionado en el apartado anterior ahora vamos a estudiar el efecto del número de capas en la red. Probaremos de 1 a 4 capas densas. La primera tendrá tamaño 128, cada capa adicional tendrá como tamaño la mitad de la capa anterior. Representa la evolución durante las épocas[5] del acierto para el conjunto de entrenamiento y de validación para todos los entrenamientos según el tamaño del vocabulario. Explica lo que observas en las gráficas sobre la evolución del acierto. Mira también el informe de clasificación para los que consideres los dos mejores modelos. Según lo que has observado ¿que parámetros elegirías para entrenar un modelo para estos datos? ¿Por qué?

[5]El método `fit` retorna un objeto con información sobre el entrenamiento, en el campo `history` encontrarás un diccionario con listas que guardan los datos del entrenamiento por cada época.

CAPÍTULO 3

Diseño y uso de redes neuronales con Keras

Existen múltiples librerias para la definición y entrenamiento de redes neuronales, las más utilizadas son Tensorflow y PyTorch, pero existen otras y muchas más se han quedado por el camino o han sido absorbidas por otros desarrollos. El elegir una u otra librería es más una cuestión de gustos (o manías), ya que todas permiten hacer básicamente lo mismo. En algunos de los ejercicios de esta lista de problemas usaremos la librería Keras. Esta se ha elegido por razones de simplicidad. Esta librería funciona por encima de Tensorflow y permite trabajar de manera más amigable renunciando a algo de flexibilidad.

Para usar las librerias de tensorflow y keras deberís importarlas primero, podéis hacerlo añadiendo al principio del notebook:

```
import tensorflow as tf
from tensorflow import keras
```

3.1. Modelos en Keras

La principal diferencia de trabajar con una librería de redes neuronales respecto a hacerlo con el MLP de `scikit-learn` es que podemos definir el modelo que queremos entrenar de manera flexible y utilizando todo el potencial que nos permiten las redes usando todos los tipos de capas que existen (o nos inventemos). Para ello hemos de definir un modelo (`keras.Model`) que indique qué tipo de entrada y salida tiene.

Lo que hay entre la entrada y salida es un grafo de computación que definiremos a partir de las capas que permite Keras. Tenemos dos maneras básicas de definir estos grafos, a partir de una secuencia de capas, si la computación es simplemente una transformación secuencial de la entrada o usando las capas de manera funcional si tenemos un grafo arbitrario definiendo las diferentes ramas que tenga el grafo. Obviamente también podemos definir una red secuencial usando las capas de manera funcional.

Los modelos más sencillos se definen a partir de una secuencia de capas. Se pueden definir usando el objeto `keras.Sequential` que es una sublase de `keras.Model`. Este permite ir apilando las capas en el orden que corresponda usando el metodo `add`. Por ejemplo:

```
model = keras.Sequential()
model.add(keras.Input(shape=(256)))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dense(16, activation='relu'))
model.add(keras.layers.Dense(2, activation='linear'))
```

Esto define un MLP con una entrada de tamaño 256 y tres capas densas (totalmente conectadas con la siguiente) con diferentes tamaños y funciones de activación. El tamaño de la salida lo determina la última capa que corresponde con la capa de salida.

Si queremos un modelo más complejo que sea un grafo arbitrario debemos crearlo manualmente usando las capas como si fueran funciones, por ejemplo:

```
input = keras.Input(shape=(256))
rama1 = keras.layers.Dense(128, activation='relu')(input)
rama1 = keras.layers.Dense(64, activation='relu')(rama1)
output1 = keras.layers.Dense(1, activation='linear')(rama1)
rama2 = keras.layers.Dense(64, activation='relu')(input)
rama2 = keras.layers.Dense(32, activation='relu')(rama2)
output2 = keras.layers.Dense(1, activation='tanh')(rama2)
```

Con esto tenemos una red que tiene una entrada de tamaño 256 y que se conecta con dos ramas, una que tienen dos capas ocultas que acaba en una capa densa con una neurona con función de activación lineal y otra que tiene dos capas ocultas y acaba en una capa densa con una neurona con función de activación tanh. Tenemos, por lo tanto, una red que tiene dos salidas que solo tienen en común la capa de entrada.

Si definimos así la red luego tendremos que usar el objeto `Model` indicando que elementos forman la entrada y cuáles la salida.

```
model = keras.Model(input=input, output=[output1, output2])
```

Usando el método `summary` de `Model` obtendremos una descripción del modelo con sus capas, sus tamaños de entrada y salida y su número de parámetros.

3.2. Capas

Exiten un gran número de capas disponibles en Keras, cada una de ellas tiene una funcionalidad y propósito específico, con su propio conjunto de parámetros. Podéis consultar sus detalles en la documentación https://keras.io/api/layers/. Como guía básica estas son las que utilizaremos:

- `Dense`, es la capa totalmente conectada, como parámetros básicos tendremos su tamaño y la función de activación (si usamos una de las usuales). Podemos indicar también diferentes tipos de regularización.
- `Conv1D`, `Conv2D`, son capas convolucionales para usar con secuencias o imágenes, como parámetros tendremos su tamaño, el tamaño de los kernels de convolución, el paso con el que se traslada la convolución, como se tratan las convoluciones en los bordes de la matriz y la función de activación a usar.

- `Dropout`, es una capa que descarta aleatoriamente parte de las conexiones y permite regularizar la red, como parámetro tendremos la probabilidad con la que una conexión es anulada.
- `Embedding`, capa que transforma vectores con valores enteros que representan índices a vectores reales de cierto tamaño. Solo deberíamos tener que indicar el tamaño de los vectores de salida, ya que habitualmente no es directamente la capa de entrada de la red. Se puede ver como una transformación de una codificación de valores (por ejemplo one hot encoding) a valores reales.
- `Flatten`, capa que adapta las dimensiones de una capa para que pueda conectarse con capas que admiten entradas en una sola dimensión

Podéis ver las características de otras capas en la documentación.

3.3. Entrenamiento

Una vez tengamos el modelo definido hemos de indicar cuál es la función de error que querremos optimizar, que puede ser una de las funciones de error ya definidas o una que definamos nosotros y que algoritmo de optimización queremos usar. Estos parámetros se indican usando el método `compile` de `Model`. Las más comunes se pueden indicar usando el string que las representa o se puede pasar el objeto correspondiente si queremos cambiar sus parámetros por defecto.

Para las funciones de error, tenemos las que ya conocemos para regresión y clasificación:

- Regresión: `mean_squared_error`, `mean_absolute_error`
- Clasificación: `binary_crossentropy`, `categorical_crossentropy` (si las etiquetas son un one-hot-encoding), `sparse_categorical_crossentropy` (si las etiquetas son números enteros)

Para los optimizadores tenemos tambien diferentes opciones, lo habitual es usar `adam` que es un algoritmo por descenso de gradiente estocástico que es adaptativo, en el sentido que decide automáticamente cuál es tamaño del paso que se usa en cada época por cada variable.

Podemos hacer por ejemplo

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
```

Si los parámetros no son correctos ocurrirá un error al entrenar el modelo.

El entrenamiento del modelo se puede hacer manualmente o automáticamente mediante el método `fit`. Al invocar este método podemos indicar entre otros:

- `x`, matriz de datos
- `y`, columna respuesta (etiquetas o valores numéricos)
- `batch_size`, tamaño del batch de cada paso de entrenamiento
- `epochs`, número de épocas que se usarán en el entrenamiento
- `validation_split`, el porcentaje de datos a usar para validación si no se tiene un conjunto de validación

- `validation_data`, tupla con los datos de validación si se tienen

Estos son los parámetros mínimos para poder entrenar un modelo.

Adicionalmente, Keras permite usar terminación temprana e ir salvando el mejor modelo que se ha obtenido durante el entrenamiento. Esto se realiza utilizando el parámetro `callbacks` que inserta llamadas durante el entrenamiento para estos propósitos. Este parámetro es una lista con instancias de objetos de la clase `keras.callbacks.Callback`. Estas son las que nos interesarán:

- `keras.callbacks.EarlyStopping`, que termina el entrenamiento cuando pasa alguna condición, generalmente que la función de pérdida en el conjunto de validación no mejora durante cierto número de épocas. Tiene un parámetro `patience` que indica el número de pasos a esperar desde que la condición de finalización se cumple.
- `keras.callbacks.ModelCheckpoint`, que permite salvar el modelo cada vez que se obtiene uno que mejora la función de pérdida de manera que podamos recuperarlo al final del entrenamiento. Esto nos interesa porque el modelo en la última época no tiene por qué ser el mejor.

Nota: los objetos de los callbacks han de ser siempre nuevos, no se pueden reusar entre diferentes entrenamientos.

Por ejemplo, podemos realizar un entrenamiento de la siguiente manera:

```
early = keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
check = keras.callbacks.ModelCheckpoint('modelckpt',
                                        monitor='val_loss',
                                        save_best_only=True,
                                        save_weights_only=True)
res = model.fit(X_train, y_train, batch_size=64, epochs=100
                validation_data=(X_val, y_val), callbacks=[early, check])
```

Esto entrenará un modelo con batches de 64 ejemplos como mucho durante 100 épocas monitorizando la pérdida del conjunto de validación y parando si esta no mejora durante 10 épocas (patience) y salvando el modelo cada vez que la función de pérdida en el conjunto de validación mejore.

El entrenamiento retorna un objeto con la información sobre la optimización. En el atributo `history` tenemos un diccionario que guarda el valor de la función de pérdida sobre el conjunto de entrenamiento y validación que podemos representar para ver su evolución.

Una vez hemos a acabado el entrenamiento podemos cargar el último punto de restauración salvado usando el método `load_weights` de la siguiente manera:

```
model.load_weights('./modelckpt').expect_partial()
```

Podemos obtener predicciones a partir del modelo usando el método `predict` pasándole un nuevo conjunto de ejemplos.

```
pred = model.predict(X)
```

Dependiendo de la salida de la red necesitaremos adaptar la salida adecuadamente si queremos por ejemplo obtener en informe de clasificación que nos da scikit learn. Por ejemplo, si es un problema de

clasificación binaria y queremos las etiquetas deberemos convertir las predicciones a 0/1, lo podemos hacer simplemente con el cálculo:

```
pred_bin = (pred>0.5)*1
```

Si tenemos una salida por clase podemos calcular el índice de la clase con la función `argmax` de la siguiente manera:

```
pred_label = np.argmax(pred, axis=1)
```