

2021

Machine Learning

CÉSAR SAN JUAN PASTOR

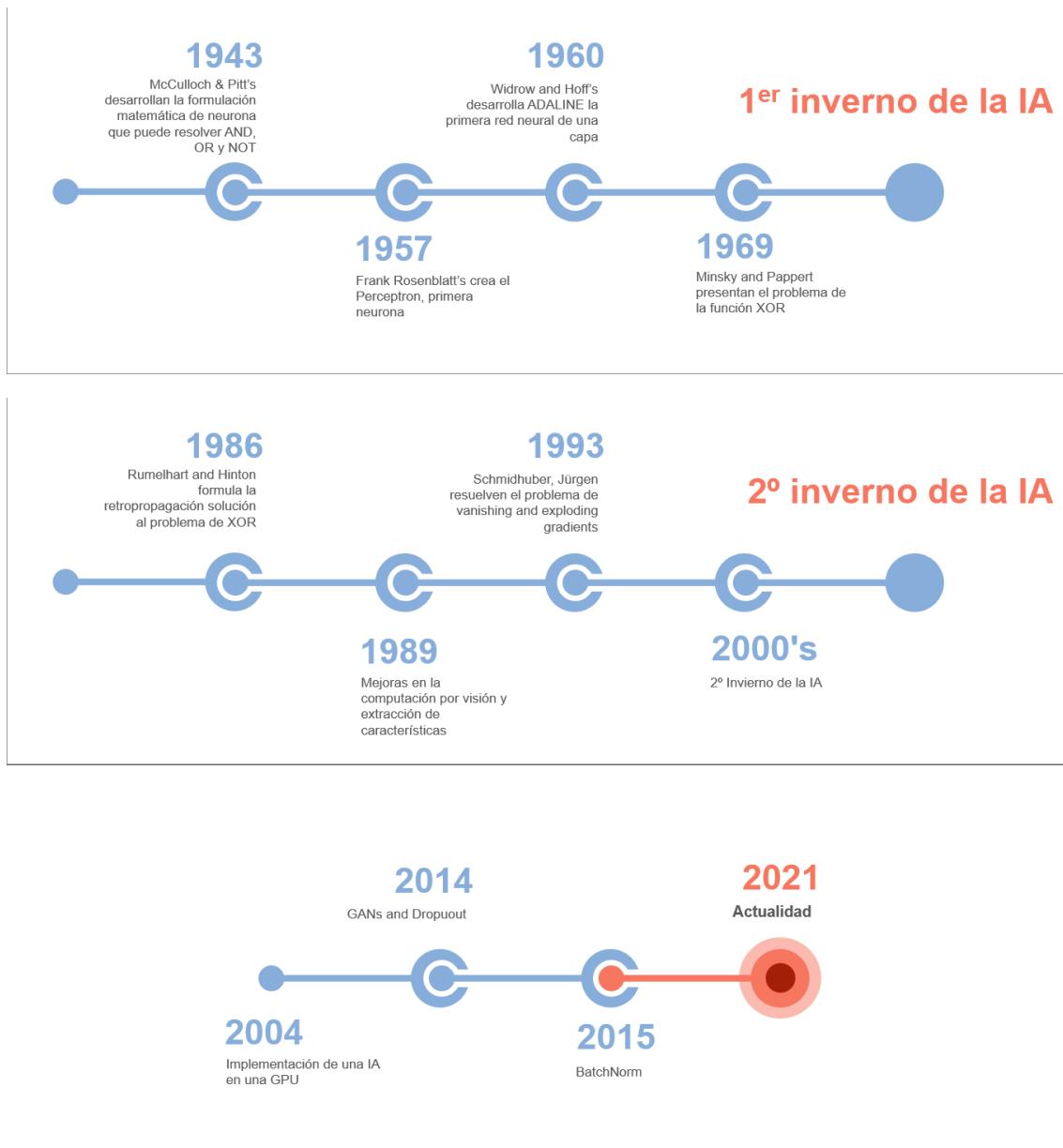
I.E.S. ARCIPRESTE DE HITA | Dpto. de Informática

Contenidos

Cap 0.- Time Line	2
Cap 1.- Introducción	3
Cap 2.- Conceptos Generales y terminología	9
Cap 3.-Entrenamiento de una ML para clasificación	18
Ejercicios.....	23
Cap 4.- Explorando el scikit-learn: Clasificación.....	24
Ejercicios.....	37
Cap 5.- Preprocesado de datos	38
Cap 6.- Reducción de la dimensionalidad: Extracción.....	59
Cap 7.- Buenas prácticas para la evaluación del modelo y afinado de hiperparámetros	62
Cap 8.- Elección del algoritmo.....	74
Ejercicios.....	75
Cap 9.- Combinación de algoritmos en uno solo	76
Ejercicios.....	78
Cap 10.- Explorando el scikit-learn: Regresión.....	79
Ejercicios.....	80
Cap 11.- Explorando el scikit-learn: Aprendizaje no supervisado	81
Cap 12.- Redes neuronales y Aprendizaje profundo: Introducción	87
Cap 13.- Redes neuronales y Aprendizaje profundo: Ajuste de la red	109
Ejercicios.....	119
Cap 14.- Introducción a Tensorflow	120
Cap 15.-Tratamiento de imágenes: Redes Convolucionales (CNN)	139
Ejercicios.....	164
Cap 16.-Tratamiento del lenguaje natural con RNN	165
Ejercicios.....	173
Cap 17.- Generación con nuevo contenido GANs.....	174
Cap 18.- Aprendizaje reforzado.....	193
Anexo I: Algoritmo simplificado de trabajo (Sin Redes neuronales).....	197
Anexo II: Algoritmo simplificado de trabajo (Redes neuronales)	199
Anexo III: Recursos	200
Anexo IV: Índice completo	201

UT-1

Cap 0.- Time Line

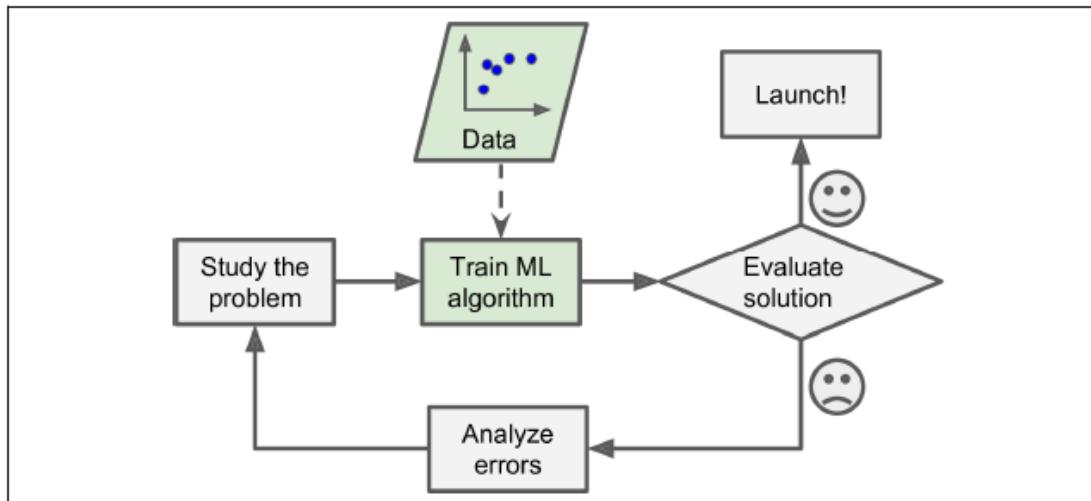


Aprendizaje Automático Inteligencia Artificial débil



Cap 1.- Introducción

Este curso está dedicado a la programación en el aprendizaje automático (Machine Learning), en el ML se intenta conseguir conocimiento de los datos a través de algoritmos matemáticos. El aprendizaje automático es el proceso por el que los datos generan conocimiento. Dicho proceso se basa en el uso de algoritmos matemáticos para reconocer patrones y hacer predicciones o clasificaciones.



En el curso usaremos como lenguaje de programación Python al ser potente y fácil de aprender. Hoy por hoy es el lenguaje más utilizado para el aprendizaje de Machine Learning. En concreto, podremos usar cualquier versión de 32 bits (x32) o 64 bits (x64) siempre que no utilicemos una versión superior 3.7 (11/2020) o 3.9 (6/2021), ya que es la última que tiene todas las librerías necesarias. Como IDE se pueden utilizar tanto PyCharm como Spyder.

Las librerías que vamos a usar son entre otras (abril 2021):

- Scipy 1. 6.2
- NumPy 1. 20. 2
- Scikit-learn 0. 24.1
- Matplotlib 3.4.0
- Pandas 1.2.3

Una vez instalado el entorno Python en nuestra máquina descargando los ficheros desde la página del proyecto (<https://www.python.org/>) podremos instalar todos ficheros que necesitemos a través de la siguiente orden:

```
pip install nueva_librería
```

Si no dominamos el lenguaje de programación Python tendremos que seguir un curso para adquirir las competencias mínimas para la programación en este lenguaje.

```
https://www.kaggle.com/learn/python
```

Consideraciones ético - morales

HitchBOT es un robot de cuerpo cilíndrico, fabricado con un cubo de plástico, con brazos y piernas unidos, y una pantalla LED que mostraba ojos y una boca.

Estaba programado para pedir a las personas que lo recogieran y los llevara con ellas. Cruzó con éxito Canadá y algunos países europeos. En agosto de 2015, apareció decapitado en Filadelfia. ¿Hicieron algo malo los vándalos al

decapitar a HitchBOT más allá de destruir la propiedad privada? ¿Tenían la obligación moral de no interferir en su estado normal de funcionamiento y dejarle continuar su viaje por el mundo?

Replicantes de la película Blade Runner (1982). Supongamos que un día se pudiera fabricar una "persona" de "plástico". La IA fuerte ha triunfado. Piensa,

siente y habla como nosotros. Se diferencia en los materiales, se diferencia en cómo llega a este mundo. Roba, mata, crea, ama, siente, ... como nosotros.

¿Cuál es la ley que se le debería aplicar a las máquinas? ¿Qué significa matar una máquina? ¿Qué derechos morales tendría dicha máquina? ¿Es solo una máquina? Un punto de vista bajo el cual es relevante analizar las respuestas es empleando el argumento aristotélico de equidad: los mismos casos se deberían tratar del mismo modo.

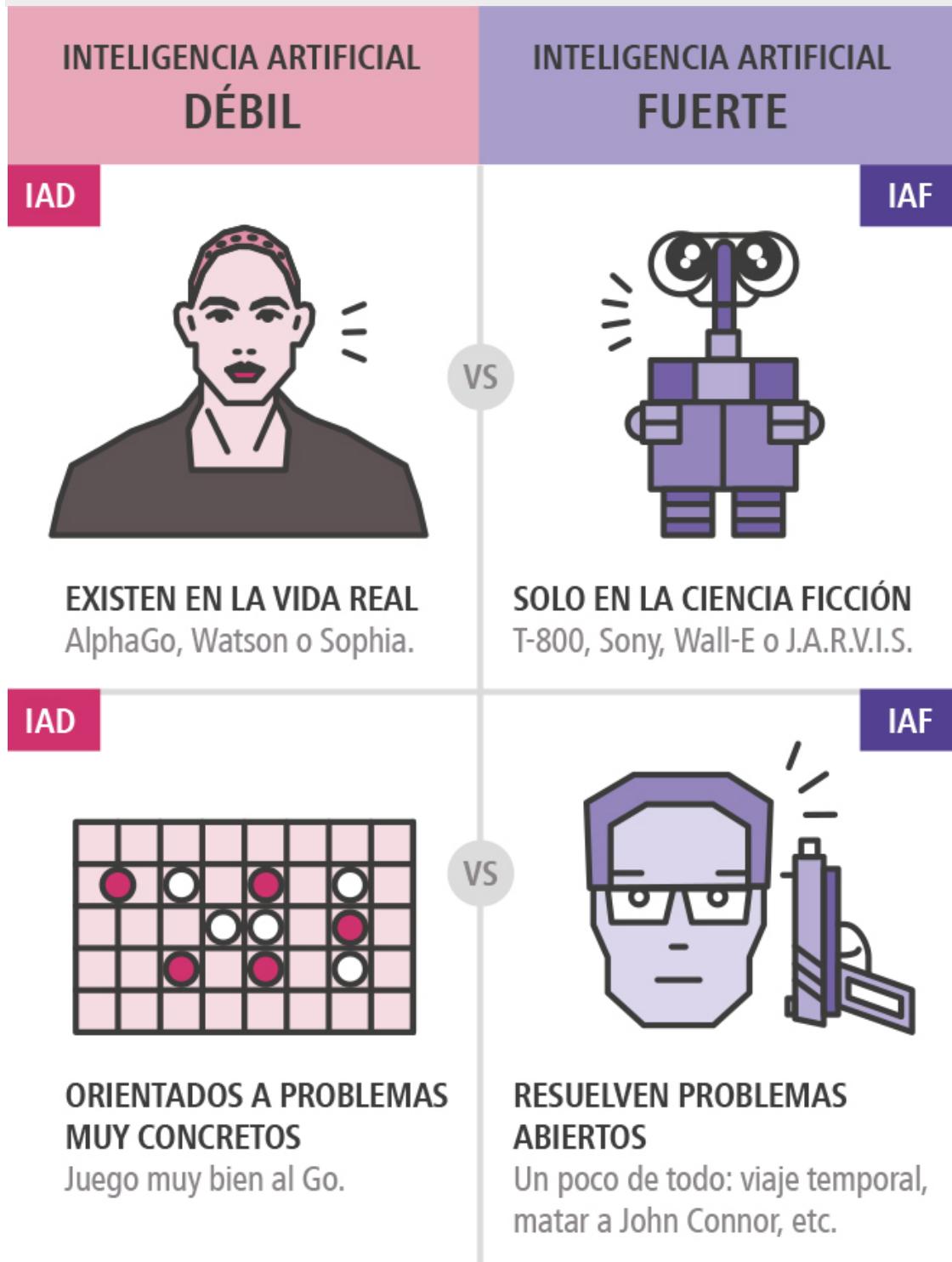
El coche autónomo ¿Quién es el responsable? ¿El propietario del vehículo? ¿El fabricante del vehículo? ¿El desarrollador del software?

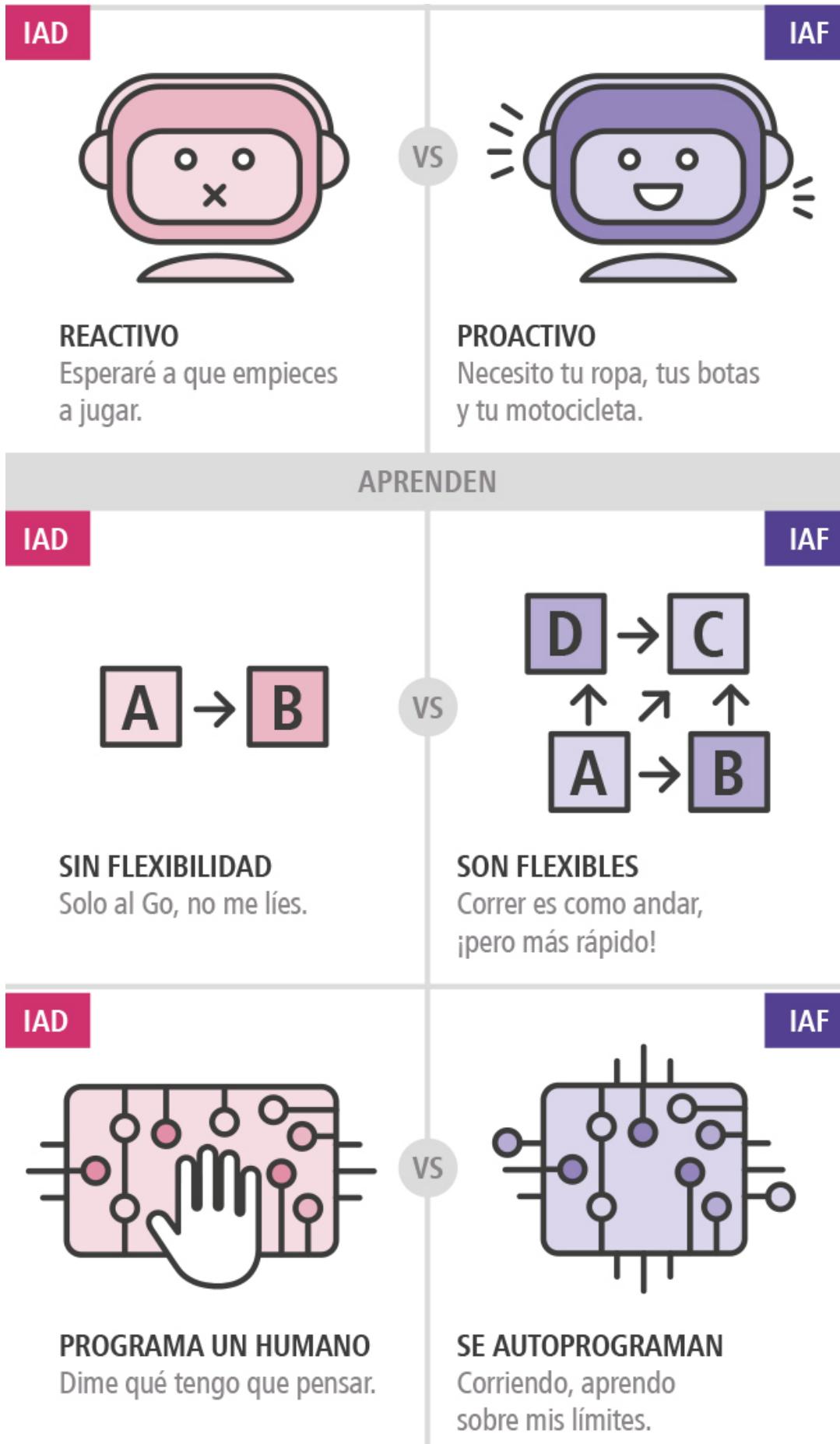
Los sistemas de reconocimiento ¿Queda justificado el uso de estos sistemas? ¿Se podrían implantar, pero restringiendo su uso?

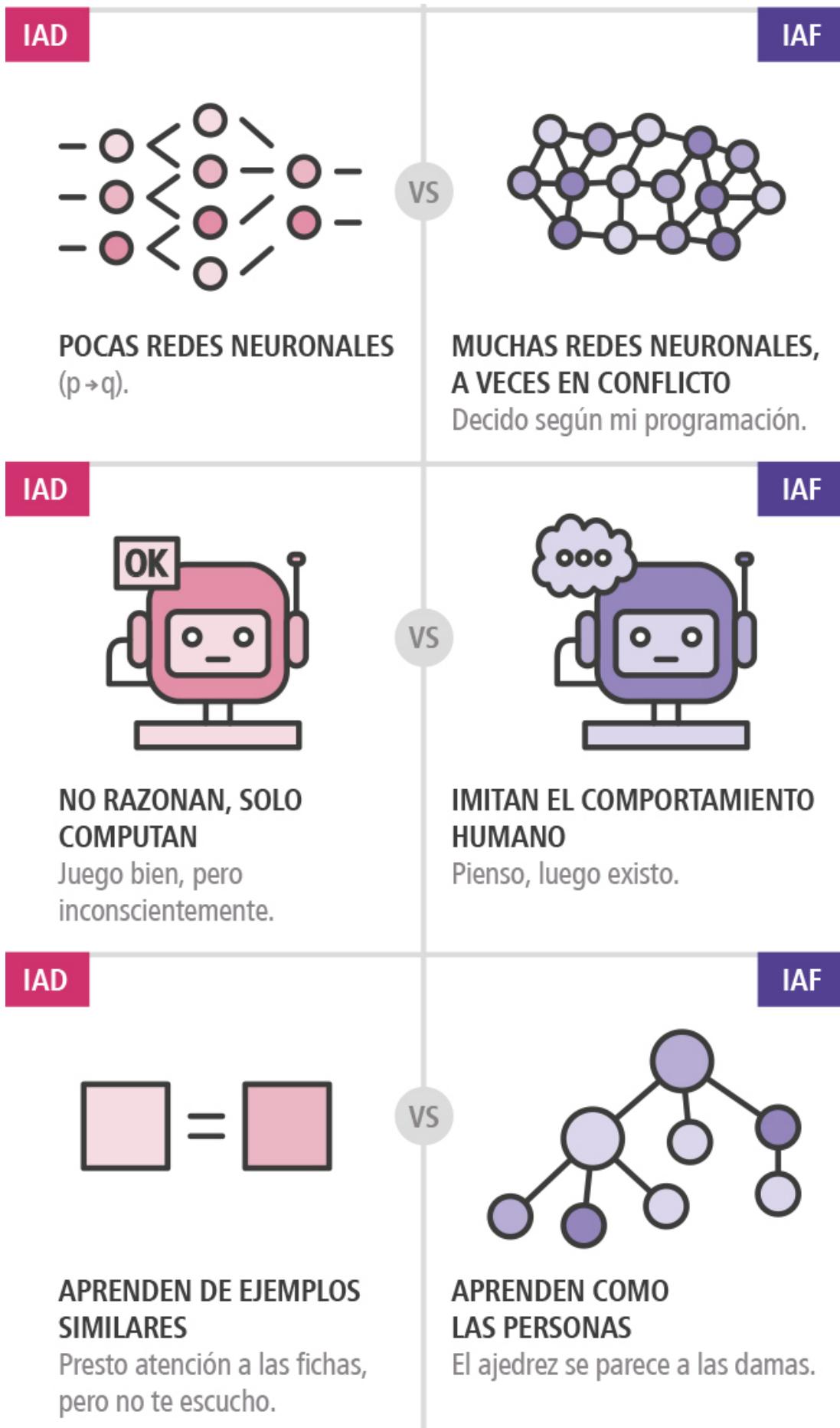
Los algoritmos. ¿Qué autoridad o poder de decisión debe tener los algoritmos? ¿Cómo los algoritmos deben rendir cuentas y a quién? ¿Cuáles decisiones deben ser justificadas y cómo?

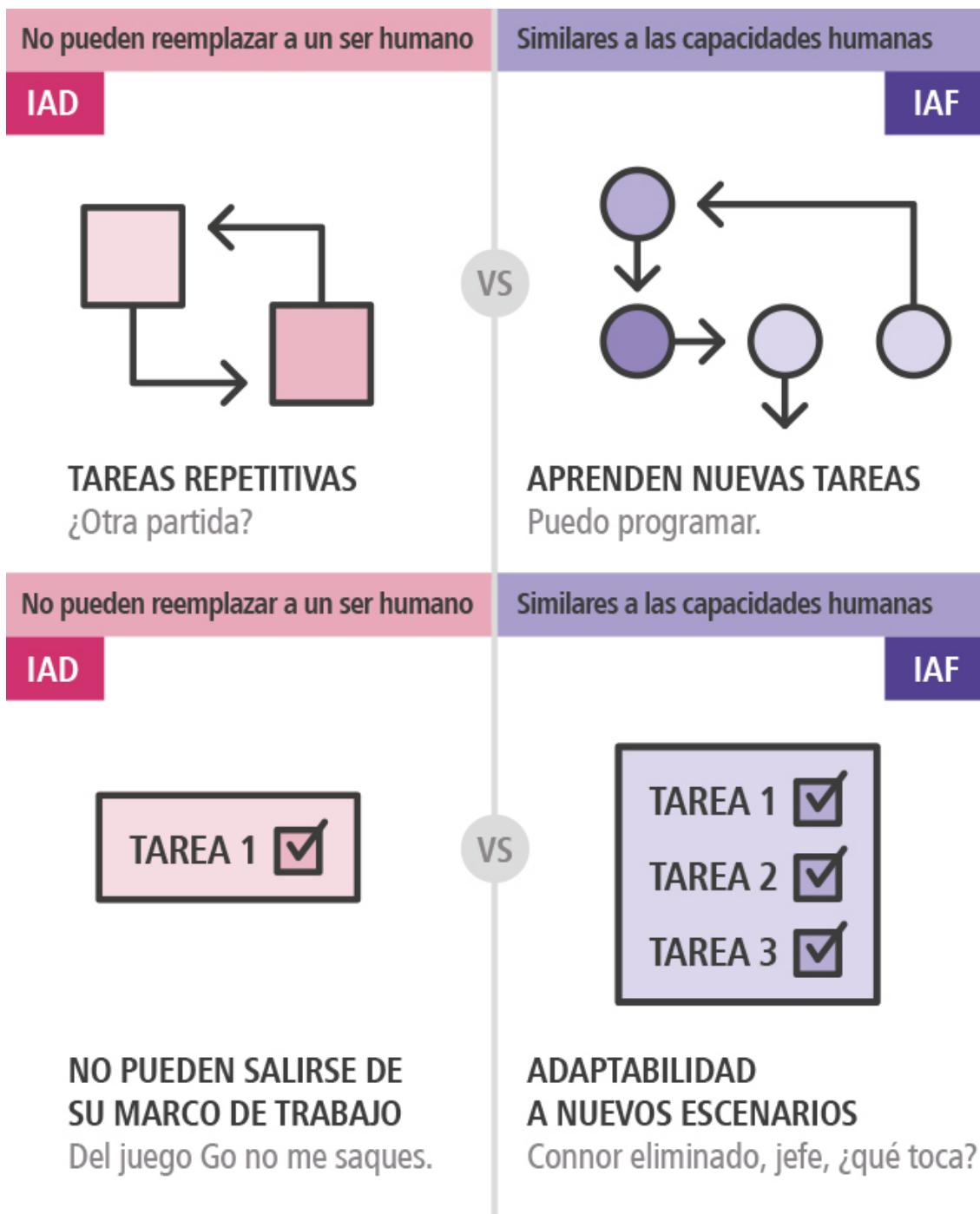
Debemos reflexionar sobre estos temas

IA débil vs fuerte





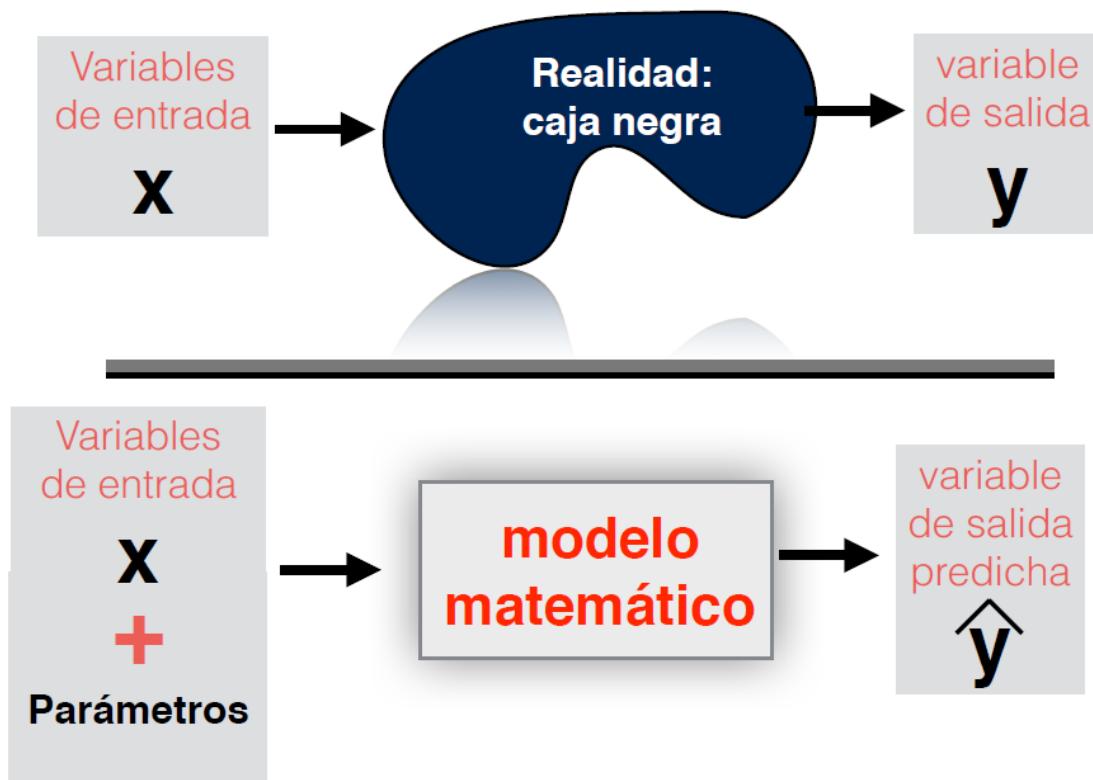




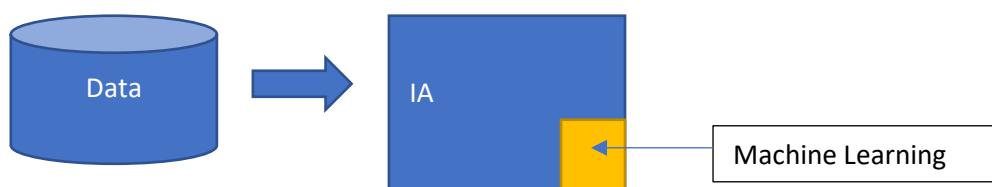
Cap 2.- Conceptos Generales y terminología

Qué es el Machine Learning

Los métodos de aprendizaje automático son ejemplos notables de la llamada IA débil. En los métodos de aprendizaje automático el dominio de aplicación está delimitado y es conocido. Estos métodos elaboran un modelo matemático de esta realidad

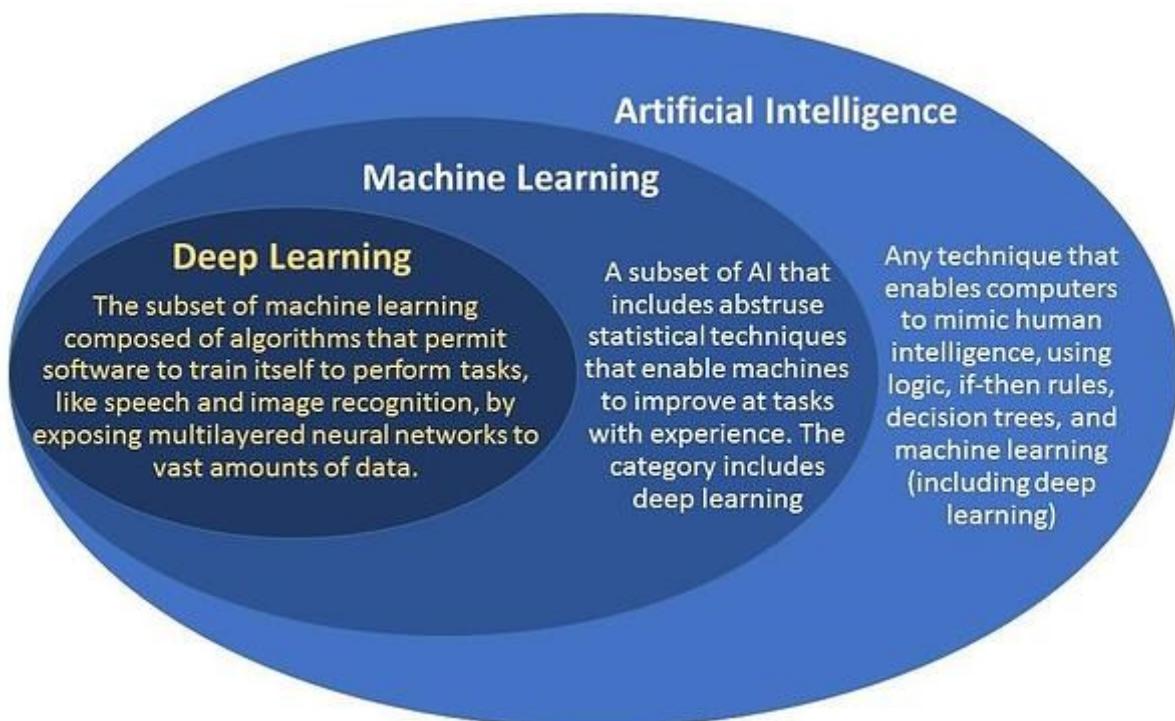


"Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed" Arthur L. Samuel, AI pioneer, 1959



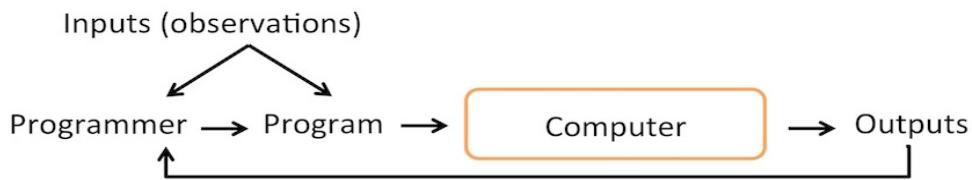
Como se ha mencionado el ML es proceso de obtención de conocimiento a partir de los datos, pero existen diferentes tipos de ML en función de las características de los datos.

Debemos distinguir entre IA, ML y DP (Deep Learning).



Como podemos apreciar en el gráfico, la inteligencia artificial es la ciencia que intenta imitar el comportamiento humano en ordenadores. De ésta, el machine Learning es la parte de la IA que intentan obtener conocimiento de los datos existentes, buscando patrones. Y el Deep Learning es una parte del ML encargado de aprender por sí mismo conocimiento de gran cantidad de datos sin un entrenamiento anterior.

La revolución del paradigma de programación

The Traditional Programming Paradigm

Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed
 – Arthur Samuel (1959)

Machine Learning

Sebastian Raschka, 2016

Aplicaciones del ML

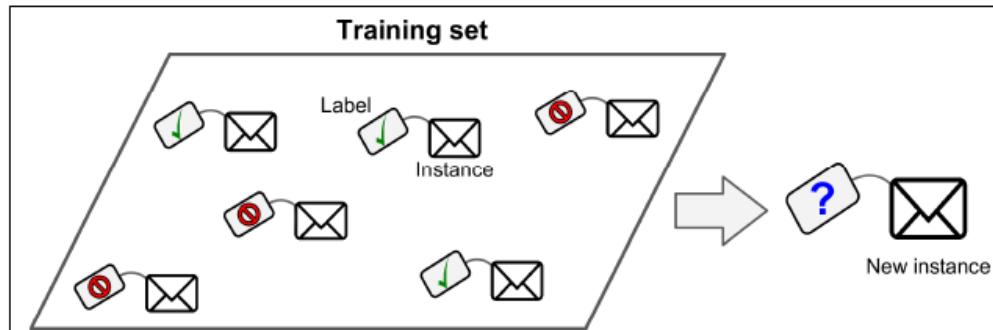
- Detección de spam
- Detección de caras (iPhone X)
- Búsquedas en la web (DuckDuckGo, Bing, Google)
- Predicciones en los partidos
- Detección de fraudes
- Detección de stocks
- Asistentes virtuales (Apple Siri, Amazon Alexa, ...)
- Recomendaciones de productos (Netflix, Amazon)
- Conducción automática (Uber, Tesla)
- Traducción (Google translate)
- Análisis de sentimientos
- Análisis de medicamentos
- Diagnósticos

Tipos de ML según el tipo de aprendizaje.

Tipos de variables en el ML.

- **Cuantitativas.** Valores numéricos: edad, ingresos, altura.
 - **Continuas**
 - **Discretas**
- **Ordinales.** Expresan un orden: ranking
- **Cualitativas.** Expresan una calidad: color, sexo.

- **Supervisado:** El aprendizaje supervisado parte de datos etiquetados, con lo que buscaremos predicciones actuales y futuras o clasificación de los datos de entrada. En este tipo, se tienen un conjunto de ejemplos etiquetados ya conocidos con los que entrenaremos el modelo. Las etiquetas deben ser discretas (contables). Por ejemplo, clasificar correo spam o determinar la clase de una imagen. La segunda posibilidad es la predicción o regresión, donde a partir de un conjunto de variables se consigue una predicción. Las variables se suelen llamar características y la variable de salida es la variable objetivo. Ha aparecido un nuevo tipo de aprendizaje supervisado: Rankin en el que se clasifican en un orden un conjunto de valores de entrada.



Un modelo de regresión predice valores continuos. Por ejemplo, los modelos de regresión hacen predicciones que responden a preguntas como las siguientes:

¿Cuál es el valor de una casa en California?

¿Cuál es la probabilidad de que un usuario haga clic en este anuncio?

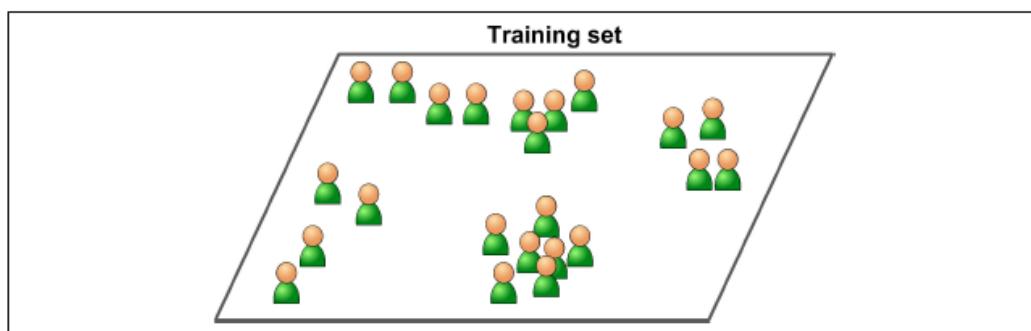
Un modelo de clasificación predice valores discretos. Por ejemplo, los modelos de clasificación hacen predicciones que responden a preguntas como las siguientes:

¿Un mensaje de correo electrónico determinado es spam o no es spam?

¿Esta imagen es de un perro, un gato o un hámster?

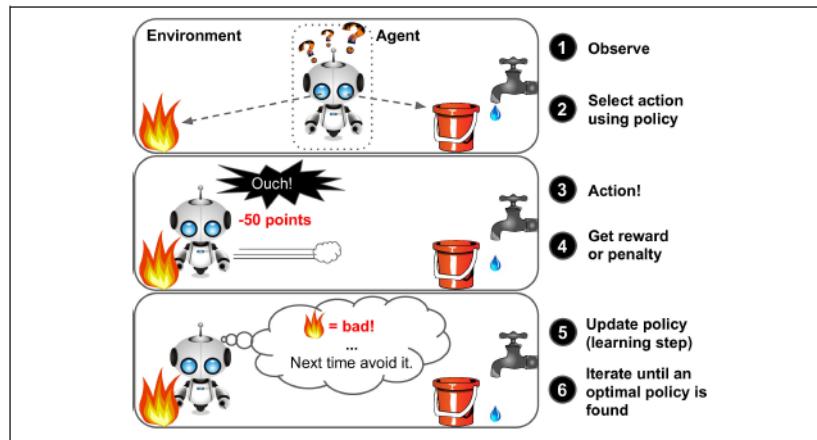
- **No supervisado:** El aprendizaje no supervisado parte de datos sin etiquetas para buscar una estructura interna o clasificación sin una función de recompensa.

El ejemplo principal es el Clustering o agrupación. Cada clúster o conjunto tendrá características similares. Por ejemplo, se podrá determinar grupos de compradores basados en sus intereses. También se aplica esta técnica a la disminución de la dimensionalidad, en la que se reducen el número de variables de entrada sin pérdida en la información subyacente, este proceso permite eliminar gran parte del ruido.



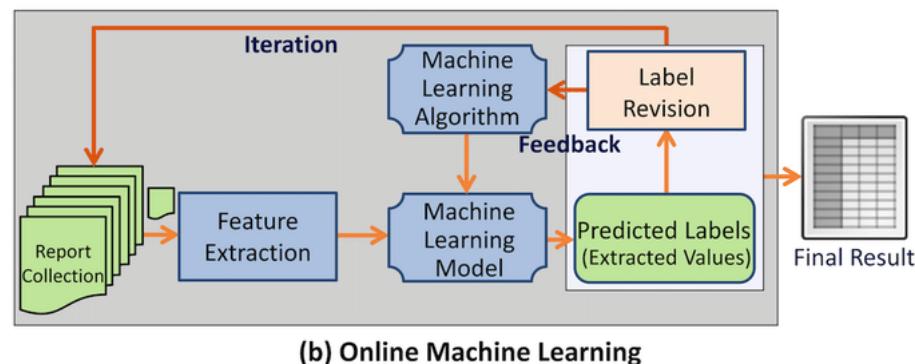
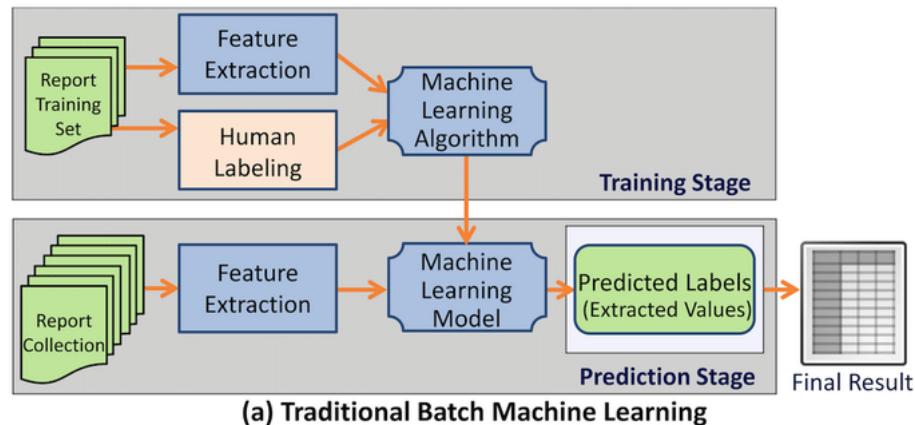
- **Aprendizaje reforzado:** Es el proceso de decisión mediante sistema de recompensas para aprender una serie de acciones. El objetivo es hacer un sistema que mejore su rendimiento basado en las interacciones con el entorno. El feedback se realiza a través de medidas que nos dicen cómo de buena es la acción a través de una función de recompensa. El agente aprende a través de la interacción con el entorno y un proceso

de prueba y error. Un ejemplo típico es un motor de ajedrez en el que agente tratará de maximizar la función de recompensa (ganar la partida) a través de interacciones con el entorno (jugadas). Un concepto importante es que la recompensa se puede obtener de forma inmediata o al final (como es el caso del ajedrez).



Otras clasificaciones del ML

- Aprendizaje online o aprendizaje batch. En el primero el sistema aprende indefinidamente de los datos que llegan y se adapta a las nuevas situaciones rápidamente, en el segundo hay que volver a entrenar el modelo completamente con los datos anteriores y con los actuales para que se adapte.



- Aprendizaje basado en instancias o basado en modelos. En el primero se utilizarán los ejemplos para buscar semejanzas, en el segundo se buscarán patrones para realizar la tarea deseada.



Terminología y nomenclatura

La nomenclatura en el ML es fundamental y por consiguiente debemos estudiar y aprenderla muy bien ya que será utilizada a lo largo de todo el curso.

Ejemplos	A	B	C	D	Etiqueta
1	1	2	3	1	A
1	1	1	2	3	B

Características

Clase

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} \quad X_{j \rightarrow \text{columna}}^i \rightarrow \text{fila}$$

X_4^2

$$X \Rightarrow \text{matriz} \quad x \rightarrow \text{Vector} \quad X_j^i \rightarrow \text{elemento}$$

$$X_2 = [1 \ 1 \ 2 \ 3] \quad \dim(X) = 4 \times 2 \quad \text{Característica: } \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

- Ejemplo: una fila.
- Modelo: algoritmo matemático encargado de clasificar o predecir.
- Entrenamiento: ajuste de los parámetros del modelo.

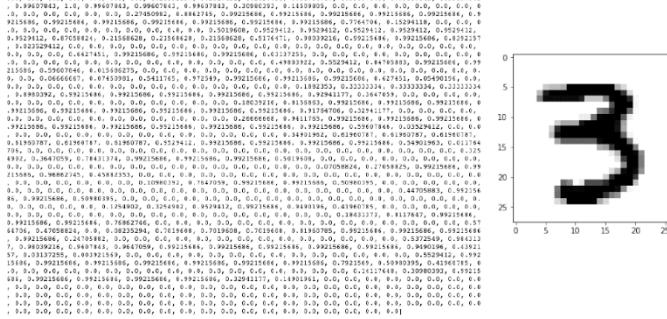
- Característica: Una columna de la tabla, una variable, una entrada.
- Objetivo (y): variable de salida, variable a predecir.
- Función de pérdida: función de coste, función de error. Función que determina si el algoritmo se está acercando al objetivo o alejando de él.
- Predicción. Salida predicha por la red.

Iris.csv
database.sqlite

Ejemplo	Características					Etiqueta
	# Id	# SepalLeng...	# SepalWid...	# PetalLeng...	# PetalWidth...	
1	5.1	3.5	1.4	0.2	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	0.4	Iris-setosa

Representación de los Datos en el ML

La característica principal del ML son los datos, y estos deben tener una representación adecuada para ser tratados. Las características tienen que ser generalmente numéricas, y los datos que representan pueden ser de diversa índole. Así encontramos representaciones con datos **estructurados** como es el Dataset iris, o representaciones con métodos **desestructurados**.



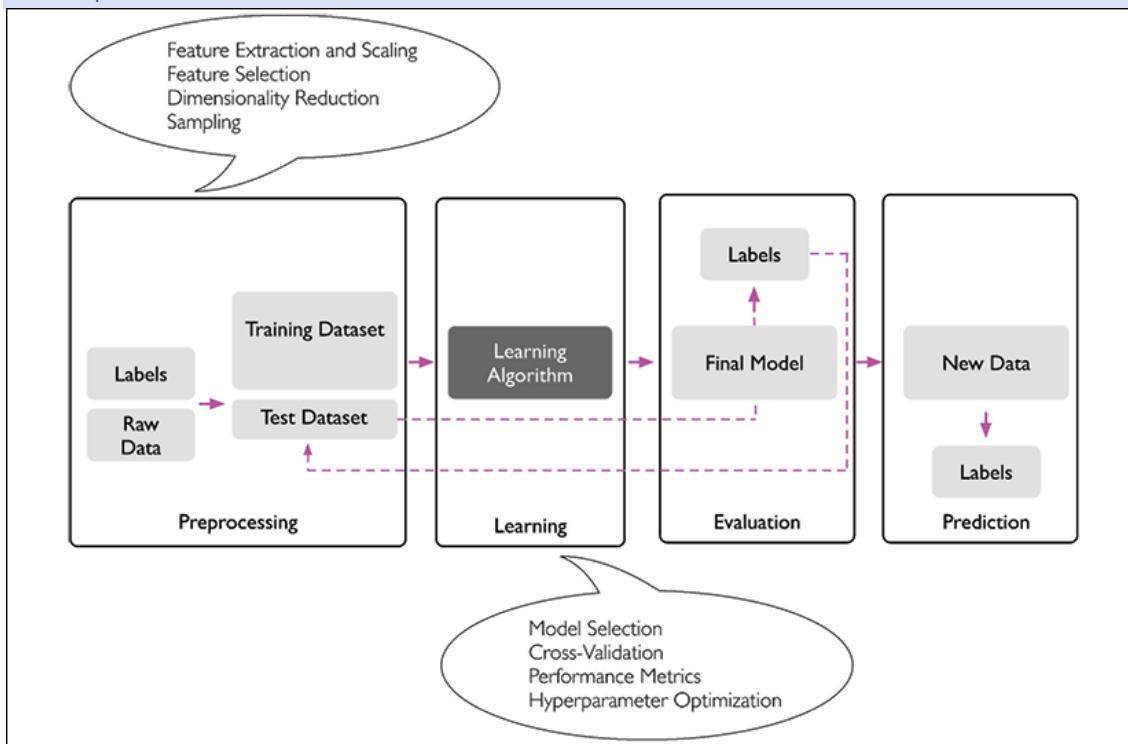
Los desafíos de la ML

Para que podamos desarrollar un buen trabajo en el ML debemos afrontar una serie de retos presentes en el proceso de aprendizaje.

1. **Datos insuficientes.** La captura de datos es primordial para cualquier proyecto de ML, no solo la cantidad debe ser adecuada, sino también la calidad de los mismos. En general serán necesarios miles de ejemplos para poder hacer converger a nuestros modelos. Si bien el que no haya suficientes ejemplos es un hándicap, hay algoritmos que podrán sacar partido de ellos aun siendo insuficientes.
2. **Datos no representativos.** Es imprescindible que los datos sean representativos de los casos que queremos representar.
3. **Baja calidad de los datos.** Si los datos son erróneos o contienen ruido el algoritmo no podrá converger de forma eficiente. Por lo que limpiar los datos y adecuarlos es una tarea primordial antes de todo.
4. **Características irrelevantes.** Nuestro sistema solo será capaz de aprender adecuadamente si los datos son correctos, y si hay características no necesarias deben ser eliminadas antes de empezar el proceso.
5. **Sobreajuste de los datos.** El sobreajuste se produce cuando un modelo no generaliza bien para datos no presentados en el entrenamiento, este es un grave problema que deberemos tratar.

6. **Subajuste de los datos.** En este caso el modelo es demasiado simple para recoger todas las características de los datos.

Pasos para el ML.



Preprocesado

Es una de las etapas más importantes. En esta etapa vamos a preparar los datos para los algoritmos, de esta fase dependerá el éxito de las siguientes. Muchos algoritmos requieren que las características estén en la misma escala o sean comparables (por ejemplo, distribuidos según la Normal). O puede que haya datos redundantes y se tenga que hacer una reducción de la dimensionalidad eliminando los superfluos.

En esta fase dividiremos los datos en dos conjuntos uno de entrenamiento y otro de pruebas.

Selección y entrenamiento del modelo predictivo

Elegiremos en esta fase el modelo que más se ajusta a los datos, siendo muy importante comparar el resultado de varios algoritmos para encontrar el mejor. Pero para poder comparar el resultado es imprescindible elegir una manera de medir el rendimiento entre todos los algoritmos y que sea comparable entre ellos.

Otro punto a tener en cuenta es que no podemos esperar que los parámetros por defecto de un algoritmo hagan el mejor trabajo, por lo que habrá que probar diferentes combinaciones de los parámetros del modelo para ajustarse a los datos, es lo que se denomina ajuste de los hiperparámetros.

Entrenar un modelo simplemente significa aprender (determinar) valores correctos para todas las ponderaciones y las ordenadas al origen de los ejemplos etiquetados. En un aprendizaje supervisado, el algoritmo de un aprendizaje automático construye un modelo al examinar varios ejemplos e intentar encontrar un modelo que minimice la pérdida. Este proceso se denomina minimización del riesgo empírico.

Evaluación del modelo y predicción

La única manera de saber si un modelo generaliza bien con datos no utilizados es evaluando el modelo. Para ello utilizaremos los datos de pruebas para estimar el error de generalización y si es aceptable, usar el modelo pasar a las predicciones o clasificaciones reales.

Si el error de generalización sobre los datos de prueba es muy bajo, pero sobre los datos de test es alto indicará que nuestro modelo se ha sobreajustado.

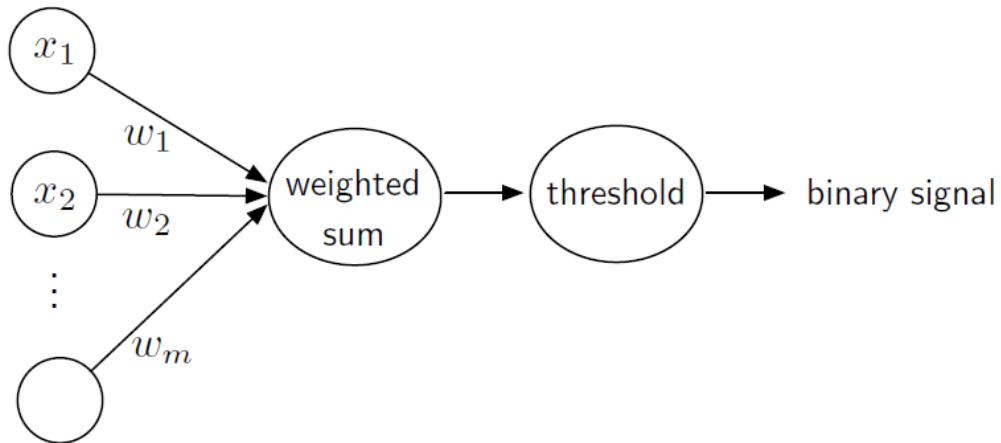
Instalación de Python

- Descargar la versión 3. 7 x64 e instalar los paquetes vía: **pip install paquete**
- Actualizar el instalador de Python: **pip install paquete –upgrade** si es necesario antes
- Se puede usar la distribución anaconda como remplazo a la instalación manual
 - **conda install paquete / conda update paquete**

Cap 3.-Entrenamiento de una ML para clasificación

Definición formal de neurona

El concepto matemático de neurona viene del fisiológico. El funcionamiento tradicional de una neurona se basa en recibir estímulos de diferentes fuentes y si la activación supera un umbral la neurona se activará.



Del mismo modo, la neurona artificial tiene una serie de entradas, cada una de ellas ponderarán con un peso (w) su entrada (x) y la función de activación determinará con todos los datos si se debe transmitir o no el resultado a la salida. Explicado de forma matemática es lo siguiente:

$\phi(z)$ es la función de activación

$$w = \begin{bmatrix} w_1 \\ \dots \\ w_n \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix} \quad z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

si $\phi(z) > \text{umbral}$ then 1
else -1

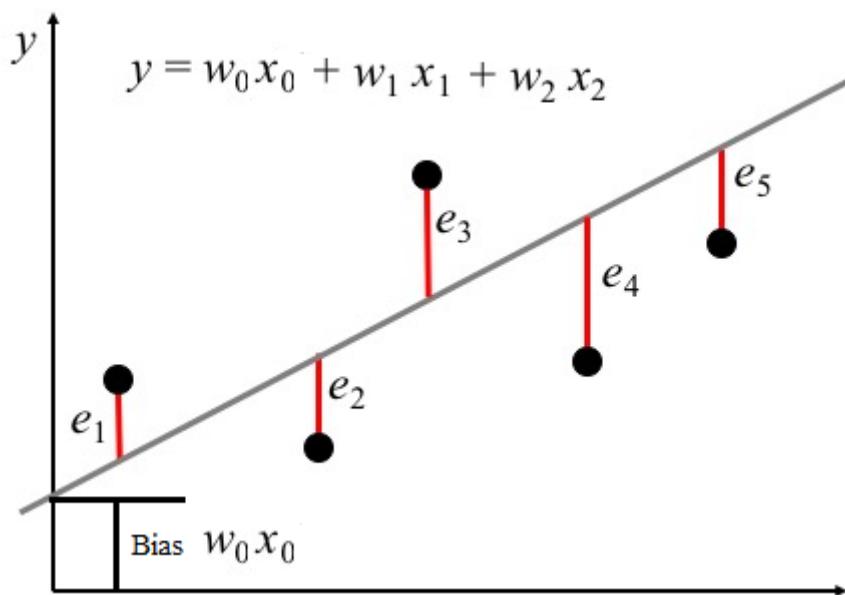
$$z_1 = w_0 x_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

$w_0 x_0$ es la tendencia o bias

si $\phi(z_1) > \text{umbral}$ then 1
else -1

Según el cálculo de matrices:

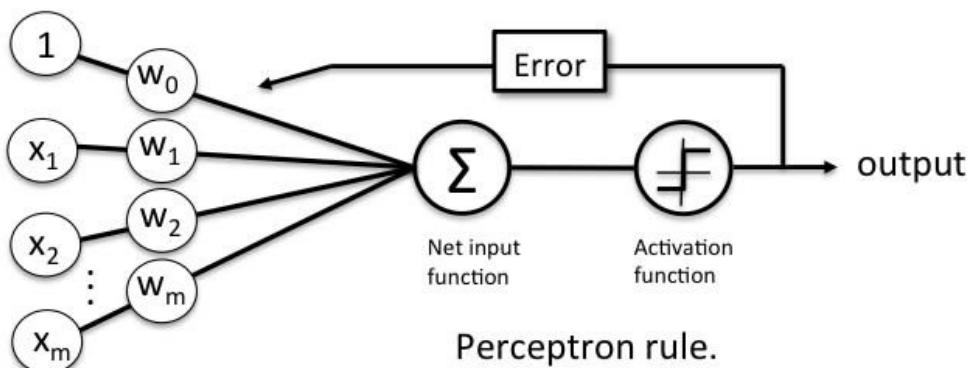
$$Z = W^T \cdot X = \sum_{j=0}^n w_j x_j \quad T \text{ indica transpuesta}$$



A la “neurona” matemática se la añade una entrada más X_0 con valor 1 y con peso W_0 para modelizar el desplazamiento a lo largo del eje y, como se puede ver en el dibujo anterior.

Programando un Perceptrón

1. Inicialización de pesos (w) a cero o un valor aleatorio.
2. Para cada ejemplo.
 - a. Calcular el valor de salida.
 - b. Actualizar los pesos $w_j := w_j + \Delta w_j$.
 - i. El incremento de cambio (Δw_j) se calcula como: La tasa de aprendizaje por La diferencia entre el valor real menos el predicho, todo ello por el valor de entrada. $\Delta w_j = \eta * (real - pred) * x_j$
 - ii. Todos los pesos se actualizan a la vez, por lo que el cálculo del cambio solo se hace una vez.
 - iii. La tasa de cambio (η) suele estar entre 0 y 1.
 - iv. La convergencia del Perceptrón solo será posible si los datos son linealmente separables.



Código 001. Py

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad Y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad W = [0 \quad 0]$$

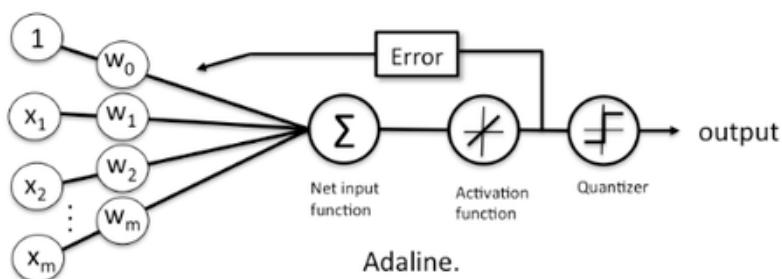
Hay que recordar que los pesos (W) no se pueden inicializar a cero todos si queremos que converja el algoritmo.

Podemos extender el modelo de Perceptrón único a más de dos clases creando tantos perceptrones como clases, creando nuevas características en todos ellos en la que la clase correspondiente será positiva y las del resto negativas.

Código 002.py

Convergencia del aprendizaje (Adaline)

Adaline es una mejora del Perceptrón, en la que la principal diferencia es que los pesos se actualizan en base a una función lineal de activación en vez de la función del Perceptrón basada en un umbral. Se sigue usando la función de umbral de activación para hacer predicciones.



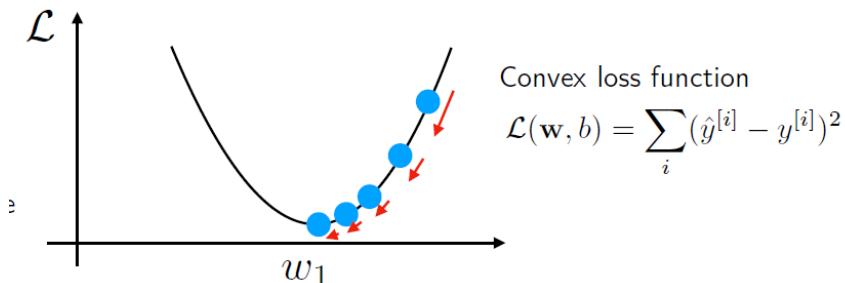
El mecanismo general para la convergencia es el descenso del gradiente.

Descenso del gradiente: Minimizar la función de coste

En el caso de Adaline, la función de coste se definirá:

$$J(w) = \frac{1}{2} \sum_i (y(i) - \phi(z(i)))^2 \quad y(i) \text{ salida esperada} \quad \phi(z(i)) \text{ salida predicha}$$

$z(i)$ Valor de la suma ponderada de las entradas



Para encontrar el incremento o decremento se utilizará la técnica del descenso del gradiente (el gradiente se encuentra como la derivada con respecto a los pesos). De cada peso se tiene que derivar la función de coste con respecto a los pesos (W) (el 1/2 es para que la deriva sea más

sencilla). La idea del algoritmo es dar pasos pequeños en la dirección contraria al gradiente para llegar el mínimo local, ese incremento se conoce como la tasa de cambio (η).

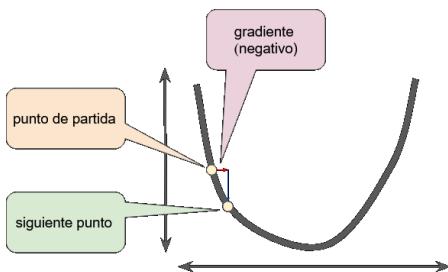


Ilustración 1.- Gradiente

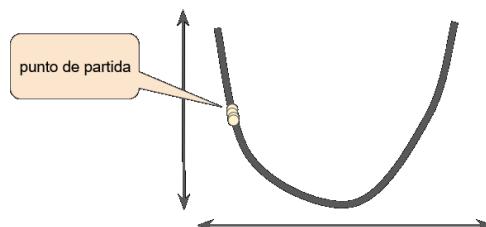


Ilustración 2.- Tasa de Cambio

El cómputo se hace calculando el gradiente para cada peso, encontrando la derivada parcial para cada peso de la función de coste, multiplicándolo por la fase de aprendizaje y por el valor de la entrada:

$$\Delta w_j = \eta \sum_i (y^i - \phi(z^i)) x_j^i$$

En la práctica se requiere experimentación para encontrar una buena tasa de aprendizaje.

Código 003.py

Mejorar el descenso de gradiente con el escalado de las características

El orden de magnitud de las variables influye en los procedimientos estadísticos. Por ejemplo, si tenemos una muestra donde se recogen el peso y la altura de ciertos individuos es importante las unidades en cómo se miden las variables. Si la altura la cuantificamos en metros, los individuos posiblemente estarían en el intervalo [0; 2] mientras que el peso, si fuese medido en kilogramos, en el intervalo [0; 200]. Una variable tiene un rango 100 veces más grande que la otra.

- **Estandarización por rangos.** Se reemplazan las variables por

$$x_i^{new} = \frac{x_i^{old} - \min(x_i^{old})}{\max(x_i^{old}) - \min(x_i^{old})}$$

- **Z-score estandarización.** Consiste en hacer que las variables tengan media 0 y desviación estándar 1, usando la fórmula de la tipificación

$$x_i^{new} = \frac{x_i^{old} - \bar{x}_i}{s_i}$$

donde \bar{x}_i es la media muestral de la variable x_i y s_i su desviación estándar muestral.

- **Escalamiento decimal.** Se trata de dividir por potencias de 10, esto es:

$$x_i^{new} = \frac{x_i^{old}}{10^m}$$

siendo m el número de dígitos del mayor dato en valor absoluto.

La mayoría de los algoritmos se beneficiarán de que las características sean comparables o lo que es lo mismo, se hayan escalado para converger más rápidamente. Veamos la técnica llamada **estandarización**: En la que se cambiarán los datos de una característica para ajustarse a la Normal. Se hace calculando la media y la desviación típica de la característica y a continuación a cada dato se le resta la media y se le divide por la desviación típica.

$$x'_j = \frac{x_j - \eta_j}{\sigma_j}$$

Descenso de gradiente estocástico (SGD)

El descenso del gradiente es computacionalmente muy costoso en escenarios con muchos ejemplos, una alternativa es el SGD. En vez de actualizar los pesos basándose en la suma de los errores acumulados de todos los ejemplos, se actualizarán de forma incremental con cada uno.

$$\eta(y_i - \phi(z_i))x_i$$

Código 004.py

Este nuevo método suele converger más rápidamente al haber más actualizaciones en los pesos, pero es importante que los ejemplos se muestren de forma aleatoria, por lo que es normal mezclar en cada vuelta del conjunto de datos. Además, la tasa de aprendizaje generalmente no es fija, suele descender según pasa el tiempo:

$$\frac{c_1}{\text{num}_{\text{iter}} + c_2} \quad \begin{array}{l} c_1 \text{ es un cte} \\ c_2 \text{ es una cte.} \end{array}$$

Otra ventaja del SGD es que se puede entrenar online con los ejemplos que van llegando al ser un algoritmo acumulativo.

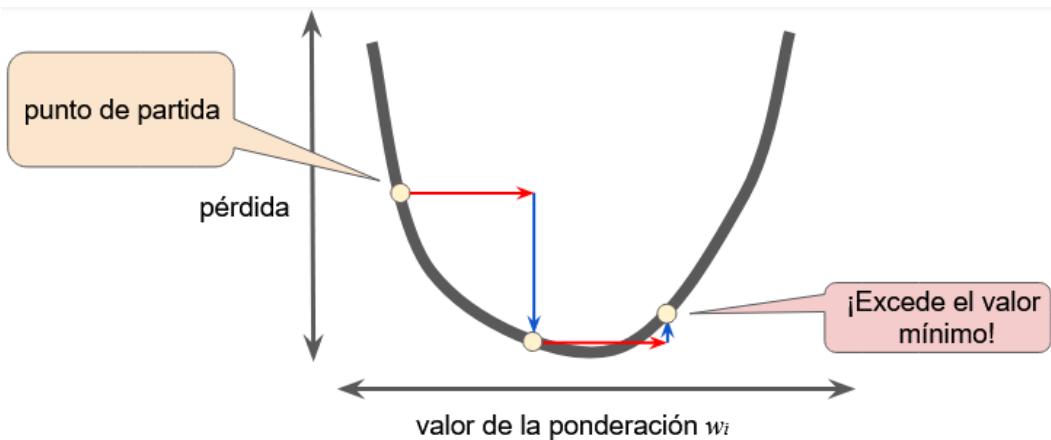


Ilustración 3.- Tasa de aprendizaje no adecuada

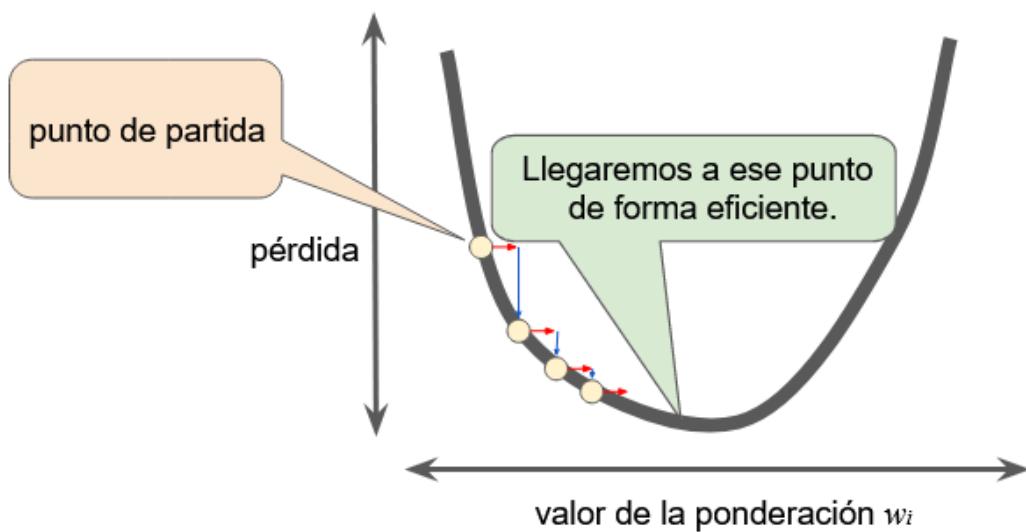


Ilustración 4.- Tasa de aprendizaje adecuada

Ejercicios

- Partiendo del Dataset de Iris, realizar las siguientes combinaciones de características para intentar separarlas con un AdalineGD estandarizado y no
 - ∞ sepal length in cm, sepal width in cm
 - ∞ sepal width in cm, petal width in cm
 - ∞ petal length in cm, petal width in cm
- Usar los Dataset data_1.csv y data_2.csv para hacer un estudio con AdalineGD y Perceptron

UT-2

Cap 4.- Explorando el scikit-learn: Clasificación

Eligiendo un algoritmo de clasificación

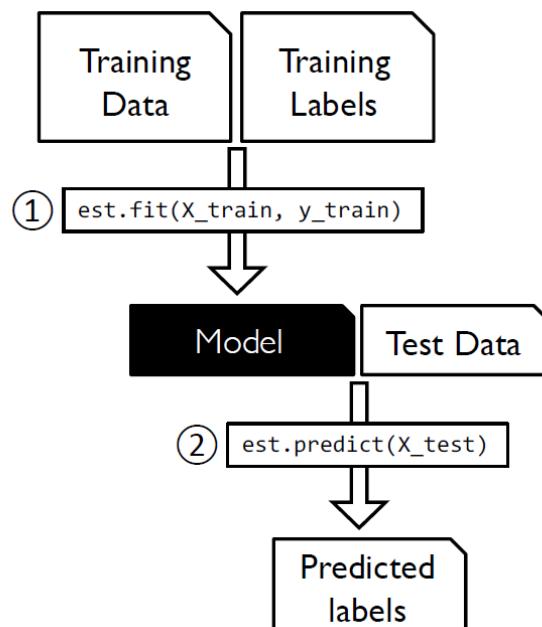
Cuando nos enfrentemos a la elección del algoritmo tendremos en cuenta que no existe ningún algoritmo que se ajuste a todos los posibles escenarios. Es recomendable probar varios y comparar el rendimiento entre ellos para elegir el que mejor se adapte. Pero para poder realizar esta comparación, hay que usar métricas semejantes en todos los algoritmos. En todo caso el proceso es el siguiente:

- Elegir las características y recoger los ejemplos etiquetados.
- Elegir la métrica.
- Elegir un algoritmo de clasificación y optimización (mejor redes neuronales que se verá más adelante)
- Evaluar el rendimiento del modelo.
- Optimizar el algoritmo cambiando los hiperparámetros.

Entrenando un Perceptrón con scikit-learn

Código 005.py

La librería scikit-learn ofrece una gran variedad de algoritmos de aprendizaje ya implementados junto con cantidad de funciones para preprocesar los datos, optimizar los algoritmos y evaluar los modelos. Para la mayoría de algoritmos es necesario que las características categóricas se codifiquen como números enteros.



Cuando trabajamos con ML, ya hemos dicho que el algoritmo se debe validar, con lo que es necesario dividir los casos de entrada en dos conjuntos, uno de entrenamiento y otro de pruebas.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3, random_state=1, stratify=y)
```

En este caso hacemos que el conjunto de pruebas tenga un tamaño del 30%. Esta función además mezcla los datos antes de dividirlos con lo que evitamos muchos problemas de convergencia. El parámetro **random_state** tiene el mismo significado que en los puntos anteriores, se utiliza para inicializar el generador aleatorio y así poder reproducir los resultados. El parámetro **stratify** hace que los datos de entrenamiento y prueba tengan las mismas proporciones de elementos según, en este caso, las etiquetas (y).

Como se comentó en puntos anteriores, la estandarización puede mejorar la convergencia del algoritmo.

```
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Es importante remarcar que el escalador se debe ajustar con los datos de entrenamiento y se deben adaptar ambos conjuntos de datos con el mismo sin modificar, tal y como vemos en el código anterior, para que sean comparables.

Dijimos en la implementación del Perceptrón, que solo se podían separar dos clases por Perceptrón, esta librería soporta división de varias clases con el mecanismo que se comentó en los puntos anteriores (poner a 1 la clase correspondiente y a cero las demás, creando varios perceptrones, uno por clase) de forma automática.

Esta librería también incorpora gran cantidad de métricas, algunas como funciones externas, pero además los propios algoritmos incorporan un método para realizar la métrica.

```
ppn = Perceptron(eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
print("Tasa:", ppn.score(X_test_std, y_test))

y_pred = ppn.predict(X_test_std)
print("Tasa:", accuracy_score(y_test, y_pred))
```

En este caso utilizamos ambas aproximaciones, la función **accuracy_score** de la librería y el método **score** del Perceptrón.

El uso en la métrica del error producido frente a la exactitud del algoritmo es cuestión de gustos ya que ambos están relacionados según: *exactitud = 1 – error*.

Uno de los problemas más graves del ML es el sobreajuste del algoritmo. Significa que en los datos de prueba funciona muy bien pero cuando alimentamos el modelo con datos nuevos falla. Se hablará más delante de este problema.

Clasificación por Regresión Logística

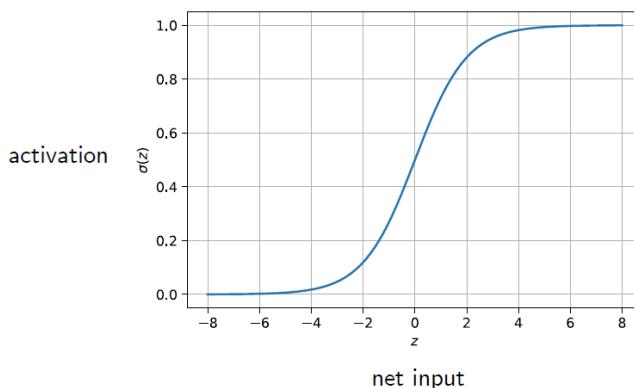
La RL es un modelo de clasificación (no de predicción). Es uno de los más utilizados en la industria. Es un modelo de clasificación binaria que se puede generalizar para múltiples clases: Regresión logística multinomial o Softmax.

Definiremos la probabilidad de un suceso Como: $p/(1-p)$, donde p es la probabilidad de que sea positivo el evento a predecir. Crearemos la función *logit* Como: $\log(p/(1-p))$. Esta función coge valores en el rango 0...1 y los transforma a reales. Como nos interesa predecir la probabilidad que un evento sea de una clase en particular usaremos la función inversa a la *logit* llamada sigmoide: $f(z):1/(1+e^{-z})$, donde z es la entrada de la red.

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

$$z = w^t x = w_0 x_0 + w_1 x_1 + \dots + w_n x_n$$

$w_0 x_0$ es la tendencia (*bias*), x_0 vale 1



Podemos concluir que esta función (sigmoide) coge un número real y lo transforma en un valor entre 0...1 siendo el valor $y=0,5$ cuando $x=0$. En la RL se usa la función sigmoide como función de activación.

La salida de la función sigmoide se interpreta como la probabilidad de que un ejemplo sea de la clase correspondiente para esas características y parametrizadas por esos pesos. De hecho, es tan importante saber la predicción como el porcentaje de certeza: en la predicción del tiempo, en la predicción de enfermedades.

Encontrando los pesos en la función de coste.

Para poder derivar la función de coste y utilizar el descenso de gradiente definiremos la función

$$L(w) = \prod \left(\phi(z^i)^{y^i} (1 - \phi(z^i))^{1-y^i} \right)$$

$$a^b c^d$$

Y aplicaremos logaritmos ya que son más fáciles de maximizar, además se reduce la posibilidad de un desbordamiento numérico y los productos se convierten en sumas que son menos costosas.

$$l(w) = \log(L(w)) = \sum y^i \log(\phi(z)) + (1 - y) \log(1 - \phi(z))$$

$$b \log(a) + d \log(c)$$

$$x = a^b c^d \leftrightarrow \log(x) = \log(a^b) \log(c^d) = b \log(a) + d \log(c)$$

En el ML, lo general es minimizar, por lo que la función de coste se describe usando derivadas:

$$\sum [-y^i \log(\phi(z^i)) - (1 - y^i) \log(1 - \phi(z^i))]$$

Código 006.py

Código 007.py

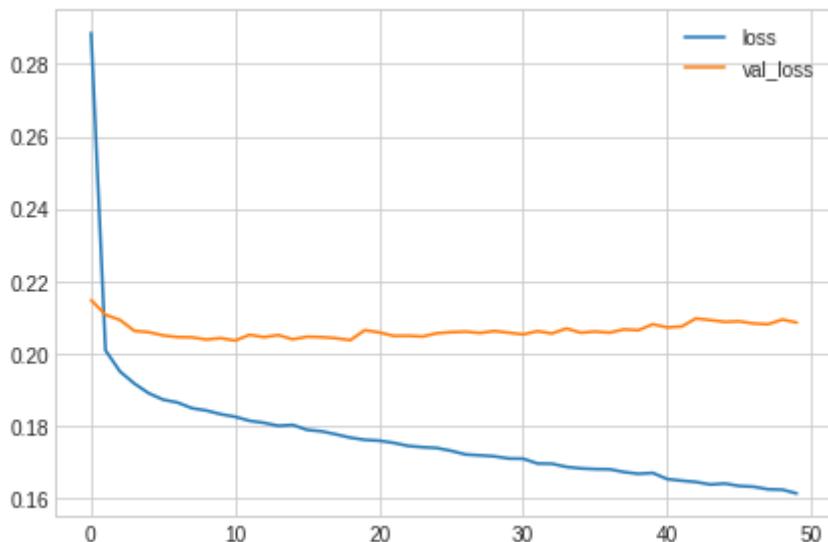
Parámetros importantes en la RL

- **C** (float), default=1.0. Es la inversa de la regularización, debe ser un valor real positivo. Valores más pequeños implican mayor regularización.
- **solver**{‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’}, default=’lbfgs’. Algoritmo a usar para realizar el ajuste.
 - Para pequeños Dataset y problemas binarios se utiliza liblinear, si el Dataset es grande mejor usar ‘sag’ y ‘saga’ que son más rápidos
 - Para problemas con más de dos clases se tiene que usar ‘newton-cg’, ‘sag’, ‘saga’ y ‘lbfgs’
 - Los algoritmos ‘newton-cg’, ‘lbfgs’, ‘sag’ y ‘saga’ pueden usarse con la regularización L2
 - Los algoritmos ‘liblinear’ y ‘saga’ permiten también a regularización L1
 - Los algoritmos ‘sag’ y ‘saga’ convergerán solo si las características están aproximadamente en la misma escala.
- **multi_class**{‘auto’, ‘ovr’, ‘multinomial’}, default=’auto’. La opción ovr se debe usar para problemas de multiclass en los que cada característica es binaria. La opción multinomial no está disponible para el solver liblinear
- **n_jobsint**, default=None. Número de cpus a usar de forma paralela. Se usará solo si se elige multi_class=’ovr’, es ignorado para ‘liblinear’ -1 significa usar todos los procesadores posibles.
- **penalty**{‘l1’, ‘l2’, ‘elasticnet’, ‘none’}, default=’l2’. Usado para determinar la regularización a usar.

Controlar los problemas de los modelos con la regularización

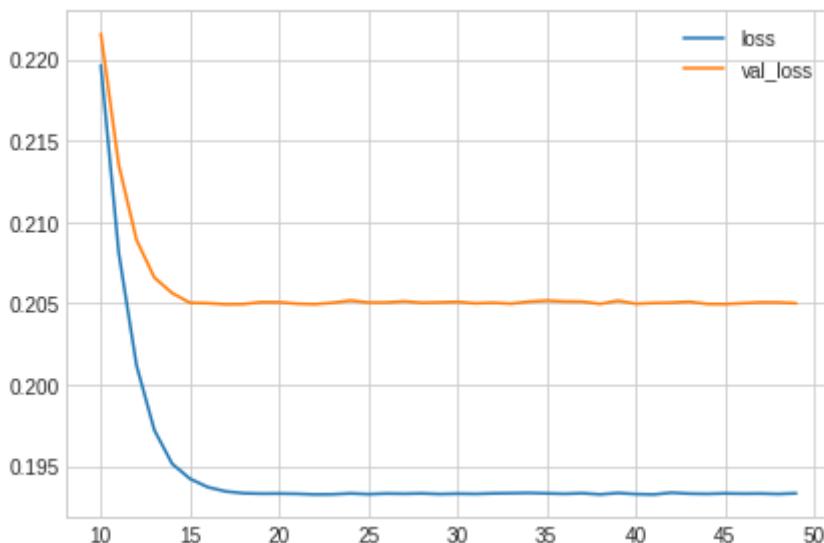
La regularización pretende reducir la capacidad del modelo y / o reducción de la varianza de las predicciones para mejorar la convergencia. Puede ser entendida como el proceso de agregar información (cambiar la función objetivo) para evitar el sobreajuste.

El sobreajuste es un problema común en el ML. Significa que el modelo se ajusta muy bien a los datos de entrenamiento, pero no también a datos nuevos (test).



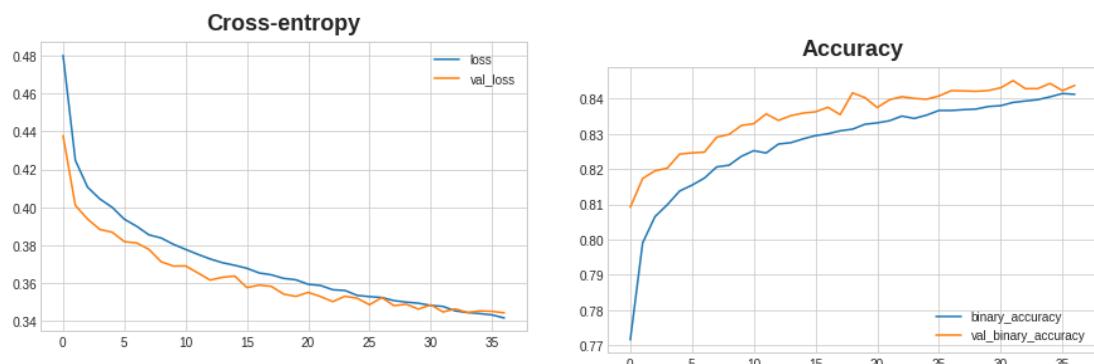
Podemos ver en la gráfica anterior una situación de claro sobre ajuste ya que la separación entre ambas curvas se incrementa y la curva naranja no mejora el ajuste desde el diez en adelante. En este caso deberemos reducir la complejidad del modelo para evitar el sobre ajuste, regularizar, o cualquier otro método que intente evitarlo.

El modelo también puede sufrir de Subajuste, lo que significa que el modelo no es lo suficiente complejo para capturar la estructura de los datos.



En la gráfica anterior podemos ver como las curvas son paralelas a partir del 15 y la distancia entre ambas es muy elevada, con lo que se puede ver un claro ejemplo de Subajuste. Se podría intentar aumentar la complejidad del modelo, etc.

Las dos siguientes imágenes muestran ejemplos de curvas de aprendizaje que no tienen ningún tipo de problemas, ambas tienden a juntarse cada vez más y aumentando la eficiencia continuamente.



Equilibrio tendencia-varianza (Bias - Variance)

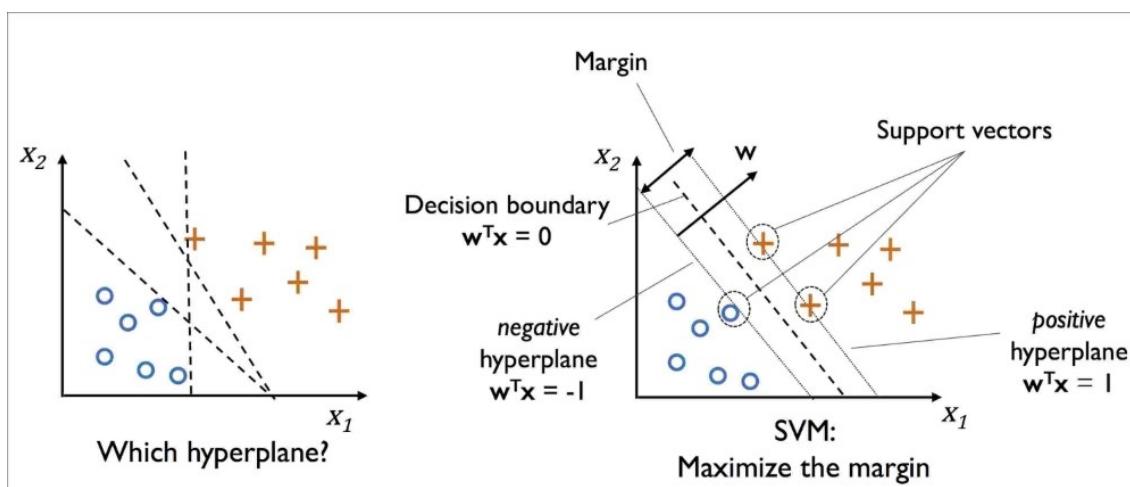
Se utiliza este término para describir el rendimiento de un modelo. Diremos que una alta varianza es proporcional a un sobreajuste. La varianza mide la consistencia del modelo y es sensible al orden de los datos de entrada.

Por el contrario, la tendencia mide cómo de lejos estás las predicciones de la realidad y no es susceptible al orden o a diferentes conjuntos de entrenamiento. La tendencia es la medida del error sistemático que no está relacionado con la aleatoriedad.

Una buena manera de encontrar un buen equilibrio ante la tendencia y la varianza es la regularización.

Máquinas de vector soporte

Las MVS pueden ser consideradas una extensión del Perceptrón. Se basan en minimizar los errores de clasificación maximizando el margen. El **margen** se define como la distancia entre el hiperplano de separación (bordes de decisión) y los ejemplos que están más cerca al mismo que se denominan: vectores de soporte. Este tipo de algoritmo se puede utilizar tanto para clasificación lineal o no lineal, regresión o detector de valores extremos.



Maximizando el margen

La idea de maximizar el margen es que éste tenderá a tener un error de generalización menor. Las MVS utilizan de base la siguiente ecuación:

$$\frac{W_t(x_{pos} - X_{neg})}{\|w\|} = \frac{2}{\|w^2\|}$$

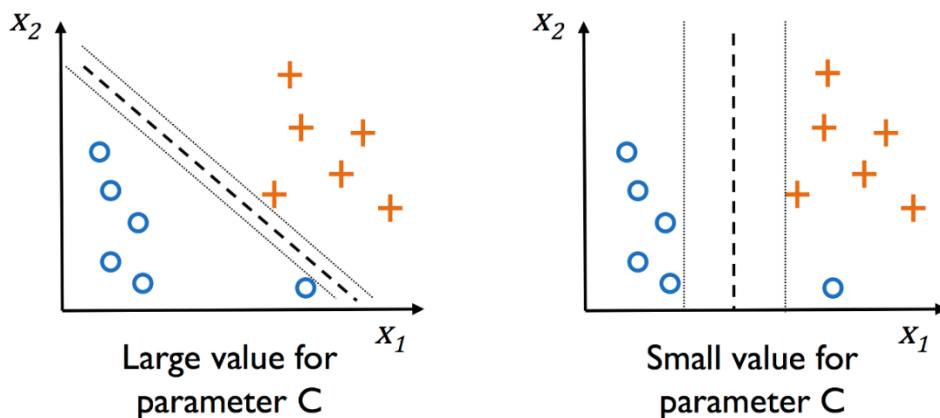
Pero en la práctica es menos costoso utilizar $\frac{1}{2}\|w^2\|$ que es el recíproco de la segunda parte de la ecuación.

Variable débil (ε)

Se comprobó que usando una nueva variable (slack ε) en la fórmula anterior hacía que las restricciones lineales pudieran ser relajadas para casos no lineales y permitir la optimización de los errores de clasificación, teniendo en cuenta una penalización. Se utiliza la siguiente fórmula:

$$\frac{1}{2}\|w^2\| + C \sum_i \varepsilon^i$$

En esta fórmula aparece el parámetro **C** que se comporta con un control de la penalización. En concreto valores pequeños de C dan valores grandes de penalización en la clasificación de los errores y viceversa (recordar que es el recíproco, de la ecuación). En definitiva, se puede controlar la anchura del margen con el parámetro C según la siguiente figura. Si el modelo MVS está sobre ajustado, se puede intentar regularizar reduciendo el valor de C



En la práctica real, tanto la regresión lineal logística como las SVM tienen un rendimiento similar.

Código 008.py

Parámetros importantes en las MVS

- **C** (float), default=1.0. Es la inversa de la regularización, debe ser un valor real positivo. Valores más pequeños implican mayor regularización
- **kernel**{‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’}, default=‘rbf’. Algoritmo a usar para realizar el ajuste.
- **degree** int, default=3. Coeficiente del polinomio a usar si el kernel es poly
- **coef0** float, default=0.0. Valor del término independiente si el kernel es poly
- **probability** bool, default=False. Indica si hay que estimar la probabilidad de una clase

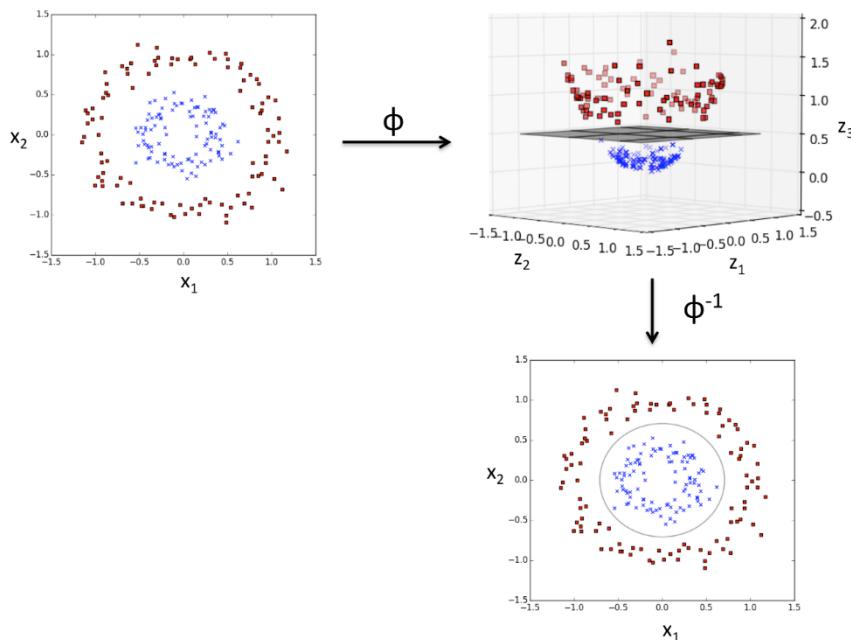
Resolviendo problemas no lineales con el Kernel MVS

Una razón de que MVS es tan popular es por la posibilidad de usar diferentes kernel para solucionar los problemas de clasificación que no son linealmente separables. La idea principal es crear combinaciones no lineales de las características originales y proyectarlas a un espacio dimensional mayor que sea separable linealmente.

El problema de construir esas características es que son computacionalmente muy costosas, por lo que hay que buscar algún atajo. En concreto, uno de los kernel más usado es el RBF (Radial basic function) o kernel Gausiano que se expresa simplificado así:

$$k(x^i, x^j) = \exp(-\gamma \|x^i - x^j\|^2)$$

Donde $\gamma = \frac{1}{2\sigma^2}$ es un parámetro libre para optimizar. El mayor problema de este kernel es la complejidad computacional del mismo.



El parámetro γ (gamma) establecido a un valor pequeño hará que los bordes del área sean más suaves, mientras que un valor mayor los convertirá en más abruptos. Hay que tener en cuenta que si un modelo se ajusta mucho a los datos de entrenamiento sufrirá overfit (sobreajuste). Si el modelo está sobre ajustado, hay que reducirlo, si está subajustado hay que aumentarlo.

```
svc = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svc = SVC(kernel='rbf', random_state=1, gamma=100, C=1.0)
```

Otra aproximación es añadir más características en forma de características polinomiales tal y como hemos hecho anteriormente en la regresión logística y aplicar este modelo. En este caso es tan simple como añadir al modelo SVC el *kernel='poly'*, indicando los parámetros *degree*, *coef0* (este parámetro controla la influencia entre las características polinomiales de alto grado frente a las que tienen un grado bajo) y *C*. Si el modelo se sobreajuste, siempre podremos reducir el coeficiente del kernel para ajustarlo.

Para terminar, hay que recordar que: empezar por el kernel lineal con *LinearSVC* mejor que *SVC(kernel='linear')* ya que el primero es muchísimo más eficiente. Si no se ajusta bien, podemos pasar al *kernel='rbf'* que suele funcionar en la mayoría de los casos, pero a costa del tiempo de cálculo.

Implementaciones alternativas

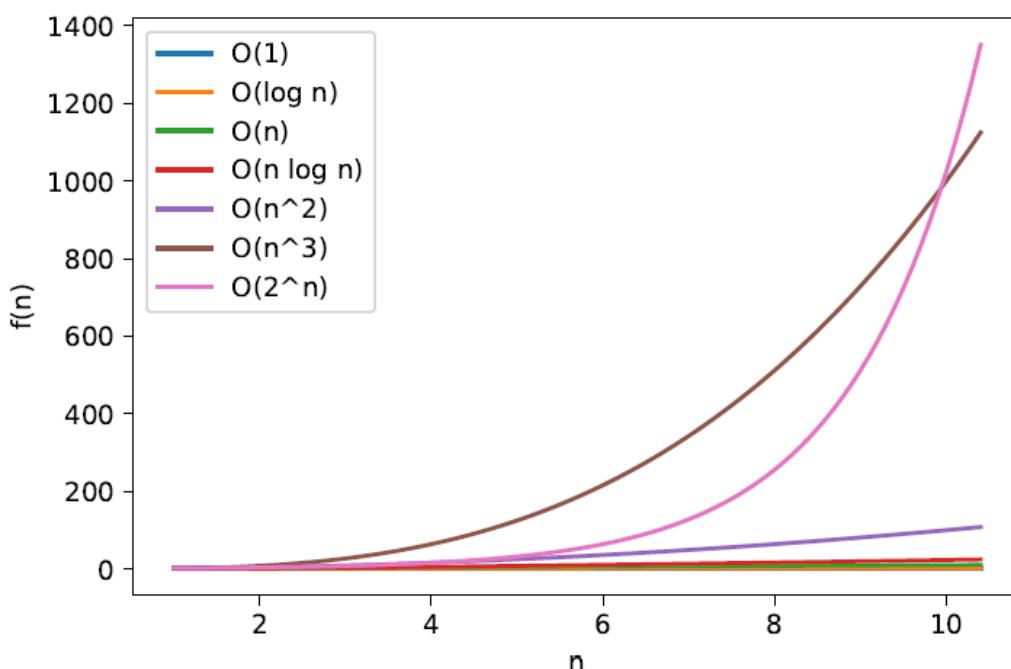
Algunos conjuntos de datos son muy grandes como para gestionarlos en memoria, por lo que la librería scikit-learn proporciona implementaciones alternativas eficientes. En concreto podemos usar el **SGDClassifier** dentro de los modelos lineales. Esta clase es similar al algoritmo de descenso estocástico del gradiente y se puede inicializar tanto al Perceptrón como al algoritmo de regresión logística o a una SVM.

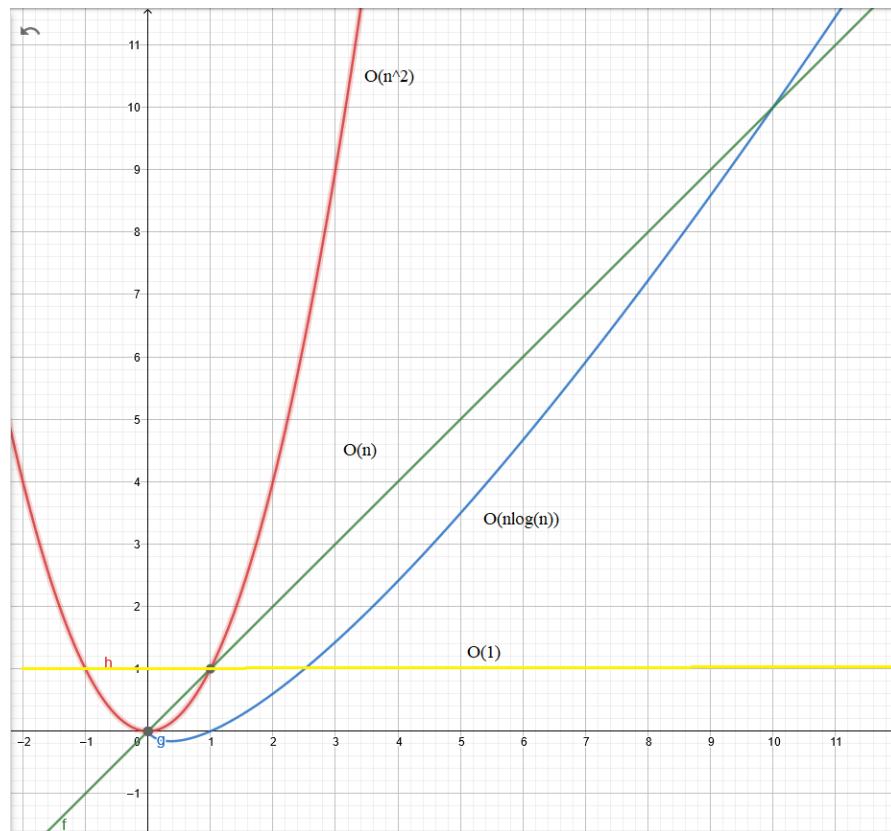
```
from sklearn.linear_model import SGDClassifier
ppn = SGDClassifier(loss='perceptron')
lr = SGDClassifier(loss='log')
svm = SGDClassifier(loss='hinge') # Similar MVC
```

Órdenes de complejidad

Los algoritmos informáticos o matemáticos se miden según su orden de complejidad O . Este se calcula en función del tiempo que tardaría en hacer dicha operación en función del número de elementos y viene expresado de forma de expresión matemática: $O(1)$, $O(n)$, $O(n^2)$, $O(n \log n)$, en el primer caso indica que el tiempo del algoritmo es constante independientemente del número de elementos, en el segundo que el tiempo crece de forma lineal y en el tercero de forma exponencial al cuadrado. Eso significa que para 5 y 25 elementos el primero tardará siempre lo mismo, el segundo tardará 5 en el primer caso y 25 en el segundo; y para el tercer orden de complejidad tardará 5^2 en el primer caso de ejemplos y 25^2 en el segundo, si lo computamos como segundos: 25 en el primero y 625 en el segundo caso.

Una vez entendidos los órdenes de complejidad diremos que: el algoritmo LinearSVC tiene un orden de $O(m \times n)$, el SGDClassifier de $O(m \times n)$ y el SVC desde $O(m^2 \times n)$ hasta $O(m^3 \times n)$, donde m es el número de ejemplos y n el número de características.

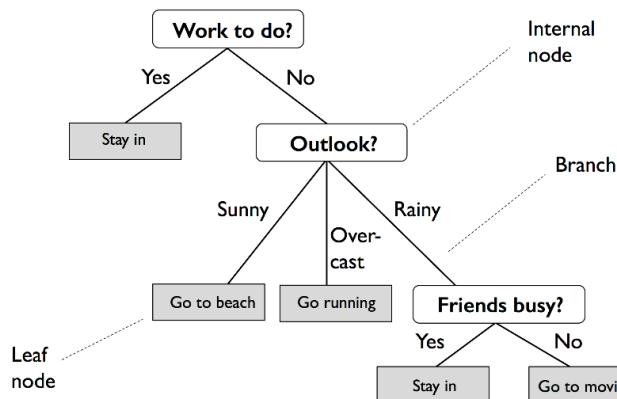




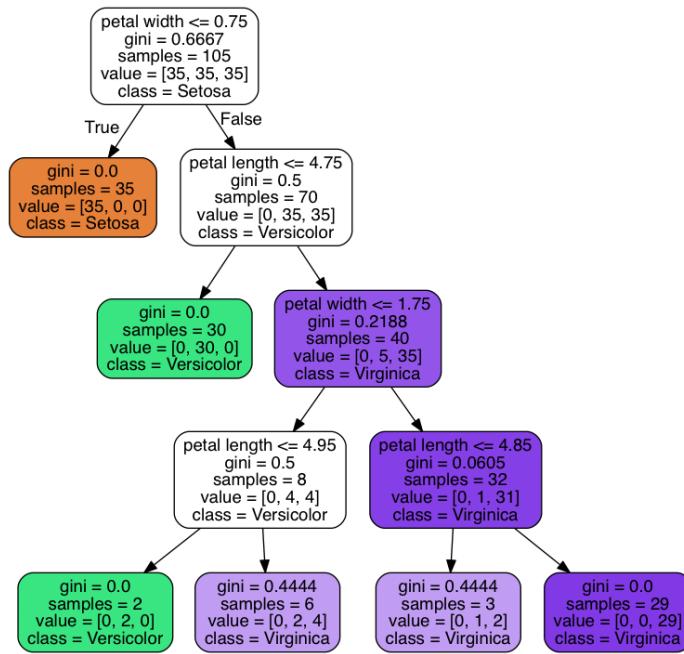
En la gráfica superior podemos comprobar como para dos o menos elementos el orden $O(n \log(n))$ es el mejor, siendo incluso mejor para menos de diez elementos que el $O(n)$. A partir de ahí vemos que $O(n)$ es la mejor aproximación ya que $O(1)$ solo se puede conseguir en lecturas/escrituras pero no en algoritmos complejos.

Árboles de decisión

Un árbol de decisión es un algoritmo que divide los datos en función de una serie de preguntas.



Se empieza desde la raíz y se va dividiendo los datos de las características buscando la mayor ganancia de información (I_g). Este proceso se hace de forma iterativa durante la división de los datos. Si no ponemos límites a la iteración llegaremos al sobreajuste.



La función objetivo de este algoritmo es la siguiente:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_{padre}} I(D_{left}) - \frac{N_{right}}{N_{padre}} I(D_{right})$$

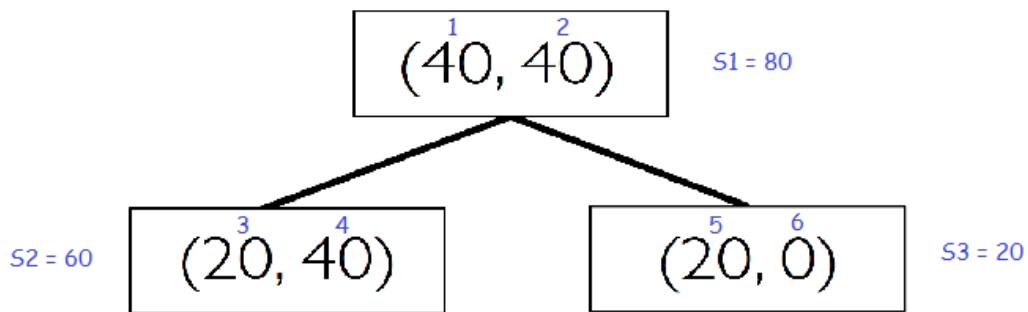
Dónde $I(D_x)$ es el cálculo de la impureza de la solución en el padre o en cada nodo, N se refiere al número total de ejemplos en cada nodo. Para el cálculo de la impureza se utilizan tres aproximaciones diferentes ($p(i|t)$) se refiere a la probabilidad de una clase en el nodo correspondiente):

- Entropía: $I_h(t) = - \sum_{i=1}^{num_nodos} p(i|t) \log_2 p(i|t)$. Esta función es máxima cuando la distribución es uniforme de los ejemplos y cero cuando todos los ejemplos de un nodo son de la misma clase.
- Gini: $I_g(t) = 1 - \sum_{i=1}^{num_nodos} p(i|t)^2$. Se puede entender como un criterio para minimizar la probabilidad de errores mal clasificados. Es máxima si las clases están perfectamente mezcladas.
- Error de clasificación: $I_e(t) = 1 - \max \{p(i|t)\}$. Esta impureza se utiliza sobre todo para realizar la poda del árbol en vez de para su creación.

De los tres criterios anteriores, los dos primeros funcionan similar y se puede usar cualquiera de ambos, es mejor investigar el punto de poda (*max_depth*) para evitar el sobreajuste que el cálculo de la impureza. Generalmente el mecanismo *Gini* está en medio entre la *Entropía* y el *Error de clasificación*.

En definitiva, un árbol de decisión creará un modelo cuya estructura sean rectángulos cada vez más pequeños para ajustarse a los datos. Si no paramos en algún momento se llegará a que cada ejemplo está en una única área de clasificación. **Es muy importante destacar que no es necesario escalar los datos para este mecanismo.**

Ejemplo de clasificación con error de clasificación



$$Ie(D_p) = 1 - \max\left(\frac{1}{80}, \frac{2}{80}\right) = 1 - 0,5 = 0,5 \quad I_e(t) = 1 - \max\{p(i|t)\}$$

A
S1

$$Ie(D_l) = 1 - \max\left(\frac{3}{60}, \frac{4}{60}\right) = 1 - \frac{4}{6} = \frac{1}{3} \quad I_e(t) = 1 - \max\{p(i|t)\}$$

B
S2

$$Ie(D_r) = 1 - \max\left(\frac{5}{20}, \frac{6}{20}\right) = 1 - \frac{2}{2} = 0 \quad I_e(t) = 1 - \max\{p(i|t)\}$$

C
S3

$$IG = 0,5 - \frac{60}{80} * \frac{1}{3} - \frac{20}{80} * 0 = 0,25 \quad IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_{padre}} I(D_{left}) - \frac{N_{right}}{N_{padre}} I(D_{right})$$

A S1 B S1 C

Si queremos usar cualquier otra función solo hay que cambiar los cálculos de **Ie** en todos.

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='gini', max_depth=4,
                               random_state=1)
```

Este algoritmo utiliza dos hijos exclusivamente, no siendo posible aumentar el número de ellos. Hay otros algoritmos que pueden tener más de dos hijos.

Bosques aleatorios

Este algoritmo es conocido por su buena escalabilidad y su facilidad de uso. Se le puede considerar como un conjunto de árboles de decisión.

El algoritmo utiliza un parámetro importante que es el número de árboles (K). la mayor ventaja es que no hay que preocuparse por los hiperparámetros. En general, Cuantos más árboles más rendimiento en el clasificador, pero más coste computacional.

```
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(criterion='gini', n_estimators= 25,
                                 random_state=1, n_jobs=-1)
```

En el código anterior el parámetro **n_estimators** indica el número de árboles, y el **n_jobs** el número de procesadores a usar para realizar el cálculo (paralelismo de cómputo).

K-vecinos (KNN)

Este algoritmo es sustancialmente diferente a los estudiados hasta ahora, es un ejemplo de aprendizaje perezoso. No aprende a discriminar una función de los datos de entrada, en vez de eso memoriza el conjunto de datos.

Este algoritmo se basa en los siguientes pasos:

1. Elegir el número de k vecinos y la métrica de distancia.
2. Encontrar los k-vecinos más cercanos en función de la métrica.
3. Asignar la clase en función del voto mayoritario.

Este algoritmo hace conjuntos de k-vecinos basándose en la distancia que nosotros elijamos. Se adapta muy bien a recoger datos de prueba on-line pero el coste computacional se incrementará de forma lineal con el número de nuevos casos, por lo que el espacio en memoria puede ser un problema con grandes conjuntos de prueba.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
```

La correcta elección del parámetro k (número de vecinos $n_neighbors$ en el código anterior) es crucial para un buen funcionamiento, así como de la métrica. Generalmente la distancia euclídea es una buena aproximación, pero deberemos estandarizar los datos.

Existen dos parámetros claves, el número de vecinos (k) y la métrica. Si la métrica incluye algún otro parámetro también será necesario establecerlo, como es el caso del código anterior.

$$\text{Minkowski} = d(x^i, y^j) = \sqrt[p]{\sum_k |x_k^i - x_k^j|^p}$$

Como vemos es necesario establecer el parámetro p de la métrica para que se pueda utilizar. En concreto si $p=2$, es la definición de métrica euclídea.

```
knn = KNeighborsClassifier(n_neighbors=5, metric= 'euclidean')
```

Esta definición es la misma que el primer código.

Métricas existentes para valores reales

Identificador	Nombre de clase	args	Función de distancia
“euclidean”	EuclideanDistance		<code>sqrt(sum((x - y)^2))</code>
“manhattan”	ManhattanDistance		<code>sum(x - y)</code>
“chebyshev”	ChebyshevDistance		<code>max(x - y)</code>
“minkowski”	MinkowskiDistance	<code>p</code>	<code>sum(x - y ^p)^(1/p)</code>
“wminkowski”	WMinkowskiDistance	<code>p, w</code>	<code>sum(w * (x - y) ^p)^(1/p)</code>
“seuclidean”	SEuclideanDistance	<code>V</code>	<code>sqrt(sum((x - y)^2 / V))</code>
“mahalanobis”	MahalanobisDistance	<code>V o VI</code>	<code>sqrt((x - y)' V^-1 (x - y))</code>

Existe un problema con este algoritmo: qué pasa si los vecinos están muy separados entre sí en un espacio de alta dimensionalidad. En este caso se recurren a las técnicas de reducción de dimensionalidad o a la reducción de características.

Ejercicios

- Usar los Dataset data_1.csv y data_2.csv para hacer un estudio con todos los algoritmos vistos en la unidad
- Usar el Dataset social_network.csv para hacer un estudio con todos los algoritmos vistos en la unidad
- Usar el Dataset test_data.txt, train_data.txt para hacer un estudio con todos los algoritmos vistos en la unidad

UT-3

Cap 5.- Preprocesado de datos

¿Qué son los datos? Definiremos datos como el conjunto de símbolos o signos que representan una calidad o modelo de la realidad.

La calidad de los datos y la cantidad de información que contienen son puntos clave para el buen funcionamiento de los algoritmos de ML. El Preprocesado de los datos es crucial. Asegurar que los datos son adecuados y correctos es el primer paso para encontrar una buena solución.

La información a nivel de datos, será para nosotros la variación dentro una característica, es decir, la diversidad de valores que hay en una característica. Generalmente podremos encontrar una medida de la información a través de la varianza estadística, cuanto mayor sea la varianza, mayor será la información contenida en el atributo.

Algunos atributos tienen una distribución de los datos asimétrica y se pueden transformar mediante funciones matemáticas a una distribución más general, por ejemplo, usando logaritmos.

<https://scikit-learn.org/stable/modules/preprocessing.html#>

El trabajo de analista de datos es principalmente conectar el problema de empresa con los datos y con la tecnología.

Hay que tener en cuenta que el preprocesado de datos para el análisis de datos es diferente que para machine Learning

Antes de realizar las tareas de este capítulo, tendremos en cuenta los siguientes puntos en cuanto a nuestros datos como paso preliminar:

- Deberían estar en una forma estructurada, por ejemplo, las fechas tendrían que estar separadas por campos en vez de en uno único.
- Los nombres de las columnas son fáciles y no contienen caracteres separadores.
- Cada columna tiene un identificador único.
- Crear las columnas agregadas necesarias.

Problemas comunes de los datos

Los defectos más comunes que encontramos en los datos son:

- **Ausencia de valores:** Es muy común que, si estamos trabajando con un conjunto de datos grande, algunos valores estén vacíos. Esto puede ocurrir por infinidad de motivos (ausencia de medición real, error al almacenar la información, error al recuperarla, etc.) El principal problema de estas ausencias es que impiden que el sistema de aprendizaje automático pueda entrenarse correctamente ya que la ausencia de datos no es numéricamente tratable. Para solucionar este problema existen varias aproximaciones que pueden ser utilizadas dependiendo de los datos que estemos tratando:
 - Interpolación: Si estamos tratando con datos temporales, una política que se suele llevar a cabo es la de interpolar el dato ausente teniendo en cuenta los datos próximos.

- Rellenar con un valor fijo: Como puede ser la media, la moda o incluso 0.
- Rellenar utilizando regresión: Existen técnicas que tratan de intentar predecir el valor perdido utilizando el resto de las variables de nuestro conjunto de datos. Uno de estos métodos es MICE (Multivariate Imputation by Chained Equation).
- Considerar el vacío como una categoría: Si la variable donde faltan datos es categórica, podemos añadir una categoría extra que aglutine todos aquellos cuyo campo está vacío.
- Eliminar el registro completo: En el caso de que ninguna de las técnicas anteriores sea adecuada para llenar los valores vacíos, en ocasiones se opta por descartar ese registro y trabajar exclusivamente con aquellos que sí que estén completos
- **Inconsistencia de datos:** Ocurre en muchas ocasiones que cuando estamos procesando datos detectamos errores en el formato o en el tipo de alguno de ellos. Esto puede ser debido a un error de lectura de los datos o a un mal almacenamiento de estos. Por ejemplo: fechas que siempre comienzan por el día del mes y en ciertos registros empiezan por el año, valores que deberían ser numéricos que incluyen otro tipo de caracteres, etc. Existen multitud de validaciones que es necesario comprobar y tratar de solucionar para que los datos sean coherentes. En este punto, dependiendo de la variable que estemos analizando, se deberán aplicar unas técnicas de validación u otras, que en ocasiones puede llegar a requerir conocimiento experto del problema.
- **Valores duplicados:** Puede suceder que en alguna ocasión encontramos registros duplicados en nuestro conjunto de datos. Es importante detectar estos registros y eliminar todos aquellos que se repitan más de una vez. No hacer esto podría suponer que el elemento duplicado sea tenido más en cuenta que el resto de los datos por parte del método de aprendizaje automático y, por lo tanto, éste se entrena de manera sesgada.
- **Outliers:** Debido a errores de almacenamiento, de medición o de inserción de los datos, pueden aparecer outliers (datos anómalos) en alguno de nuestros campos. Estos valores pueden distorsionar mucho la distribución de los datos haciendo que todo el proceso de aprendizaje se vea afectado. Existen muchas técnicas destinadas a intentar detectar estos datos anómalos.

Procedimientos comunes

```
import pandas as pd

df = pd.read_csv('iris.csv')
df.head()

Id SepalLength[cm] SepalWidth[cm] PetalLength[cm] PetalWidth[cm] Species
0 1 5.1 3.5 1.4 0.2 Iris-setosa
1 2 4.9 3.0 1.4 0.2 Iris-setosa
2 3 4.7 3.2 1.3 0.2 Iris-setosa
3 4 4.6 3.1 1.5 0.2 Iris-setosa
4 5 5.0 3.6 1.4 0.2 Iris-setosa

df.shape
```

(150, 6)

```
pd.read_csv(
    filepath_or_buffer: Union[str, pathlib.Path, IO[~AnyStr]],
    sep=',',
    delimiter=None,
    header='infer',
    names=None,
    index_col=None,
    usecols=None,
    squeeze=False,
    prefix=None,
    mangle_dupe_cols=True,
    dtype=None,
    engine=None,
    converters=None,
    true_values=None,
    false_values=None,
    skipinitialspace=False,
    skiprows=None,
    skipfooter=0,
    nrows=None,
    na_values=None,
    keep_default_na=True,
    na_filter=True,
    verbose=False,
    skip_blank_lines=True,
    parse_dates=False,
    infer_datetime_format=False,
    keep_date_col=False,
    date_parser=None,
    dayfirst=False,
    cache_dates=True,
    iterator=False,
    chunksize=None,
    compression='infer',
    thousands=None,
    decimal: str = '.',
    lineterminator=None,
    quotechar='''',
```

```
y = df['Species'].values
y
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

```
X = df.iloc[:, 1:5].values
```

```
X[:5]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
```

Explorando los datos

Familiarizarse con los atributos y su significado es primordial para poder hacer una preprocesado de datos adecuado.

Dos atributos numéricicos

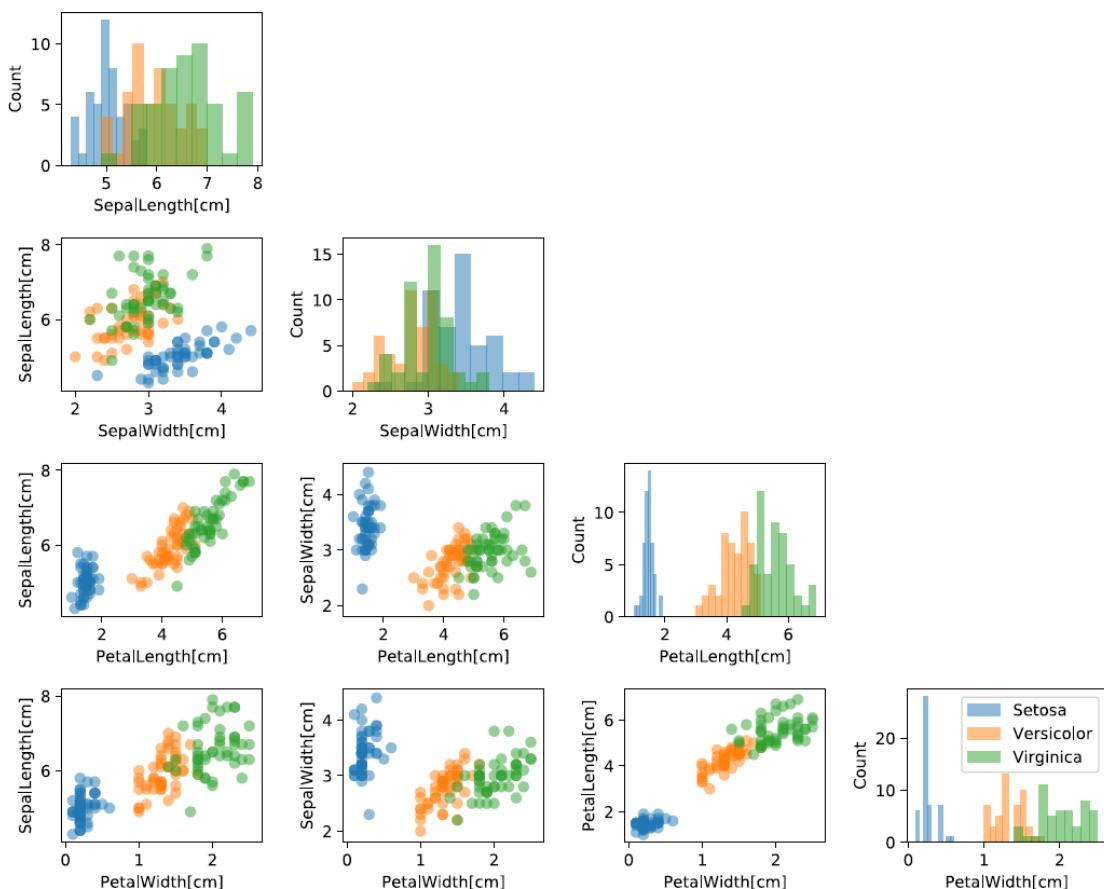
En este caso se recomienda hacer gráfico **scatter** entre ambos atributos y ver la distribución.

```
import matplotlib.pyplot as plt
from mlxtend.data import iris_data
from mlxtend.plotting import scatterplotmatrix

names = df.columns[1:5]

fig, axes = scatterplotmatrix(X[y==0], figsize=(10, 8), alpha=0.5)
fig, axes = scatterplotmatrix(X[y==1], fig_axes=(fig, axes), alpha=0.5)
fig, axes = scatterplotmatrix(X[y==2], fig_axes=(fig, axes), alpha=0.5, names=names)

plt.tight_layout()
plt.legend(labels=['Setosa', 'Versicolor', 'Virginica'])
plt.show()
```



Dos atributos categóricos.

En este caso se recomienda crear una tabla de contingencia. Una tabla de contingencia es una matriz que muestra la frecuencia de los datos para cada combinación de dos atributos.

```
tbl_contingencia = pd.crosstab(df.atributo_1, df.atributo_2)
prob_tbl_contingencia = tbl_contingencia / tbl_contingencia.sum()
sns.heatmap(prob_tbl_contingencia, annot=True, center=0.5)
plt.show()
```

Un atributo numérico y otro categórico.

En este caso se recomienda transformar el atributo numérico en categórico y usar el método anterior (**bins** indicará el número de conjuntos a crear).

```
dato_transformado_a_categorico = pd.cut(df.attr_numerico, bins=5)
```

Ausencia de valores

Podemos encontrar datos faltantes representados como espacios en blanco, valores NaN o como NULL, desafortunadamente la mayoría de las herramientas que usaremos no saben lidiar con este tipo de datos y habrá tratarlos de forma correcta.

Los valores nulos se pueden clasificar en tres categorías: **MCAR**, **MAR** y **MNAR**. MCAR indica que el valor es completamente aleatorio y no depende de nada y puede aparecer en cualquier objeto. MAR indica que hay objetos en los que los nulos sean más probables que en otros (por ejemplo, un sensor que solo falla a partir de una cierta velocidad). MNAR indica que el valor nulo sabemos exactamente en qué objetos va a aparecer. Estos últimos son los más difíciles de tratar.

Para determinar si los valores nulos de una característica son de uno de los tres tipos, estudiaremos la correlación de dichos valores con el resto de propiedades, en caso que no hay ninguna correlación se dirá que son nulos MCAR, si hay correlación con al menos una característica de forma notable se dirá que es MAR y si la correlación es fuerte serán MNAR.

Ver: code_0009_a.ipynb

Dependiendo el tipo tendremos las siguientes opciones.

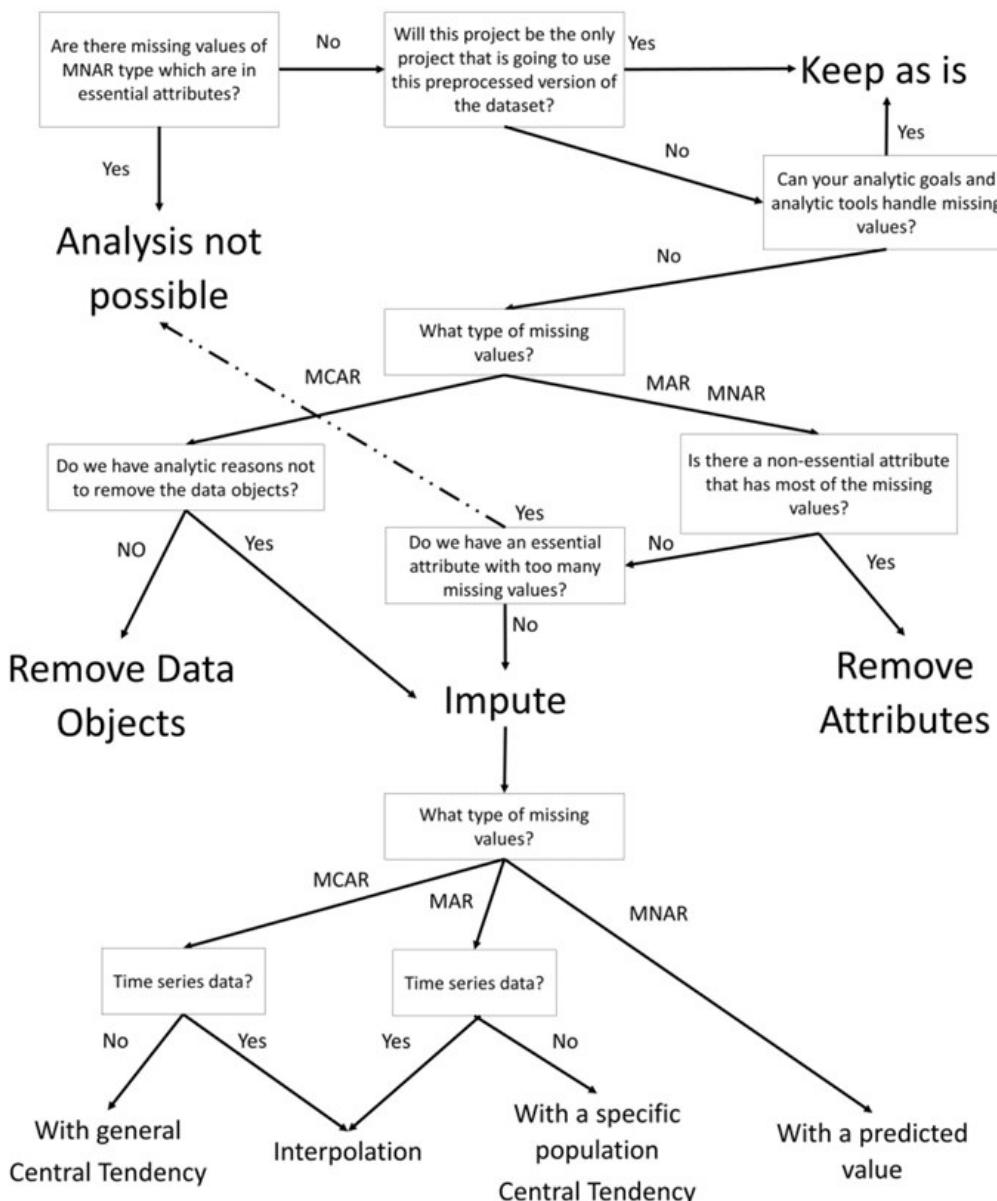
- Encontrar el número de valores nulos por característica:
dataframe.isnull().sum()
- Eliminar las filas con NaN
dataframe.dropna(axis=0)
- Eliminar las columnas con NAN
dataframe.dropna(axis = 1)
- Eliminar solo aquellas filas con todas sus columnas a NaN
dataframe.dropna(how= 'all')
- Eliminar aquellas filas con menos De un número de valores. (4 En el ejemplo)
dataframe.dropna(thresh = 4)
- Eliminar aquellas filas en las que el NaN está en una columna específica
dataframe.dropna(subset=['c'])

Pero el método de eliminar datos tiene algunas desventajas: Se puede eliminar demasiados ejemplos o se pueden quitar muchas características (columnas) de entrada.

Otra posibilidad es asignar valores a esos datos que faltan mediante técnicas de interpolación. El método más común es el de asignación a la Media (mediana, a cero, a uno, a máximo o a mínimo): simplemente se remplaza el valor faltante por la media de su característica.

```
df.fillna(df.mean())
```

En general para tipos MCAR se imputarán valores según una tendencia central (media, mediana o moda). Para valores tipo MAR mejor imputar con una tendencia central del conjunto más característico del atributo. Para MNAR una aproximación es imputar los valores faltantes con una regresión.



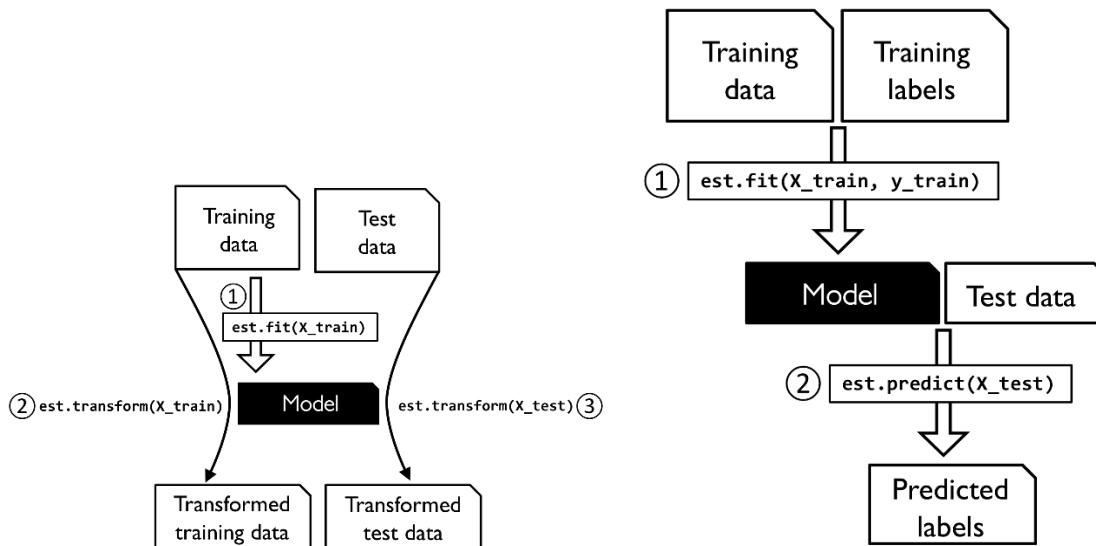
Entendiendo scikit-learn estimator

La clase **SimpleImputer** pertenece a las clases transformadoras de la librería. El método **fit** se usa sobre los datos de entrenamiento para ajustar la clase, y después usaremos el método **transform** para cambiar los datos finales.

```

from sklearn.impute import SimpleImputer
import numpy as np
imr = SimpleImputer(missing_values=np.nan, strategy='mean')
imr = imr.fit(df.values)
inputed_data = imr.transform(df.values)
  
```

Hay otro tipo de clases, los estimadores, que son muy similares a la anterior, pero añaden un método **predict** para predecir las etiquetas faltantes.



<https://scikit-learn.org/stable/modules/impute.html>

Valores extremos

Un valor extremo en ML es un valor atípico, destacado, excepcional, anormal, valor extremo.

Es decir, que los outliers en nuestro Dataset serán los valores que se “escapan al rango en donde se concentran la mayoría de muestras”. Según Wikipedia son las muestras que están distantes de otras observaciones.

La detección de los valores extremos es importante porque pueden afectar considerablemente a los resultados que pueda obtener un modelo de Machine Learning y se basan Los métodos se basan en construir bandas (intervalos) en los que deben estar los datos. Si un dato se sale de dicha banda se dictamina que se trata de un outlier.

Outliers Buenos vs Outliers Malos

Los Outliers pueden significar varias cosas:

- ERROR: Si tenemos un grupo de “edades de personas” y tenemos una persona con 160 años, seguramente sea un error de carga de datos. En este caso, la detección de outliers nos ayuda a detectar errores.
- LIMITES: En otros casos, podemos tener valores que se escapan del “grupo medio”, pero queremos mantener el dato modificado, para que no perjudique al aprendizaje del modelo de ML.
- Punto de Interés: puede que sean los casos “anómalos” los que queremos detectar y que sean nuestro objetivo (y no nuestro enemigo!)

Muchas veces es sencillo identificar los outliers en gráficas. Veamos ejemplos de outliers en 1, 2 y 3 dimensiones.

Outliers en 1 dimensión

Si analizáramos una sola variable, por ejemplo “edad”, veremos donde se concentran la mayoría de muestras y los posibles valores “extremos”. Pasemos a un ejemplo en Python.

```
import matplotlib.pyplot as plt
```

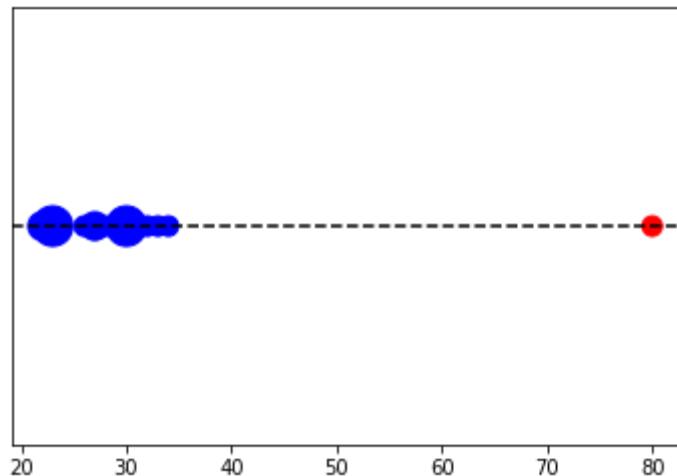
```
import numpy as np

edades =
np.array([22,22,23,23,23,23,26,27,27,28,30,30,30,30,31,32,33,34,80])
edad_unique, counts = np.unique(edades, return_counts=True)

sizes = counts*100
colors = ['blue']*len(edad_unique)
colors[-1] = 'red'

plt.axhline(1, color='k', linestyle='--')
plt.scatter(edad_unique, np.ones(len(edad_unique)), s=sizes,
color=colors)
plt.yticks([])
plt.show()
```

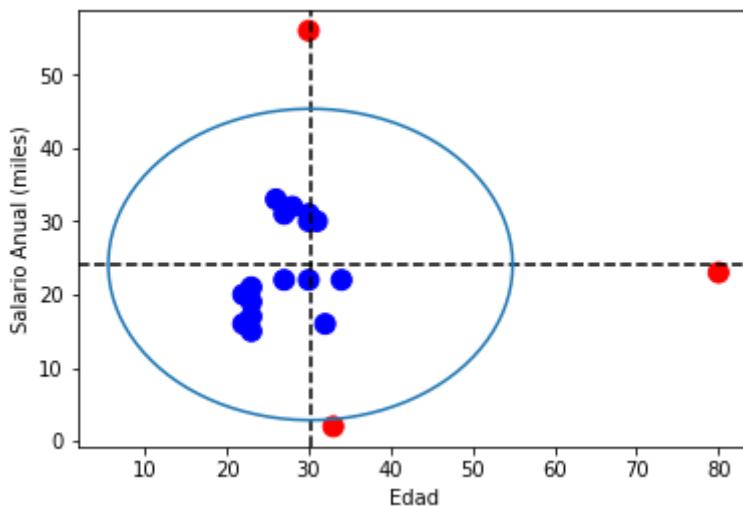
En azul los valores donde se concentra la mayoría de nuestras filas. En rojo un outliers, o “valor extremo”.

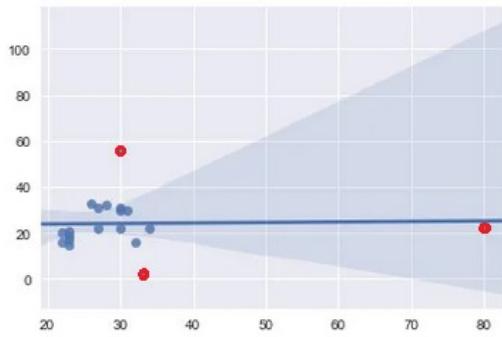


Al crear el gráfico vemos donde se concentran la mayoría de edades, entre 20 y 35 años. Y una muestra aislada con valor 80.

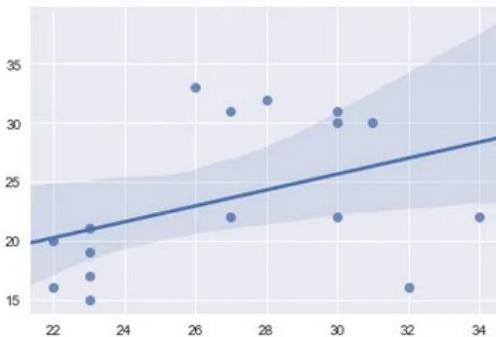
Outliers en 2 Dimensiones

Ahora supongamos que tenemos 2 variables: edad e ingresos. Hagamos una gráfica en 2D. Además, usaremos una fórmula para trazar un círculo que delimitará los valores outliers: Los valores que superen el valor de la “media más 2 desvíos estándar” (el área del círculo) quedarán en rojo.





CON OUTLIERS: La línea de tendencia se mantiene plana sobre todo por el outlier de la edad



SIN OUTLIERS: Al quitar los outliers la tendencia empieza a tener pendiente

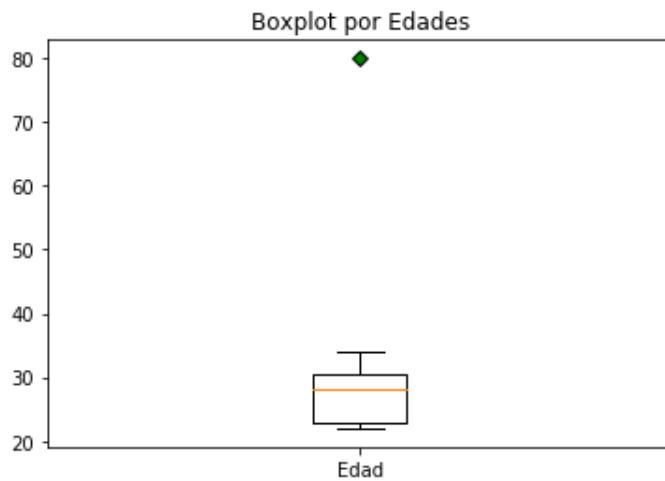
Con esto nos podemos hacer una idea de qué distinto podría resultar entrenar un modelo de Machine Learning con o sin esas muestras anormales.

Outliers en N-dimensiones

La realidad es que en los modelos con los que trabajamos constan de muchas dimensiones, podemos tener 30, 100 o miles. Entonces ya no parece tan sencillo visualizar los outliers.

Outliers con una librería

Una forma bastante interesante de conocer los outliers es la de los Boxplots, muy utilizados en el mundo financiero. En nuestro caso, podemos visualizar las variables y en esa “cajita” veremos donde se concentra el 50 por ciento de nuestra distribución (percentiles 25 a 75), los valores mínimos y máximos (las rayas en “T”) y -por supuesto- los outliers, esos “valores extraños” y alejados.



```
green_diamond = dict(markerfacecolor='g', marker='D')
fig, ax = plt.subplots()
ax.set_title('Boxplot por Edades')
ax.boxplot(edades, flierprops=green_diamond, labels=["Edad"])
```

Otras estrategias para delimitar outliers.

Una librería muy recomendada es PyOD. Posee diversas estrategias para detectar Outliers. Ofrece distintos algoritmos, entre ellos Knn que tiene mucho sentido, pues analiza la cercanía entre muestras

```
from pyod.models.knn import KNN
import pandas as pd
compras_mes = np.array([1,2,1,20,1,0,3,2,3,0,5,3,2,1,0,1,2,2,2])

X = pd.DataFrame(data={'edad':edades,'salario':salario_anual_miles,
                      'compras':compras_mes})

clf = KNN(contamination=0.18)
clf.fit(X)
y_pred = clf.predict(X)
X[y_pred == 1]
```

	compras	edad	salario
3	20	23	21
10	5	30	56
16	2	33	2
18	2	80	23

	compras	edad	salario
count	19.000000	19.000000	19.000000
mean	2.684211	30.210526	24.105263
std	4.372294	12.673590	10.913268
min	0.000000	22.000000	2.000000
25%	1.000000	23.000000	18.000000
50%	2.000000	28.000000	22.000000
75%	2.500000	30.500000	30.500000
max	20.000000	80.000000	56.000000
	compras	edad	salario
3	20	23	21
10	5	30	56
16	2	33	2
18	2	80	23

La librería PyOd detecta los registros anómalos. Para problemas en la vida real, con múltiples dimensiones conviene apoyarnos en una librería como esta que nos facilitará la tarea de detección y limpieza/transformación del Dataset.

Una vez detectados, ¿qué hago?

Según la lógica de negocio podemos actuar de una manera u otra: dejarlos, modificarlos o eliminarlos.

Ampliación

<https://machineLearningmastery.com/model-based-outlier-detection-and-removal-in-python/>

https://scikit-learn.org/stable/auto_examples/applications/plot_outlier_detection_wine.html

Datos categóricos

Generalmente, cuando recibimos un conjunto de datos, además de encontrar valores numéricos, en la mayoría de las ocasiones debemos trabajar con datos no estructurados, que son aquellos datos que no se rigen por un esquema determinado, como pueden ser los textos, imágenes, vídeos, etc. Todos estos datos, si queremos poder utilizarnos para entrenar nuestro modelo deben ser convertidos a números.

Es muy común en la recogida de datos que varias características sean categóricas en vez de numéricas. Dentro de este tipo diferenciaremos aquellas que tengan que estar ordenadas (XL > L > M > S) de aquellas que no tienen orden expreso (rojo, blanco, verde).

En cualquier caso, tendremos que hacer un mapeo de los valores de las categorías a valores numéricos. Teniendo en cuenta que esta asignación puede imponer un orden que antes no existía. Si pasamos los colores a valores numéricos (rojo=0, blanco=1, etc.) El rojo será menor que el blanco y así sucesivamente, lo que introduce información no existente en los datos y nos presentará un problema.

Mapeado manual

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df['categorical_1'] = df['categorical_1'].map(size_mapping)
```

```
class_mapping = {
    label: idx for idx, label in enumerate(np.unique(df['classlabel']))}
df['classlabel'] = df['classlabel'].map(class_mapping)
```

En caso de necesitar obtener los valores anteriores simplemente haremos:

```
inverse_mapping = [v: k for k,v in size_mapping.items()]
df['categorical_1'] = df['categorical_1'].map(inverse)
```

Mapeo con LabelEncoder

El proceso de codificación de etiquetas es muy normal, por lo que se ha implementado una clase **LabelEncoder** que realiza todo el proceso.

```
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
# reverse mapping
class_le.inverse_transform(y)
```

En el código anterior, hay que fijarse en la llamada **fit_transform**, que aúna las llamadas a **fit** y **transform** que anteriormente hacíamos de forma separada.

Codificación OneHot

La codificación manual o con la clase LabelEncoder introduce información no existente en los datos tal y como comentábamos al principio del punto. De hecho, cometemos uno de los errores más comunes: proporcionar orden a unas etiquetas que no tenían orden. Al asignar un valor numérico, hemos dado un orden implícito, aunque no queramos.

Para solucionar este problema podemos aplicar el codificador **OneHot** que creará una nueva característica (columna) por cada uno de los valores de la etiqueta a convertir, poniéndolo a 1 en la correspondiente y en el resto a cero.

Aplicando el codificador a una sola columna

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

colores = np.array(['rojo', 'rojo', 'verde', 'rojo', 'azul'])
precios = np.array([200, 100, 200, 333, 444])
```

```

etiquetas = np.array(['a', 'a', 'b', 'b', 'b'])
df = pd.DataFrame(data={'color': colores, 'precio': precios,
                       'etiqueta': etiquetas})

enc = OneHotEncoder(handle_unknown='ignore')
enc_df = pd.DataFrame(enc.fit_transform(df[['color']]).toarray())
df = df.join(enc_df)
df = df.drop(['color'], axis=1)
print(df)

```

Desde Pandas

```

import numpy as np
import pandas as pd

colores = np.array(['rojo', 'rojo', 'verde', 'rojo', 'azul'])
precios = np.array([200, 100, 200, 333, 444])
df = pd.DataFrame(data={'color': colores, 'precio': precios})
print(df)

df = pd.get_dummies(df, columns=["color"], prefix=["Type_is"])
print(df)

```

Aplicando el codificador a varias columnas a la vez.

```

import numpy as np
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

colores = np.array(['rojo', 'rojo', 'verde', 'rojo', 'azul'])
precios = np.array([200, 100, 200, 333, 444])
etiquetas = np.array(['a', 'a', 'b', 'b', 'b'])

df = pd.DataFrame(data={'color': colores, 'precio': precios,
                       'etiqueta': etiquetas})
c_transf = ColumnTransformer([('onehot', OneHotEncoder(), [0, 2]),
                             ('nothing', 'passthrough', [1])])

df = pd.DataFrame(c_transf.fit_transform(df[['color', 'precio',
                                             'etiqueta']]))

print(df)

```

Realizando el mismo ajuste a través de Pandas.

```

import numpy as np
import pandas as pd

colores = np.array(['rojo', 'rojo', 'verde', 'rojo', 'azul'])
precios = np.array([200, 100, 200, 333, 444])
etiquetas = np.array(['a', 'a', 'b', 'b', 'b'])
df = pd.DataFrame(data={'color': colores, 'precio': precios,
                       'etiqueta': etiquetas})
print(df)

X = df[['color', 'precio', 'etiqueta']].values
df = pd.get_dummies(df, columns=["color", "etiqueta"],
                     prefix=["P", "E"])

```

```
print(df)
```

Cuando utilizamos el codificador OneHot, deberemos de tener en cuenta que introduce multicolinealidad y puede presentarnos un problema para varios métodos muy comunes. Si las características están fuertemente correlacionadas, será computacionalmente muy costoso y puede provocar estimaciones inestables. Para reducir la correlación, podemos eliminar una de las nuevas características creadas, de esta forma no se pierde información (la que eliminamos está implícita en el resto de características).

Para eliminar una de las categorías redundantes se debe poner el atributo **categories** a **auto** y **drop** a **first**.

```
color_ohe = OneHotEncoder(categories='auto', drop='first')
c_transf = ColumnTransformer([('onehot', color_ohe, [0, 3]),
                             ('nothing', 'passthrough', [1, 2])])
c_transf.fit_transform(X).astype(float)
```

Count Encoding

Esta codificación remplaza cada categoría con el número de veces que aparece en el conjunto de datos.

Target Encoding

Esta codificación reemplaza cada con el valor medio de la característica.

CatBoost Encoding

Similar al anterior, pero en vez de tener en cuenta todos los ejemplos, se tiene en cuenta solo los ejemplos anteriores para el cálculo de la media.

Separando el conjunto de datos en datos de entrenamiento y datos de prueba

La forma más eficiente para dividir el conjunto de datos es con la función: **train_test_split**. A esta función se le pueden pasar diferentes Dataset con el mismo número de filas y los dividirá por los mismos índices. Esta función es muy útil cuando tenemos los datos y las etiquetas en ficheros diferentes.

Otro de los problemas que se presentan en los datos es la desigualdad de ejemplos entre las diferentes categorías a predecir, para mitigar errores, esta función puede dividir los ejemplos en función de la frecuencia de un conjunto de datos que se pasen en el parámetro **stratify**.

Normalización

Normalización significa comprimir o extender los valores de la variable para que estén en un rango definido. Sin embargo, una mala aplicación de la normalización, o una elección descuidada del método de normalización puede arruinar tus datos, y con ello tu análisis.

No existe un método ideal de normalización que funcione para todas las variables. Es trabajo del Data Scientist conocer cómo se distribuyen los datos, saber si existen anomalías, comprobar rangos, etc. Con este conocimiento, se puede seleccionar la mejor técnica para no distorsionar los datos.

La normalización de las características es crucial en el Preprocesado de datos. Aunque algunos algoritmos no lo necesiten (árboles de decisión y bosques aleatorios) la mayoría sí.

Hay dos aproximaciones de normalización: escalado y la estandarización. En la primera se pasan todos los valores a un rango de 0..1 y en la segunda se transformar a la distribución Normal. El

segundo método permite a los algoritmos aprender los pesos más rápidamente, pero puede hacer bajar el rendimiento del mismo. Además, mantiene más información sobre los extremos. Pero en ambos casos se puede dar un problema: La estandarización puede (posiblemente [peligrosamente]) distorsionar tus datos

$$\text{Escalado: } x_{esc}^i = \frac{x^i - x_{min}}{x_{max} - x_{min}}$$

$$\text{Estandarización: } x_{std}^i = \frac{x^i - \mu_x}{\sigma_x}$$

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler

df_wine = load_wine()
X, y = df_wine.data, df_wine.target
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=0,
                                                    stratify=y)

mms = MinMaxScaler()
X_train_esc = mms.fit_transform(X_train)
X_test_esc = mms.transform(X_test)

stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

Las clases más usadas son MinMaxScaler, StandarScaler y Normalizer. MinMaxScaler se corresponde con el proceso de escalado presentado anteriormente. La clase StandarScaler se corresponde con el proceso anterior de estandarización. Estas clases se ejecutan sobre la característica (columna). Por el contrario, Normalizer es similar a StandarScaler en cuanto a concepto, pero se ejecuta sobre el ejemplo (la fila).

Es importante remarcar que para usar el método **fit** deben utilizarse los datos de entrenamiento y ejecutará tanto para los datos de prueba como los de entrenamiento.

```
mms = MinMaxScaler()
mms.fit (X_train)
X_train_norm = mms.transform(X_train)
X_test_norm = mms.transform(X_test)
```

Como norma general necesitaremos hacer un escalado cuando el rango de todos los atributos del Dataset deba ser similar, mientras que haremos una estandarización cuando la media y la desviación estándar de las características deba ser similar.

Para terminar, la librería implementa otro método como **RobustScaler** muy recomendado si trabajamos con conjuntos de datos muy pequeños. Se basa en eliminar los valores medianos y escalar el resto de acuerdo al primer y tercer cuartil.

Ajustes de problemas del modelo

Ya sabemos que, si un modelo predice los datos mucho mejor en los de entrenamiento que en los de test, es muy posible que sea un sobreajuste. La posible razón es que nuestro modelo es muy complejo para los datos de entrenamiento. Las soluciones más comunes para esto son:

- Conseguir más datos para entrenar el modelo.
- Introducir una penalización a la complejidad del modelo vía regularización.
- Elegir un modelo más simple.
- Reducir la dimensionalidad de los datos.

Regularización L1 y L2

La regularización es una manera para manejar la colinealidad (alta correlación entre las características), filtrar el ruido y prevenir el sobreajuste.

La correlación de dos atributos indica la semejanza entre ambas variables. Este valor va desde menos uno (-1) hasta uno (1), siendo el cero el punto donde mayor independencia hay entre las variables y los extremos donde mayor. La matriz de correlación nos indica esta relación, estableciendo un valor de 0,7 (tanto positivo como negativos) como valor para estudiar la correlación entre dos atributos.

Matriz de correlación

```
import numpy as np
np.random.seed(10)

# generating 10 random values for each of the two variables
X = np.random.randn(10)

Y = np.random.randn(10)

# computing the correlation matrix
C = np.corrcoef(X,Y)

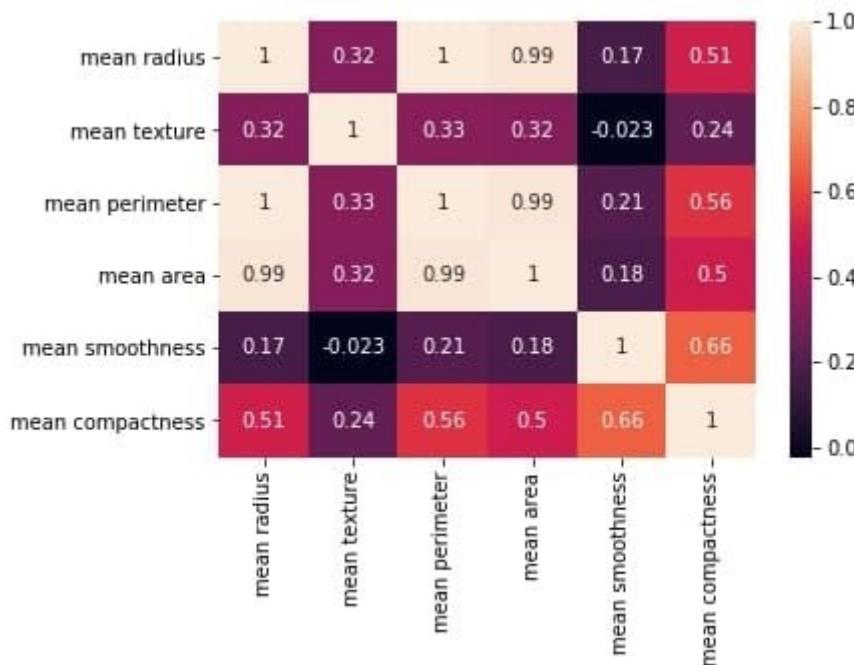
print(C)
```

Matriz de correlación para más de dos variables

```
from sklearn.datasets import load_breast_cancer
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

breast_cancer = load_breast_cancer()
X = breast_cancer.data
features = breast_cancer.feature_names
df_X = pd.DataFrame(X, columns=features)

# taking all rows but only 6 columns
df_small = df_X.iloc[:, :6]
correlation_mat = df_small.corr()
sns.heatmap(correlation_mat, annot=True)
plt.show()
```



Hay que tener en cuenta los siguientes puntos con respecto a las matrices de correlación como la que se muestra arriba:

- Cada celda de la cuadrícula representa el valor del coeficiente de correlación entre dos variables.
- El valor en la posición (a, b) representa el coeficiente de correlación entre los elementos de la fila a y la columna b. Será igual al valor en la posición (b, a)
- Es una matriz cuadrada – cada fila representa una variable, y todas las columnas representan las mismas variables que las filas, de ahí el número de filas = número de columnas.
- Es una matriz simétrica – esto tiene sentido porque la correlación entre $a \rightarrow b$ será la misma que la de $b \rightarrow a$.
- Todos los elementos diagonales son 1. Dado que los elementos diagonales representan la correlación de cada variable consigo misma, siempre será igual a 1.
- Los marcadores de los ejes denotan el rasgo que cada uno de ellos representa.
- Un valor positivo grande (cercano a 1,0) indica una fuerte correlación positiva, es decir, si el valor de una de las variables aumenta, el valor de la otra variable aumenta también.
- Un valor negativo grande (cercano a -1,0) indica una fuerte correlación negativa, es decir, que el valor de una de las variables disminuye al aumentar el de la otra y viceversa.
- Un valor cercano a 0 (tanto positivo como negativo) indica la ausencia de cualquier correlación entre las dos variables, y por lo tanto esas variables son independientes entre sí.
- Cada celda de la matriz anterior también está representada por sombras de un color. En este caso, los tonos más oscuros del color indican valores más pequeños, mientras que los tonos más brillantes corresponden a valores más grandes (cerca de 1). Esta escala se da con la ayuda de una barra de color en el lado derecho del gráfico.

Regularización L1 y L2

La regularización introduce información adicional para penalizar los valores extremos de los parámetros. La regularización más común es la llamada L2.

$$\frac{\lambda}{2} \|w\|^2 = \frac{\lambda}{2} \sum_{j=1}^n w_j^2$$

Para que la regularización funcione es imprescindible que las características sean comparables (se hayan escalado) las características.

Incrementar el valor de λ , aumenta la fuerza de la regularización. El término C de la Regresión Logística está relacionado con λ en que es el inverso. Por lo que si decrece C estará aumentando la fuerza de la regularización.

En este sentido, las regularizaciones L1 y L2 pueden ayudar a mejorar el modelo. La regularización L2 reduce la complejidad del modelo penalizando los valores altos de los pesos, con lo que se tiende a que los pesos sean cercanos a cero. Por el contrario, la regularización L1 hace que muchos de los pesos valgan cero, con lo que el efecto es que algunas características no cuentan y por tanto es una reducción de las mismas.

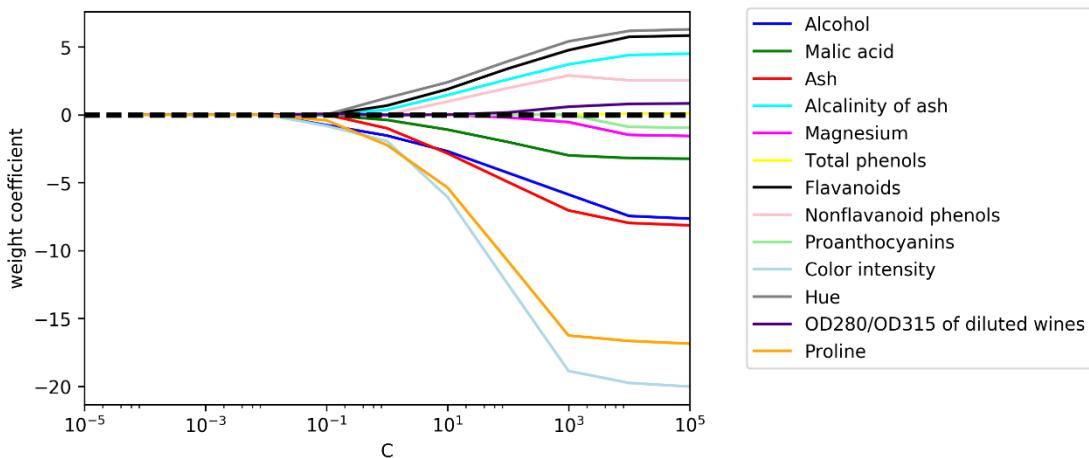
Ambas penalizaciones incluyen un parámetro C que determina la cantidad de regularización. Cuanto más pequeño es este parámetro, más fuerte es la regularización.

Código 009.py

No todos los algoritmos son capaces de usar las regularizaciones L1 y L2, de hecho, **Ibfsgs** no la soporta actualmente y hay que utilizar **liblinear** en su lugar.

```
lr = LogisticRegression(penalty='l1', C=1.0, solver='liblinear',
                       multi_class='ovr')
# Note that C=1.0 is the default. You can increase
# or decrease it to make the regularization effect.
# Smaller values specify stronger regularization.
lr.fit(X_train_std, y_train)
print('Training accuracy:', lr.score(X_train_std, y_train))
print('Test accuracy:', lr.score(X_test_std, y_test))
```

En la siguiente imagen vemos el efecto del parámetro C sobre los pesos de los coeficientes.



Para saber el valor de los pesos de un modelo podemos usar **intercept_** para el valor de W_0 y **coef_** para el valor del resto de pesos W_j (lr.coef_).

A medio camino entre la regularización L1 y la L2 se encuentra la regularización elástica. Esta regularización tiene un parámetro que determina la cantidad de regularización. En caso de valer 0, es equivalente la L2, en caso de valer 1 es equivalente a L1, y el resto de valores entre cero y uno hacen que se apliquen de forma proporcional las regularizaciones L2 y L1.

Algoritmos de selección de características

La reducción de la dimensionalidad mediante selección de características tiene dos aproximaciones: La primera (selección) en la que se van eliminando características hasta que llegamos al número pedido, y la segunda (extracción) en la que se generan el número pedido de nuevas características a partir de las existentes.

Para todos los casos, hay que indicar un criterio por el que seleccionar características y el más general es el que minimice el porcentaje de eficiencia del modelo antes y después de la eliminación.

Hay muchos algoritmos implementados en la librería para la reducción de características. Se puede ampliar en:

https://scikit-learn.org/stable/modules/feature_selection.html

Eliminando características con baja varianza

VarianceThreshold es una aproximación muy simple que elimina aquellas características que tienen varianza cero, o el mismo valor en todas las características.

```
from sklearn.feature_selection import VarianceThreshold
X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
print(sel.fit_transform(X))
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

Como era de esperar elimina la primera característica que tiene todo ceros.

Selección de características invariables

Este mecanismo selecciona las mejores características basándose en los test de invarianza estadística. Tenemos las siguientes aproximaciones:

- **SelectKBest** selecciona las mejores características.

Este algoritmo utiliza una función de selección que debe ser:

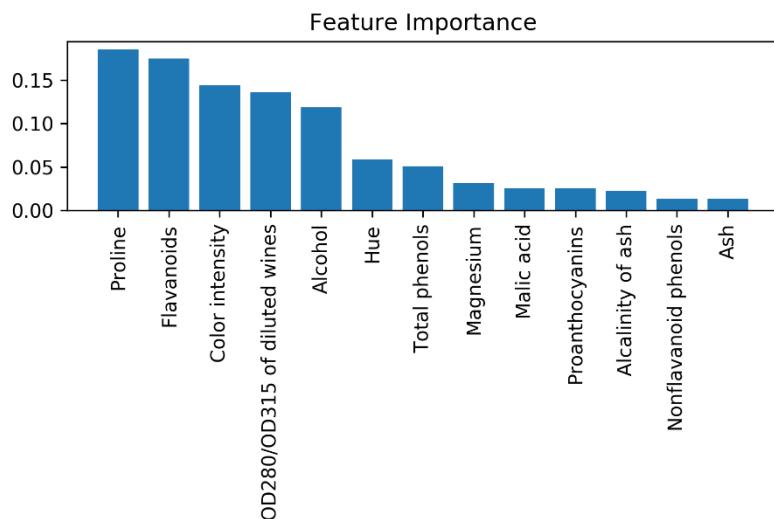
- Para problemas de regresión: f_regression, mutual_info_regression
- Para problemas de clasificación: chi2, f_classif, mutual_info_classif

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif,
mutual_info_classif
```

```
X, y = load_iris(return_X_y=True)
print(X.shape) # (150, 4)
funciones_de_seleccion = [chi2, f_classif, mutual_info_classif]
for funcion in funciones_de_seleccion:
    # K número de características
    X_new = SelectKBest(funcion, k=2).fit_transform(X, y)
    print(X_new.shape) # (150, 2)
    # métrica y comprobar
```

Evaluando la importancia de las características con bosques aleatorios

Los bosques aleatorios pueden determinar la importancia de cada característica sin hacer supuestos sobre si los datos son linealmente separables o no. Una vez realizado el bosque podremos ver las características más relevantes con el atributo **feature_importances**.



```
df_wine = pd.read_csv('wine.data', header=None)

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=0,
                                                    stratify=y)

feat_labels = df_wine.columns[1:]

forest = RandomForestClassifier(n_estimators=500,
                                 random_state=1)

forest.fit(X_train, y_train)
importances = forest.feature_importances_

indices = np.argsort(importances)[-1:]

for f in range(X_train.shape[1]):
    print("%2d %-*s %f" % (f + 1, 30,
                           feat_labels[indices[f]],
                           importances[indices[f]]))

plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]),
        importances[indices],
```

```
    align='center')

plt.xticks(range(X_train.shape[1]),
           feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()
```

El mayor problema que tiene este mecanismo para la elección de características es que si dos de ellas están altamente correlacionadas, una obtendrá una alta puntuación a cambio de no recoger todas las características de los datos de la otra. Si estamos es entrenando un modelo para realizar clasificaciones o predicciones no debe importarnos este problema, pero si lo que necesitamos es caracterizar los datos sí.

Cap 6.- Reducción de la dimensionalidad: Extracción

Una forma alternativa de reducir la dimensionalidad es la extracción de características. Se diferencia con la selección del punto anterior en que se genera un espacio nuevo de dimensiones a partir del original, pero con menos dimensiones manteniendo toda la información del espacio original. Cuanto mayor es el espacio de dimensional mayor es el espacio entre puntos y más cerca de los bordes estarán los ejemplos.

En resumen, cuanto mayor número de dimensiones tenemos, mayor es el riego de sobreajuste del algoritmo y cuanto más reduzcamos las dimensiones más rápido se efectuará el entrenamiento, pero no siempre nos llevará a un resultado más óptimo.

El mayor problema de estos algoritmos es determinar el número de componentes en la reducción. Hay que tener en cuenta que cuantas menos características elijamos mayor cantidad de información se perderá. Una pérdida del cinco por ciento es aceptable en casi todos los problemas.

PCA (Análisis de componentes principales)

La idea del algoritmo es buscar una matriz de transformaciones desde el espacio original al espacio buscado (W), realizar la transformación $z = Wx$, donde z es el nuevo vector y x el antiguo. El algoritmo PCA intenta buscar las direcciones de máxima varianza en el espacio de partida y proyectarlo en otro con menos dimensiones. Los ejes (componentes principales) serán ortogonales en el nuevo espacio, se interpretan como las direcciones de máxima varianza con respecto a los ejes ya definidos siendo ortogonales.

La primera componente elegida tendrá la mayor varianza, el resto serán las que tengan mayor varianza siempre que no estén correlacionadas (sean ortogonales) con respecto al resto de componentes (incluso si las características iniciales estaban correlacionadas).

Hay que tener en cuenta que las direcciones (las componentes extraídas) son muy dependientes al escalado de los datos, por lo que se tienen que estandarizar los mismos antes de hacerlo si queremos dar la misma importancia a todas las características originales. El algoritmo PCA es el siguiente:

1. Estandarizar los datos originales.
2. Construir la matriz de covarianzas.
3. Descomponer la matriz en los eigenvectores y los eigenvalores.
4. Ordenar los eigenvalores en orden decreciente.
5. Seleccionar los k eigenvectores.
6. Construir la matriz de proyección W .
7. Transformar el espacio inicial (X) en el de destino (Z).

Para el proceso anterior tenemos la librería scikit-learn que automatiza gran parte del trabajo.

```
# df_wine = pd.read_csv('wine.data', header=None)

df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols',
                   'Proanthocyanins',
                   'Color intensity', 'Hue',
                   'OD280/OD315 of diluted wines', 'Proline']

df_wine.head()
```

```

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.3,
                      stratify=y,
                      random_state=0)

# Standardizing the data.
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)

lr = LogisticRegression(multi_class='ovr', random_state=1,
                        solver='lbfgs')
lr = lr.fit(X_train_pca, y_train)

```

Código 010.py

Si estamos interesados en conocer los cambios en la varianza de cada componente principal, podemos inicializar en la clase **PCA** el atributo **n_components** a **None** y acceder a dichos datos a través de la propiedad **explained_variance_ratio_**. Del mismo modo, se puede establecer el atributo **n_components** a un valor decimal entre 0 y 1 para indicar la pérdida esperada en vez del número de componentes (**n_components=0.95** indica una pérdida del 5% máximo).

Compresión supervisada mediante análisis discriminante lineal (LDA)

La principal diferencia con el algoritmo anterior de este mecanismo se basa en usar un método supervisado frente al no supervisado del anterior. Esta técnica la podemos usar con datos no regularizados.

Para que el método sea óptimo hay que tener en cuenta que las clases deben tener idéntica matriz de covarianza y los ejemplos deben ser estadísticamente independientes. Pero incluso sin que se cumplan estas restricciones el algoritmo funciona bien.

Algoritmo:

1. Estandarizar las características.
2. Para cada clase, realizar el vector medio.
3. Construir la matriz de dispersión entre clases y de cada clase.
4. Descomponer la matriz en los eigenvectores y los eigenvalores.
5. Ordenar los eigenvalores en orden decreciente.
6. Seleccionar los k eigenvectores.
7. Construir la matriz de proyección W.
8. Transformar el espacio inicial (X) en el de destino (Z).

Podemos comprobar que el algoritmo es muy similar al PCA excepto que tiene en cuenta la información de las etiquetas en el paso dos.

Podemos implementar desde cero el algoritmo o hacer uso de la librería scikit-learn.

Código 011.py

Uso del análisis de componentes principales con Kernel (mapeo no lineal)

Con este algoritmo podemos gestionar problemas no lineales proyectándolos en un espacio mayor que sea lineal y posteriormente realizar la reducción de dimensionalidad con PCA al espacio deseado. El mayor desafío es la capacidad de cálculo necesaria para realizarlo.

Código 012.py

Otros algoritmos de reducción de la dimensionalidad

- PCA Incremental. Permite dividir el conjunto de entrada en partes más pequeñas y alimentar el algoritmo con ellas. De esta manera se puede entrenar de forma online y con Dataset muy grandes.
- LLE. Se mide primero cada instancia de forma lineal con sus vecinos y se busca una representación de menor dimensionalidad que preserve dichas relaciones.
- Proyecciones aleatorias. Se elige al azar las dimensiones.
- MDS. Reduce la dimensionalidad preservando la distancia entre las instancias.
- Isomapas. Crea un grafo de interconexión con los vecinos e intentar mantener la distancia geodésica.

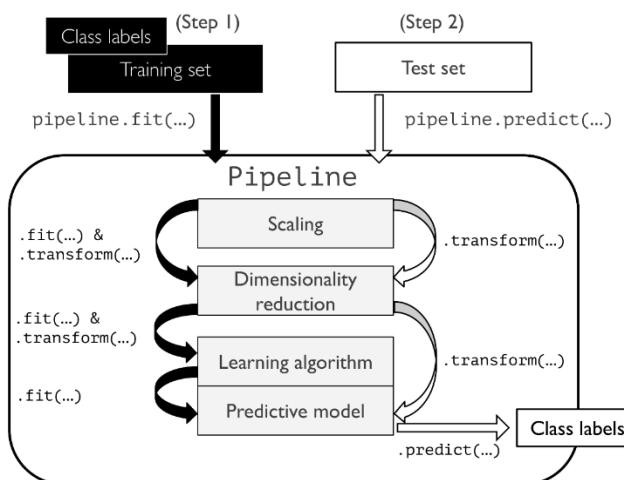
Cap 7.- Buenas prácticas para la evaluación del modelo y afinado de hiperparámetros

Herramientas: Tuberías

Reutilizar estados del modelo, variables, pasos de ejecución y demás artificios del machine Learning es muy común. Para tal fin se ha creado una nueva clase que va a automatizar todos los pasos de forma secuencial.

Código 013.py

Ejercicio: Determinar cuál es el mejor algoritmo a usar en el ejemplo del código 0013 y el tamaño óptimo de componentes para la PCA.



Las tuberías son imprescindibles para el tratamiento secuencial de todos los datos, pero si queremos hacer tratamiento sobre columnas de forma individual o conjunta podemos usar la clase **ColumnTransformer**. Se puede aplicar a cada columna una tubería o proceso independiente, se puede aplicar a un conjunto de índices, se puede eliminar una o varias columnas con la palabra clave *drop* y se pueden dejar sin hacer nada a otras con *passthrough*.

```

cl = ColumnTransformer(transformers=[(
    'cat', OneHotEncoder(), [2, 3]),
    ('drop', [5],
    remainder='passthrough')

data = cl.fit_transform(data)
  
```

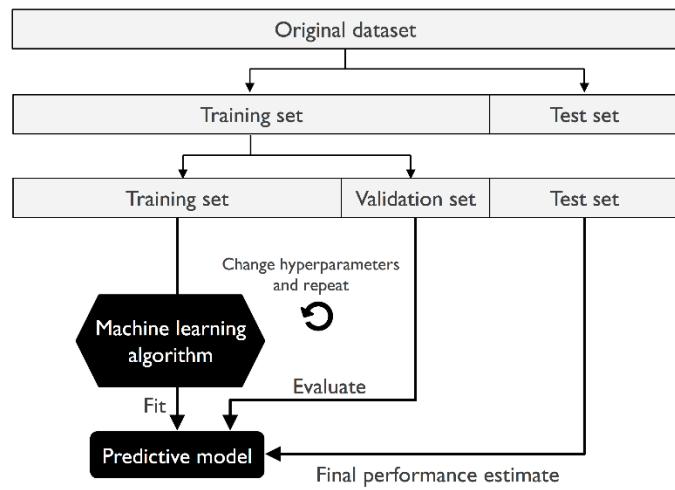
Ajustado de los Hiperparámetros

Los hiperparámetros por defecto no se ajustarán de la mejor manera, hay que buscar el que es mejor para nuestro problema. El ajuste se hace por un proceso de prueba y error, pero hay que tener cuidado cómo hacerlo. Tenemos tres aproximaciones: Holdout, K-Fold y búsqueda en Grid.

Holdout

Este método hace que los datos iniciales se dividan en tres conjuntos en vez de dos: entrenamiento, validación y test. Los dos primeros se utilizarán para calibrar el modelo y buscar

los mejores hiperparámetros, el tercero se usará para comprobar el rendimiento una vez determinado esos hiperparámetros.



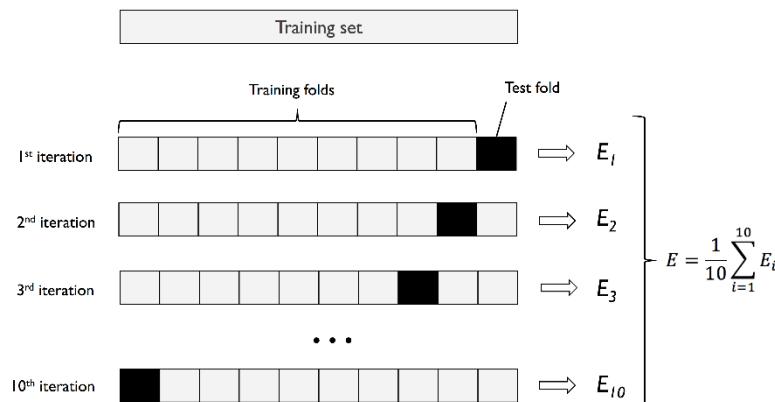
El problema de este mecanismo es que la estimación del rendimiento final es muy dependiente de la partición que hagamos de los datos.

Ejercicio: Implementar este algoritmo de forma manual con el ejemplo Código 0013.py

K-Fold

La segunda aproximación dice que dividimos el conjunto de entrenamiento en particiones (k) del mismo tamaño, dejando una de ellas para la estimación del rendimiento. Si cada vez que buscamos calibrar los hiperparámetros cogemos un conjunto diferente de las particiones podremos hacer hasta k vueltas sin interferencias.

Para finalizar, una vez encontrado los hiperparámetros, volveremos a entrenar el modelo con todos los datos de entrenamiento en vez de con las particiones y usaremos los de validación para comprobar la eficiencia.



Un buen número de particiones es 10, teniendo que incrementar este número si el conjunto de entrada es muy pequeño, o de disminuirlo si el conjunto de entrada es muy grande. Un caso especial es en el que hacemos el mismo número de particiones que de ejemplos de entrada, con lo que cada partición solo contendrá un caso. Esto se hará para conjuntos extremadamente pequeños de entrada.

Este algoritmo es recomendable si el número de ejemplos es muy bajo.

```
df = pd.read_csv('wdbc.data', header=None)
print(df.head())
X, y = df.iloc[:, 2:].values, df.iloc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)

kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
score_actual = componentes_actual = 0
componentes = 1
for k, (train, test) in enumerate(kfold):
    pipe_lr = make_pipeline(StandardScaler(),
                           PCA(n_components=componentes),
                           LogisticRegression(random_state=1,
                           solver='lbfgs'))
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k + 1,
                                                       np.bincount(y_train[train]), score))
    if score > score_actual:
        score_actual = score
        componentes_actual = componentes
    componentes += 1

print("Número de componentes:", componentes_actual)
pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=componentes_actual),
                        LogisticRegression(random_state=1,
                        solver='lbfgs'))
pipe_lr.fit(X_train, y_train)
print("Aciertos:", pipe_lr.score(X_test, y_test))
```

Ajustado de los Hiperparámetros con búsqueda en Grid

https://scikit-learn.org/stable/modules/grid_search.html#grid-search

La idea de este algoritmo es proporcionar todos los valores de los hiperparámetros que queramos probar y el sistema hará todas las combinaciones posibles, proporcionando el que mejor se ajuste. Una vez finalizado veremos la eficacia final con los datos de test.

```
df = pd.read_csv('wdbc.data', header=None)
print(df.head())
X, y = df.iloc[:, 2:].values, df.iloc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)

pipe_svc = make_pipeline(StandardScaler(), SVC(random_state=1))
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
param_grid = [{ 'svc_C': param_range,
                'svc_kernel': ['linear', 'sigmoid'],
                },
                { 'svc_C': param_range,
                'svc_gamma': param_range,
                }]
```

```

'svc__kernel': ['rbf']
}

# Probamos con dos tipos diferentes de parámetros
gs = GridSearchCV(estimator=pipe_svc,
                    param_grid=param_grid,
                    scoring='accuracy',
                    refit=True,
                    cv=10,
                    n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)

clf = gs.best_estimator_

# clf.fit(X_train, y_train)
# note that we do not need to refit the classifier
# because this is done automatically via refit=True.
print('Test accuracy: %.3f' % clf.score(X_test, y_test))

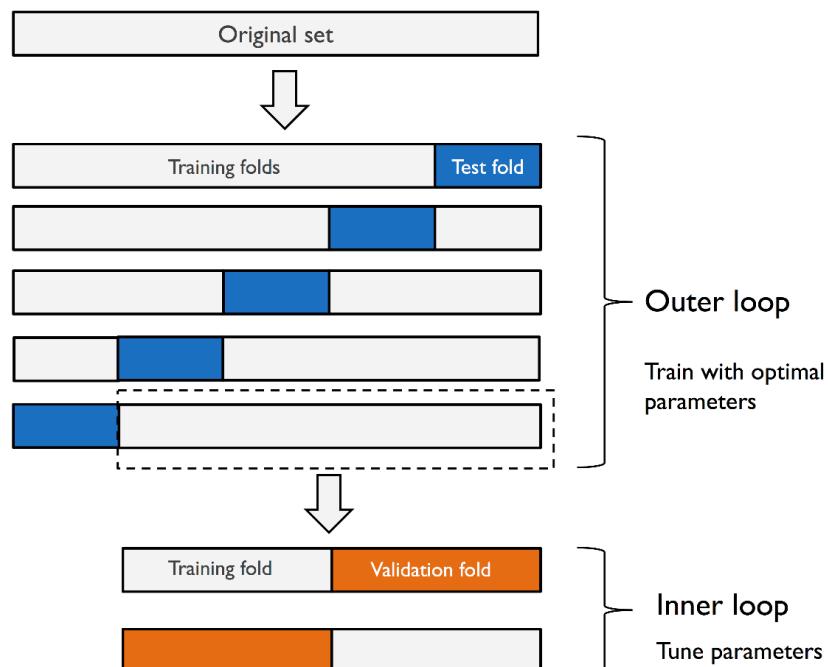
```

Se accede al algoritmo mejor estimado a través de **best_estimator_**.

Aunque esta aproximación es la mejor que se podría utilizar, es computacionalmente muy costosa, por lo que se puede realizar una aproximación diferente, en la que se eligen al azar entre los parámetros y sus combinaciones para reducir el coste computacional. (Para una mayor ampliación ver el manual de la librería scikit-learn **RandomizedSearchCV**).

Búsqueda en Grid avanzada

Usando el algoritmo K-Fold junto con la selección en Grid podemos hacer una selección del algoritmo de aprendizaje bastante acertada. En concreto tendremos dos bucles anidados, uno externo de selección de bloque y otro interno para afinar los hiperparámetros. En la siguiente figura vemos la idea. La clase **GridSearchCV** hace las particiones de forma automática con el parámetro **cv**.



```
df = pd.read_csv('wdbc.data', header=None)
print(df.head())
X, y = df.iloc[:, 2:].values, df.iloc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)

def imprimir(gs):
    gs = gs.fit(X_train, y_train)
    print(gs.best_score_, end='->')
    print(gs.best_params_)
    scores = cross_val_score(gs, X_train, y_train, scoring='accuracy',
                             cv=5)
    print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
                                             np.std(scores)))

pipe_svc = make_pipeline(StandardScaler(), SVC(random_state=1))
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
param_grid = [{ 'svc__C': param_range,
                'svc__kernel': ['linear', 'sigmoid']},
              { 'svc__C': param_range,
                'svc__gamma': param_range,
                'svc__kernel': ['rbf']}]

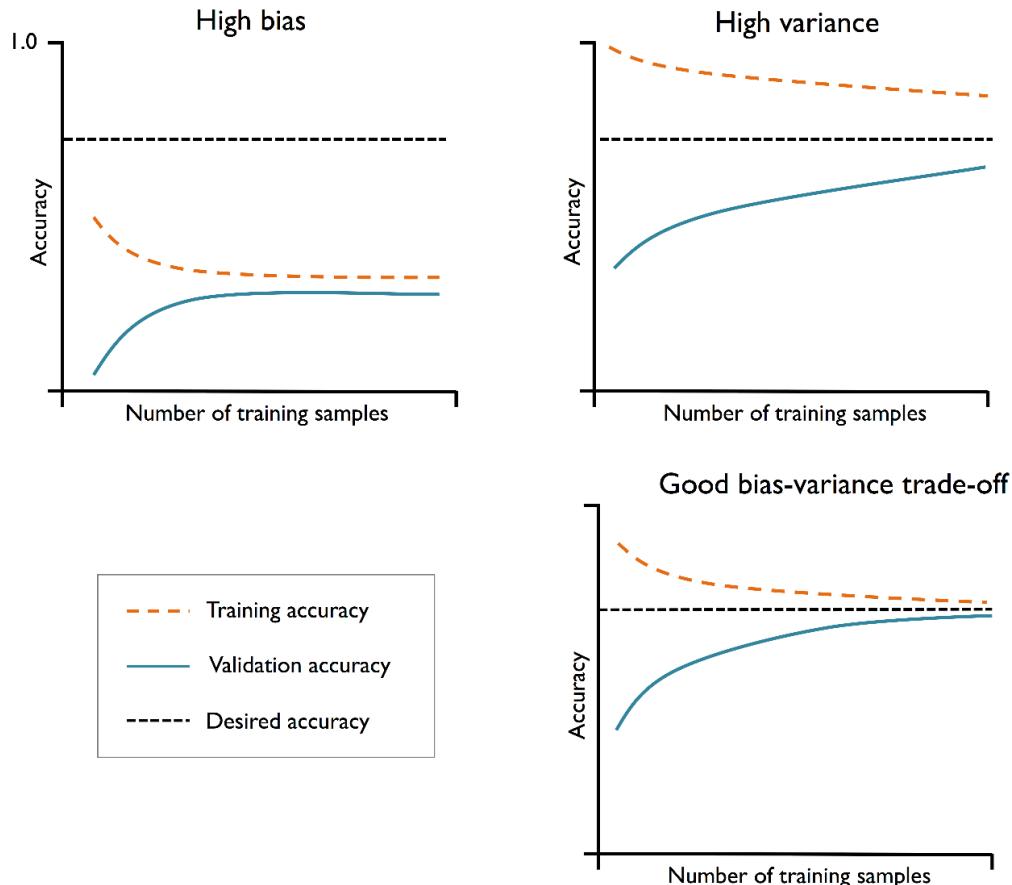
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  refit=True,
                  cv=10,
                  n_jobs=-1)
imprimir(gs)

gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
                  param_grid=[{ 'max_depth': [1, 2, 3, 4, 5, 6, 7,
                                              None]}],
                  scoring='accuracy',
                  cv=2)
imprimir(gs)
```

Evaluar el modelo: Curvas de aprendizaje y validación

Curva de aprendizaje

Con esta curva dibujaremos el rendimiento del modelo en función del número de ejemplos. Podremos fácilmente determinar qué tipo de problema estamos teniendo.



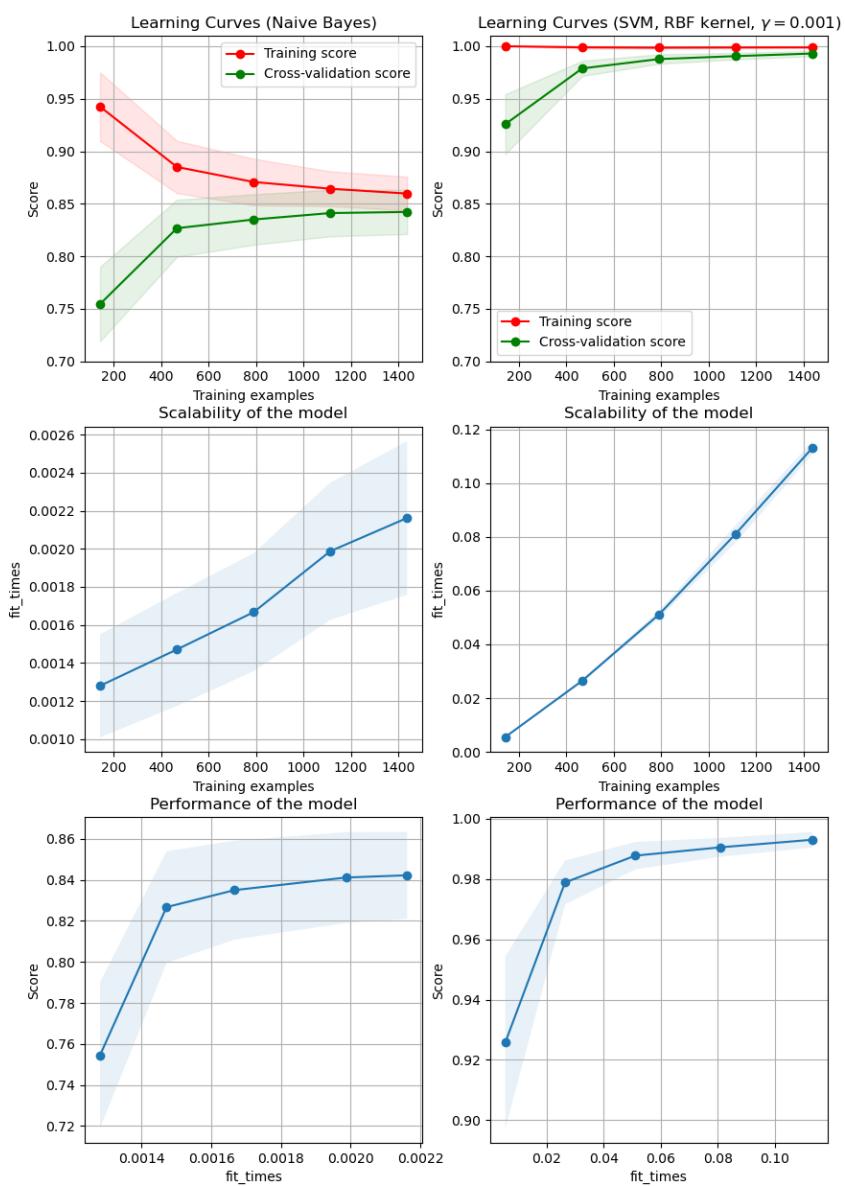
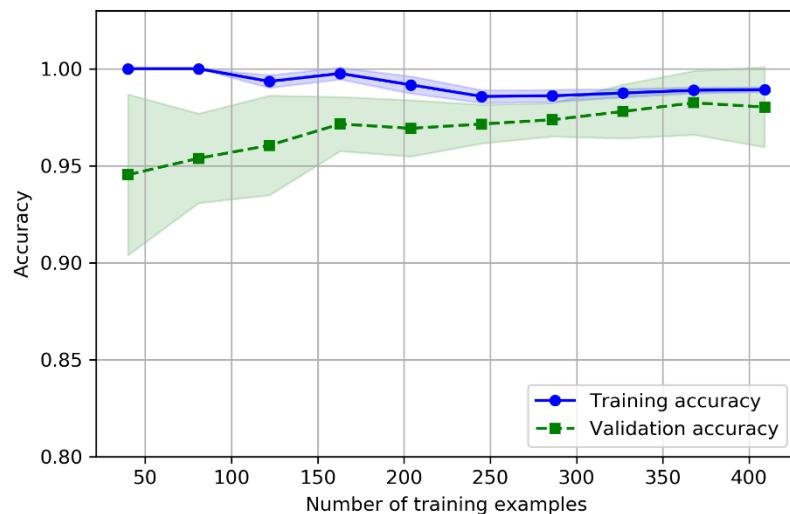
Traducción: Bias es tendencia

En la parte superior izquierda vemos que las curvas se ajustan muy lejos del desempeño esperado (línea horizontal de puntos), lo que indica un claro problema de Subajuste, por lo que deberemos aumentar el número de columnas creando nuevas características o minimizando la regularización de los datos.

Por otro lado, en la superior derecha es un claro sobreajuste al ser el espacio entre ambas gráficas muy elevado. En este punto o buscamos más datos o reducimos el nivel de complejidad del modelo o incrementamos la regularización de los datos.

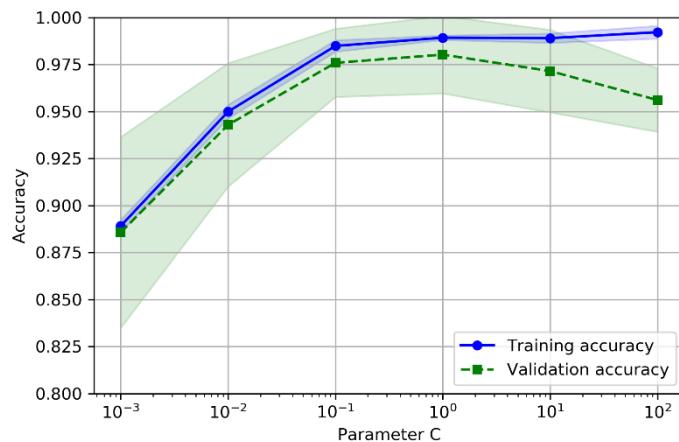
En el ejemplo siguiente podemos ver como a partir de 250 ejemplos la desviación con respecto a la media se ensancha (ver sombreado de color verde) lo que indica un sobreajuste.

Código 014.py



Curva de validación

Otro caso diferente es en el que cambiamos el valor de un hiperparámetro y comprobamos el funcionamiento del modelo. En el ejemplo siguiente vemos como a partir de uno, las curvas se separan indicando potenciales problemas. En este caso elegiremos un valor entre 0,01 y 0,1. (Si no sabemos el nombre que se usa internamente para los parámetros utilizaremos: `nombre_estimador.get_params().keys()`)



Código 015.py

Evaluar el modelo: Uso de diferentes métricas

En la librería scikit-learn, la métrica conseguida a través del método `score` sigue la regla de mayor es mejor. Si un método devuelve un valor mayor que otro método, este primero será mejor que el segundo.

Matriz de confusión

		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

La matriz de confusión nos sirve para evaluar el rendimiento de un algoritmo de aprendizaje. Es una matriz simple que enfrenta los casos positivos (TP) frente a los falsos positivos (FP) y los casos negativos (TN) frente a los falsos negativos (FN)

En la librería scikit-learn proporcionan una función que nos muestra la matriz de confusión para un algoritmo.

```

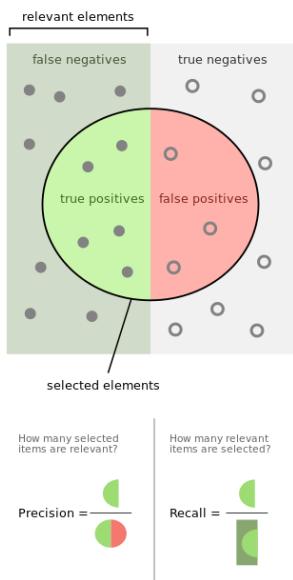
df = pd.read_csv('wdbc.data', header=None)
X, y = df.iloc[:, 2:].values, df.iloc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)

pipe_svc = make_pipeline(StandardScaler(), SVC(random_state=1))
pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)

```

Optimización del Recall y Precisión de un modelo

El error (ERR) y la precisión (ACC) proporcionan información general acerca de los ejemplos que están mal clasificados. El error (ERR) se puede entender como la suma de todas las falsas predicciones divididas por el número total de predicciones, y la precisión se calcula como uno menos el error.



$$ERR = \frac{FP + FN}{FP + FN + TP + TN} \quad ACC = 1 - ERR$$

Para hacer comparaciones, es mejor usar las tasas de verdaderos positivos y de falsos positivos:

$$FPR = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{FN + TP}$$

En un caso de detección de tumores malignos, si estuviéramos más interesados en la detección de casos malignos, es muy importante minimizar el número de falsos positivos. Al contrario que la FPR, la TPR muestra información sobre el número de positivos que se han identificado correctamente frente al total de positivos.

Las métricas que se suelen utilizar son PRE (precisión) y REC (Recall) y están relacionadas con las FPR y TPR de la siguiente manera:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{FN + TP}$$

La métrica REC nos ayuda a minimizar el problema de los casos no detectados a costa de predecir como enfermos a pacientes sanos. Si optimizamos para PRE entonces si predecimos que un paciente tiene un tumor maligno, será a costa de que habrá mayor cantidad de pacientes con dicho tumor que no se han detectado. La métrica PRE indicará la tasa de aciertos, es decir si predice que hay 80 cincos, cuántos eran de verdad cinco. Y la métrica REC detecta el porcentaje total de aciertos, es decir, si detecta 60 cincos y había 120 en todo el conjunto cuál es el porcentaje de aciertos.

Por tanto, se suele utilizar una métrica que balancea tanto PRE como REC llamada F1.

$$F1 = 2 \frac{PRE \times REC}{PRE + REC}$$

Todas las métricas anteriores están implementadas en la librería de forma estándar.

```
print('Precision: %.3f' % precision_score(y_true=y_test,
                                           y_pred=y_pred))
print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('F1: %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

Que podremos usar dentro de la clase **GridSearchCV** a través del parámetro de *scoring*.

Evaluación del modelo: Dibujando las curvas ROC

Dibujar curvas ROC para diferentes modelos nos puede ayudar a elegir el mejor para el problema que estemos tratando. Estas curvas muestran el rendimiento con respecto a las métricas TRP y FPR anteriores.

Una gráfica perfecta es la que se dibuja en la siguiente imagen como un ángulo recto de puntos que parte del (0, 0) y sube hasta el (0, 1). Del mismo modo una gráfica no funcional es la representada como la diagonal de segmentos en el gráfico siguiente. A partir de ahí, cuanto más se acerque a la gráfica de puntos mejor será el algoritmo.

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import roc_curve, auc

df = pd.read_csv('wdbc.data', header=None)
X, y = df.iloc[:, 2:].values, df.iloc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)

pipe_lr = make_pipeline(StandardScaler(),
                       PCA(n_components=2),
                       LogisticRegression(penalty='l2',
                                           random_state=1, solver='lbfgs', C=100.0))

X_train2 = X_train[:, [4, 14]]
cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))

fig = plt.figure(figsize=(7, 5))
mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas = pipe_lr.fit(X_train2[train],
                          y_train[train]).predict_proba(X_train2[test])
```

```

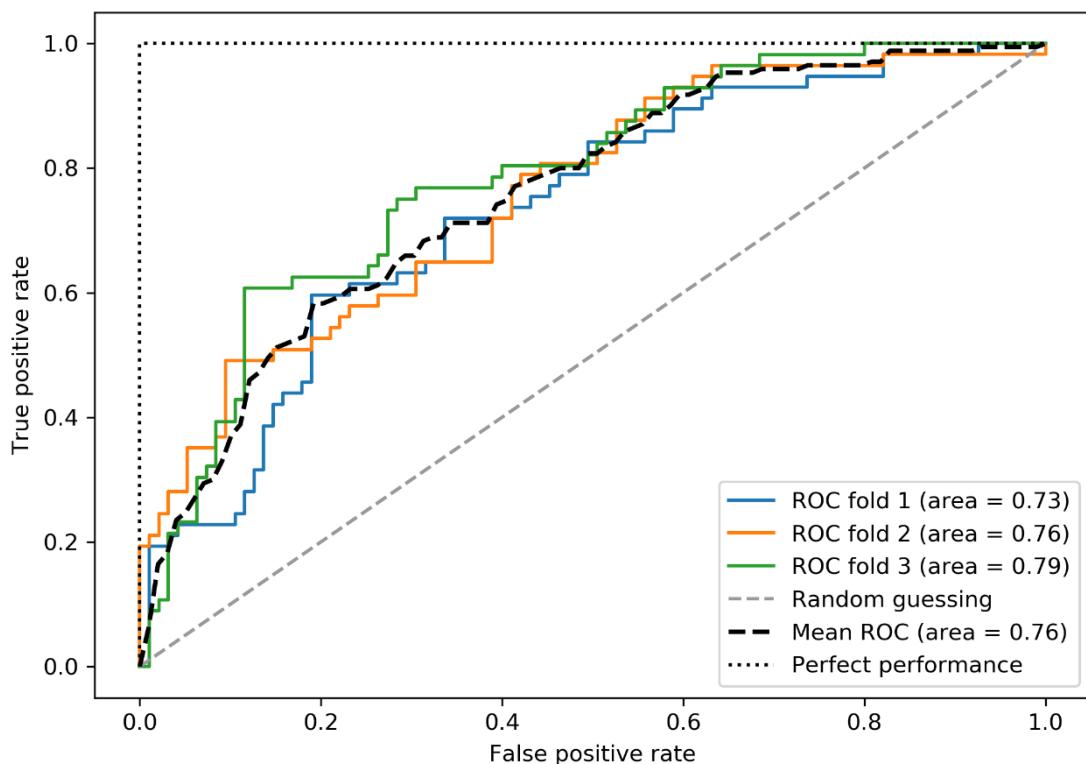
fpr, tpr, thresholds = roc_curve(y_train[test], probas[:, 1],
                                  pos_label=1)
mean_tpr += np.interp(mean_fpr, fpr, tpr)
mean_tpr[0] = 0.0
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label='ROC fold %d (area = %0.2f)' % (i + 1,
                                                          roc_auc))

plt.plot([0, 1], [0, 1],
         linestyle='--',
         color=(0.6, 0.6, 0.6),
         label='Random guessing')
mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--', label='Mean ROC (area = %0.2f)' %
         mean_auc, lw=2)
plt.plot([0, 0, 1],
         [0, 1, 1],
         linestyle=':',
         color='black',
         label='Perfect performance')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.legend(loc="lower right")

plt.tight_layout()
plt.show()

```



Asociada a las curvas ROC, está el área por debajo de la curva, que es también un indicador. Para calcularlo se utiliza la función `roc_auc_score` que será uno cuando el algoritmo sea perfecto y 0,5 en el peor de los casos.

Evaluar el modelo: Métricas para clasificación de múltiples clases

La librería scikit-learn también implementa métricas para este tipo de clasificación. Hay dos métricas importantes: Micro-media y Macro-media. La primera es útil si queremos dar un peso equitativo a cada predicción, mientras que la segunda da pesos similares a cada clase, no a cada ejemplo. Se puede establecer a través del parámetro *average*.

```
pre_scorer = make_scorer(score_func=precision_score,
                           pos_label=1,
                           greater_is_better=True,
                           average='micro')
```

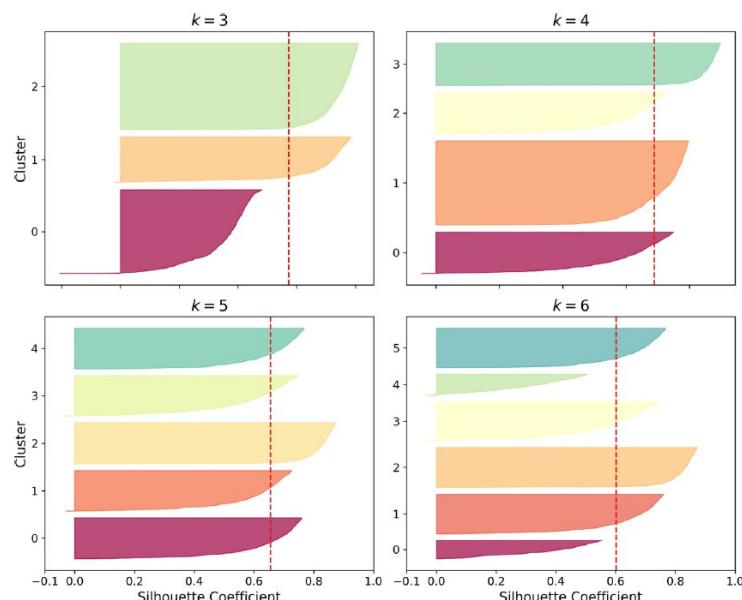
Evaluar el modelo: Gestión del problema de desequilibrio

Cuando una clase predomina sobre las demás dentro de la muestra de datos, simplemente prediciendo dicha clase se llegará como mínimo al mismo rendimiento que el porcentaje de la clase en la muestra. Es decir, si en la muestra hay un 90% de casos positivos, si predecimos todos como positivos tendremos al menos un acierto del 90%, pero no se habrá aprendido nada por parte del modelo.

Para evitar este problema podemos asignar una penalización mayor a las predicciones erróneas que haga el modelo estableciendo el parámetro **class_weight = 'balanced'**. O podemos crear nuevos ejemplos a partir de los datos en la clase minoritaria para igualar el porcentaje de casos, a través de la función **resample**. Por último, podemos eliminar casos de la clase mayoritaria hasta igualar el porcentaje de casos, siempre que la cantidad de ejemplos nos lo permita.

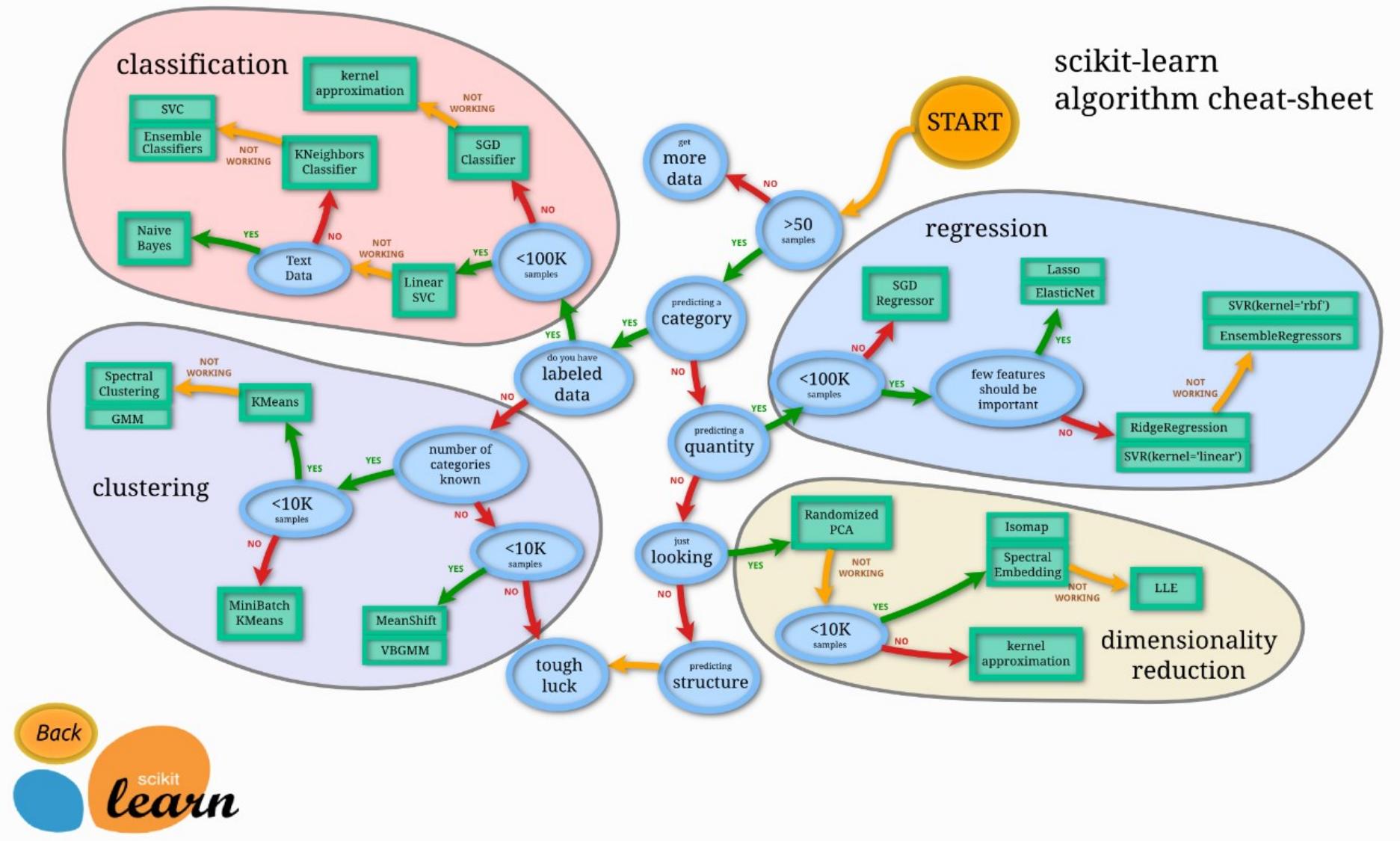
Evaluar el modelo: Otras métricas

Se puede usar **Silhouette_score** para los algoritmos de Clustering. Más precisa pero computacionalmente más costosa. Con esta métrica se puede imprimir el coeficiente silhouette en función del número de clústeres para determinar el número de los mismos. Se elegirá el tamaño más equilibrado.



Para la figura anterior, tanto 4 como 5 son buenas elecciones, pero $k=5$ está más balanceado.

Cap 8.- Elección del algoritmo



Ejercicios

Clasificación de setas

<https://www.kaggle.com/uciml/mushroom-classification>

Este Dataset incluye la descripción de 23 especies de setas sacados de la guía americana de setas en la que se incluye si son venenosas o no. Se pretende con este ejemplo:

- Qué algoritmos son los que tienen un mejor desempeño.
- Qué características son las más indicativas.

Código 016.py

Clasificación de asteroides

<https://www.kaggle.com/shrutimehta/nasa-asteroids-classification>

Este Dataset está basado en el servicio NeoWs de información para la detección de potenciales asteroides peligrosos. Se pretende con este ejemplo:

- Determinar los potencialmente peligrosos.
- Que características son las más indicativas.

Código 017.py

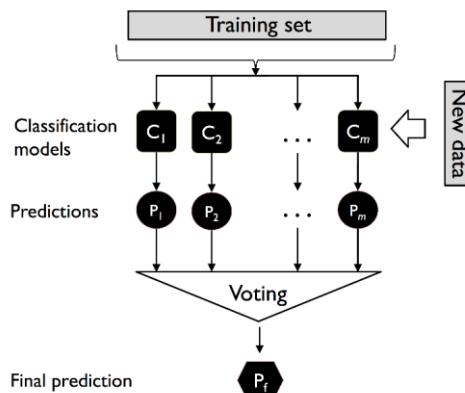
Cap 9.- Combinación de algoritmos en uno solo

El propósito de combinar diferentes algoritmos en uno solo (meta-clasificador) es mejorar sustancialmente la capacidad predictiva de cada uno de los algoritmos individuales. Estadísticamente se demuestra que la combinación de varios algoritmos tendrá menos error que el mejor de ellos de forma individual.

Voto

El procedimiento a la hora de clasificar es preguntar a todos los algoritmos implicados y hacer que voten para determinar la clase final. La votación generalmente está ponderada por pesos pudiendo controlar la importancia de cada algoritmo de forma individual. Esta votación se podrá hacer de forma unánime (todos dan la misma respuesta), de forma mayoritaria (la más votada, todos votan) o de forma plural (la más votada, algunos no votan).

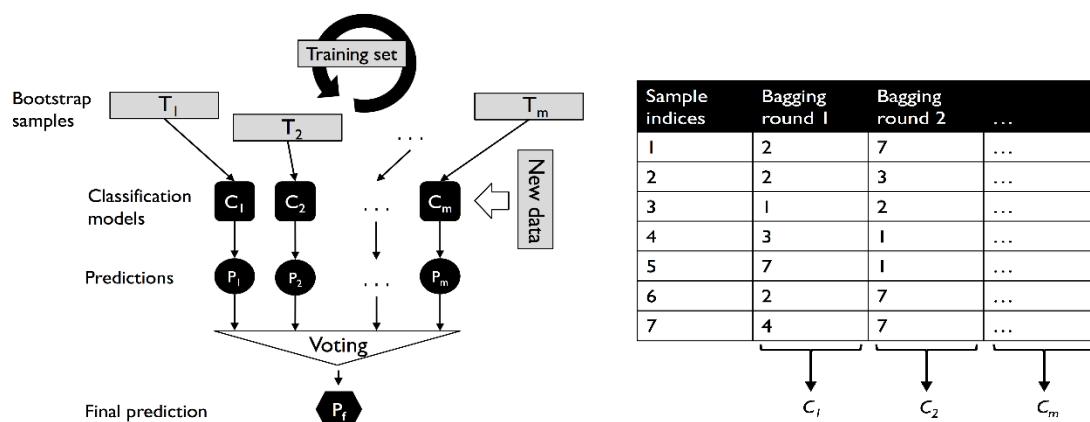
A la hora de crear este nuevo algoritmo se pueden utilizar diferentes algoritmos para crearlo, o un único algoritmo con diferentes hiperparámetros. Dependerá de la situación qué elegir.



<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html>

Bagging

Es un meta-algoritmo que está muy cerca del algoritmo de voto de la sección anterior. Pero en vez de utilizar el mismo conjunto de entrenamiento para todos los algoritmos interiores, se elegirán un conjunto de ejemplos al azar con repetición para entrenar cada algoritmo diferente.



Código 018.py

En la práctica, los problemas complejos con alta dimensionalidad pueden hacer que los algoritmos simples se sobreajuste, y aquí es donde este algoritmo toma ventaja. Podemos reducir la varianza de forma efectiva, pero a cambio la tendencia no.

AdaBoost

Esta aproximación se basa en usar algoritmos muy básicos, cercanos a la elección aleatoria. La idea principal es enfocarse en aquellos ejemplos que no se han podido clasificar de forma correcta para que el siguiente los tenga en cuenta.

- 1º. Se elige un subconjunto sin remplazo con el que se entrena el primer algoritmo.
- 2º. Se elige un segundo subconjunto sin remplazo y se añade un 50% de los ejemplos que previamente se han clasificado de forma incorrecta por el primero para el segundo algoritmo.
- 3º. Se eligen los ejemplos que los dos algoritmos anteriores no han clasificado igual y se le proporciona a un tercer algoritmo.
- 4º. Se combinan las decisiones de los tres algoritmos mediante la mayoría de voto.

El incremento de rendimiento en este algoritmo se basa en que puede mejorar la tendencia a la vez que la varianza comparada con el algoritmo de Bagging.

Código 019.py

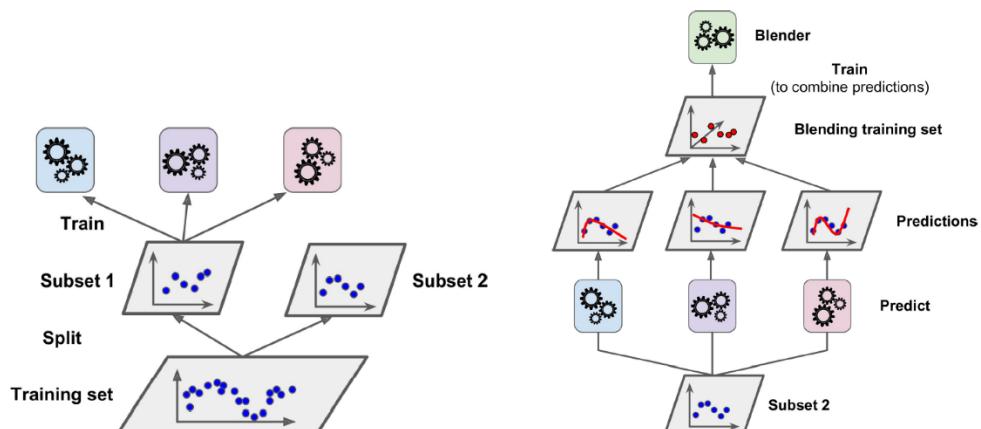
GradientBoost y HistGradientBoostingClassifier

Estos dos algoritmos usan una aproximación similar al AdaBoost en cuanto al tipo de algoritmo, pero difieren significativamente en cómo se llega a ajustar los pesos internos, utilizando una aproximación de mínimo gradiente.

El segundo algoritmo solo difiere del primero en que es significativamente más rápido si el número de ejemplos es superior a 10000.

Stacking

La idea es muy simple: dividir el proceso en dos pasos, en el primero se entrena un conjunto de modelos diferentes y dan una predicción, en la segunda fase se recogen las predicciones de esos modelos y se entrena otro para que tome la decisión final. Para que sea completo, el conjunto de entrenamiento se divide en dos, una parte se dedica al entrenamiento de la capa uno y el resto se utiliza para hacer las predicciones que la segunda capa usará para entrenarse.



Ejercicios

En este ejemplo usaremos la base de datos de MNIST para realizar un reconocimiento de caracteres.

Estudiar el código para reforzar las técnicas hasta ahora vistas.

Código 020.py

Más ejercicios

1. Evaluar el rendimiento del clasificador *KNeighborsClassifier* usando un Grid para ajustar los hiperparámetros **weights** ({‘uniform’, ‘distance’}) y **n_neighbors**.
2. Analizar el Dataset del hundimiento del Titanic
 - <https://www.kaggle.com/c/titanic/data>
 - <https://www.kaggle.com/vaishnavikhilari/titanic-survival-prediction>
3. Analizar los Dataset del directorio ch09\DatasetExtras
4. Analizar el siguiente notebook
 - a. <https://www.kaggle.com/abbascanguven/churn-prediction-train-val-test-split-score-0-97>

UT-4

Cap 10.- Explorando el scikit-learn: Regresión

La regresión intenta predecir el valor de una variable de salida a partir de las variables de entrada, dentro de una escala continua en vez de un conjunto de etiquetas categóricas (clasificación). Procedimentalmente hablando es similar a la clasificación como veremos en el ejemplo.

De momento no se usarán para la clasificación algoritmos de redes neuronales, será más adelante.

Regresión lineal

Intenta utilizar elemento geométrico recto (línea, plano, hiperplano) para intentar las predicciones. La regresión lineal se puede dividir en simple o múltiple dependiendo del número de variables de entrada, pero en ambos casos intentará predecir la variable objetivo. De hecho, la regresión simple es un caso específico de la más general la múltiple.

Una de las características más notables de los algoritmos de regresión es que generalmente no es necesario que las variables estén distribuidas según la normal.

Código 021.py

Los modelos de regresión lineal (**LinearRegression**) son muy dependientes de los valores extremos. Para intentar paliar el efecto de los mismos se introduce la clase **RANSACRegressor** que utiliza un algoritmo de detección de extremos bastante potente, pero no debemos olvidar que habrá que estudiar en profundidad el problema y los extremos para determinar qué hacer con ellos. Esta clase introduce varios hiperparámetros que son simples de entender, pero uno de ellos, **residual_threshold**, es dependiente del problema que se esté tratando y no se puede determinar más que con el mecanismo de prueba y error. El uso de esta clase reduce el efecto potencial de los valores extremos en la solución, pero no podemos estar seguros que la conclusión alcanzada sea la óptima. De hecho, en el código de ejemplo vemos que las métricas son peores.

Para finalizar, como cualquier modelo, habrá que medir el rendimiento del mismo, usando como norma general la diferencia entre la distancia vertical entre el valor predicho y el real. Para las mediciones podemos usar la métrica MSE (muy usado para validar modelos vía gridsearch o validación cruzada) o la métrica R² que es una estandarización de la métrica MSE.

Métodos regularizados

Al igual que en el modelo de clasificación podemos usar las regularizaciones L2 (Lasso), L1(Ridge) o Elastic para hacer la regresión.

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
y_train_pred = lasso.predict(X_train)
y_test_pred = lasso.predict(X_test)
print(lasso.coef_)
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)
from sklearn.linear_model import Lasso
```

```
lasso = Lasso(alpha=1.0)
from sklearn.linear_model import ElasticNet
elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Regresión polinomial

La regresión lineal da por supuesto que los datos se pueden ajustar con una línea (2D) un plano (3D) etc. En este caso en vez de usar una línea para ajustar los ejemplos, utilizaremos un polinomio para tal fin. La aproximación es la misma que la llevada en la clasificación con los mismos problemas de sobre ajuste, por lo que la elección del grado del polinomio debe hacerse de forma muy cuidadosa.

```
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

Código 022.py

Regresión usando Bosques aleatorios

El modelo usará la misma técnica que se vio en la clasificación, ajustando un conjunto de árboles de decisión independientes. La ventaja de este método es que no necesita ningún tipo de transformación de las características

Código 023.py

Regresión usando SVM

También se pueden usar las máquinas de soporte vector para problemas de regresión. Con este algoritmo es muy importante escalar las características.

Código 024.py

Ejercicios

A partir de las características de un artículo de machine Learning intentaremos predecir, cuantas veces será compartido en Redes Sociales ([artículos_ml.csv](#)).

- Con una variable: Word count
- Con dos variables:
 - Word count
 - Una calculada como la suma de: # of Links" + # of comments +# Images video

A partir de las características del fichero de coches [mtcar.csv](#)

- Implementar los cuatro algoritmos y comparar las métricas
- Hacer predicciones de al menos 8 coches

UT-5

Cap 11.- Explorando el scikit-learn: Aprendizaje no supervisado

Las técnicas que nos permiten descubrir la estructura intrínseca de los datos sin conocer de antemano la solución se encuadran dentro del aprendizaje no supervisado.

Algoritmo K-Means

Este algoritmo es extremadamente rápido y fácil de implementar. Para cada bloque se buscará el centroide (el punto en media) o el mediode (el punto más representativo) y se minimizará la distancia con el resto de puntos para formar los grupos. Este algoritmo es especialmente bueno para divisiones esféricas. El principal problema es que debemos establecer a priori el número de grupos (k) a crear, no se deducen de los datos además de que alguno pueda estar vacío y que no se ajusta bien a datos con diferentes tamaños, densidades o formas no esféricas.

Podemos describir este algoritmo como un problema de optimización iterativa para minimizar la suma de los errores al cuadrado.

Cuando apliquemos este algoritmo a los datos reales, deberemos asegurarnos que todas las características están medidas aplicando la misma escala, por lo que habrá que hacer una estandarización o un escalado Max-min.

```
from sklearn.cluster import KMeans  
  
km = KMeans(n_clusters=3, init='random', n_init=10,  
             max_iter=300, tol=1e-04, random_state=0)  
  
y_km = km.fit_predict(X)
```

Algoritmo k-Means++

Mejora el algoritmo anterior aplicando la estrategia de situar los centroides lo más separados posibles unos de otros, haciendo que converja mejor y sea más consistente en sus resultados. Lo único que necesitamos es establecer el parámetro `init='k-means++'` en la creación de la clase o no poner nada ya que es el valor por defecto.

Algoritmo FCM

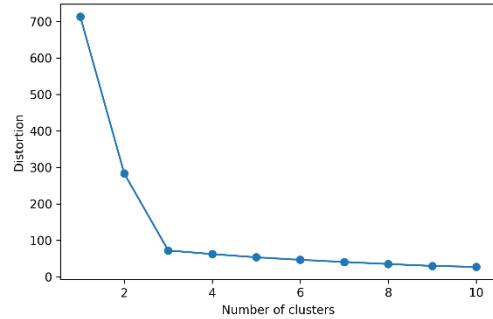
Este algoritmo en vez de utilizar una asignación dura de puntos (solo en un clúster) permite que un ejemplo pertenezca a varios clústeres (asignación débil). Es muy similar algoritmo de K-Means proporcionando porcentajes de proximidad para un ejemplo en cada clúster.

No está implementado bajo scikit-learn: <https://pythonhosted.org/scikit-fuzzy/>

Elección del número de clústeres

Uno de los mayores retos de estos algoritmos es determinar el número de clúster a crear. La respuesta dependerá del problema que estemos tratando, pero podemos usar un conjunto de técnicas gráficas (métricas) para determinar el número más apropiado.

Para este tipo de técnicas se usa la medida de inercia, accesible a través del parámetro `inertia_` de los algoritmos. Se define la inercia como la distorsión sobre la métrica SSE. Se puede usar el parámetro `y` representar cómo varía con respecto al número de clúster, y fijar como óptimo aquel valor en el que la pendiente cambie drásticamente a otra más plana (El tres en la imagen de la izquierda).

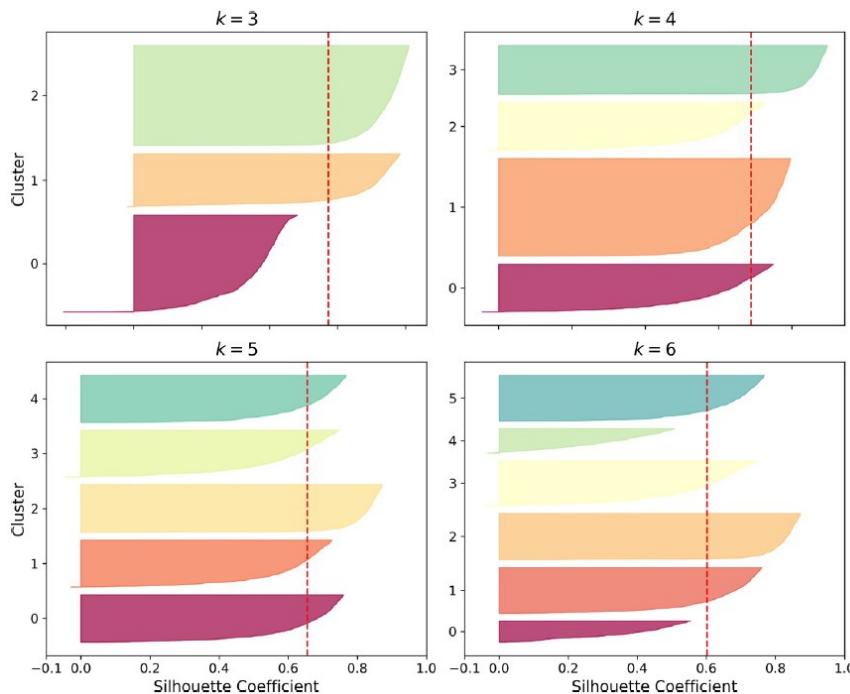


```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=150, n_features=2, centers=3,
                   cluster_std=0.5, shuffle=True, random_state=0)
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i, init='k-means++', n_init=10,
                 max_iter=300, random_state=0)
    km.fit(X)
    distortions.append(km.inertia_)

plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.tight_layout()
plt.show()
```

La segunda aproximación es a través de los gráficos silhouette. Éstos intentan describir el tamaño de cada clúster con respecto a los demás basados en un coeficiente. En la siguiente figura vemos la representación para 4 tipos de clúster, siendo $k=4$ y $k=5$ los que deberíamos elegir, estando mejor situado $k=5$ al tener todos las clases por encima de la media del coeficiente silhouette (línea de puntos) y estar más balanceado.



```

import numpy as np
from matplotlib import cm
from sklearn.metrics import silhouette_samples
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=150, n_features=2, centers=3,
                   cluster_std=0.5, shuffle=True, random_state=0)

km = KMeans(n_clusters=3,
             init='k-means++',
             n_init=10,
             max_iter=300,
             tol=1e-04,
             random_state=0)
y_km = km.fit_predict(X)

cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X, y_km, metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper),
             c_silhouette_vals, height=1.0,
             edgecolor='none', color=color)
    yticks.append((y_ax_lower + y_ax_upper) / 2.)
    y_ax_lower += len(c_silhouette_vals)

silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg, color="red", linestyle="--")

plt.yticks(yticks, cluster_labels + 1)

```

```

plt.ylabel('Cluster')
plt.xlabel('Silhouette coefficient')

plt.tight_layout()
plt.show()

```

Algoritmo de Árbol Jerárquico

Una de las ventajas de este algoritmo es que nos permite dibujar lo que se llaman dendrogramas. Las dos aproximaciones más comunes de los árboles jerárquicos son acumulativos y divisibles. En los divisibles empezamos con un clúster grande y lo vamos dividiendo haciéndolos más pequeños según las características, en la segunda aproximación (acumulativos) es al revés, se parte de un clúster por cada ejemplo y se va agrupando en clústeres.

Del mismo modo para los algoritmos acumulativos las dos implementaciones que se utilizan son single linkage y complete linkage. En el primero calculamos las distancias entre los miembros más similares para cada par de clústeres y los unimos. En el segundo en vez de calcular las distancias entre los más similares, se miden los menos similares.

Para hacer uso de estos algoritmos bajo scikit-learn se utilizará la clase AgglomerativeClustering.

```
from sklearn.cluster import AgglomerativeClustering
```

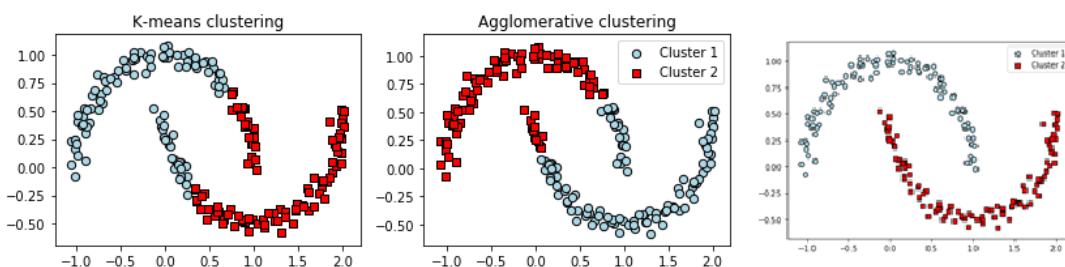
```

ac = AgglomerativeClustering(n_clusters=3,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Cluster labels: %s' % labels)

```

Algoritmo DBSCAN

Este algoritmo no predispone nada sobre la forma de los clúster, pudiendo ser de cualquier forma (no esféricos como en el k-Means). Asigna clústeres en función de la densidad de zonas a partir de un radio específico bien separadas con áreas de baja densidad de puntos. En el siguiente gráfico vemos la comparación de todos los métodos, y el tercero (DBSCAN) se ajusta perfectamente.



```

from sklearn.cluster import DBSCAN

X, y = dataset.make_moons(n_samples=150, noise=.05)

db = DBSCAN(eps=0.2, min_samples=5, metric='euclidean')
y_db = db.fit_predict(X)
plt.scatter(X[y_db == 0, 0], X[y_db == 0, 1],

```

```

        c='lightblue', marker='o', s=40,
        edgecolor='black',
        label='Cluster 1')
plt.scatter(X[y_db == 1, 0], X[y_db == 1, 1],
            c='red', marker='s', s=40,
            edgecolor='black',
            label='Cluster 2')
plt.legend()
plt.tight_layout()
plt.show()

```

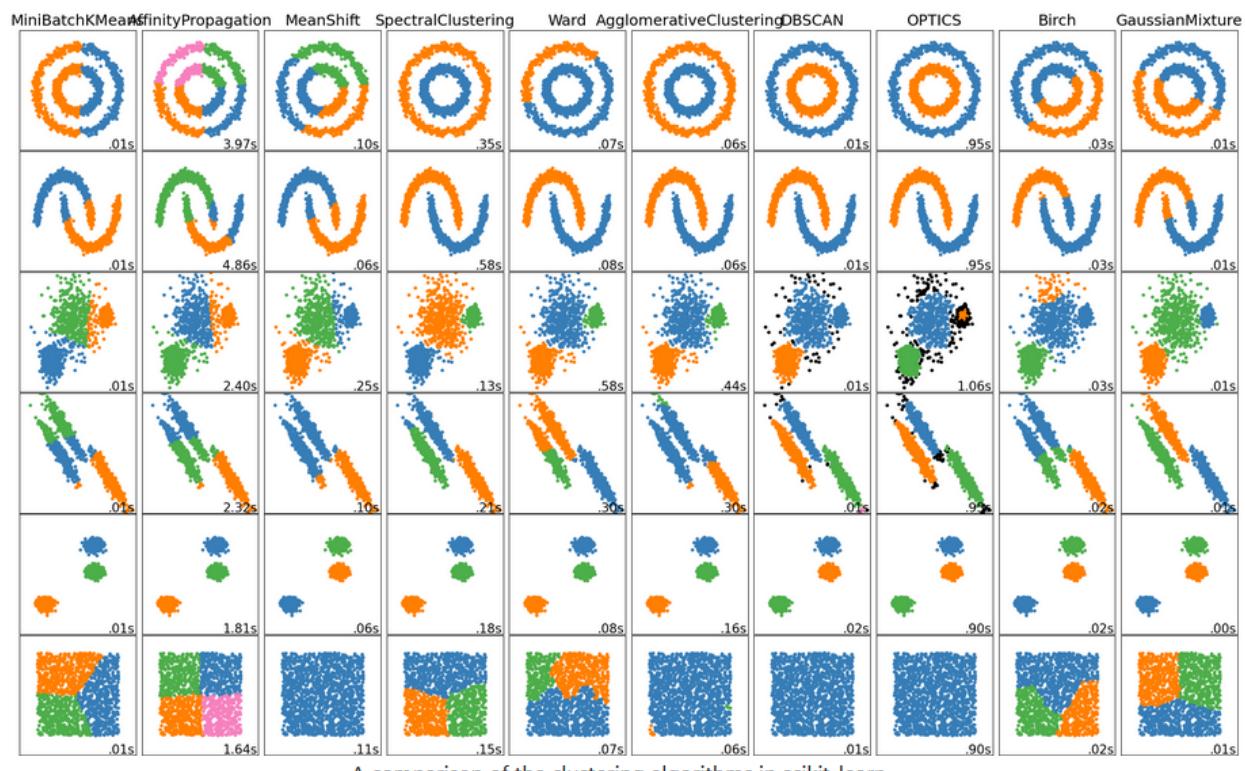
Otros algoritmos

Existen multitud de algoritmos de clúster implementados en la librería, es recomendable visitar la página:

<https://scikit-learn.org/stable/modules/clustering.html>

Method name	Parameters	Scalability	Use case	Geometry (metric used)
K-Means	number of clusters	Very large n samples, medium n_clusters with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium n samples, small n_clusters	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters or distance threshold	Large n samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters or distance threshold, linkage type, distance	Large n samples and n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large n samples, medium n_clusters	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large n samples, large n_clusters	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large n clusters and n_samples	Large dataset, outlier removal, data reduction.	Euclidean distance between points

En la siguiente figura se representa en cada fila un tipo de datos y en cada columna el algoritmo correspondiente, siendo los colores cómo quedarían divididos los clústeres.



UT-6

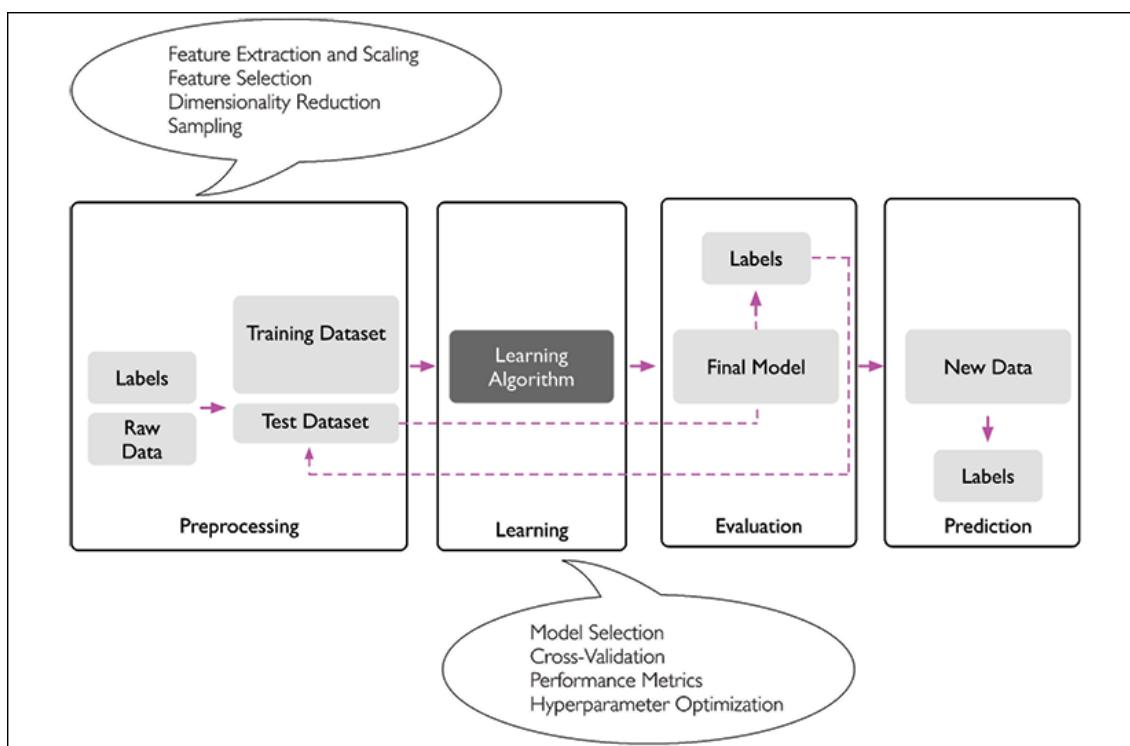
Cap 12.- Redes neuronales y Aprendizaje profundo: Introducción

Conceptos anteriores

El uso de NN en el ML se denomina como ya sabemos Deep Learning, pero este concepto viene de la cantidad de capas que podemos usar de redes para crear nuestro algoritmo. Deep Learning es técnicamente un algoritmo multicapa para aprender representaciones de los datos.

La regla general para distinguir entre el análisis de datos y el ML que veremos a continuación es la cantidad de datos. Si hay gran cantidad de datos, gran variedad y gran velocidad se debería usar ML frente al análisis de datos.

Los algoritmos que vamos a ver en el resto de curso basados en NN se sitúan en la fase de aprendizaje del ML, con lo que no podemos olvidar los conceptos que hasta ahora hemos aprendido del resto de fases.



Preprocesado de datos

- Tratamiento de los valores que faltan
- Tratamiento de los valores extremos
- Tratamiento de los datos categóricos
- Creación de los conjuntos de test, validación y entrenamiento
- Normalización de los datos
- Regularización

Reducción de la dimensionalidad

- PCA
- LDA
- Bosques aleatorios

Evaluación del modelo

- Curvas de aprendizaje
- Curvas de validación
- Uso de las métricas
 - Matriz de confusión
 - Recall, Precisión y F1
 - Curvas ROC

Ajuste de los hiperparámetros

- Holdout
- K-fold
- Búsquedas en Grid

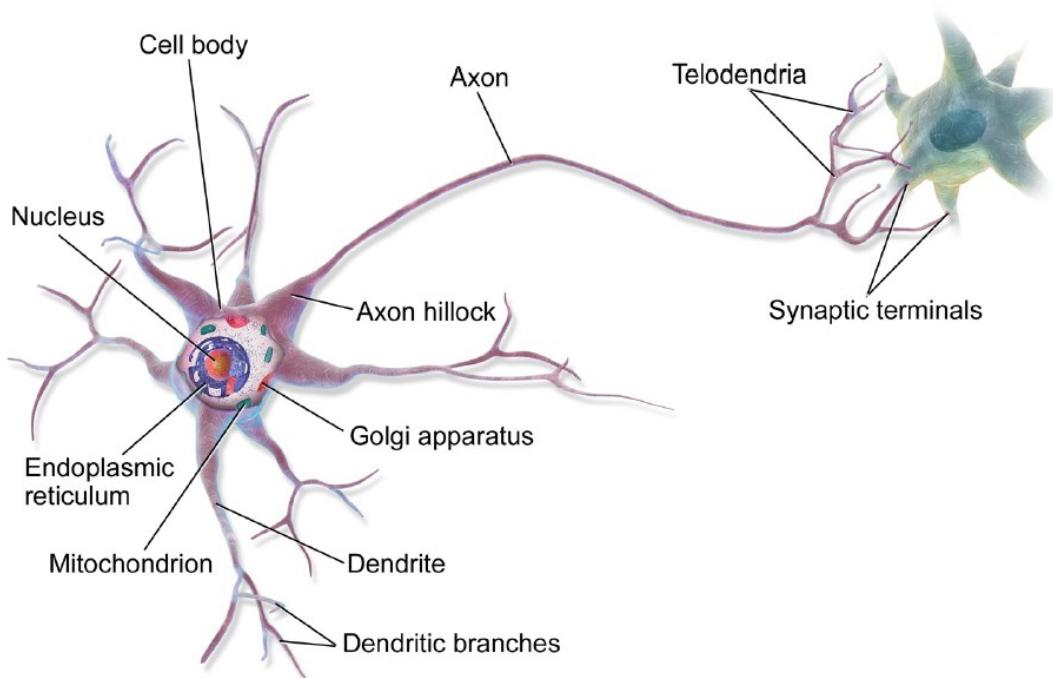
Problemas en el modelo

- Sobreajuste
- Subajuste

Introducción

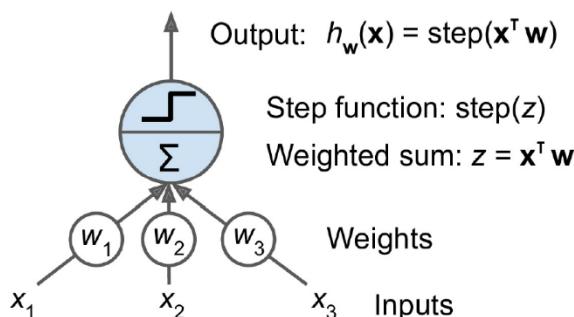
La idea de los algoritmos neuronales partió del estudio del cerebro y su funcionamiento, tomando como unidad de procesamiento la neurona y tratando de realizar un algoritmo que reflejase las relaciones existentes en el cerebro.

neurotransmitters, as some of them inhibit the neuron from firing).



En general, cada neurona está conectada a miles de otras con complejas reacciones químicas y físicas que hacen que la información fluya a través de ellas. Cada neurona recibe el estímulo de las que tiene de entrada y si la cantidad de estimulación es la suficiente se activa y transmitirá un impulso eléctrico a través de su axón que llegará a las neuronas a las que esté conectado dicho axón.

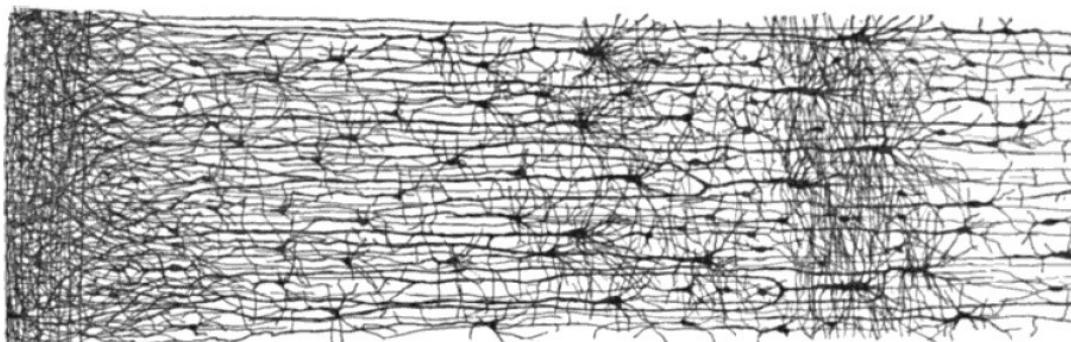
Si transformamos el modelo biológico en uno matemático: cada neurona artificial recibirá un conjunto de entrada de datos que evaluará a través de una función de activación. Si dicha función sobrepasa un umbral desencadenará la transmisión de su información a las neuronas que sean destino.



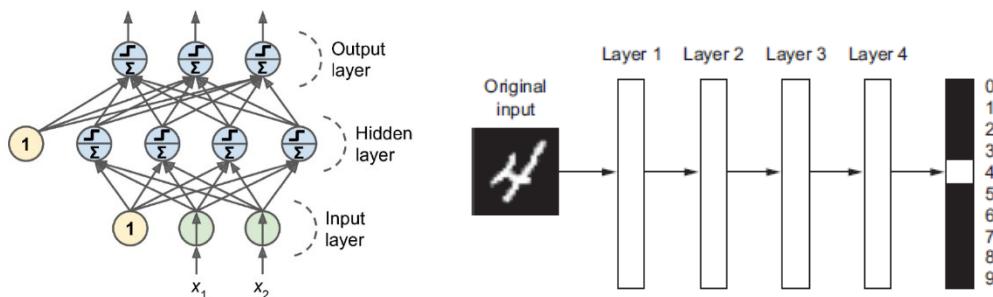
En la imagen anterior se implementa un modelo matemático del funcionamiento de una neurona (Perceptrón). Ese modelo se implementó al comienzo del curso.

Pero nuestro cerebro está formado por una cantidad ingente de neuronas, pero formando estructuras reconocibles. Estas estructuras están siendo estudiadas por los neurobiólogos, así como sus relaciones. Pero a nosotros nos interesa la estructura más interna, que se da a lo largo de todos los componentes del cerebro: las capas.

Las neuronas se organizan al segundo nivel organizativo en capas. Unas neuronas se activarán o no y pasarán la información a la siguiente capa, que será procesada del mismo modo y pasada a la siguiente, así hasta conseguir un resultado final.

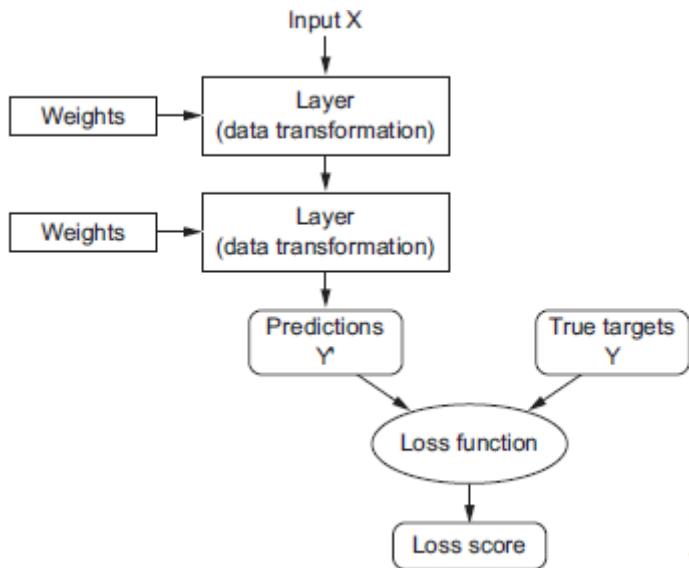


Si modelamos esta estructura organizativa para que sea funcional en una computadora, tendremos algo similar a la siguiente figura:



En la que aparecen tres capas, una de entrada, una oculta intermedia y otra de salida. En este caso cada neurona de una capa está conectada con todas las neuronas de la siguiente capa para

transmitir su activación si corresponde, y recoge la información de todas las neuronas de la capa anterior (se denomina capa densa). Lo que hace una capa está almacenado en los pesos de la capa, y aprendizaje significa buscar esos pesos.



Pero el modelo matemático es un poco más complejo, en el que se añade a cada entrada un peso específico para poder ponderar la importancia de la entrada en la decisión, y además se le añade un término independiente (representado por el círculo con el uno) para tener en cuenta la tendencia del ajuste (bias en inglés).

El algoritmo matemático intentará encontrar el valor de los pesos de cada conexión para que se ajusten lo más posible a la resolución de un problema dado, que se presentará a través de ejemplos de entrada. En la figura anterior a través de pares x_1 y x_2 .

Sin entrar en grandes detalles matemáticos, el algoritmo usado para determinar los pesos está basado en el descenso de gradiente (que debe calcular derivadas parciales para encontrar la dirección en la que se minimiza el error).

Para controlar la salida de una red neuronal, necesitamos medir la distancia entre lo predicho y lo real. Este trabajo lo realiza la función de pérdida (loss). El hecho más importante es utilizar esta distancia como feedback de la red para ajustar el valor de los pesos de todas las capas en la dirección inversa, es decir, de abajo a arriba. Este ajuste se hace mediante la función de optimización. El algoritmo usado es el **algoritmo de Backpropagation**. Con cada ejemplo, los pesos se ajustan un poco, y después de muchas vueltas se llega al ajuste deseado.

De forma un poco más detallada:

- El conjunto de todos los datos de entrada se utilizará varias veces (época), el número de épocas que usemos para entrenar la red será determinante. Los datos de entrada se agrupan en bloques (mini-batch) para tratarlos a la vez.
- Cada bloque (mini-batch) es pasado a la capa de entrada y se realiza toda la propagación a través de toda la red hasta encontrar un resultado.
- A continuación, se estima el error entre el resultado del punto anterior y el resultado real a través de una función de pérdida (loss).

- Se encuentra la cantidad que cada conexión aporta al error en la capa actual, utilizando la regla de la cadena (derivación). Este es el punto más costoso del entrenamiento y gracias a la regla de la cadena se puede hacer que sea computacionalmente tratable.
- Se calcula cuánto contribuye cada entrada de la capa anterior al error utilizando la regla de la cadena, se repite este paso empezando en la capa de salida y hasta la capa de entrada en orden inverso al de propagación.
- Se encuentra el gradiente y se modifican los pesos de las conexiones para ajustarlos a los datos esperados.

En resumen, el algoritmo realiza una predicción, calcula el error de cada una de las entradas en cada capa de forma inversa, encuentra los valores para minimizar dicho error y modifica los pesos para ajustarlos a lo esperado.

Como hemos visto para una red neuronal son necesarios los siguientes elementos:

- Capas que tendrá el modelo
- Para cada capa
 - Nombre de la capa (name) opcional.
 - Número de neuronas
 - Estructura de conexiones entre las neuronas
 - Función de activación. Las funciones de activación más comunes son:
 - Sigmoide
 - Tangente hiperbólica
 - RELU
- Para la capa de entrada
 - Atributo `input_shape` o `input_dim` indicando las dimensiones de las características. El primero espera una matriz y el segundo el número de neuronas de entrada
- Para la capa de salida
 - Número de salidas, que será igual al número de clases a predecir.
 - Función de activación.
- Función de pérdida o de medición del error, métrica y optimizador.

El algoritmo para un modelo de Red Neuronal sería el siguiente:

- Cargar los datos bajo NumPy o un Dataset.
- Crear los conjuntos de Entrenamiento, Test y validación (si son NumPy esto es opcional).
- Crear el modelo: Secuencial, API Funcional o API Dinámica.
- Compilar el modelo pasando la función de pérdida, el optimizador y la lista de métricas.
- Ajustar el modelo (`fit`) pasando los datos de entrenamiento y validación (o el porcentaje si son en NumPy) y el número de épocas y el tamaño del mini-batch (`batch_size`)
- Comprobar las curvas de aprendizaje con el objeto `history` devuelto del `fit`
- Comprobar las métricas
- Evaluar el modelo con los datos de test
- Grabar el modelo y predecir.

Primer ejemplo de NN

```
import pandas as pd
from keras.utils import np_utils
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler, normalize
from tensorflow import keras

data = pd.read_csv("iris.data", header=None, encoding='utf-8')

data.loc[data[4] == "Iris-setosa", 4] = 0
data.loc[data[4] == "Iris-versicolor", 4] = 1
data.loc[data[4] == "Iris-virginica", 4] = 2

X = data.iloc[:, :4].values
y = data.iloc[:, 4].values

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1, stratify=y)
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

y_train = np_utils.to_categorical(y_train, num_classes=3)
y_test = np_utils.to_categorical(y_test, num_classes=3)

model = keras.Sequential()
model.add(keras.layers.Dense(10, input_dim=4, activation='relu'))
#empezar con 100 50 y pasar a 10 5
model.add(keras.layers.Dense(5, activation='relu'))
model.add(keras.layers.Dense(3, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy'])
history = model.fit(X_train, y_train, validation_data=(X_test,
y_test), batch_size=20, epochs=300) # 150 a 300

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

y_proba = model.predict([[0.22, 0.33, 0.23, 0.56]])
print(y_proba)

# prediction=model.predict(X_test)
# length=len(prediction)
# y_label=np.argmax(y_test, axis=1)
# predict_label=np.argmax(prediction, axis=1)
#
# accuracy=np.sum(y_label==predict_label)/length * 100
# print("Accuracy of the Dataset",accuracy )
```

Código 025.py

Hola mundo de las NN

La BBDD de caracteres del MNIST es la base de muchos de los algoritmos de NN para el aprendizaje, vamos a utilizarla ahora.

```
from keras.datasets import mnist
from keras.utils.np_utils import to_categorical
from keras import models
from keras import layers

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
# Las imágenes son una matriz de 28x28

print(train_images.shape)      # (60000, 28, 28)
print(len(train_labels))       # 60000
print(train_labels)            # array([5, 0, 4, ..., 5, 6, 8],
dtype=uint8)
print(test_images.shape)        # (10000, 28, 28)
print(len(test_labels))        # 10000
print(test_labels)             # array([7, 2, 1, ..., 4, 5, 6],
dtype=uint8)

# pasamos de 28x28 a 768
train_images = train_images.reshape((60000, 28 * 28))
# se pasa a float32 por eficiencia en las GPU y CPU
# normalización al rango [0..1]
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

digit = train_images[4]
import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()

network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
                       input_shape=(28 * 28,)))  # 768 entradas
network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])

network.fit(train_images, train_labels, epochs=5, batch_size=128)

test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
```

Código 025_b.py

Representación de los datos en NN

Los datos en redes neuronales se representan generalmente bajo Arrays de diferentes dimensiones llamados Tensores:

- Escalares (Tensores de 0D)

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

- Vectores (Tensores de 1D)

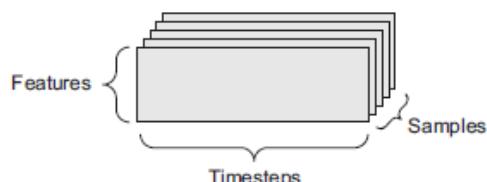
```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

- Matrices (Tensores 2D)

```
>>> x = np.array([[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]])
>>> x.ndim
2
```

- Tensores en 3D

```
>>> x = np.array([[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]],
[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]],
[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]]])
>>> x.ndim
3
```



En Deep Learning podremos llegar a manejar tensores 5D para vídeos

Los tensores tienen los siguientes parámetros a tener en cuenta.

- Número de ejes (o rango), atributo **ndim**. En general, el primer eje (axis=0) son los ejemplos
- Dimensión. Una tupla que indica el número de ejemplos en cada uno de los ejes (**shape**)
- Tipo. Indica el tipo almacenado, uint8, uint32, float32, float64.

Caso Especial: Tratamiento de las imágenes

Las imágenes tienen una estructura peculiar, generalmente vienen en formato de 2D más los canales de color, por lo que al cargar las imágenes nos encontraremos con (ejemplos, x, y, canales), donde canales serán 1 para monocromo y 3 o 4 para color, generalmente 3 ya que el

canal alfa se suele eliminar. Para usar este tipo de datos en nuestra NN debemos aplanar las dimensiones x e y en una sola de x*y con `reshape()`.

Del mismo modo, los vídeos vendrán en un formato (ejemplo, frame, x, y, canales)

Anatomía de una red neuronal

Las redes neuronales están formadas por los siguientes elementos:

- **Capas.** Es la estructura fundamental, recoge un tensor de entrada, le aplica una serie de operaciones matemáticas y genera uno o más tensores de salida. Generalmente tiene un estado, los pesos.
- **Modelos.** Es un grafo acíclico de capas que van a procesar los datos. El grafo más general es la pila de capas o secuencial. La topología que implementemos será lo que se llama espacio de la hipótesis. Es muy importante acertar con la topología para el problema que tenemos que desarrollar.
- **Función de pérdida y optimizador.** La primera define la distancia entre lo predicho y lo real, y la segunda se encarga de usar los datos de la primera para ajustar los pesos. Es muy importante elegir la función de pérdida adecuada para el problema.
- **Función de activación.** Función que determinará en una neurona si la suma de los pesos por las entradas hace que se active la neurona y pase el resultado a la siguiente.

Clasificación con NN

Las redes neuronales también se pueden usar de forma muy eficiente para tareas de clasificación, tanto binarias como cualquier otro tipo. Lo único que debemos tener en cuenta es que se usará como función de activación Softmax (también llamada logistic) para la capa de salida.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers		One per input feature (e.g., 28 x 28 = 784 for MNIST) Depends on the problem, but typically 1 to 5 Depends on the problem, but typically 10 to 100	
# output neurons	1	1 per label	1 per class
Problem type	Last-layer activation		Loss function
Binary classification	sigmoid		<code>binary_crossentropy</code>
Multiclass, single-label classification	softmax		<code>categorical_crossentropy</code>
Multiclass, multilabel classification	sigmoid		<code>binary_crossentropy</code>
Regression to arbitrary values	None		<code>mse</code>
Regression to values between 0 and 1	sigmoid		<code>mse or binary_crossentropy</code>

Implementación de un Perceptrón multicapa (MLP) con Keras: Clasificación

La programación de redes neuronales es un gran trabajo y conlleva muchas subtareas. La elección del API a utilizar es muy importante y se deben considerar diferentes factores. Las

librerías más populares son Keras, Tensorflow y PyTorch. La primera de alto nivel y las dos siguientes de más bajo nivel. Tensorflow es un proyecto libre de Google mientras que PyTorch está desarrollado por Facebook, ambas librerías se han igualado bastante con la versión 2 de Tensorflow. Además, esta versión de Tensorflow incorpora Keras como parte de su implementación, con lo que facilita el acceso a la misma. En este curso usaremos Keras y Tensorflow.

Código 025_c.py

Hemos implementado nuestra primera red neuronal con éxito, y podemos pasar al siguiente nivel, pero antes tendremos en cuenta los siguientes aspectos:

- Si el resultado no es el esperado, podemos afinar los hiperparámetros de la red.
 - Afinar Learning rate.
 - Elegir otro optimizador (hay que reiniciar el parámetro anterior antes del cambio).
 - Cambiar el número de capas.
 - Cambiar el número de neuronas por capa.
 - Cambiar las activaciones de las capas.
 - Cambiar el tamaño de batch (mini-batch)
- Pero lo más importante, es no realizar este paso de afinado de los hiperparámetros con los ejemplos de test, utilizar otro conjunto diferente.

Código 025_d.py

Código 025_e.py

Resumen

- Generalmente se necesita Preprocesado de los datos. Las secuencias de palabras pueden ser codificadas como vectores binarios, pero hay más opciones.
- El modelo de capas apiladas con la función de activación relu puede resolver una gran cantidad de problemas.
- En el problema de clasificación binaria, la red debe terminar con una capa densa con una única neurona y con la función sigmoid. La salida será un valor entre 0 y 1 indicando la probabilidad de que sea correcta. Para este problema usaremos la función de pérdida binary_crossentropy.
- El optimizador rmsprop es generalmente una buena opción para cualquier problema.
- Hay que monitorizar siempre la salida para comprobar el rendimiento.
- Si estamos clasificando varias clases, la última capa debe ser densa con tantas neuronas como clases a clasificar
- En caso de un problema de multiclasiificación pero para una única etiqueta, la rede debe usar Softmax como función de activación.
- Para problemas categóricos la función de pérdida debe ser crossentropy.
- Hay dos formas de tratar las etiquetas en un problema de multiclasiificación
 - Codificar como OneHot y usar categorical_crossentropy
 - Codificar usando enteros y usar sparse_categorical_crossentropy

- Si hay que clasificar los datos en una gran cantidad de categorías, hay que evitar que las capas intermedias tengan pocas neuronas para evitar cuellos de botella

Referencia

- <https://keras.io/losses>
- <https://keras.io/optimizers>
- <https://keras.io/metrics>

Regresión con NN

Las redes neuronales se pueden utilizar para problemas de regresión (predicción) tanto binaria como múltiple. En este último caso deberemos tener tantas neuronas de salidas como clases a predecir.

En general, no necesitaremos una función de activación ya que el fin es predecir, con lo que nos interesa que se transmita toda la información siempre. Se puede considerar que la salida sea siempre positiva por razones de eficiencia por lo que se recomienda usar la función RELU para este tipo de tareas, o en caso que tengamos que acotar los resultados de salida entre dos valores fijos usaremos la función sigmoide (0..1) o la tangente hiperbólica (-1..1). En caso de la función de pérdida se podrá usar la función MSE (Error medio al cuadrado) pero si el conjunto de datos tiene gran cantidad de extremos (outliers) se usará mejor MAE (Error absoluto medio).

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	<code>binary_crossentropy</code>
Multiclass, single-label classification	softmax	<code>categorical_crossentropy</code>
Multiclass, multilabel classification	sigmoid	<code>binary_crossentropy</code>
Regression to arbitrary values	None	<code>mse</code>
Regression to values between 0 and 1	sigmoid	<code>mse</code> or <code>binary_crossentropy</code>

Implementación de un Perceptrón multicapa (MLP) con Keras: Regresión

La principal diferencia entre la regresión y la clasificación es el hecho que la capa de salida debe tener una única neurona, no tiene función de activación y la función de pérdida es MSE.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import keras

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu",
                      input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])

model.compile(loss="mean_squared_error", optimizer="sgd")

history = model.fit(X_train, y_train, epochs=20,
                     validation_data=(X_valid, y_valid))

mse_test = model.evaluate(X_test, y_test)
print("loss:", mse_test)

X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
print(y_pred)
```

Código 025_f.py

Ejercicio:

- Probar con tres capas ocultas en vez de dos
- Cambiar el número de neuronas: 32 y 64
- Usar la función tanh de activación en vez de relu

Código 025_g.py

Resumen

- Para la regresión se utilizan funciones de pérdidas diferentes que para la clasificación.
La más usada es la MSE.
- La métrica más usada es MAE
- Cuando las entradas están en diferentes escalas hay que normalizarlas.
- Cuando haya pocos datos se debe usar el procedimiento K-Fold para el entrenamiento.

Almacenamiento del modelo

Nuestro modelo una vez compilado y entrenado deberá ser almacenado para su uso posterior y cargado por el programa que haga uso de él. A continuación, se muestran las instrucciones para dicha tarea.

```
model.save("my_keras_model.h5")
model = keras.models.load_model("my_keras_model.h5")
```

El almacenamiento incluye.

- Los modelos de arquitectura.
- Los valores de peso del modelo (que se aprendieron durante el entrenamiento).
- La configuración de entrenamiento del modelo (lo que pasó a 'compilar'), si corresponde
- El optimizador y su estado, si corresponde (esto le permite reiniciar el entrenamiento donde lo dejó).
- No guardará funciones de activación u optimizadores personalizados.

Este procedimiento funciona para el modelo secuencial o para el modelo por API, pero no para el modelo creado a través de herencia.

También se puede exportar un modelo completo al formato 'SavedModel' de Tensorflow que es un formato de serialización independiente compatible con las implementaciones de Tensorflow que no sean Python.

```
# Exportar el modelo a 'SavedModel'
keras.experimental.export_saved_model(model, 'path_to_saved_model')

# Recrea exactamente el mismo modelo
new_model =
    keras.experimental.load_from_saved_model('path_to_saved_model')
```

Para más información visite la guía de referencia

https://www.tensorflow.org/guide/keras/save_and_serialize

Uso de llamadas externas (callbacks)

El método de Keras **fit** permite un parámetro (**callbacks**) que contendrá una matriz con los nombres de las funciones a llamar al comienzo y al final de cada entrenamiento, cada época e incluso de cada mini-batch.

Podemos hacer uso de callbacks predefinidas (**ModelCheckpoint**, **EarlyStopping**) o nuestras propias callbacks.

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10,
callbacks=[checkpoint_cb])
```

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
validation_data=(X_valid, y_valid), callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5")
```

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                 restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

<https://keras.io/api/callbacks/>

Tensorboard

Tensorboard es una aplicación que sirve para visualizar los datos de aprendizaje de un modelo, su uso es muy sencillo y debería ser usada por todos los programadores de NN.

El primer paso es decirle al método **fit** que guarde los datos en un directorio, es recomendable que esté vacío, pero no imprescindible.

```
log_dir = "logs/fit/" +
          datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

tensorboard_callback = tf.keras.callbacks.TensorBoard(
    log_dir=log_dir, histogram_freq=1)

model.fit(x=x_train, y=y_train, epochs=5,
           validation_data=(x_test, y_test),
           callbacks=[tensorboard_callback])
```

Una vez ejecutado el programa se creará un nuevo directorio contenido los datos de log. A continuación, hay que lanzar el server para la visualización desde un terminal.

`tensorboard --logdir ch14/logs/fit`

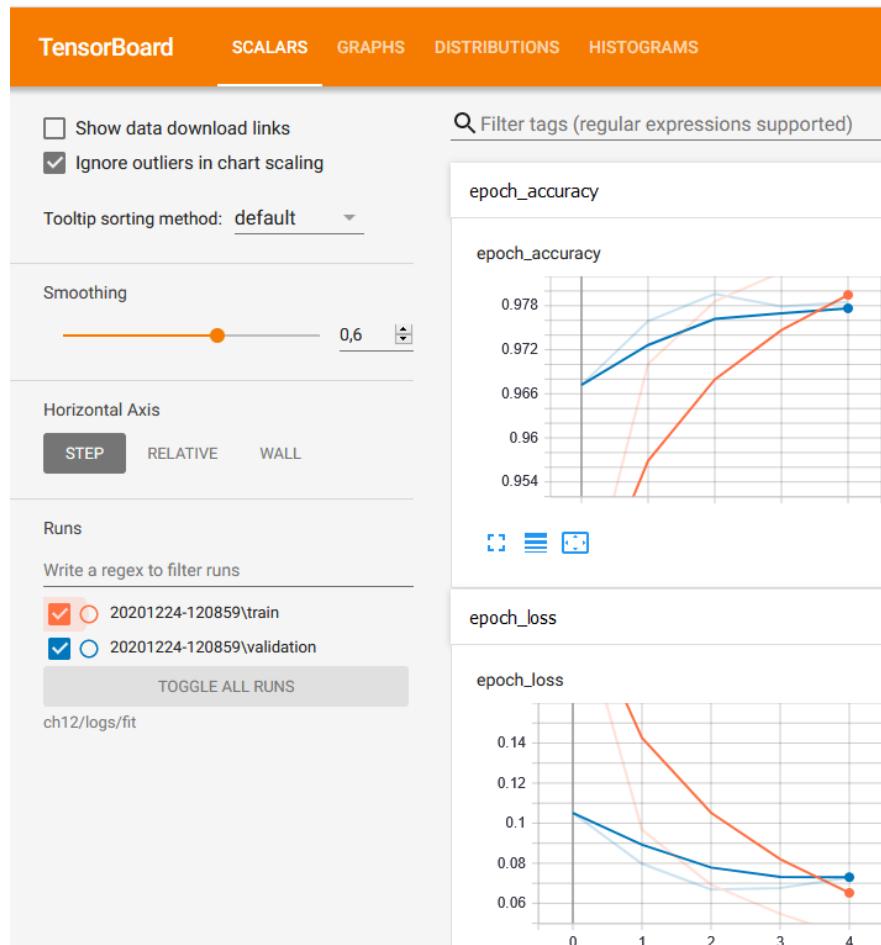
Y acceder al servidor a través de un navegador web.

<http://localhost:6006/>

Opcionalmente se puede usar con los blocs de Jupyter.

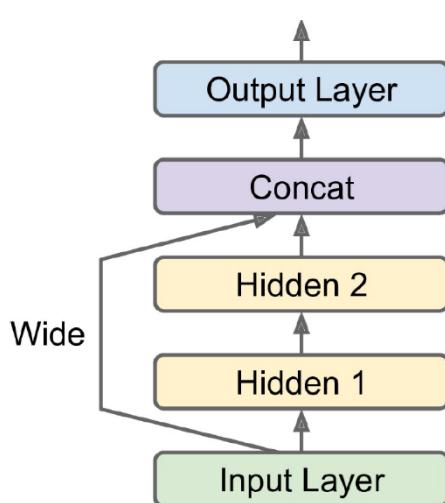
```
%load_ext tensorboard
%tensorboard --logdir=~/my_logs --port=6006
```

Código 026.py



El API Funcional Keras

El modelo Secuencial de Keras es muy estático y predefinido. Si necesitamos una mayor flexibilidad a la hora de crear nuestras redes, tendremos que hacer uso del API de más bajo nivel. Con éste podemos hacer conexiones inter-capas no consecutivas, salidas de capas intermedias, y mucho más.



Supongamos que necesitamos implementar el modelo que aparece a la izquierda, en donde la última capa oculta recibe datos directamente de la capa de entrada y de la capa anterior (hidden 2). Este modelo sería imposible de implementar directamente con las clases de Keras con lo que deberemos hacer uso del API.

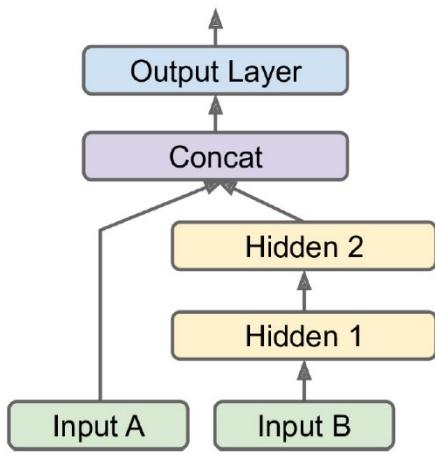
En la implementación siguiente podemos ver cómo haciendo uso de **Concatenate** creamos una nueva capa que recibe la entrada de varias. También observamos cómo se usan variables para crear las capas y se les pasa en el constructor la capa de las que reciben las entradas.

Se crea la capa de entrada con el tamaño de las características (`input_`), después se crean dos capas ocultas en secuencia (`hidden1` que recibe de `input_` y `hidden2` que recibe de `hidden1`), a continuación, se crea la capa de concatenación con la entrada de las capas `input_` y `hidden2` y por último se

crea la capa de salida recibiendo los datos de concat. Con toda la definición se crea el modelo pasando las entradas y las salidas.

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Cabe destacar que en este API las capas de las que se reciben los datos se pasan como si fuera una llamada a una función, no en el constructor (ver los segundos paréntesis después de los datos del constructor).

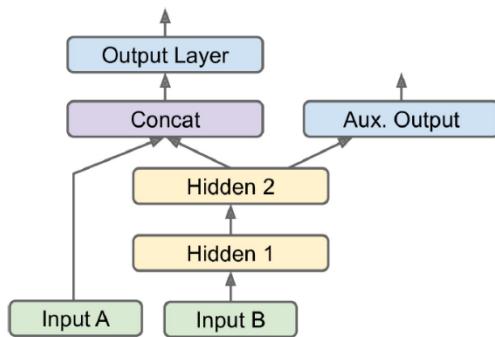


```
input_A = keras.layers.Input(shape=[5],
                             name="wide_input")
input_B = keras.layers.Input(shape=[6],
                             name="deep_input")
hidden1 = keras.layers.Dense(30,
                            activation="relu")(input_B)
hidden2 = keras.layers.Dense(30,
                            activation="relu")(hidden1)
concat = keras.layers.concatenate(
    [input_A, hidden2])
output = keras.layers.Dense(1,
                           name="output")(concat)
model = keras.Model(
    inputs=[input_A, input_B],
    outputs=[output])
```

En este segundo modelo Podemos ver que se definen dos capas de entrada `input_A` e `input_B` que aparte de utilizarlas como en el ejemplo anterior, se deben indicar a la hora de crear el modelo. Esta definición nos obliga que a los métodos `fit`, `evaluate` o `predict`, haya que pasarle una tupla con dos pares de matrices para cada una de las entradas.

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```



Del mismo modo que hemos usado varias entradas podemos necesitar tener salidas intermedias o incluso varias salidas. Ejemplos de esta situación son: Querer localizar y clasificar, tener varias tareas independientes sobre los mismos datos, regularizar y necesitar los datos intermedios, etc.

Realizar estas salidas es muy fácil, se crea la estructura como ahora y se añade en el modelo a la lista de salidas. Deberemos proporcionar tantas

funciones de pérdida como salidas hayamos creado y el sistema utilizará ambas para crear una función compuesta de salida entre ambas, incluso podemos ponderar el peso de cada función en la función compuesta final a través del parámetro **loss_weight**.

```

input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(
    inputs=[input_A, input_B],
    outputs=[output, aux_output])

model.compile(loss=["mse", "mse"],
              loss_weights=[0.9, 0.1], optimizer="sgd")
  
```

Al tener ahora dos salidas, deberemos proporcionar a los métodos **fit** y **evaluate** dos conjuntos de etiquetas de salida, y los datos que podemos recoger a través del método **evaluate** serán los datos de ambas funciones de pérdida y de la compuesta. A través del método **predict** recogeremos ambas salidas en el orden definido en el momento de la compilación.

```

history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))

total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])

y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
  
```

El método **fit()** necesita un conjunto de datos de validación diferentes a los de test, éstos se pueden crear antes y proporcionar a través de **validation_data** o se puede proporcionar un único conjunto de datos e indicar el porcentaje de validación a través de **validation_split** (solo si los datos se pasan como Arrays NumPy).

Este API presenta gran cantidad de funciones, pero una que nos puede ayudar mucho a visualizar al estructura del modelo es **keras.utils.plot_model(modelo,"ruta.png")** que creará un gráfico con la estructura del modelo.

Modelado dinámico con el API

Las dos aproximaciones a la creación de redes a través de Keras vistas son declarativas, es decir una vez definidas no pueden cambiar su forma ni sus conexiones. Habrá momentos en los que desearemos que nuestras redes sean completamente dinámicas y tengan una u otra forma en función del programa correspondiente. Tal vez necesitamos incrementar las capas en un bucle, o crear una u otra capa según una condición, esta funcionalidad se presenta a través del subclasing de Keras.

La idea es extender la clase **Model** de Keras e implementar los métodos necesarios para el funcionamiento. Las capas se crearán en el **constructor** y a continuación se realizará la asignación en el método **call** pero con toda la funcionalidad del lenguaje, pudiendo usar bucles, condiciones, etc.

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(units,
                                          activation=activation)
        self.hidden2 = keras.layers.Dense(units,
                                          activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

Hay que tener en cuenta que cualquier modelo de Keras puede ser usado a su vez como capa de otro modelo.

Afinado de los hiperparámetros

Tal y como vimos en los capítulos referidos a la regresión y clasificación, en las redes neuronales hay que afinar los hiperparámetros también. El mayor problema de las NN es la cantidad de ellos que hay.

Una primera aproximación, pero computacionalmente muy costosa, es utilizar un modelo en Grid para realizar dicha búsqueda. En caso que así lo hagamos, será mejor usar una búsqueda aleatoria en Grid que una secuencial. Dependiendo los parámetros que ajustemos, el modelo y los datos puede durar hasta varios días el procedimiento. Una vez que termine podremos usar los mismos parámetros (**best_score** y **best_params_**) para encontrar la mejor solución.

```
import numpy as np
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV
from tensorflow import keras
from keras.datasets import imdb

numero_de_palabras_a_recoger = 10000
```

```

def vectorize_sequences(sequences,
dimension=numero_de_palabras_a_recoger):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3,
input_shape=[numero_de_palabras_a_recoger]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model

(train_data, train_labels), (test_data, test_labels) =
imdb.load_data(num_words=numero_de_palabras_a_recoger)
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')

x_val = x_train[:numero_de_palabras_a_recoger]
partial_x_train = x_train[numero_de_palabras_a_recoger:] # estos
datos para afinar los hiperparámetros no todos
y_val = y_train[:numero_de_palabras_a_recoger]
partial_y_train = y_train[numero_de_palabras_a_recoger:]

keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
param_distrib = {"n_hidden": [0, 1, 2, 3],
                 "n_neurons": np.arange(1, 100),
                 "learning_rate": reciprocal(3e-4, 3e-2),
                 }
rnd_search_cv = RandomizedSearchCV(keras_reg, param_distrib,
n_iter=10, cv=3)
rnd_search_cv.fit(x_train, y_train, epochs=100,
validation_data=(x_val, y_val),
callbacks=[keras.callbacks.EarlyStopping(patience=10)])

print(rnd_search_cv.best_estimator_)
print(rnd_search_cv.best_params_)
print(rnd_search_cv.best_score_)

```

Cualquier parámetro extra que se pase al método **fit** se pasará también al modelo subyacente.

Parada temprana

Los cálculos sobre las NN son muy costosos, y podemos hacer que los algoritmos paren cuando la tasa de aprendizaje deje de mejorar en un porcentaje o cantidad

```

rnd_search_cv.fit(x_train, y_train, epochs=100,
                  validation_data=(x_val, y_val),
callbacks=[keras.callbacks.EarlyStopping(patience=10)])

```

Patience indica el número de épocas en las que no se mejora. Podemos usar el parámetro **min_delta** que indicará el cambio mínimo que se tiene que producir y **monitor** la cantidad a monitorizar.

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

Búsqueda manual

Aunque la búsqueda en Grid se puede utilizar para problemas simples, no es el método más apropiado para problemas más complejos. En estos casos la idea es buscar una zona de parámetros que parezca buena, y sobre esa zona realizar una búsqueda más exhaustiva. Hay multitud de librerías que nos pueden ayudar a tal fin: Hyperopt, Hyperas, kopt, Talos, Keras Tuner, Scikit-optimize (skopt), Spearmint, Hyperbanda, Sklearn-Deep y muchas compañías están desarrollando servicios en red para este fin: Googel, Arimo, SigOpt, etc...

Independientemente de las librerías y servicios en red, podemos tener en cuenta algunos consejos para hacerlo nosotros de forma manual.

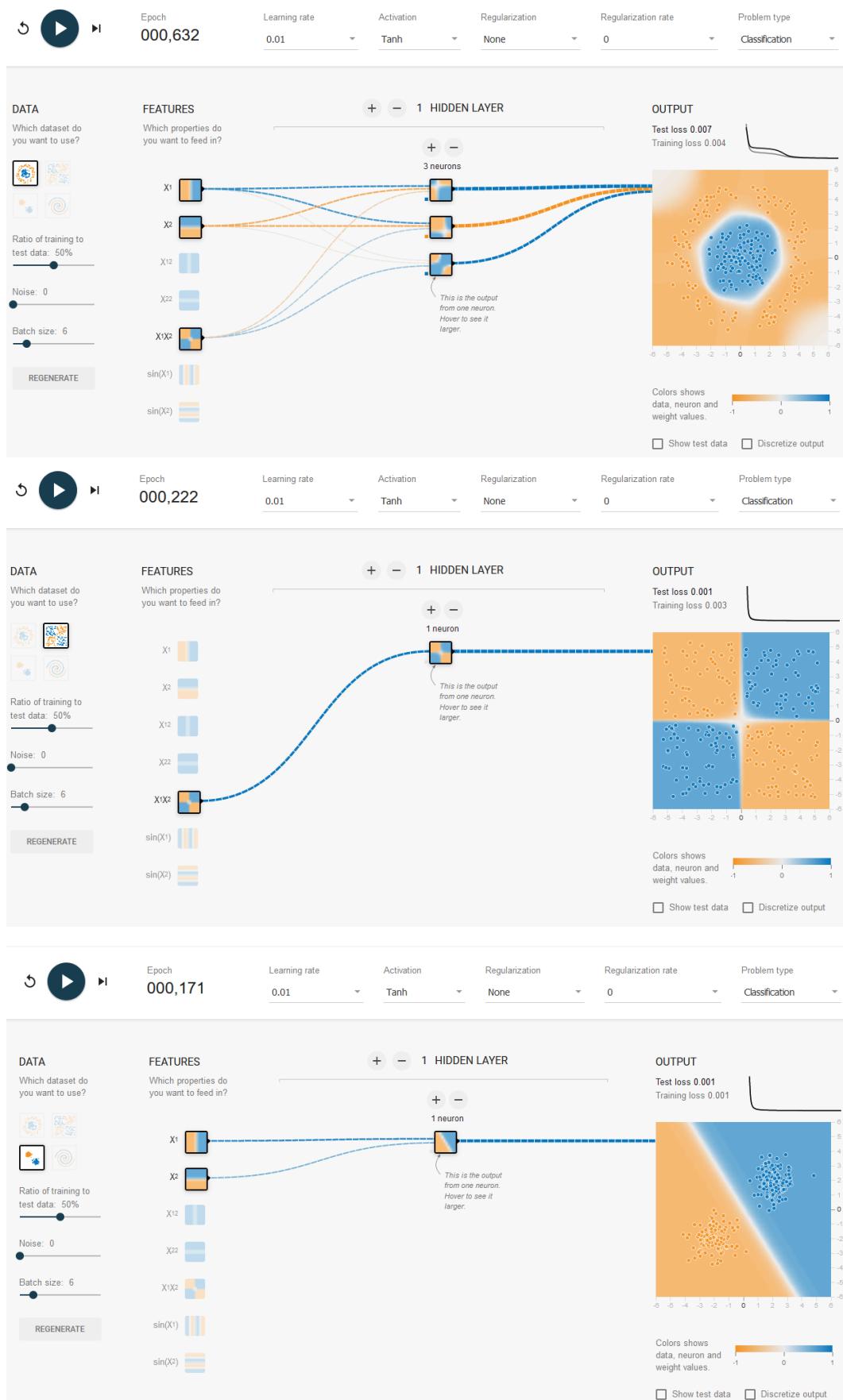
- Número de capas ocultas. En teoría una capa únicamente es suficiente para funciones complejas si se proporcionan el número suficiente de neuronas, pero será menos eficiente que si aumentamos el número de capas con menos neuronas por capa. Cada capa oculta que añadimos agrega un nivel de abstracción y reconocimiento al problema, por lo que la estructura de la NN debe ser acorde a la estructura jerárquica subyacente del problema. Cada capa superior captura detalles de más alto nivel utilizando lo reconocido en la capa inferior. Además, cuantas más capas haya más generalizarán un problema y se podrá reutilizar la estructura y lo aprendido en otro tipo de problemas. Se pueden reutilizar conjuntos de capas ya aprendidas en otros problemas sin tener que volver a enseñar al modelo, es lo que se denomina: aprendizaje transferido. Para muchos problemas podemos empezar con una o dos capas ocultas e ir aumentando el número de las mismas. Por ejemplo, en problemas de reconocimiento son necesarias cientos de capas (no densas) para que den un buen rendimiento.
- Número de neuronas por capa. El número de neuronas en las capas de entrada y salida vienen determinado por el problema. El número de neuronas de las capas intermedias se solía usar una estructura piramidal en la que empezábamos con muchas neuronas y se iban reduciendo paulatinamente capa a capa. Actualmente, se cree que se puede mantener el mismo número de neuronas en todas las capas ocultas, por lo que solo habrá que tener en cuenta un número, no uno por cada uno.
- En general será mejor empezar con un modelo con más capas y neuronas de las necesarias y utilizar la parada temprana y otras normas de regularización para prevenir el sobreajuste. Del mismo modo, normalmente será mejor incrementar el número de capas que el número de neuronas de la capa.
- Tasa de aprendizaje. La tasa de aprendizaje óptima suele estar a la mitad de la máxima tasa de aprendizaje (el algoritmo empieza a divergir). Un buen método para determinar la tasa de aprendizaje es empezar con una muy baja e ir incrementando la tasa en un constante en cada iteración, si dibujamos la función de pérdida frente a la tasa de aprendizaje (con escala logarítmica en la tasa de aprendizaje) se verá un agujero al principio, pero poco después se estabilizará. La tasa de aprendizaje óptima está un poco

antes que el punto en donde la gráfica empieza a subir de nuevo, aproximadamente diez veces menos.

- Optimizador. Es muy importante y veremos nuevos optimizadores más adelante.
- Tamaño del bloque. Puede generar un gran impacto en el tiempo de ejecución del algoritmo de aprendizaje. Cuanto mayor sea este tamaño menos tiempo tardará en ejecutarse, pero peor será la generalización que se obtendrá por el modelo. Se puede utilizar tamaños más grandes al principio y si vemos que no es estable bajar el tamaño del bloque, y establecer la tasa de aprendizaje también ya que están muy relacionados.
- La función de activación. RELU es una buena opción para las capas ocultas, para la de salida dependerá del problema.
- Número de iteraciones (epoch). No es un problema establecerlo al número tan grande como deseemos si utilizamos la parada temprana.

<https://playground.tensorflow.org>





Cap 13.- Redes neuronales y Aprendizaje profundo: Ajuste de la red

Las redes neuronales que pueden ser sustituidas por algoritmos de regresión o clasificación no suelen ser muy profundas (muchas capas) por lo que no presentan graves problemas. Es cuando avanzamos en el entrenamiento de redes más grandes cuando aparecen ciertos problemas con los que tenemos que lidiar:

- Desajustes en los crecimientos de los gradientes. Referentes a este punto encontramos dos problemas, que el gradiente se hace cada vez más y más pequeño en la Backpropagation, con lo que las capas más cercanas a la entrada no se actualizan, o lo contrario, el gradiente se hace cada vez más grande y haciendo que nunca converja el problema.
- Puede ser que no tengamos acceso a suficientes datos etiquetados, o que tengamos muchos sin etiquetar pero que sea muy costoso etiquetarlos.
- El entrenamiento puede que nos lleve mucho tiempo.
- Un modelo con cientos de miles de parámetros puede sobreajustarse a los datos muy fácilmente, sobre todo si no hay suficientes datos o son muy ruidosos.

Para prevenir que un modelo aprenda patrones irrelevantes en los datos de test, la mejor aproximación es conseguir más datos para entrenarlo. Si esto no es posible, reduciendo la capacidad de la red podremos forzar a que se centre en los patrones relevantes.

Reducir el tamaño de la red

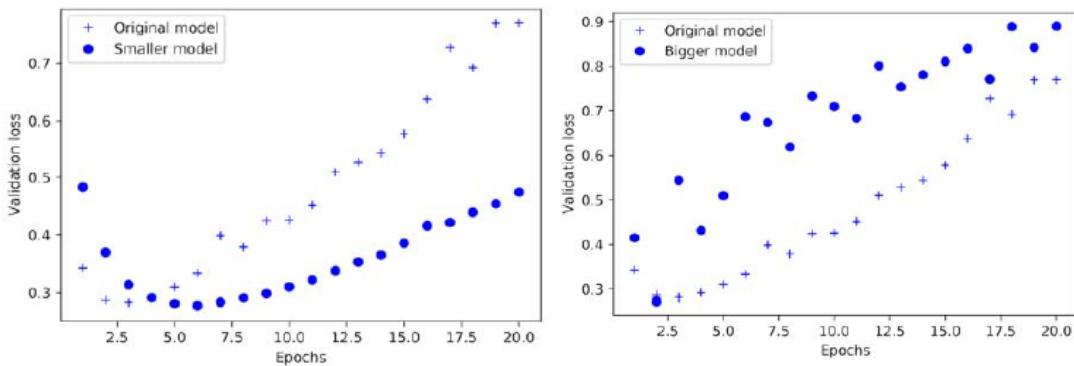
La manera más simple para prevenir el sobreajuste es reducir el tamaño del modelo, o bien el número de capas o el número de neuronas por capa.

El verdadero problema en las redes neuronales es la generalización, ya que ajustarse a los datos de entrenamiento se puede conseguir “fácilmente” pero que una vez ajustado, los resultados sean tan buenos en los datos de prueba es lo difícil. No hay una manera exacta de determinar el número de capas o de sus componentes de forma fácil.

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



Desajustes en los gradientes.

El algoritmo de Backpropagation va desde la capa de salida y va subiendo por la estructura hasta la capa de entrada, aplicando el reajuste de pesos de abajo a arriba. Desafortunadamente, el gradiente que se va calculando al aplicarlo en este orden hace que las capas más cercanas a la entrada queden prácticamente invariadas, con lo que la solución nunca converge (desvanecimiento del gradiente). El caso opuesto también se da, en el que los cambios se hacen cada vez mayor, siendo enorme en las capas más cercanas a la entrada (estallido del gradiente) haciendo que el problema tampoco converja. Por si no fuera poco, existe un tercer problema en las NN de gran tamaño, referente al hecho de que cada capa aprenderá a ritmos muy diferentes.

El problema parece ser que viene del uso de la función sigmoide junto con la estandarización a la normal de los datos haciendo que en valores muy cercanos se saturen. Para dar solución a estos problemas han aparecido diversos métodos que vamos a explorar.

Inicialización Glorot y He

La idea de estos mecanismos es inicializar los pesos de forma aleatoria según una función específica (no vamos a entrar a describir la función correspondiente).

Por defecto Keras usa la inicialización Glorot que puede ser cambiada a `Hi` de la siguiente forma:

```
keras.layers.Dense(10, activation="relu",
                   kernel_initializer="he_normal")
```

También se puede usar la activación LeCun, muy similar a Glorot.

```
keras.layers.Dense(10, activation="relu",
                   kernel_initializer="lecun_normal")
```

Inicialización	Función de activación
Glorot	None, tanh, logistic, softmax
He	ReLU and variants
LeCun	SELU

Funciones de activación que no saturan

Los problemas con el gradiente son en parte por la función de activación y sus características, por lo que han ido apareciendo otras funciones que intentan dar solución a este problema.

- RELU. Esta función no es perfecta, pero es el primer paso. Esta función permite una buena aproximación, pero también sufre de un defecto: algunas neuronas “morirán” no aportarán nada a la solución, siendo el valor de su salida siempre cero.

- Leaky RELU. Define un nuevo hiperparámetro alfa que se puede afinar, controlando la cantidad de ajuste que realiza. Generalmente este parámetro se suele establecer a 0.01 que asegura un ajuste. Este valor introduce a veces momentos en el que el aprendizaje parece que no avanza y después de una pocas épocas despierta y sigue adelante. Últimos estudios aseguran que un valor de 0.2 es mejor que 0.01, pero dependerá de nuestro problema.

```
keras.layers.Dense(10, kernel_initializer="he_normal"),
keras.layers.LeakyReLU(alpha=0.2),
```

- ELU. Esta variante parece que es mejor que todas las variantes de RELU, que mejora los tiempos de aprendizaje y se ajusta mejor a los datos de test. También tiene un hiperparámetro alfa que podemos controlar, generalmente se establece a 1. El mayor problema de esta función de activación y su mayor coste computacional, pero a cambio converge antes que las versiones de RELU, pero aun así es mucho más lenta que RELU.
- SELU. Es una variante de ELU escalada. La ventaja con esta función de activación es que la propia red se auto normaliza manteniendo la media en cero y la varianza en uno, solventando los problemas de gradiente. Pero para que esto suceda se deben dar las siguientes condiciones:
 - Las entradas deben estar estandarizadas (media cero y varianza uno).
 - Cada capa oculta debe ser inicializada con LeCun normal.
 - La arquitectura debe ser secuencial.
 - Solo se garantiza la autonormalización si todas las capas son densas.

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Una vez vistas las diferentes funciones de activación se nos presenta la duda de cuál de ellas utilizar en cada problema, y en general depende del mismo, pero podemos conocer la siguiente secuencia de uso: SELU > ELU > Leaky RELU > RELU > Tang > logistic. Pero hoy en día por las aceleraciones hardware que están implementadas se suele utilizar RELU de largo.

Normalización Batch (Batch Normalización)

Esta operación permite lidiar con todos los problemas de las grandes NN de una sola vez, pero no de forma excluyente, se pueden usar las técnicas anteriores. Se puede hacer justo antes de cada capa oculta (aunque hay estudios que defienden que se haga después). Se basa en centrar en cero las entradas y normalizar, escalarlas y desplazarlas utilizando nuevos parámetros por cada capa. Además, tiene un uso más, si se añade como primera capa después de la entrada o del aplanado (Flatter) conseguiremos que se normalicen las entradas y no será necesario que lo hagamos nosotros antes.

Este procedimiento mejora significativamente el rendimiento de las NN, puede mejorar hasta niveles del 2% solo añadiendo las capas sin afinar nada. La desventaja es que añade complejidad al modelo, aumentando el tiempo de procesamiento, pero a cambio disminuye el número de épocas para converger.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
```

```

        keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal"),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(10, activation="softmax")
    )

```

Es importante que la Normalización esté después de la capa Flatter para que afecte a todas las entradas.

Este procedimiento tiene un par de hiperparámetros que podemos ajustar. Generalmente desearemos utilizar el momento. Este valor suele estar muy cercano a uno 0.9 y se añadirán más nueves al valor cuanto más pequeño es el tamaño del mini-batch (bloque de aprendizaje).

Recorte del gradiente

Otra técnica muy popular es recortar el tamaño del gradiente durante la Backpropagation, evitando el problema de la explosión del gradiente. Con esta técnica se limita el valor máximo que puede tener el cambio.

```

optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)

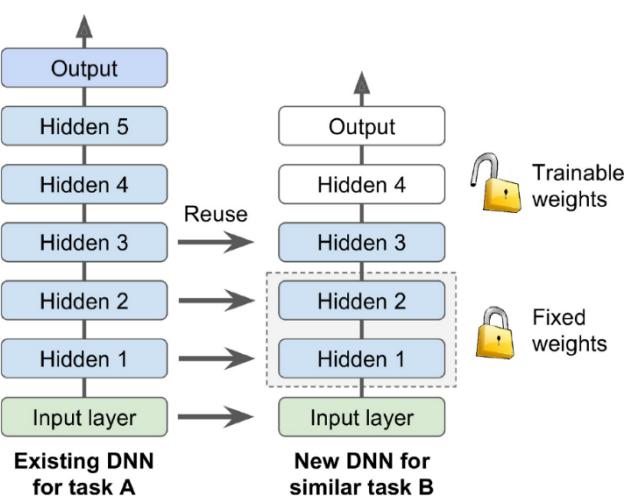
```

El problema es que al realizar el corte se pueden presentar otros problemas, como que no se preserve la orientación o se elimine el valor de una de las direcciones. Así un vector vale (0.9,100) y se aplica el recorte quedará en (0.9,1) cambiando la dirección del vector. Del mismo modo si se hace un corte normalizado el resultado será (0.008, 1) con lo que la primera componente prácticamente se anula.

Uso de capas ya entrenadas

En general no es una buena idea entrenar una red grande si existe otra que haga un trabajo similar. El entrenamiento puede ser una tarea muy costosa y si podemos reutilizar partes de otros modelos ya entrenados y validados se adelantará mucho, esta técnica se llama transferencia del aprendizaje.

La idea principal es transferir las capas más cercanas a la entrada a nuestro modelo manteniendo los pesos. A continuación, añadimos las capas superiores que creamos convenientes y entrenamos el modelo con nuestros datos, forzando al cambio a las capas que hemos añadido.



Este procedimiento es muy válido y se utiliza mucho, lo único que deberemos tener algunas consideraciones:

- Si los modelos no tienen las mismas características de entrada, habrá que realizar un proceso de adaptación de nuestros ejemplos al nuevo número de entradas.
- Generalmente la capa de salida debe ser remplazada para que se adapte al problema que estamos resolviendo.

El mecanismo explicado es simple, pero nos podemos encontrar con que nuestro modelo no se ajusta, con lo que iremos liberando capas fijas de arriba debajo de pocas en pocas y entrenando de nuevo para que mejore el ajuste, cuantos más ejemplos de entrada tengamos más capas podremos liberar. No nos podemos olvidar ajustar la tasa de aprendizaje correctamente, generalmente a un valor bajo.

Tenemos que tener especial cuidado con los estudios que se hacen en Deep Learning, en los que se muestran algoritmos que funcionan muy bien con ciertos parámetros, o ciertos ajustes que son los que hay que usar. Estos estudios muestran los resultados, pero muy pocas veces dicen los intentos que se han tenido que hacer hasta encontrar esos resultados, por lo que un valor presentado parece que es el mejor pero lo que pasa es que los fallos no se muestran, por lo que no se puede evaluar correctamente el funcionamiento.

Para finalizar esta sección, no hay que obviar que esta técnica no funciona bien con pequeñas redes densas, mejora mucho con redes convolucionales profundas.

Si tenemos muchos ejemplos sin etiquetas y es muy costoso, podemos intentar entrenar un autoencoder o una GANs y reusar las capas más bajas del autoencoder o del discriminador de la GAN y añadir una capa de salida, usando los ejemplos que tengamos etiquetados para finalizar el entrenamiento.

Optimizadores rápidos

Entrenar DNN puede ser computacionalmente muy costoso y llevar una gran cantidad de tiempo. Para acelerar el proceso podemos utilizar alguna de las siguientes técnicas:

- Aplicar una buena técnica de inicialización.
- Usar una buena función de activación.
- Usar Normalización batch.
- Reusar capas de modelos ya entrenados.
- Utilizar algoritmos de optimización rápidos. Este es el tema que vamos a tratar en este punto.

Hay varios algoritmos más rápidos que el clásico GD o SGD (con un learning rate óptimo de 0.1).

- NAG (Nesterov accelerated Gradient). Es una pequeña variación del anterior que hace que sea más rápido que el GD o SGD.

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9,
                                 nesterov=True)
```

- AdaGrad. Este algoritmo escala la dirección del vector gradiente a en la dirección más empinada. Funciona muy bien para problemas cuadráticos, pero se para muy pronto para DNN. No se debe utilizar en problemas de Deep Learning ya que presenta el riesgo que pare demasiado pronto en un mínimo local en vez del global.
- RMSProp. Este algoritmo mejora los problemas del anterior introduciendo un nuevo hiperparámetro beta que generalmente se establece a 0.9.

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

- ADAM. Combina las ideas de momento y RMSPROP en un único algoritmo, por lo que incorpora dos nuevos hiperparámetros pero generalmente los valores mostrados dan buenos resultados.

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9,
```

```
beta_2=0.999)
```

- NADAM. Es el algoritmo anterior con una nueva optimización, que generalmente hace que converja algo más rápido

Una vez definido y establecido el optimizador se pasará como parámetro a la función compile.

```
model.compile(loss="mse", optimizer=optimizer)
```

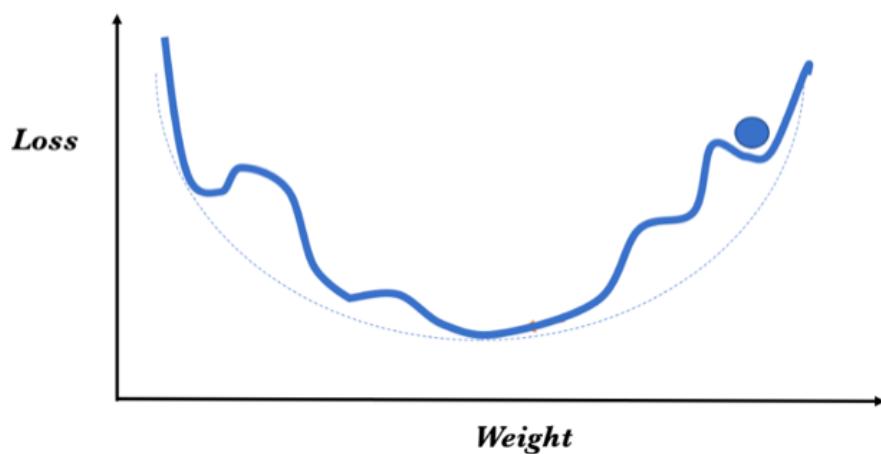
Comparación de los optimizadores

Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
Nadam	***	** or ***
AdaMax	***	** or ***

Momento

Los optimizadores también permiten un parámetro llamado momento. Este parámetro permite tener en cuenta los gradientes anteriores ya calculados para modificar el valor actual de aprendizaje. La idea es que si llevamos mucha inclinación acumulada el salto deberá ser mayor, al igual que cuando dejamos caer una bola por una pendiente.

La pregunta es para qué esta optimización. La respuesta es sencilla, las funciones pueden tener mínimos locales en los que queden estancados los algoritmos, y esta “aceleración” extra permitirá saltarlos.



En la imagen vemos que si el cambio es pequeño se podría quedar atascado en el mínimo local en el que se encuentra, con la aceleración extra, es decir con un cambio mayor por los cambios anteriores saltará el desnivel y podrá seguir bajando.

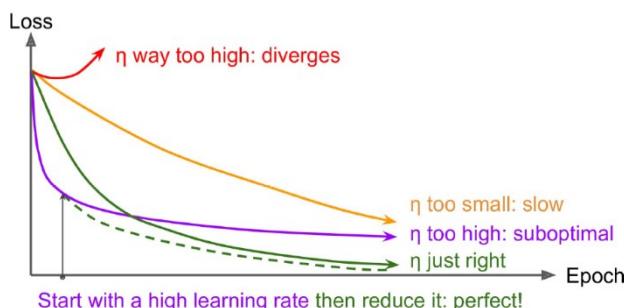
Los valores que admite el parámetro están entre 0 y 1, pero generalmente 0.9 es un buen valor.

```
optimizer = tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD")

model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Gestión de la tasa de aprendizaje

Hasta ahora hemos visto que la tasa de aprendizaje permanece constante entre las diferentes épocas. Vamos a explorar la idea de que ésta varía dependiendo del momento de aprendizaje.



La primera idea que podemos utilizar para encontrar la tasa adecuada de aprendizaje es entrenar nuestro modelo con solo unos cientos de épocas, incrementando de forma exponencial en cada una de ellas el valor de la tasa de aprendizaje. Una vez realizado se muestra la curva de aprendizaje

utilizando como tasa de aprendizaje un poco menor al punto en el que la curva vuelve a crecer. Con esta tasa entrenar de nuevo el modelo completamente.

Una segunda aproximación nos indica que empecemos con una tasa de aprendizaje elevada y la vayamos reduciendo en cada época de forma progresiva. Para esta aproximación se han creado los siguientes algoritmos:

- Power scheduling. Primero avanza muy rápido y después cada vez más lento, tiene un par de hiperparámetros a ajustar.

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

- Exponential scheduling. El descenso se hace de forma exponencial en un factor de diez cada vez.

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
lr_scheduler =
keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...],
callbacks=[lr_scheduler])
```

- Piecewise constant scheduling. Divide el entrenamiento en tres períodos en función de las épocas y utiliza para cada una de ellas una tasa de aprendizaje fija, pero diferente entre ellas.

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
lr_scheduler =
keras.callbacks.LearningRateScheduler(piecewise_constant_fn)
history = model.fit(X_train_scaled, y_train, [...],
```

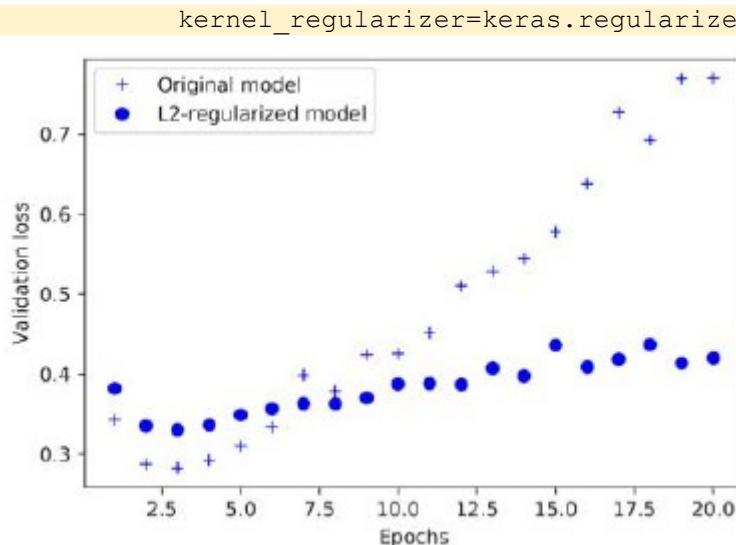
- callbacks=[lr_scheduler])
 - Performance scheduling. Mide el error de validación cada N pasos y reduce la tasa de aprendizaje en un factor (hiperparámetro) cuando el error deja cambiar.
- ```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5,
 patience=5)
```
- One cycle scheduling. Divide el aprendizaje en dos periodos, en el primero la tasa de aprendizaje se incrementará y es segundo descenderá hasta alcanzar la inicial (hará una especie de pico) La elección de los valores inicial y pico se usa la siguiente técnica. Para el pico se utilizará la técnica explicada en el primer párrafo, para el inicial se elegirá uno que sea unas diez veces menor que el pico.
- ```
class OneCycleScheduler(keras.callbacks.Callback):
    def __init__(self, iterations, max_rate, start_rate=None,
                 last_iterations=None, last_rate=None):
        self.iterations = iterations
        self.max_rate = max_rate
        self.start_rate = start_rate or max_rate / 10
        self.last_iterations = last_iterations or iterations
        // 10 + 1
        self.half_iteration = (iterations -
                               self.last_iterations) // 2
        self.last_rate = last_rate or self.start_rate / 1000
        self.iteration = 0
    def _interpolate(self, iter1, iter2, rate1, rate2):
        return ((rate2 - rate1) * (self.iteration - iter1) /
               (iter2 - iter1) + rate1)
    def on_batch_begin(self, batch, logs):
        if self.iteration < self.half_iteration:
            rate = self._interpolate(0, self.half_iteration,
                                    self.start_rate, self.max_rate)
        elif self.iteration < 2 * self.half_iteration:
            rate = self._interpolate(self.half_iteration, 2 *
                                    self.half_iteration,
                                    self.max_rate, self.start_rate)
        else:
            rate = self._interpolate(2 * self.half_iteration,
                                    self.iterations,
                                    self.start_rate, self.last_rate)
            rate = max(rate, self.last_rate)
        self.iteration += 1
        K.set_value(self.model.optimizer.lr, rate)

n_epochs = 25
onecycle = OneCycleScheduler(len(X_train) // batch_size *
n_epochs, max_rate=0.05)
history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
batch_size=batch_size,
validation_data=(X_valid_scaled, y_valid),
callbacks=[onecycle])
```

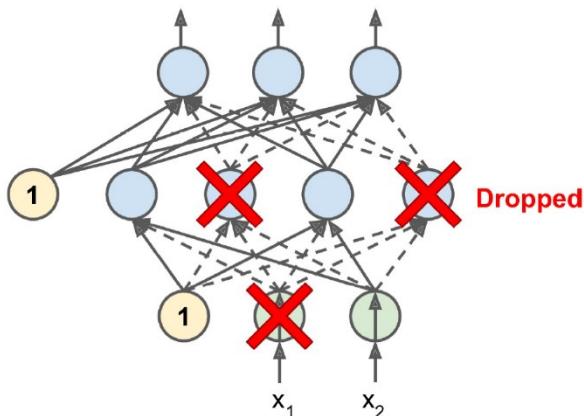
Regularización

Tal y como vimos en los capítulos primeros se pueden usar la regularización L2 (keras.regularizers.l2) y L1 (keras.regularizers.l1) para las NN o ambas a la vez (keras.regularizers.l1_l2). Hay que recordar que la penalización se utilizará solo en la fase de entrenamiento.

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
```



Dropout

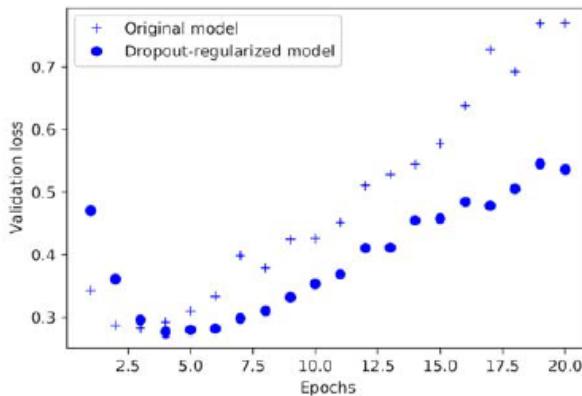


Una de las técnicas más usadas para la regularización es Dropout. Se consigue entre un 1% y 2% de mejora simplemente con añadirla sin ajustarla. La idea es ignorar ciertas neuronas en este ciclo de aprendizaje. El porcentaje de neuronas es un hiperparámetro a ajustar, pero en NN suele estar entre 10% y 50%, más cerca del 20% en redes neuronales recurrentes y cerca del 40% o 50% en las convolucionales.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Si observamos que el modelo se está sobreajustado podemos intentar incrementar el porcentaje de Dropout, en caso de Subajuste se tendrá que bajar o incluso eliminar si así fuera necesario. También es un factor el tamaño de la capa, en capas con muchas neuronas admiten mayor porcentaje de Dropout que las capas con pocas neuronas.

Importante: Si vamos a utilizar la función SELU de activación deberemos utilizar Alpha Dropout en vez de la tradicional ya que preserva la media y la desviación estándar en las entradas.



Regularización Max-Norm

Para cada neurona constriñe el valor de los pesos dentro de un radio (r) de tal manera que, al reducir r , incrementa la cantidad de regularización y mejorar el sobreajuste.

```
keras.layers.Dense(100, activation="elu",
                   kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

Resumen

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping ($+\ell_2$ reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

- Si se necesita un modelo disperso se tiene que utilizar la regularización L1.
- Si necesitamos un modelo de baja latencia (hace predicciones muy rápido) habrá que utilizar pocas capas y añadir Batch Normalization antes de cada capa y posiblemente usar una función de activación muy rápida como Leaky RELU o RELU.
- Si estamos haciendo un modelo de alto riesgo, Dropout mejorará el rendimiento.

Ejercicio

Vamos a realizar una práctica con el Dataset CIFAR10:

- a. Usar DNN con 20 capas ocultas de 100 neuronas, usar inicialización HE_normal y ELU.
- b. Usando NADAM y parada temprana entrenar la red. Tiene 60000 imágenes de 32x32 pixeles a color con 10 clases. Usaremos softmax en la capa de salida con 10 neuronas.
- c. Utilizar Batch Normalización.
- d. Cambiar la Batch Normalización por SELU.
- e. Regularizar el modelo con Dropout
- f. Ver el uso de la gestión 1cycle.

Código 027.py

Ejercicios

Clasificación de setas

<https://www.kaggle.com/uciml/mushroom-classification>

Código 028.py

Clasificación de asteroides

<https://www.kaggle.com/shrutimehta/nasa-asteroids-classification>

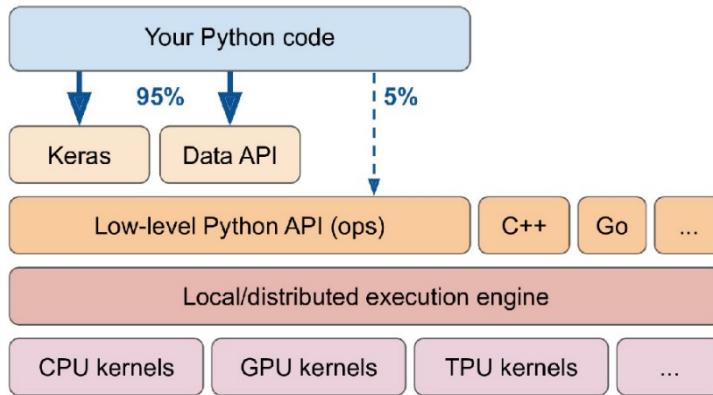
Código 029.py

Kaggle

Kaggle (<https://www.kaggle.com/learn/overview>) es un sitio de internet dedicado al aprendizaje y difusión del machine Learning. El alta es gratuita y podemos encontrar gran cantidad de información y documentación, es muy recomendable estar dado de alta y participar en las competiciones que se existen o que periódicamente se van lanzando.

Cap 14.- Introducción a Tensorflow

Hasta ahora hemos realizado nuestros modelos con el API Keras que está implementado sobre Tensorflow, pero no hemos hablado nada de este API de bajo nivel. A continuación, vemos la estructura de la librería completa.



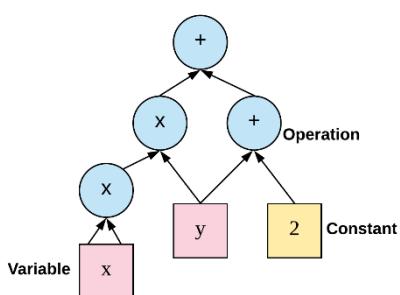
Tensorflow es una librería escalable para la programación de machine Learning desarrollada por Google y liberada en 2015. Con la versión 2 de la librería se está convirtiendo en el estándar para el ML al simplificar mucho su uso. La principal ventaja es que es eficiente sobre cualquier tipo de procesador, incluso con estructuras distribuidas. Esta librería hace un uso eficiente de las GPU de nuestra máquina si ésta tiene núcleos CUDA o TPUs así como clusters o cualquier estructura hardware de computación.

Es imprescindible tener una GPU para ML si queremos realizar algoritmos decentes. Una estimación: en una CPU normal actual puede tardar una semana en compilarse una modelo, con una RTX 2080Ti tardará unas horas.

En este momento las últimas gráficas “asequibles” que NVIDIA ha desarrollado es la serie RTX 30xx.

Otra opción es usar los servicios distribuidos para ML Learning de Google, Keras, Microsoft, etc...

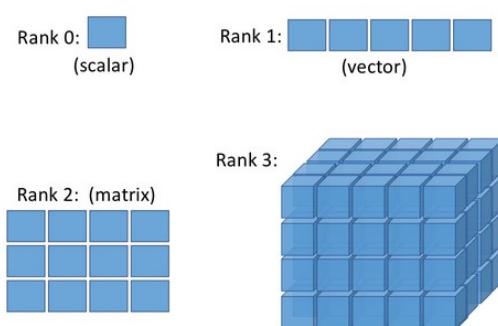
Cómo funciona



Tensorflow usa como base el gráfico computacional. Este gráfico está compuesto por nodos y cada nodo representa una operación que tendrán cero o más entradas y una salida.

En la imagen podemos ver un gráfico de una operación simple (se lee de abajo a arriba: x^2y+y+2).

La librería usa un nuevo tipo de datos para almacenar la información que necesita, el tensor. Se puede entender como una generalización de matriz desde dimensión cero (escalar) hasta la mayor dimensión que necesitemos. Los tensores se usan como entradas y salidas de los gráficos computacionales.



En el ejemplo vemos que un tensor de grado cero es un número, de grado uno es un array, de grado dos en una tabla, etc.

Siempre que usemos Tensorflow, las variables y las constantes deberán ser tensores, por lo que habrá que definirlos con las herramientas del API Tensorflow no del lenguaje que usemos de base (Python en nuestro caso).

Bajo Tensorflow, los tensores se implementan como Arrays NumPy, por lo que son directamente transportables desde esa librería a Tensorflow.

Instalación y configuración

```
pip install tensorflow
```

```
pip install tensorflow-gpu # si tenemos una GPU habilitada
```

En caso que tengamos una tarjeta gráfica con una GPU habilitada para ML podremos hacer uso de todas las aceleraciones existentes, pero para ello deberá ser compatible con NVIDIA y tener instalado el Toolkit CUDA, la librería NVIDIA cuDNN

(<https://docs.nvidia.com/cuda/index.html#installation-guides>)

Manejo inicial de TensorFlow

Código 030.py

```
# Inicio
import tensorflow as tf
import numpy as np

print('TensorFlow version:', tf.__version__) # 2.3.1

a = np.array([1, 2, 3], dtype=np.int32) # Definimos un array numpy
b = [4, 5, 6] # Definimos una lista

# Creamos los tensores correspondientes
t_a = tf.convert_to_tensor(a)
t_b = tf.convert_to_tensor(b)
# Imprimimos los tensores
print(t_a) # tf.Tensor([1 2 3], shape=(3,), dtype=int32)
print(t_b) # tf.Tensor([4 5 6], shape=(3,), dtype=int32)
# Comprobamos si son tensores dos variables
print(tf.is_tensor(a), tf.is_tensor(t_a)) # False True
# Creamos un tensor bidimensional relleno de unos
t_ones = tf.ones((2, 3))
# Vemos la dimensión del tensor
print(t_ones.shape) # (2, 3)
# Accedemos al array de numpy del tensor
print(t_ones.numpy())
"""

[[1. 1. 1.]
 [1. 1. 1.]]

# Creamos un tensor constante, con tres valores de rango uno.
const_tensor = tf.constant([1.2, 5, np.pi], dtype=tf.float32)
print(const_tensor) # tf.Tensor([1.2 5. 3.1415927],
```

```

shape=(3,), dtype=float32)

# Conversiones entre Numpy
print(type(t_ones.numpy())) # <class 'numpy.ndarray'>
print(type(np.array(t_ones))) # <class 'numpy.ndarray'>

# Acceso a los datos indexados
print(t_ones[:, 1:])
"""
tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
"""

print(t_ones[..., 1, tf.newaxis])
"""
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
"""

print(t_a + 10) # tf.Tensor([11 12 13], shape=(3,), dtype=int32)
print(tf.square(t_a)) # tf.Tensor([1 4 9], shape=(3,), dtype=int32)
print(t_ones @ tf.transpose(t_ones)) # @ equivalente a tf.matmul()
"""

tf.Tensor(
[[3. 3.]
 [3. 3.]], shape=(2, 2), dtype=float32)
"""

```

```

# Manejo del tipo de los datos y su dimensión
t_a_new = tf.cast(t_a, tf.int64) # Cambiamos el tipo a int64 del
tensor t_a
print(t_a_new.dtype) # <dtype: 'int64'>
# Redistribuimos los elementos en una matriz de 3 por 5
t = tf.random.uniform(shape=(3, 5))
t_tr = tf.transpose(t) # Hacemos la transpuesta
print(t.shape, ' --> ', t_tr.shape) # (3, 5) --> (5, 3)
t = tf.zeros((30,)) # Creamos un tensor de rango 1 con 30 ceros
t_reshape = tf.reshape(t, shape=(5, 6)) # redistribuimos en una
tensor de rango 2 de 5 por 6
print(t_reshape.shape) # (5, 6)
# Tensor con 5 dimensiones, cada uno con 3,2,1,4,1 ceros
t = tf.zeros((3, 2, 1, 4, 1))
# Eliminamos la tercera y la quinta dimensión, se empieza en cero
t_sqz = tf.squeeze(t, axis=(2, 4))
print(t.shape, ' --> ', t_sqz.shape)
# (3, 2, 1, 4, 1) --> (3, 2, 4)

```

```

# Operaciones matemáticas
tf.random.set_seed(1)
t1 = tf.random.uniform(shape=(5, 2), minval=-1.0, maxval=1.0)
t2 = tf.random.normal(shape=(5, 2), mean=0.0, stddev=1.0)
print(t1, t2)
"""

tf.Tensor(
[[-0.6697383  0.80296254]
 [ 0.26194835 -0.13090777]
 [-0.41612196  0.28500414]
 [ 0.951571   -0.12980103]
 [ 0.32020378  0.20979166]], shape=(5, 2), dtype=float32)

```

```

tf.Tensor(
[[ 0.40308788 -1.0880208 ]
[-0.06309535  1.3365567 ]
[ 0.7117601  -0.48928645]
[-0.7642213  -1.0372486 ]
[-1.2519338   0.02122428]], shape=(5, 2), dtype=float32)
"""
t3 = tf.multiply(t1, t2).numpy()
print(t3)
"""
[[-0.26996338 -0.87363994]
[-0.01652772 -0.17496566]
[-0.296179   -0.13944866]
[-0.7272108   0.13463594]
[-0.40087393  0.00445268]]
"""
t4_a = tf.math.reduce_mean(t1, axis=0) # por columnas axis=0
t4_b = tf.math.reduce_mean(t1, axis=1) # por filas axis=1
print(t4_a, t4_b)
"""
tf.Tensor([0.08957257 0.2074099 ], shape=(2,), dtype=float32)
tf.Tensor([ 0.06661212  0.06552029 -0.06555891  0.41088498
0.26499772], shape=(5,), dtype=float32)
"""
t5 = tf.linalg.matmul(t1, t2, transpose_b=True)
print(t5.numpy())
"""
[[-1.1436033   1.1154624  -0.8695717  -0.3210435   0.8555103 ]
[ 0.2480186  -0.19149339  0.2504958  -0.0644026  -0.3307204 ]
[-0.47782415  0.40717956 -0.43562764  0.02238911  0.5270062 ]
[ 0.52479297 -0.23352614  0.74080014 -0.5925749  -1.1940588 ]
[-0.09918742  0.26019508  0.12526006 -0.46231267 -0.39642125]]
"""
t6 = tf.linalg.matmul(t1, t2, transpose_a=True)
print(t6.numpy())
"""
[[-1.7107549   0.30218127]
[ 0.37133017 -1.0489656 ]]
"""
norm_t1 = tf.norm(t1, ord=2, axis=1).numpy()
print(norm_t1)
"""
[1.045609   0.29283747  0.5043658   0.9603831   0.38280934]
"""
print(np.sqrt(np.sum(np.square(t1), axis=1))) # Por filas axis=1
"""
[1.045609   0.29283747  0.5043658   0.9603831   0.38280934]
"""
print(np.sqrt(np.sum(np.square(t1), axis=0))) # Por columnas axis=0
"""
[1.3032044   0.89664596]
"""

```

Existen muchas más operaciones (tf.add(), tf.multiply(), tf.square(), tf.exp(), tf.sqrt(), etc.) en NumPy (tf.reshape(), tf.squeeze(), tf.tile()). Cuando las funciones tienen nombres diferentes en NumPy y Tensorflow (tf.reduce_mean(), tf.reduce_sum(), tf.reduce_max(), y tf.math.log() que son las equivalentes de np.mean(), np.sum(), np.max() y np.log()) es por una buena razón.

El API de Keras tiene sus propias funciones de bajo nivel den tf.keras y si queremos escribir código portable de Keras deberemos usarlas en vez de las de Tensorflow.

Una variable en Tensorflow es un objeto de Python. Conforme se crean capas, modelos, optimizadores, y otras herramientas relacionada las diferentes clases rastrean dichas variables para tenerlas en cuenta.

```
# Variables y constantes
v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
v.assign(2 * v)
print(v) # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)
print(v) # => [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.])
print(v) # => [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])
print(v) # => [[100., 42., 0.], [8., 10., 200.]]

a = tf.Variable(initial_value=3.14, name='var_a')
b = tf.Variable(initial_value=[1, 2, 3], name='var_b')
c = tf.Variable(initial_value=[True, False], dtype=tf.bool)
d = tf.Variable(initial_value=['abc'], dtype=tf.string)

w = tf.Variable([1, 2, 3], trainable=False)
print(w.assign([3, 1, 4], read_value=True))
w.assign_add([2, -1, 2], read_value=False)

init = tf.keras.initializers.GlorotNormal()
w = tf.Variable(init(shape=(2, 3)))
w = tf.Variable(tf.random.uniform((3, 3)))

ct = tf.constant([1, 2, 3])
print(ct) # tf.Tensor([1 2 3], shape=(3,), dtype=int32)
```

Tensorflow incorpora más estructuras de datos que las estudiadas hasta ahora, es recomendable ver la guía para su compresión.

- Sparse tensors (tf.SparseTensor)
- Tensor arrays (tf.TensorArray)
- Ragged tensors (tf.RaggedTensor)
- String tensors (tf.strings)
- Sets (tf.sets)
- Queues (tf.queue)

```
# Divisiones e uniones
tf.random.set_seed(1)
t = tf.random.uniform((6,))
print(t.numpy())
# [0.16513085 0.9014813 0.6309742 0.4345461 0.29193902 0.64250207]
t_splits = tf.split(t, 3) # Dividimos en bloques de tres
for item in t_splits:
    print(item.numpy())
"""
[0.16513085 0.9014813 ]
[0.6309742 0.4345461]
[0.29193902 0.64250207]
```

```
"""
tf.random.set_seed(1)
t = tf.random.uniform((5,))
print(t.numpy())
# [0.16513085 0.9014813  0.6309742  0.4345461  0.29193902]
# Hacemos dos tensores de 3 y 2.
t_splits = tf.split(t, num_or_size_splits=[3, 2])
for item in t_splits:
    print(item.numpy())
"""
[0.16513085 0.9014813  0.6309742 ]
[0.4345461  0.29193902]
"""
A = tf.ones((3,))
B = tf.zeros((2,))
C = tf.concat([A, B], axis=0)
print(C.numpy()) # [1. 1. 1. 0. 0.]
A = tf.ones((3,))
B = tf.zeros((3,))
S = tf.stack([A, B], axis=0)    # Por columnas axis=0
print(S.numpy())
"""
[[1. 1. 1.]
 [0. 0. 0.]]
"""
S = tf.stack([A, B], axis=1)    # Por filas axis=1
print(S.numpy())
"""
[[1. 0.]
 [1. 0.]
 [1. 0.]]
"""

```

Personalizando Keras con Tensorflow

Tanto las funciones de activación como las funciones de pérdidas están implementadas bajo Keras y ya hemos visto su uso a la hora de la compilación del modelo. Pero qué pasa si queremos hacer nuestras propias funciones, en este caso tendremos que acudir al API de Tensorflow para crearlas y posteriormente utilizarlas dentro de la compilación.

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)

model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

Para la personalización de cualquier función, simplemente creamos nuestra función y la pasamos como parámetro a la hora de compilar el modelo (**loss**, optimizar). Pero hemos de ser muy cuidadosos, dentro de la función utilizaremos solo llamadas a Tensorflow y operaciones vectoriales en vez de hacerlas nosotros, tal y como vemos en el ejemplo (**error = y_true - y_pred**, esta operación resta toda la matriz **y_true** de la **y_pred** en una sola operación).

```
def my_softplus(z):
    return tf.math.log(tf.exp(z) + 1.0)
```

```

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)
def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
def my_positive_weights(weights):
    tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)

layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)

```

Personalización de la métrica

Se puede personalizar la métrica de la misma manera que se ha hecho la función de pérdidas, pero en este punto queremos ir un paso más allá y crear una clase personalizada.

```

def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss =
            threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss,
                       linear_loss)
    return huber_fn

class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total",
                                     initializer="zeros")
        self.count = self.add_weight("count",
                                    initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(
            tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}

model.compile(loss=HuberLoss(2.), optimizer="nadam")

```

Personalización de capas

Se pueden crear capas personalizadas para usar como cualquier otra ya existente. Es recomendable que se creen los atributos de la capa en el método **build()** en vez del **init()** tal y

como vemos en el ejemplo. El método **get_config()** se tiene que implementar para que se pueda serializar la capa y almacenar mediante **save()** de forma correcta. El método **call()** será llamado para obtener el resultado de la activación de la capa.

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel",
            shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units],
            initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1]
+ [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
"activation":
            keras.activations.serialize(self.activation)}
```

Para crear una capa con múltiples entradas y con varias salidas implementaremos la clase según el siguiente ejemplo. Vemos como las entradas de **call** y **compute_output_shape** deben ser una tupla con todas las entradas, y en el método **compute_output_shape** se devolverá también una tupla con todas las salidas.

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1]
```

Para finalizar, recordar que las capas son objetos como cualquier otro y exponen dos propiedades muy interesantes: **input** y **output** que dan acceso a las entradas de la capa y a las salidas de la misma.

Algunas capas, en particular la capa BatchNormalization y la capa Dropout , tienen comportamientos diferentes durante el entrenamiento y la inferencia. Para tales capas, es una práctica estándar exponer un argumento de training (booleano) en el método **call()**.

Al exponer este argumento en `call()`, habilita los bucles de entrenamiento y evaluación integrados (por ejemplo, `fit()`) para usar correctamente la capa en el entrenamiento y la inferencia.

```
class CustomDropout(keras.layers.Layer):
    def __init__(self, rate, **kwargs):
        super(CustomDropout, self).__init__(**kwargs)
        self.rate = rate

    def call(self, inputs, training=None):
        if training:
            return tf.nn.dropout(inputs, rate=self.rate)
        return inputs
```

Creación de bloques de capas

Podemos crear capas que contengan otras capas y actúen como bloques para crear estructuras complejas o repetitivas tal y como vemos en el ejemplo anterior.

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [
            keras.layers.Dense(
                n_neurons,
                activation="elu",
                kernel_initializer="he_normal")
            for _ in range(n_layers)]
    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

Otro ejemplo.

```
class MLPBlock(keras.layers.Layer):
    def __init__(self):
        super(MLPBlock, self).__init__()
        self.linear_1 = Linear(32)
        self.linear_2 = Linear(32)
        self.linear_3 = Linear(1)

    def call(self, inputs):
        x = self.linear_1(inputs)
        x = tf.nn.relu(x)
        x = self.linear_2(x)
        x = tf.nn.relu(x)
        return self.linear_3(x)
```

Un ejemplo completo: XOR

Código 031.py

Manejo de datos con Tensorflow

Generalmente cuando trabajamos con Deep Learning el aprendizaje se hace de forma incremental por la gran cantidad de datos (no entran todos en memoria) con un algoritmo similar al SGD. Solo en aquellos casos en el que el conjunto de datos es lo suficientemente pequeño y se puede acomodar en la memoria como un tensor podremos usar el 'método **fit** de forma eficiente, para el resto de casos deberemos de hacer bloques de tratamiento (**batch**) y realizar todas las operaciones (regularizar, codificar, etc.) a cada uno de ellos. El repetir estas tareas es tedioso, por lo que Tensorflow proporciona un mecanismo para realizar un conjunto de operaciones que se realizará en cadena, las tuberías o pipelines.

La estructura de datos que usar Tensorflow para la gestión de conjuntos de datos se denomina Dataset. Es similar a la estructura existente en los lenguajes de programación para acceder a las bases de datos (recordset). Podemos crear un Dataset desde un array NumPy, una lista mediante la orden **tf.data.Dataset.from_tensor_slices()** y hacer bloques mediante **batch**.

```
a = [1.2, 3.4, 7.5, 4.1, 5.0, 1.0]
ds = tf.data.Dataset.from_tensor_slices(a)
for item in ds:
    print(item)
ds_batch = ds.batch(3)
for i, elem in enumerate(ds_batch, 1):
    print('batch {}:{}.'.format(i), elem.numpy())
```

Generalmente los datos están repartidos en varios tensores, y debemos unirlos antes de poder usarlos. Podemos acceder mediante las funciones **zip**, la función **from_tensor_slices** o incluso realizar algún cálculo sobre los datos de entrada.

```
tf.random.set_seed(1)

t_x = tf.random.uniform([4, 3], dtype=tf.float32)
t_y = tf.range(4)

ds_x = tf.data.Dataset.from_tensor_slices(t_x)
ds_y = tf.data.Dataset.from_tensor_slices(t_y)

## zip:
ds_joint = tf.data.Dataset.zip((ds_x, ds_y))

for example in ds_joint:
    print(' x: ', example[0].numpy(),
          ' y: ', example[1].numpy())

## from_tensor_slices:
ds_joint = tf.data.Dataset.from_tensor_slices((t_x, t_y))

for example in ds_joint:
    print(' x: ', example[0].numpy(),
          ' y: ', example[1].numpy())

## realizando cálculos sobre un tensor unido
ds_trans = ds_joint.map(lambda x, y: (x*2-1.0, y))

for example in ds_trans:
    print(' x: ', example[0].numpy(),
          ' y: ', example[1].numpy())

ds_filter = ds_joint.filter(lambda x: x < 10)
```

```
for item in Dataset.take(3):
    print(item)
```

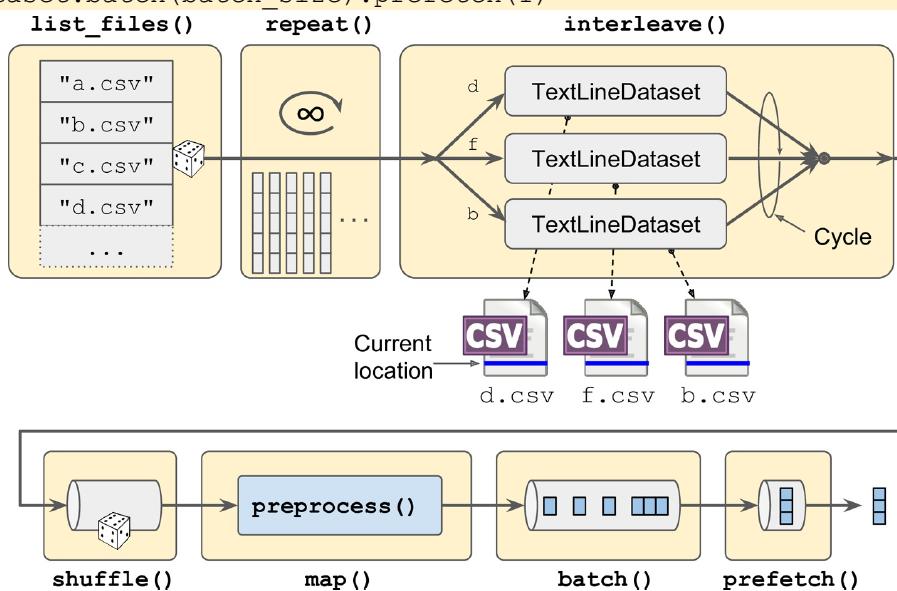
Es muy importante ser muy cuidadosos con la gestión de los datos. Por ejemplo, si mezclamos los datos de los tensores antes de unirlos perderemos la relación de orden entre ambos tensores, pero si primero los unimos en un único tensor y a continuación mezclamos, el orden se mantendrá. Ponemos este ejemplo en conocimiento ya que para el Deep Machine Learning es muy importante que los ejemplos estén bien mezclados para que los algoritmos converjan y sean eficientes una vez entrenados. Así si al mezclar tenemos que tener muy en cuenta el número de ejemplos que tenemos, ya que si es muy pequeño y hacemos bloques de tratamiento también pequeños el efecto sobre el aprendizaje será muy negativo, pero si el conjunto de datos es grande no importará el tamaño de bloque que usemos.

```
ds = ds_joint.shuffle(buffer_size=len(t_x))
```

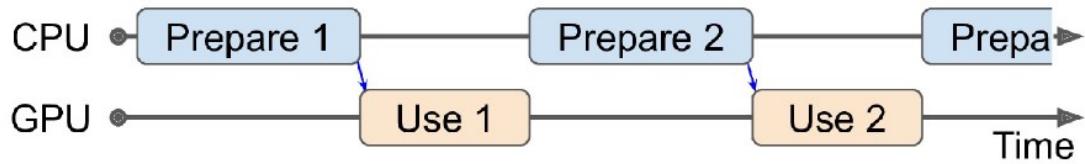
En resumen, tendremos en cuenta que el orden a realizar en una tubería debería ser primero mezclar, luego dividir y por último repetir el proceso.

Acceso a varios ficheros

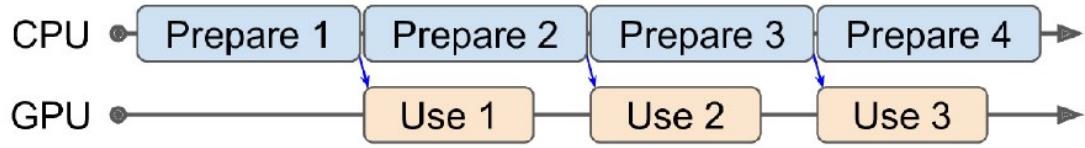
```
def preprocess():
    ...
    filepath = "Datasets/housing/my_train_*.csv"
    repeat=1
    n_readers=5,
    n_read_threads=None
    shuffle_buffer_size=10000
    n_parse_threads=5
    batch_size=32
    Dataset = Dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    Dataset = Dataset.map(preprocess,
        num_parallel_calls=n_parse_threads)
    Dataset = Dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    Dataset.batch(batch_size).prefetch(1)
```



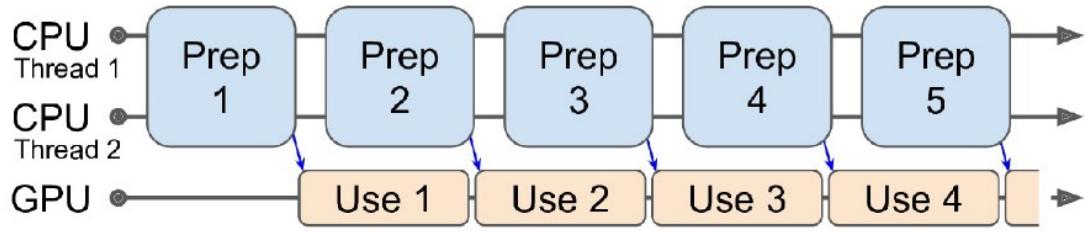
Without prefetching



With prefetching



With prefetching + multithreaded loading & preprocessing



Datasets existentes en Tensorflow

Tensorflow incorpora Dataset para probar nuestros algoritmos de forma eficiente sin que tengamos que descargarlos de ningún sitio. Cada Dataset tiene sus conjuntos de datos y características, así como están divididos en uno o varios conjuntos para test, validación y aprendizaje, depende del conjunto que usemos deberemos explorarlo.

Para usarlos tenemos que instalarla el paquete **tensorflow-datasets** a través de la herramienta pip. Podemos ver el conjunto instalado mediante la siguiente instrucción.

```
import tensorflow_datasets as tfds
print((tfds.list_builders()))
```

Para poder utilizar un Dataset primero hay que descargarlo a la máquina y se hará solo la primera vez que llamemos a la función de acceso.

```
celeba_bldr = tfds.builder('celeb_a')
```

Podremos inspeccionar el Dataset a través de las siguientes instrucciones.

```
print(celeba_bldr.info.features)
print(celeba_bldr.info.features.keys())
print(celeba_bldr.info.features['image'])
print(celeba_bldr.info.features['attributes'].keys())
print(celeba_bldr.info.citation)

celeba_bldr.download_and_prepare() # Descargar y prepara los datos
```

A continuación, vemos los conjuntos de ejemplos que incorpora.

```
Datasets = celeba_bldr.as_Dataset(shuffle_files=False)
print(Datasets.keys())# dict_keys(['test', 'train', 'validation'])
ds_train = Datasets['train']
```

Podemos realizar todo el trabajo anterior en una única instrucción.

```
mnist, mnist_info = tfds.load('mnist', with_info=True,
                               shuffle_files=False)

print(mnist_info)
print(mnist.keys())
```

Funciones con Tensorflow

Podemos crear funciones para usar con Tensorflow de forma muy fácil usando el decorador **@tf.function** y el cuerpo como una función de Python normal. Para un correcto funcionamiento el contenido de la función debería usar solo funciones de Tensorflow y operaciones matriciales. No hemos entrado en las diferencias entre TF1 y TF2 en cuanto a los gráficos estáticos y dinámicos, pero solo tenemos que saber que los estáticos de TF1 son mucho más eficientes que los nuevos de TF2, por lo que se puede convertir el gráfico TF2 en TF1 a través de la herramienta AutoGraph que hace uso del decorador construyendo de forma automática el gráfico estático de la función.

```
@tf.function
def compute_z(a, b, c):
    r1 = tf.subtract(a, b)
    r2 = tf.multiply(2, r1)
    z = tf.add(r2, c)
    return z

tf.print('Scalar Inputs:', compute_z(1, 2, 3))
tf.print('Rank 1 Inputs:', compute_z([1], [2], [3]))
tf.print('Rank 2 Inputs:', compute_z([[1]], [[2]], [[3]]))
```

Cada vez que se llame a la función con el decorador, se comprobará si es la primera vez o no. En caso de ser la primera vez se creará el gráfico estático y se usará, si no lo es se reutilizará el ya creado. Cada vez que llamemos a la función con diferentes tipos de valores se creará un gráfico nuevo, pero si queremos limitar el tipo de datos a usar con nuestra función se utilizará la siguiente nomenclatura.

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None],
                                             dtype=tf.int32),
                           tf.TensorSpec(shape=[None],
                                             dtype=tf.int32),
                           tf.TensorSpec(shape=[None],
                                             dtype=tf.int32)))
def compute_z(a, b, c):
    r1 = tf.subtract(a, b)
    r2 = tf.multiply(2, r1)
    z = tf.add(r2, c)
    return z

tf.print('Rank 1 Inputs:', compute_z([1], [2], [3]))
tf.print('Rank 1 Inputs:', compute_z([1, 2], [2, 4], [3, 6]))
```

```
# Error
tf.print('Rank 2 Inputs:',
         compute_z([[1], [2]], [[2], [4]], [[3], [6]]))
```

Código 032.py

Hay que tener algunas normas en cuanto a las funciones:

- Un gráfico solo puede tener llamadas a otras funciones de Tensorflow.
- Se pueden llamar a otras funciones Python si solo tienen código Tensorflow.
- Si se crean variables deben ser las primeras líneas, o mejor hacerlas fuera de la función.
- Se debe proporcionar el código fuente de forma externa para que funcione, no se puede dar código Python compilado.
- Los bucles for de Python solo se capturarán si son sobre Dataset o Tensores (usar tf.range en vez de range).
- Usar versiones vectorizadas de las operaciones mejores que los bucles al mejorar la eficiencia.

Cálculo de gradientes bajo Tensorflow

Para entrenar una red neuronal hay que calcular el gradiente de los pesos para poder actualizarlos correctamente mediante el algoritmo de Backpropagation, permitiendo que converjan en una solución. Generalmente este cálculo se hace de forma automática a través del método **fit()** de la capa Keras, pero podemos personalizarlo nosotros.

Tensorflow proporciona una implementación de la derivación a través del método de cadena, y nos proporciona un mecanismo para acceder a los valores de dichas derivadas intermedias. Para poder hacer uso de estos valores (para poder ver qué está funcionando mal) se deben grabar a través de **tf.GradientTape**.

```
import tensorflow as tf

w = tf.Variable(1.0)
b = tf.Variable(0.5)
print(w.trainable, b.trainable)

x = tf.convert_to_tensor([1.4])
y = tf.convert_to_tensor([2.1])

with tf.GradientTape() as tape:
    z = tf.add(tf.multiply(w, x), b)
    loss = tf.reduce_sum(tf.square(y - z))

dloss_dw = tape.gradient(loss, w) # Gradiente usado
tf.print('dL/dw : ', dloss_dw)
```

Una vez que hemos hecho uso del gradiente éste se borra y no vuelve a estar accesible a menos que lo hagamos persistente pasando el parámetro **persistent=True** al constructor de **GradientTape()** pero lo utilizaremos exclusivamente cuando tengamos que almacenar más de un gradiente ya que consume muchos recursos. También tendremos que modificar el procedimiento anterior si las variables no son modificables o estamos usando otros objetos

Tensor, en concreto tendremos que añadir `tape.watch(var)` para que se guarde el valor de dicha derivada.

```
with tf.GradientTape() as tape:
    tape.watch(x)
    z = tf.add(tf.multiply(w, x), b)
    loss = tf.square(y - z)

dloss_dx = tape.gradient(loss, x)
tf.print('dL/dx:', dloss_dx)
```

Para terminar, podremos aplicar el gradiente a los pesos junto con la función de optimización a través del siguiente código.

```
optimizer = tf.keras.optimizers.SGD()

optimizer.apply_gradients(zip([dloss_dw, dloss_db], [w, b]))

tf.print('Updated w:', w)
tf.print('Updated bias:', b)
```

Un ejemplo de un ciclo de entrenamiento completo

```
class CustomModel(keras.Model):
    def train_step(self, data):
        # Unpack the data. Its structure depends on your model and
        # on what you pass to `fit()`.

        x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)    # Forward pass
            # Compute the loss value
            # (the loss function is configured in `compile()`)
            loss = self.compiled_loss(y, y_pred,
                                      regularization_losses=self.losses)

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        # Update weights
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        # Update metrics (includes the metric that tracks the loss)
        self.compiled_metrics.update_state(y, y_pred)
        # Return a dict mapping metric names to current value
        return {m.name: m.result() for m in self.metrics}
```

Personalizando la evaluación del modelo

Podemos personalizar también el ciclo de evaluación del modelo definiendo un modelo propio y el método correspondiente como vemos a continuación.

```
class CustomModel(keras.Model):
    def test_step(self, data):
        # Unpack the data
        x, y = data
        # Compute predictions
        y_pred = self(x, training=False)
        # Updates the metrics tracking the loss
```

```

        self.compiled_loss(y, y_pred,
                            regularization_losses=self.losses)
    # Update the metrics.
    self.compiled_metrics.update_state(y, y_pred)
    # Return a dict mapping metric names to current value.
    # Note that it will include the loss (tracked in
    #   self.metrics).
    return {m.name: m.result() for m in self.metrics}

# Construct an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(loss="mse", metrics=["mae"])

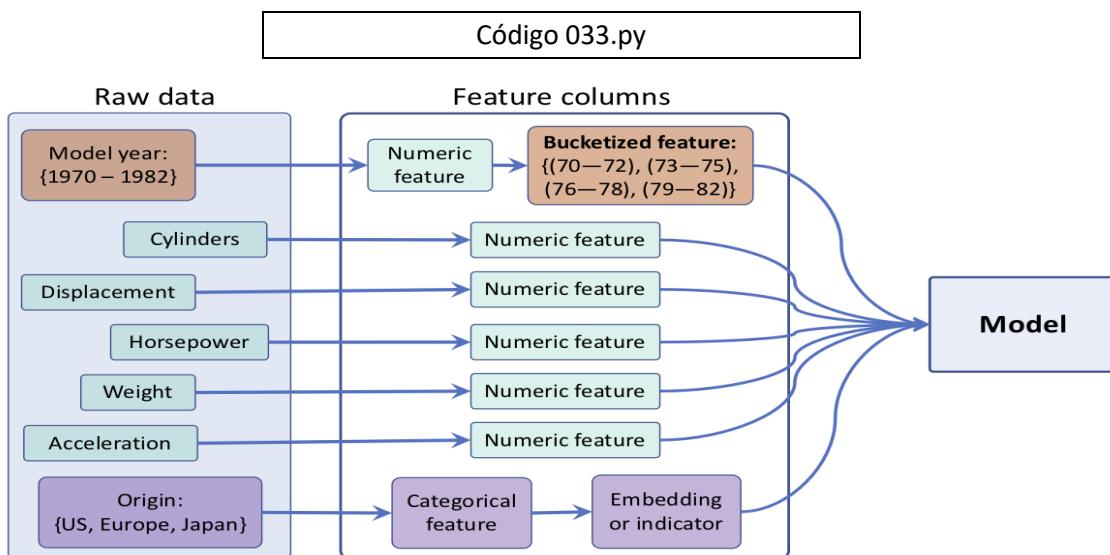
# Evaluate with our custom test_step
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
model.evaluate(x, y)

```

Estimadores (tf.estimator)

Este API encapsula todos los pasos necesarios vistos hasta ahora en un único lugar para facilitar su puesta en producción y su transportabilidad. Los estimadores añaden la capacidad de ejecutarlos en cualquier arquitectura hardware sin tener que cambiar nada en el código fuente.

Las redes neuronales trabajan con diferentes características de entrada, al igual que los estimadores, pero debemos prepararlas para poder usarlas con ellas. Los estimadores están preparados para trabajar con cualquier tipo de característica, pero antes debemos indicárselo, no es capaz de inferir el tipo del contenido de la misma.



En el siguiente código podemos apreciar cómo podemos preparar las características para que se puedan usar con estimadores, tanto numéricas, como agregadas, etc.

```

numeric_features = []
for col_name in numeric_column_names:
    numeric_features.append(
        tf.feature_column.numeric_column(key=col_name))

```

```

feature_year = tf.feature_column.numeric_column(key="ModelYear")
bucketized_features = []
bucketized_features.append(tf.feature_column.bucketized_column(
    source_column=feature_year,
    boundaries=[73, 76, 79]))

feature_origin =
    tf.feature_column.categorical_column_with_vocabulary_list(
        key='Origin',
        vocabulary_list=[1, 2, 3])
categorical_indicator_features = []
categorical_indicator_features.append(
    tf.feature_column.indicator_column(feature_origin))

```

A continuación, crearemos una función para la carga de los datos de entrenamiento y test.

```

def train_input_fn(df_train, batch_size=8):
    df = df_train.copy()
    train_x, train_y = df, df.pop('MPG')
    Dataset = tf.data.Dataset.from_tensor_slices((dict(train_x),
                                                train_y))

    # shuffle, repeat, and batch the examples
    return Dataset.shuffle(1000).repeat().batch(batch_size)

```

Definiremos el conjunto de características

```

all_feature_columns = (numeric_features +
                      bucketized_features +
                      categorical_indicator_features)

```

El estimador, indicando el directorio en donde se guardará automáticamente el modelo.

```

regressor = tf.estimator.DNNRegressor(
    feature_columns=all_feature_columns,
    hidden_units=[32, 10],
    model_dir='models/automp-g-dnnregressor/')

```

El proceso de aprendizaje

```

EPOCHS = 1000
BATCH_SIZE = 8
total_steps = EPOCHS * int(np.ceil(len(df_train) / BATCH_SIZE))
print('Training Steps:', total_steps)

regressor.train(
    input_fn=lambda:train_input_fn(df_train_norm, # Normalizado
                                   batch_size=BATCH_SIZE),
    steps=total_steps)

```

Recargamos el último modelo, ya que los estimadores guardan de forma automática el último modelo.

```
reloaded_regressor = tf.estimator.DNNRegressor()
```

```
feature_columns=all_feature_columns,
hidden_units=[32, 10],
warm_start_from='models/autompd-dnnregressor/',
model_dir='models/autompd-dnnregressor/')
```

Evaluamos el rendimiento.

```
def eval_input_fn(df_test, batch_size=8):
    df = df_test.copy()
    test_x, test_y = df, df.pop('MPG')
    Dataset = tf.data.Dataset.from_tensor_slices((dict(test_x),
                                                test_y))

    return Dataset.batch(batch_size)

eval_results = reloaded_regressor.evaluate(
    input_fn=lambda:eval_input_fn(df_test_norm, batch_size=8))

for key in eval_results:
    print('{:15s} {}'.format(key, eval_results[key]))

print('Average-Loss {:.4f}'.format(eval_results['average_loss']))
```

<https://www.tensorflow.org/guide/estimator>

El ejemplo anterior utiliza estimadores ya existentes para realizar el trabajo, pero cuando creamos nuestro propio modelo, también podemos convertirlo en un estimador y exportarlo.

```
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(2,), name='input-features'),
    tf.keras.layers.Dense(units=4, activation='relu'),
    tf.keras.layers.Dense(units=4, activation='relu'),
    tf.keras.layers.Dense(units=4, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.summary()

model.compile(optimizer=tf.keras.optimizers.SGD(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy()])

my_estimator = tf.keras.estimator.model_to_estimator(
    keras_model=model,
    model_dir='models/estimator-for-XOR/')
```

El código anterior solo crea el estimador, esto no evita que creamos nuestras funciones de entrada de datos para los conjuntos de entrada y test para posteriormente entregar y evaluar el estimador tal y como se ha hecho en los ejemplos anteriores.

Ejemplo completo con ML con Tensorflow

Código 034.py

Código 035.py

Procesamiento con hardware avanzado con Tensorflow

Uso de GPU

En caso que tengamos una tarjeta gráfica con una GPU habilitada para ML podremos hacer uso de todas las aceleraciones existentes, pero para ello deberá ser compatible con NVIDIA y tener instalado el Toolkit CUDA, la librería NVIDIA cuDNN

<https://docs.nvidia.com/cuda/index.html#installation-guides>

Para saber a qué dispositivos están asignados sus operaciones y tensores, hay que colocar `tf.debugging.set_log_device_placement(True)` como la primera declaración de su programa. Al habilitar el registro de ubicación de dispositivos, se imprimen las asignaciones u operaciones de Tensor.

<https://www.tensorflow.org/guide/gpu>

Uso de TPUs

Las TPU son unidades hardware desarrolladas específicamente para el entrenamiento de redes neuronales por Google. El soporte experimental para Cloud TPU actualmente está disponible para Keras y Google Colab.

<https://www.tensorflow.org/guide/tpu>

Uso de la nube

Tensorflow Cloud es un paquete de Python que proporciona API para una transición perfecta de la depuración local al entrenamiento distribuido en Google Cloud. Simplifica el proceso de entrenamiento de modelos de Tensorflow en la nube en una única llamada, que requiere una configuración mínima y sin cambios en su modelo. Tensorflow Cloud maneja tareas específicas de la nube, como crear instancias de VM y estrategias de distribución para sus modelos automáticamente.

https://www.tensorflow.org/guide/keras/training_keras_models_on_cloud

https://github.com/tensorflow/cloud/blob/master/src/python/tensorflow_cloud/core/tests/examples/dogs_classification.ipynb

https://www.tensorflow.org/guide/distributed_training

Existen otros mecanismos en red para implantar machine Learning.

<https://docs.microsoft.com/es-es/azure/machine-Learning/>

O para aprender como ya vimos.

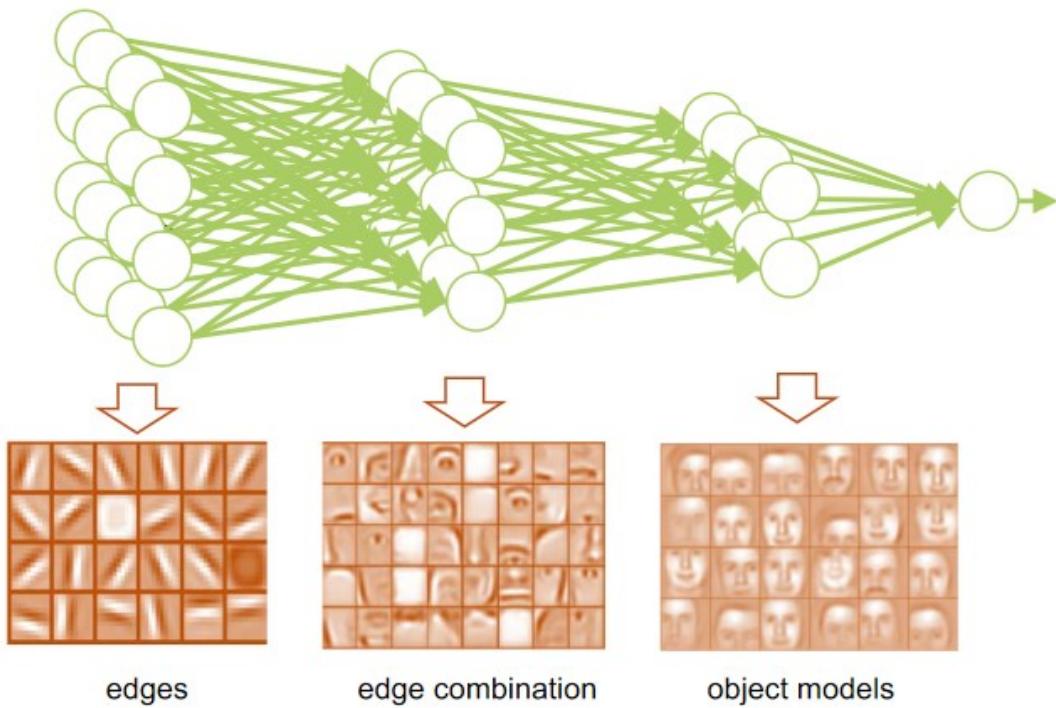
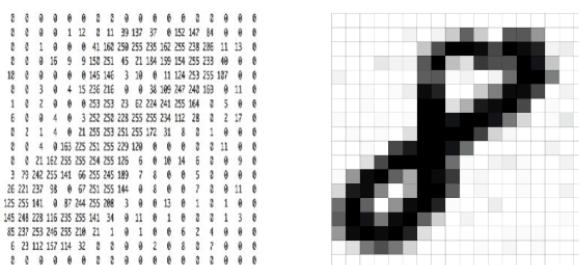
<https://www.kaggle.com/>

Cap 15.-Tratamiento de imágenes: Redes Convolucionales (CNN)

Las CNN son muy populares al poder conseguir resultados muy interesantes en el reconocimiento de imágenes, con lo que ha permitido grandes avances en áreas como conducción automática, creación de contenidos visuales, etc. Las redes CNN son capaces de aprender características de una imagen por sí solas a partir de los datos. las capas convolucionales reconocen patrones locales en pequeñas partes de la imagen (en la imagen en una zona de 5x5) en vez de aprenderlos en toda la imagen.

Estas redes hacen una suposición muy importante, es que la entrada será una imagen, con lo que la dimensión de los datos será 3D: alto, ancho y profundidad (o canales de color).

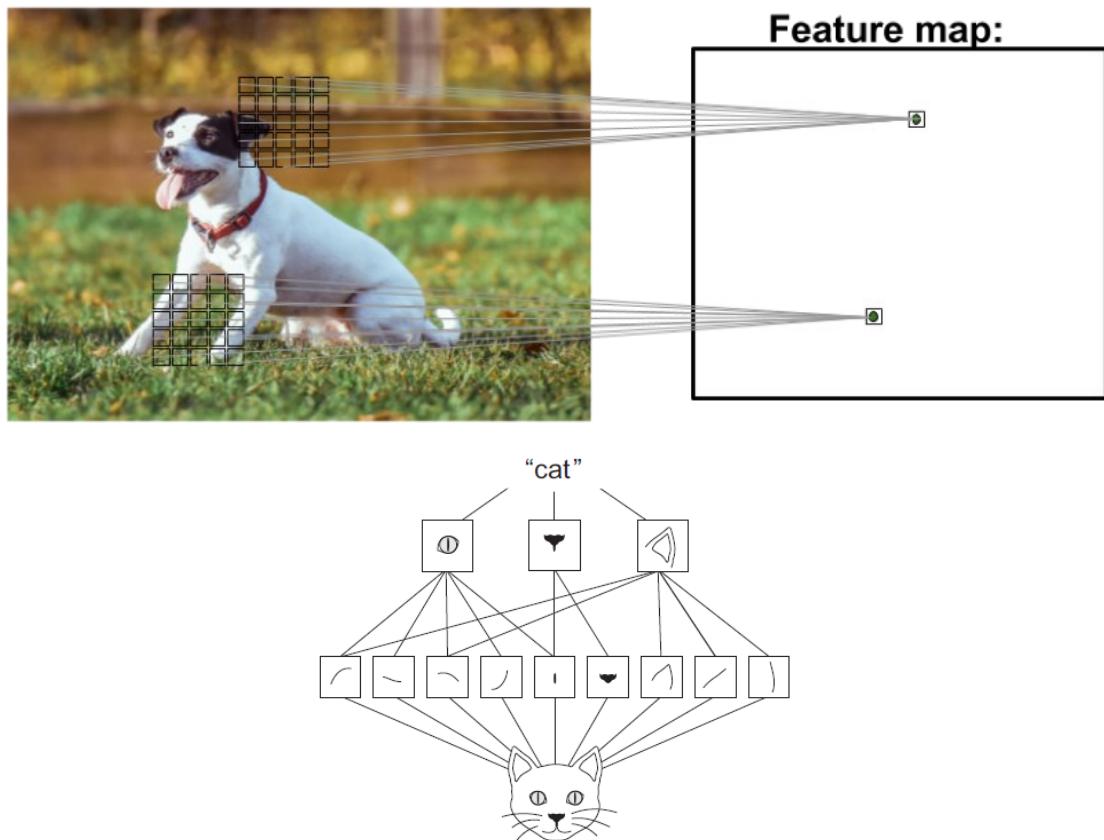
En una red neuronal, cada capa abstrae o reconoce un nivel diferente, en la primera parecerán bordes, en la segunda formas simples, en los siguientes patrones más complejos



Las CNN realizan su trabajo muy bien por dos ideas fundamentales de las mismas:

- Escasa conectividad. Un mapa de características está conectado un pequeño grupo de píxeles. Además, estos patrones son invariantes a las traslaciones.
 - Compartición de parámetros. Los mismos pesos se comparten entre diferentes grupos de píxeles.

Y estas ideas funcionan porque generalmente en una imagen los píxeles cercanos tendrán información relacionada entre ellos en vez de aquellos que están más alejados. Además, las imágenes se forman por jerarquías de patrones, que se van reconociendo según vamos avanzando por las capas convolucionales.



La primera capa reconocerá bordes, la segunda formas como ojo, boca, oreja, etc. y la última decidirá que es un gato, en el ejemplo anterior.

Generalmente las CNN están compuestas por varias capas convolucionales y capas de submuestreo, seguidas por una o más capas densas al final. Las capas de submuestreo (pooling) no tienen parámetros a entrenar, ni tienen pesos ni valor de varianza, mientras que las capas convolucionales sí.

Un ejemplo básico

```
from keras import layers
from keras import models
from keras.datasets import mnist
from keras.utils import to_categorical
# LAS NUEVAS CAPAS
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
print(model.summary())
# CAPAS YA CONOCIDAS
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
print(model.summary())

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
```

```

print(train_images.shape) # (60000, 28, 28) solo un valor, monocromo
# convertir a la convención example,height,width,channel
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255 # normalizar
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
# necesitamos que los labels con one-hot encode
print(train_labels[0]) # 5
train_labels = to_categorical(train_labels)
print(train_labels[0]) # [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5, batch_size=64)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(test_acc) # 0.9908000000000001

```

Código 035_b.py

- Las capa inicial convolucional necesita de entrada (height, width, channels) en el ejemplo anterior: **input_shape=(28, 28, 1)** Feature map.
- La salida de cada capa convolucional y MaxPooling es un tensor de dimensión 3D (height, width, channels_filters) → (conv2d (**Conv2D**) **(None, 26, 26, 32)**). El número de canales o filtros.
- El número de canales de salida se controla con el primer parámetro de las capas (32, o 64)[layers.Conv2D(32,..]
- La última capa convolucional sirve de entrada a una capa densa con el mismo número de entradas, por lo que hay que aplanarla antes con flatten.
- El tamaño de entrada se reduce por el problema de padding, si queremos el mismo debemos ajustarlo en la creación de la capa.
- La capa de salida se define con la función Softmax y con el número de clases a predecir.

La operación de Convolución.

Cada capa de convolución aprende patrones locales dentro de la imagen en pequeñas ventanas de dos dimensiones, no en toda la imagen. El propósito principal de una capa es detectar rasgos visuales. Una vez aprenda una característica será capaz de reconocerla en cualquier parte de la imagen.

Además de patrones, una capa es capaz de reconocer jerarquías de características utilizando lo aprendido por las capas superiores. Estas jerarquías serán objetos, disposiciones de objetos, etc. Por lo que las CNN son capaces de aprender patrones visuales complejos y de forma eficiente.

La operación de convolución opera sobre tensores en 3D exclusivamente en los datos de entrada, con lo que será imprescindible esta transformación antes de introducir los datos.

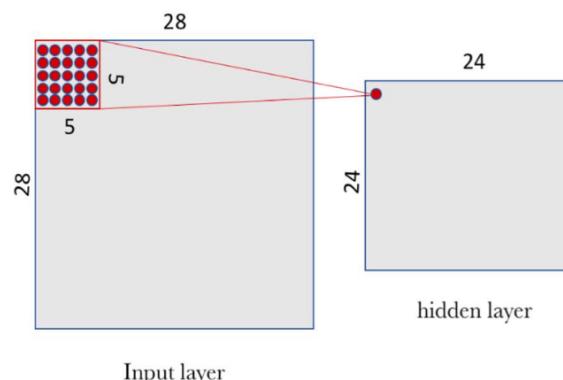
```
train_images = train_images.reshape((60000, 28, 28, 1))
```

Los tensores expresan alto, ancho y número de canales de color (1 para monocromo y tres o cuatro para color). Como vemos a la red se le proporciona una dimensión más con cada uno de los ejemplos. En este caso hay 60000 ejemplos de tensores 3D de cada una de las imágenes. Por otro lado, las etiquetas multiclase deben estar codificadas con el mecanismo OneHot (una etiqueta por columna)

```
train_labels = to_categorical(train_labels)
```

La capa CNN es muy diferente a la densa estudiada anteriormente, ya que esta conecta cada neurona con una parte muy pequeña de la imagen (3×3 o 5×5) en vez de toda la imagen.

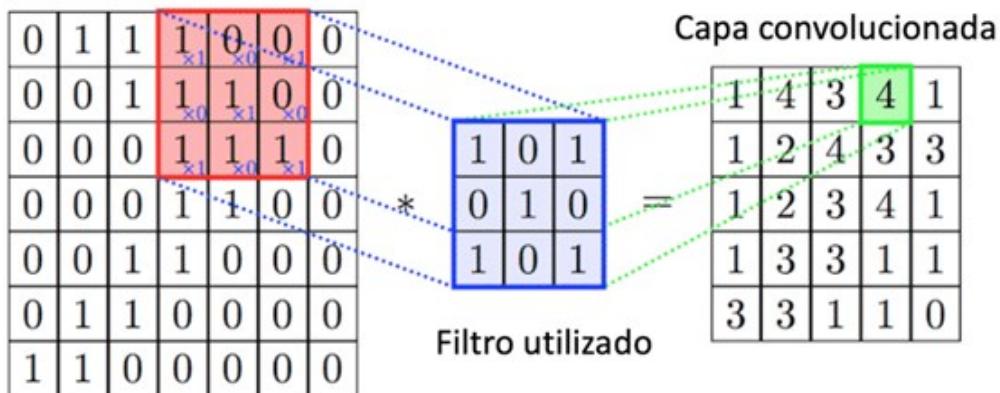
Como vemos a la derecha, la matriz de 5×5 se conectarán con la primera neurona de la siguiente capa, si desplazamos esta matriz una posición, se conectarán con la segunda neurona, así hasta terminar la fila. Una vez finalizada se desplaza la matriz una fila y a la primera columna y empezamos para la segunda fila, de esta manera se recorrerá toda la imagen para conectar la segunda capa a la primera.



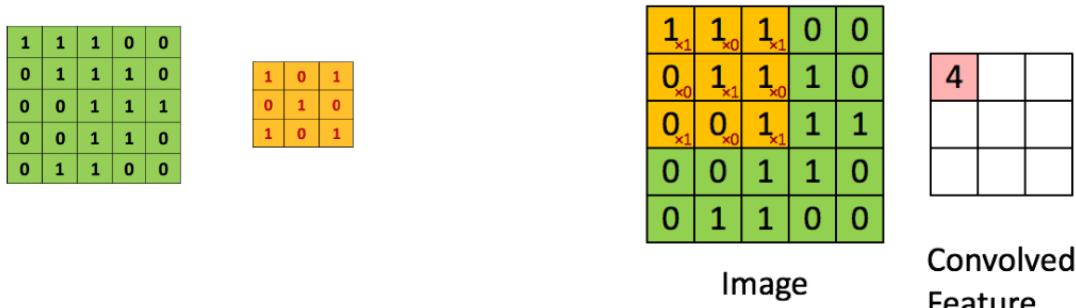
Este movimiento de desplazamiento genera dos problemas, el primero que la dimensionalidad de la capa de salida será menor que la de entrada (se gestiona con el **padding**) y segundo la cantidad de desplazamiento de la ventana puede ser variable, en el ejemplo ha sido uno, pero se gestiona con el **stride**.

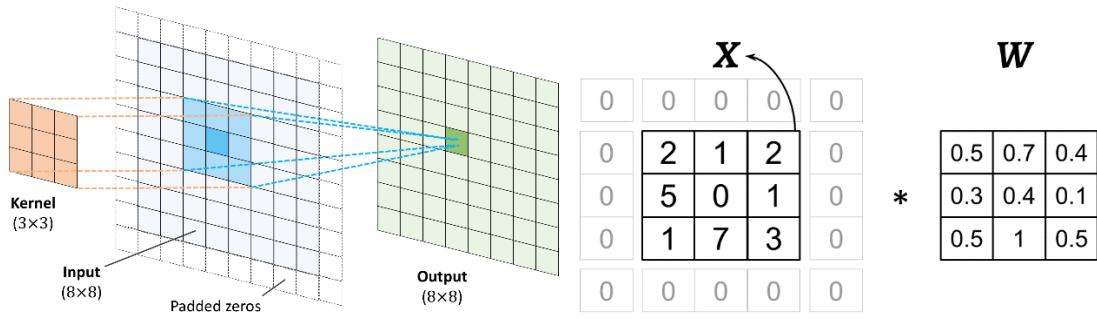
Como cada neurona está conectada con un grupo pequeño de neuronas de la capa anterior (5×5 en el ejemplo) solo tendrá una matriz de pesos de ese tamaño (25 elementos), **esta matriz es compartida por toda la capa** y se ajustará durante el aprendizaje. Es decir, se usa la misma matriz para hacer todo el recorrido de la imagen.

Capa de partida

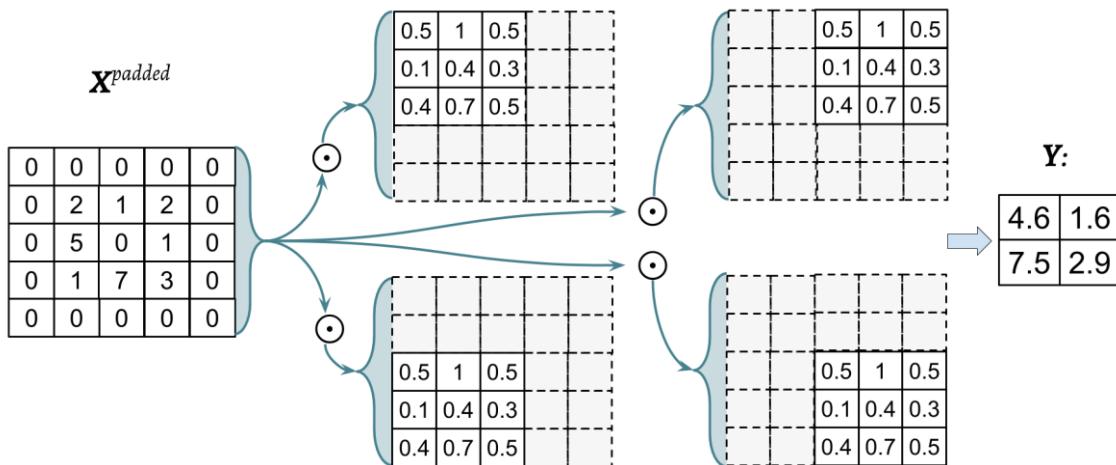


Otro ejemplo.





Para el siguiente ejemplo hacemos padding(1,1), stride=(2,2), kernel(3,3) y $X(3,3)$, con el kernel de la imagen anterior.

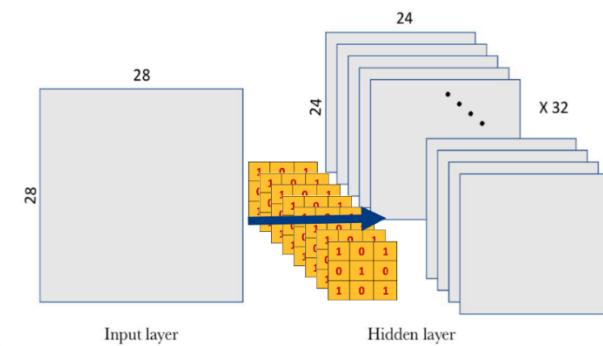


En el ejemplo anterior se asume que el padding es cero y que en cada operación el vector w^r se desplaza dos celdas a la derecha (stride=2,2). Estos dos parámetros (p y s) tendremos que ajustarlos en nuestras capas convolucionales. El desplazamiento que debe sufrir el kernel en cada operación se denomina stride y generalmente es 1.

La matriz de salida de la operación se denomina filtro, y es el resultado de mover el kernel por toda la imagen, multiplicar las dos matrices de forma escalar y sumar los resultados. En el ejemplo anterior el kernel está en azul, se está aplicando sobre la cuarta neurona de la primera capa (en rojo los datos de entrada) se multiplicarán celda a celda (escalarmente) y se sumarán (dando cuatro para la posición correspondiente)

El problema es que cada filtro solo reconoce una característica, con lo que es necesario repetir este procedimiento varias veces (32, 64) para la misma capa para que tenga en cuenta varias características, el número de filtros a definir en una capa se establecen al crear la capa.

```
model.add(layers.Conv2D(32, (3, 3), activation='relu'),
           input_shape=(28, 28, 1))
```



El conjunto de filtros de salida es lo que se denomina feature space o espacio de características.

Resumen de funcionamiento

Para trabajar con las redes convolucionales tendremos las siguientes características

- Imagen de entrada con una dimensión alto, ancho, canales
- A la imagen se le aplicará un kernel de tamaño 3x3 o 5x5
- Se le podrá añadir un número de columnas y filas de ceros en los extremos para evitar la reducción de la dimensión de salida, se llama **padding**.
- El kernel se trasladará a lo largo de toda la imagen con saltos de **stride**, generalmente igual a 1 en todas las direcciones
- Se multiplicará el valor de cada celda de la imagen con la correspondiente del kernel que cae encima y se sumará el valor de todas las celdas resultantes
- Se repetirá el paso anterior hasta cubrir toda la imagen dando otro vector de salida

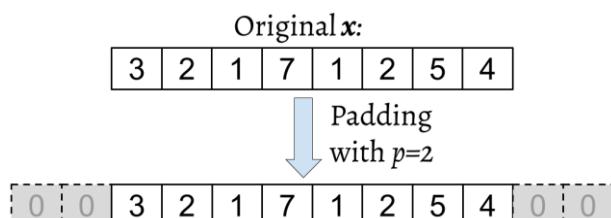
Conceptos matemáticos (en 1D por simplicidad)

La convolución es la operación fundamental en este tipo de capas y es muy importante entender cómo funciona para poder ajustar los parámetros y crear un modelo adecuado. La operación matemática que se usa es la siguiente:

$$y = x * W \rightarrow y[i] = \sum_{k=-\infty}^{k=\infty} x[i-k]w[k]$$

Donde **x** es el valor del pixel de entrada correspondiente y **w** el valor del filtro en esa posición. La operación realizará una suma de todos los elementos del filtro por los valores del pixel, obteniendo un único resultado.

En este ejemplo hemos utilizado un filtro de 3x3, al no usar padding la dimensión de salida es dos filas menos y dos columnas, partimos 7x7 de entrada y terminamos en 5x5. El efecto del padding (p) en 1D es añadir a ambos lados.

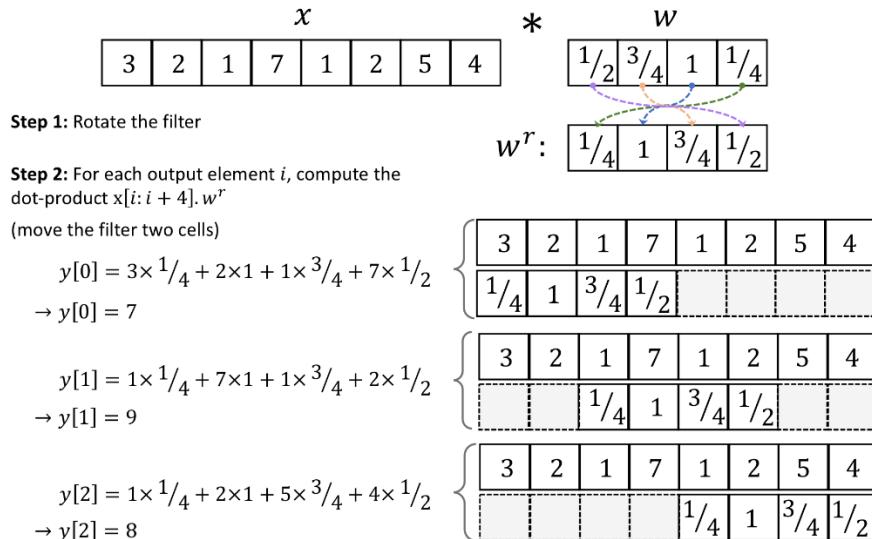


Si asumimos que el vector de entrada x (input) y el filtro w (kernel o canal) tienen n_x y m_w elementos respectivamente, y además $m_w \leq n_x$, el vector x^p (padding) tendría una dimensión

de $n+2p$ (al hacer el padding por ambos extremos). Teniendo en cuenta lo anterior, la fórmula se desarrolla así:

$$y = x * W \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i + m - k]w[k]$$

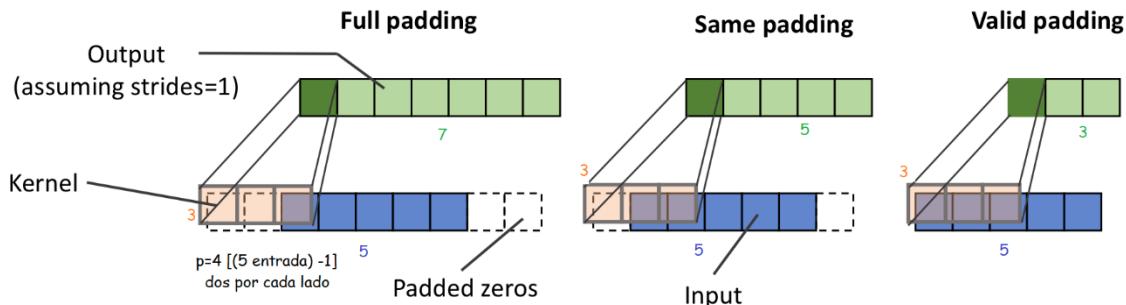
Si vemos la fórmula descubrimos que X y W se recorren de forma inversa, que es lo mismo que hacer la suma de ambos en la misma dirección con uno de los vectores rotados (w^r).



El valor del padding es muy importante ya que va a determinar el tamaño de salida de las características encontradas. Además, el tamaño del padding hace que las características sean o no tratadas de forma no uniforme, haciendo que unas se utilicen más veces en los cálculos que otras. Existen tres modos de padding que determinarán estos dos problemas.

- Full. $p = m-1$, generalmente no se usa.
- Same. Asegura que el vector de salida tiene el mismo tamaño que el de entrada. Es la opción más usada.
- Finally o Valid. Hace que $p=0$ y reduzca el tamaño del vector de salida.

El tamaño final del vector de salida viene determinado por el número de veces que el kernel



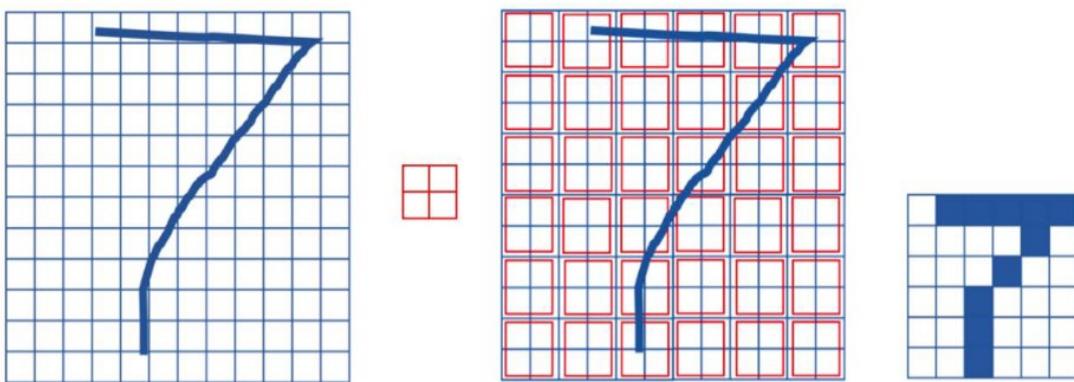
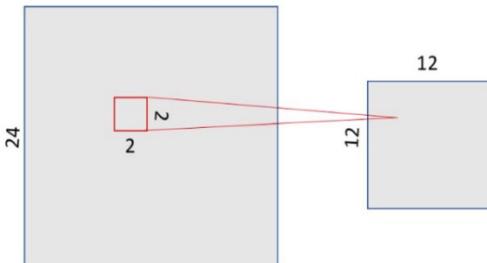
recorre el vector de entrada y se desplaza(strides).

Para el paso a 2D el único problema es saber cómo calcular la matriz w^r (rotada) que coincide exactamente con la siguiente instrucción Python: `w_rot = w[::-1, ::-1]`

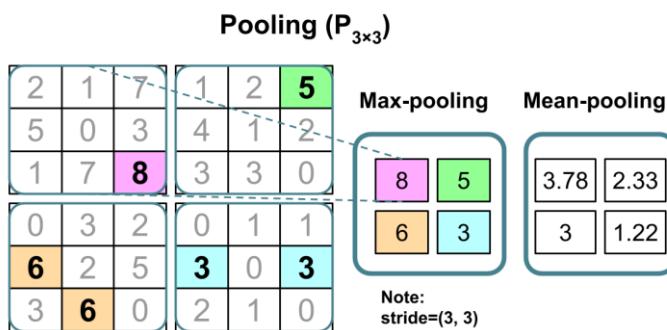
Submuestreo (Pooling)

El submuestreo realiza una simplificación de la información recogida por la capa de convolución y crean una versión condesada de la información de la misma reduciendo la dimensionalidad. Se deben aplicar justo después de cada capa de convolución.

```
model.add(layers.MaxPooling2D((2, 2)))
```



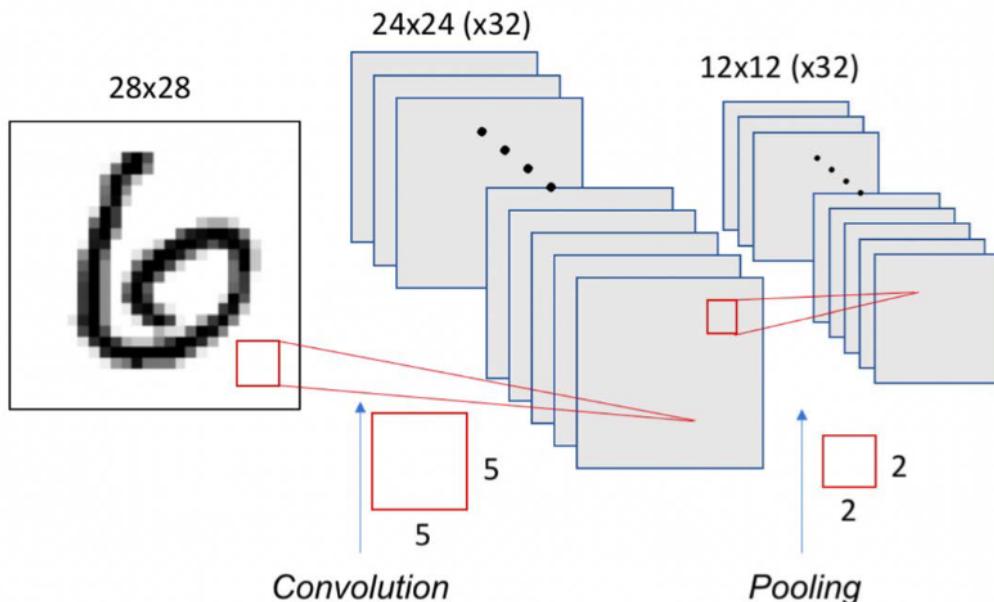
El submuestreo, adopta dos formas: **max-min** o **mean**. En la primera opción se elige el mayor de todos los implicados y en la segunda se hace la media. El funcionamiento es similar, se establece un tamaño de kernel que se aplica a la entrada, se desplaza a través de todos los valores y se va calculando el resultado final. La salida será un vector de menor tamaño que la entrada en función de la dimensión del kernel (pooling size).



El Pooling puede ser con solapamiento o sin él, en el ejemplo anterior es si solapamiento. Tradicionalmente se usa sin solapamiento, teniendo que ser la matriz kernel divisor de la matriz de entrada, pero se podría usar con solapamiento si el tamaño desplazamiento es menor que el tamaño del kernel.

Existe una tercera capa de Pooling (GlobalAvgPool2D) que muestra una única salida por cada mapa de características y por instancia, el resultado es que puede ser útil como última capa en vez de una Densa.

Como la capa convolucional tiene varios filtros (32, 64) el Pooling se aplicará a cada uno de ellos generando tantas salidas como filtros haya.



Imágenes a color

Si trabajamos con imágenes en gris solo existirá una matriz con valores, pero si estamos usando colores estas aumentan hasta 3 como mínimo (RGB) o incluso cuatro (RGBA) siendo la última la capa de transparencia.

En caso de usar imágenes a color, el procedimiento es el mismo lo único que se repetirá tantas veces como canales de entrada tengamos (RGB) y se sumarán las tres capas usando una matriz de suma que determine el modo que se pondrá dicha suma. Cada capa tiene su propio kernel.

Hay que considerar también que el tratamiento de imágenes hace que se necesita gran cantidad de memoria y de cálculo, por lo que deberemos cargar siempre nuestras imágenes como uint8, enteros de 8 bits sin signo ya que el color se especifica como un valor de 0..255 y con 8 bits es más que suficiente, aligerando de esta manera la memoria necesaria (es recomendable usar tf.images para el tratamiento de las mismas).

Se considera que una imagen es grande a partir de 256x256 píxeles para aplicar los hiperparámetros.

Extendiendo el modelo

Es muy fácil extender este modelo a más dimensiones, por ejemplo 3, en la que se trate por ejemplo video y la tercera dimensión sea cada uno de los fotogramas.

Gestión de los Hiperparámetros

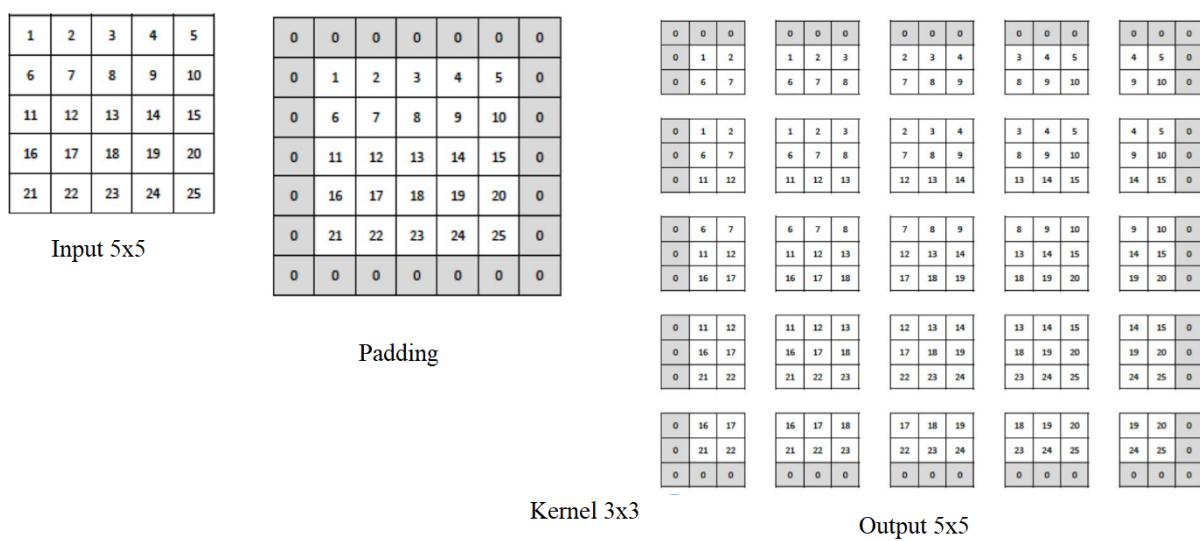
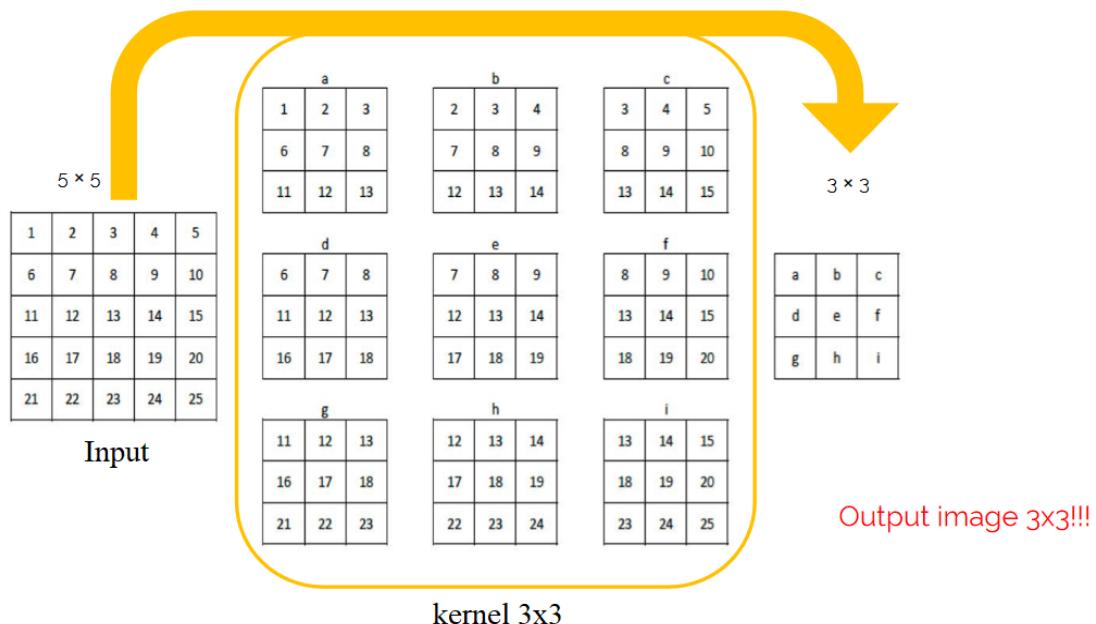
Tamaño y número de filtros

El tamaño del kernel determinará el tamaño del filtro, a mayor tamaño más información se perderá. Los tamaños recomendados son 3x3 y 5x5, pudiendo aumentar si la dimensionalidad de la imagen de entrada es muy grande.

Del mismo modo el número de filtros determinan el número de características a usar, estos valores suelen ser 32 o 64, pero se podrá aumentar o disminuir en función de la complejidad del problema.

Padding

El Padding es el número de columnas y filas rellenas de ceros que se aumentará el tensor inicial para mantener la dimensión en el tensor de salida. Recordemos que al aplicar un filtro de tamaño mayor de 1x1 la salida reducirá su dimensionalidad.



Los valores que podemos usar en el parámetro **padding** son:

- Full. $p = m-1$, generalmente no se usa.
- Same. Asegura que el vector de salida tiene el mismo tamaño que el de entrada. Es la opción más usada.
- Finally o Valid. Hace que $p=0$ y se reduzca el tamaño del vector de salida, es el defecto en Keras.

Stride

Ya hemos comentado que el stride es el número de pasos que se desplaza el kernel hacia la izquierda y hacia abajo en el desplazamiento por la imagen. Este valor suele ser una matriz indicando los pasos en cada sentido, pero por norma general no se utiliza y se usa por defecto la matriz (1,1).

Ejercicio (Code 0035_c.py):

Realizar un estudio en el conjunto de datos Fashion-Mnist

Regularización: Dropout y L1, L2

Al igual que vimos en capítulos anteriores, las CNN se benefician de este tipo de regularización. Aunque es la más usada no tiene por qué ser la única, podemos usar también L1 o L2.

```
from tensorflow import keras

dropout_layer = keras.layers.Dropout(0.5)

conv_layer = keras.layers.Conv2D(
    filters=16, kernel_size=(3, 3),
    kernel_regularizer=keras.regularizers.l2(0.001))

fc_layer = keras.layers.Dense(
    units=16, kernel_regularizer=keras.regularizers.l2(0.001))
```

Generalmente se usa un porcentaje del 50% en las capas **Dropout**. El efecto es forzar a la capa a aprender patrones más generales y robustos, mejorando además el sobreajuste. Hay que tener cuidado que no se deben poner tras la capa de entrada.

Regularización: BatchNormalization

Esta regularización tiene el mismo efecto que el estudiado en el curso con la normalización, lleva las entradas dentro de una media de cero y una desviación de 1. Estas capas aparecen justo antes de la capa Dropout y después de una Densa o convolucional.

```
from tensorflow import keras

dropout_layer = keras.layers.BatchNormalitation()
```

Regularización: Decaimiento del ratio de aprendizaje

Ya se comentaron las técnicas de decaimiento y sus beneficios. Las redes CNN también se aprovechan de ellas.

```
# opción uno
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(.. optimizer=optimizer)
# opción dos
```

```
reduce_lr = LearningRateScheduler(lambda x: 1e-1*0.9**x)
model.fit(train_images, train_labels, epochs=5, batch_size=64,
callbacks=[reduce_lr])
```

Funciones de pérdida a usar

Loss function	Usage	Examples	
		Using probabilities from_logits=False	Using logits from_logits=True
BinaryCrossentropy	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

CategoricalCrossentropy, si hemos utilizado one_encoder para la salida, en caso contrario SparseCategoricalCrossentropy.

```
from distutils.version import LooseVersion as Version

##### Binary Crossentropy
bce_probas = tf.keras.losses.BinaryCrossentropy(from_logits=False)
bce_logits = tf.keras.losses.BinaryCrossentropy(from_logits=True)

logits = tf.constant([0.8])
probas = tf.keras.activations.sigmoid(logits)

tf.print(
    'CCE (w Probas): {:.4f}'.format(
        cce_probas(y_true=[[0, 0, 1]], y_pred=probas)),
    '(w Logits): {:.4f}'.format(
        cce_logits(y_true=[[0, 0, 1]], y_pred=logits)))


##### Categorical Crossentropy
cce_probas = tf.keras.losses.CategoricalCrossentropy(
    from_logits=False)
cce_logits = tf.keras.losses.CategoricalCrossentropy(
    from_logits=True)

logits = tf.constant([[1.5, 0.8, 2.1]])
probas = tf.keras.activations.softmax(logits)

tf.print(
    'CCE (w Probas): {:.4f}'.format(
        cce_probas(y_true=[0, 0, 1], y_pred=probas)),
    '(w Logits): {:.4f}'.format(
        cce_logits(y_true=[0, 0, 1], y_pred=logits)))

##### Sparse Categorical Crossentropy
sp_cce_probas = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=False)
sp_cce_logits = tf.keras.losses.SparseCategoricalCrossentropy(
```

```

from_logits=True)

tf.print(
    'Sparse CCE (w Probas): {:.4f}'.format(
        sp_cce_probas(y_true=[2], y_pred=probas)),
    '(w Logits): {:.4f}'.format(
        sp_cce_logits(y_true=[2], y_pred=logits)))

```

Algunas veces se usa **categoriacal_cross_entropy** para la clasificación binaria, en este modelo la salida se interpreta como la probabilidad de la clase verdadera y 1-p la probabilidad de la otra clasificación, sin necesidad de usar una segunda neurona.

Resumen de parámetros en cada bloque

Este bloque se repite periódicamente tantas veces como deseemos hasta encontrar un buen ajuste, añadiendo al final de todos los bloques tres capas: **Flatter()** para cambiar la dimensionalidad y dos **Densas**,: la primera con el mismo número de neuronas que filtros del último bloque (Tener en cuenta el **MaxPooling** que reduce la dimensionalidad) y la última con el mismo número de neuronas que clases a determinar o una neurona si es un problema de clasificación binaria.

- Conv2D: tf.keras.layers.Conv2D
 - Filters. Número de características a encontrar. Suelen ir creciendo en orden de múltiplos de dos hasta las capas densas.
 - kernel_size. El tamaño es muy importante y no se debe usar grandes tamaños (5x5 o más) en cada capa, la única excepción es la primera capa que se suele utilizar con un tamaño de 5x5 y un stride de dos.
 - strides, depende de la imagen, si son pequeñas (64x64) a cero
 - padding, generalmente a SAME.
- MaxPool2D: tf.keras.layers.MaxPool2D
 - pool_size (2 para hace que se reduzca a la mitad el tamaño)
 - strides. No se suele establecer.
 - Padding. No se suele establecer.
- Normalización: tf.keras.layers.Normalization(), opcional
- Dropout: tf.keras.layers.Dropout2D
 - Rate (hasta 50%).

Ejemplo.

```

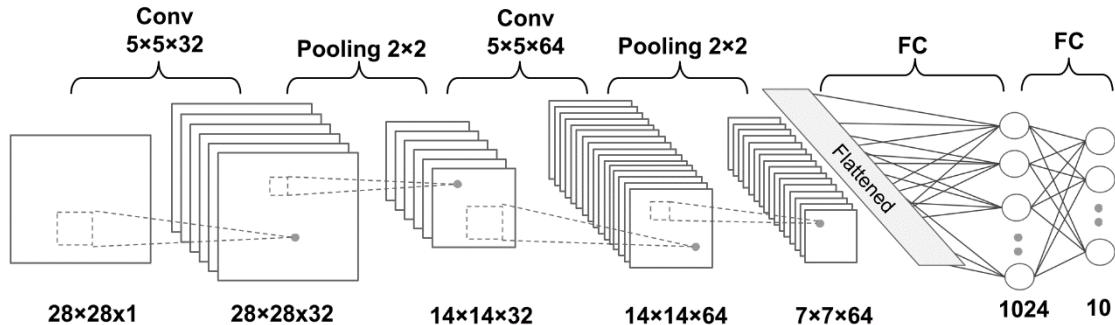
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                      input_shape=[28, 28, 1]),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
])

```

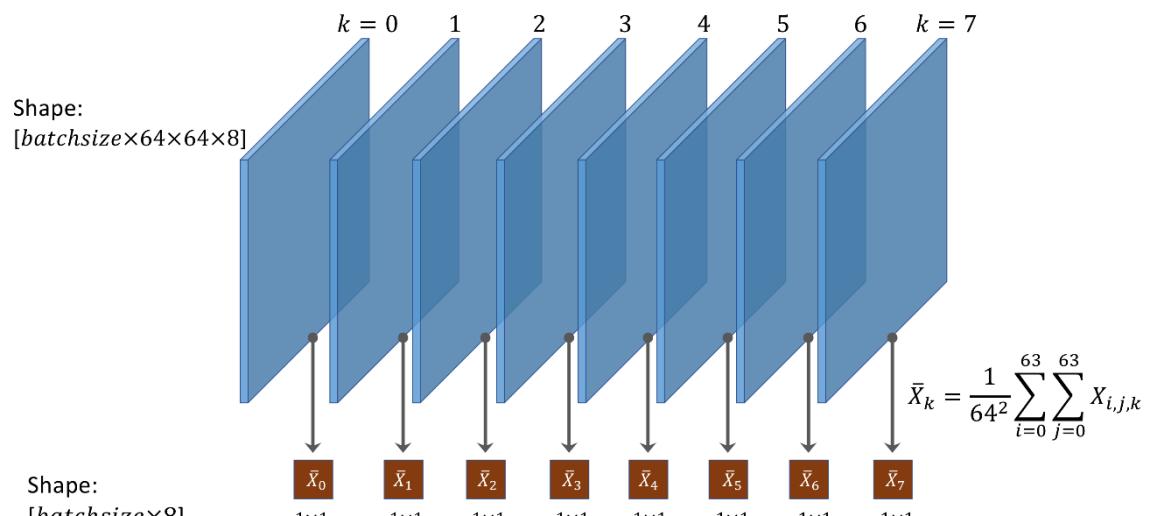
```
        keras.layers.Dense(10, activation="softmax")
    ])
```

Ejercicio (Code 0035_d.py):

Aplicar las técnicas anteriores para mejorar el resultado

Ejemplos**MNIST Caracteres numéricos**

Código 036.py

CelebA (Fotos de personas)

Código 037.py

Gatos y perros

Código 038.py

Visualización de filtros

```

from keras.models import load_model
# Preprocesses the image into a 4D tensor
from keras.preprocessing import image
import numpy as np
from keras import models
import matplotlib.pyplot as plt

model = load_model('code_0039_b.h5') # modelo ya entrenado

# imagen de prueba
img_path = 'code_0039_b.jpg'
print(model.summary())
img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
print(img_tensor.shape)
# recordar que necesitamos un tensor 4D (x, 150,150,3)
img_tensor = np.expand_dims(img_tensor, axis=0)
print(img_tensor.shape)
img_tensor /= 255. # Normalizamos como se hizo en el modelo.
plt.imshow(img_tensor[0])
plt.show()

"""
# modelo cargado
Model: "sequential"

```

Layer (type)	Output Shape	Param #
conv2d_21 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_21 (MaxPooling)	(None, 74, 74, 32)	0
conv2d_22 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_22 (MaxPooling)	(None, 36, 36, 64)	0
conv2d_23 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_23 (MaxPooling)	(None, 17, 17, 128)	0
conv2d_24 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_24 (MaxPooling)	(None, 7, 7, 128)	0
flatten_6 (Flatten)	(None, 6272)	0
dropout_3 (Dropout)	(None, 6272)	0
dense_11 (Dense)	(None, 512)	3211776
dense_12 (Dense)	(None, 1)	513

Para recoger el mapa de característica, hay que crear un nuevo modelo que recoja la imagen de muestra y como salida tenga el valor de activación de todas las capas convolucionales y pooling.

"""

```

# Cogemos solo las 8 primeras capas que son las que necesitamos,
conv2d + pooling
layer_outputs = [layer.output for layer in model.layers[:8]]
activation_model = models.Model(inputs=model.input,
outputs=layer_outputs)

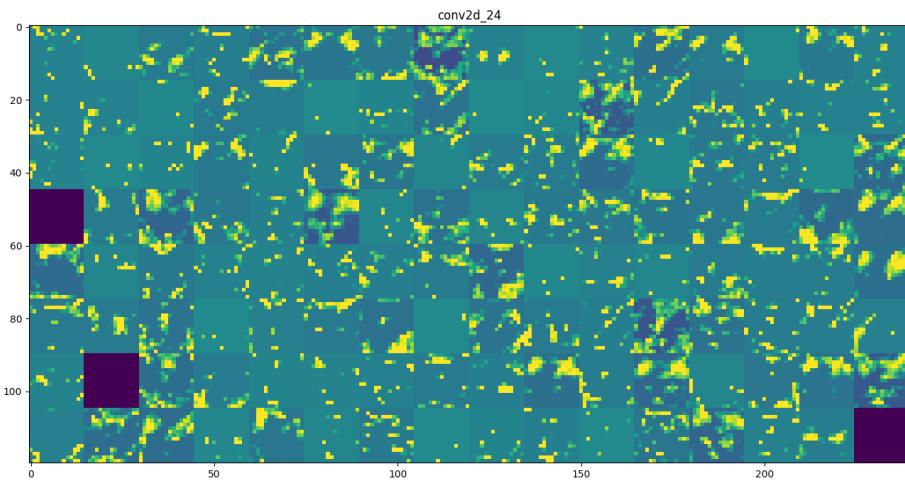
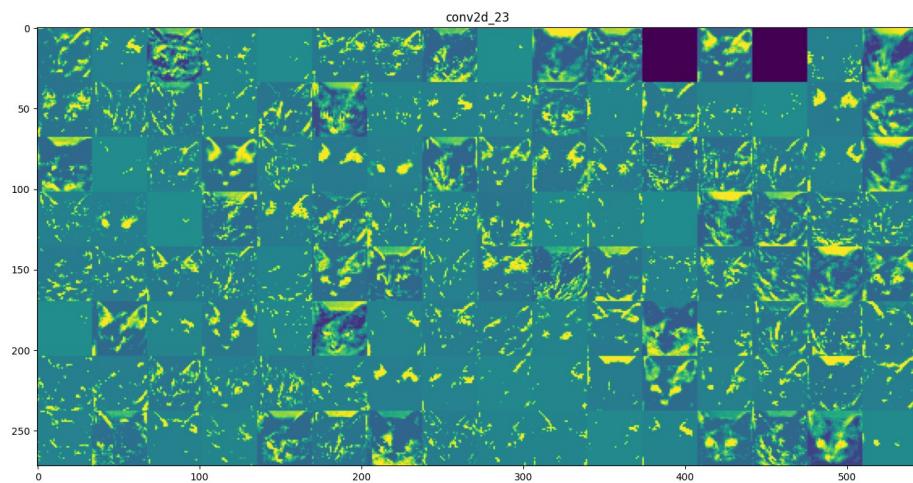
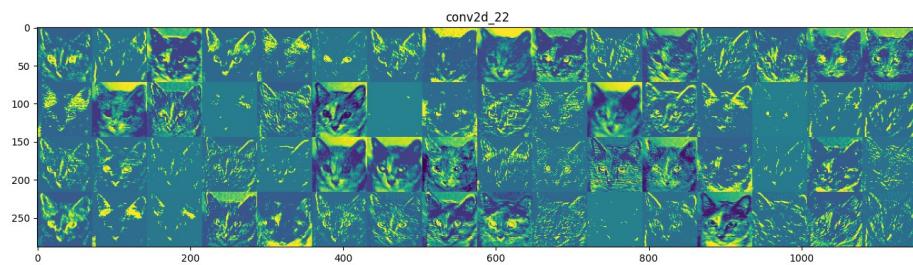
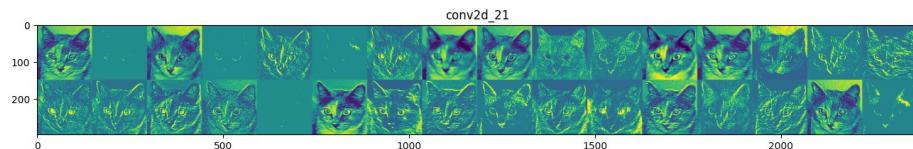
# el modelo genera 8 salidas una por cada capa
activations = activation_model.predict(img_tensor)

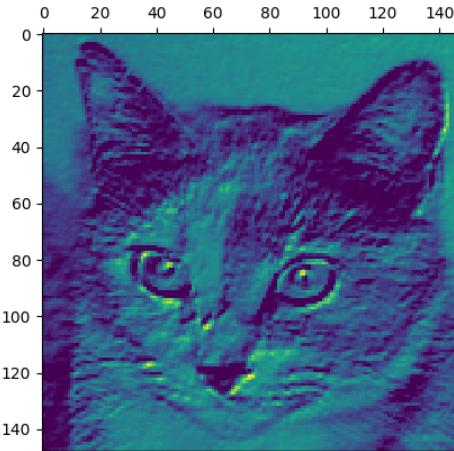
# vamos a ver la primera capa
first_layer_activation = activations[0]
print(first_layer_activation.shape)      # (1, 148, 148, 32)
# mostramos el primero canal de activación
plt.matshow(first_layer_activation[0, :, :, 0], cmap='viridis')
# mostramos el último canal de activación
plt.matshow(first_layer_activation[0, :, :, 31], cmap='viridis')
plt.show()

# Vamos a ver todas las capas
layer_names = []
for layer in model.layers[:8]: # Tenemos ocho capas en el modelo
    layer_names.append(layer.name) # nombres de capas para dibujarlas
images_per_row = 16
# Recorremos las ocho salidas
for layer_name, layer_activation in zip(layer_names, activations):
    # (1, size, size, n_features) cogemos el número de filtros de la
    # salida, la última posición
    n_features = layer_activation.shape[-1]
    # cogemos el tamaño del filtro (1, size, size, n_features).
    size = layer_activation.shape[1]
    # creamos el grid con las filas y las columnas necesarias
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))
    for col in range(n_cols):
        for row in range(images_per_row):
            # imagen sin procesar
            channel_image = layer_activation[0, :, :, col *
                                              images_per_row + row]
            # Procesamos la imagen para que se vea
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0,
                                   255).astype('uint8')
            display_grid[col * size: (col + 1) * size, row * size:
                         (row + 1) * size] = channel_image
    scale = 1. / size
    # creamos el canvas con las imágenes igual al grid
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
    plt.show()

```

Código 039_b.py





La primera capa actúa como una colección de varios detectores de bordes. En esa etapa, las activaciones retienen casi toda la información presente en la imagen inicial. A medida que asciende, las activaciones se vuelven cada vez más abstractas y menos visualmente interpretable. Comienzan a codificar conceptos de nivel superior como "oreja de gato" y "ojo de gato". Las presentaciones superiores llevan cada vez menos información sobre los contenidos visuales de la imagen, y cada vez más información relacionada con la clase de la imagen.

La escasez de activaciones aumenta con la profundidad de la capa: en la primera capa, todos los filtros son activados por la imagen de entrada; pero en las siguientes capas, cada vez hay más filtros en blanco. Esto significa que el patrón codificado por el filtro no se encuentra en la imagen de entrada.

Transferencia del aprendizaje

El reconocimiento de imágenes ha sido objeto de mucho estudio a lo largo de estos años y ha presentado una evolución muy grande, lo que ha dado como resultado gran cantidad de modelos para el reconocimiento cada uno de ellos con sus ventajas y sus inconvenientes. La característica de estos modelos es el tiempo elevado de entrenarlo en una computadora general si no tenemos una GPU avanzada, por lo que encontramos la mayoría de ellos ya entrenados y disponibles a través de `tf.keras.application`.

En el siguiente fichero encontramos todo el código del ejemplo de este punto.

Código 039_c.py

```
base_dir =
'C:/xampp/htdocs/Python/CursoEspecializacion/ut06/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

# Directorio con las imágenes de training
train_cats_dir = os.path.join(train_dir, 'cats')
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directorio con las imágenes de validation
validation_cats_dir = os.path.join(validation_dir, 'cats')
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
```

```
# Directorio con las imagenes de test
test_cats_dir = os.path.join(test_dir, 'cats')
test_dogs_dir = os.path.join(test_dir, 'dogs')

print('total training cat images :', len(os.listdir(train_cats_dir)))
print('total training dog images :', len(os.listdir(train_dogs_dir)))

print('total validation cat images :',
len(os.listdir(validation_cats_dir)))
print('total validation dog images :',
len(os.listdir(validation_dogs_dir)))

print('total test cat images :', len(os.listdir(test_cats_dir)))
print('total test dog images :', len(os.listdir(test_dogs_dir)))
```

En este ejemplo vamos a trabajar como proyectos más reales, con directorios para los datos en vez de cargarlos desde Keras, Tensorflow o un fichero csv. Hemos descomprimido el fichero zip de `cat_dog_small` y hemos dejado 13 ejemplos en cada directorio, si tenemos un ordenador con GPU no habrá que eliminar nada.

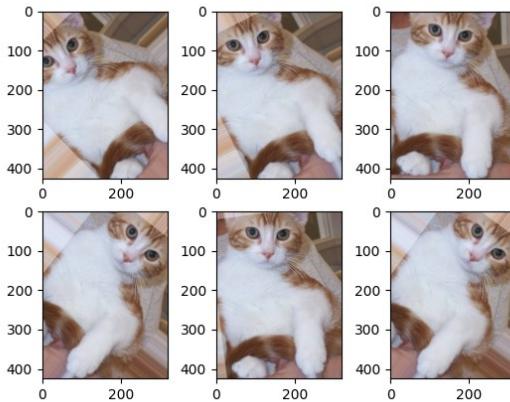
Transformación de imágenes

La idea es generar más datos de entrenamiento a partir de nuestros datos aplicando transformaciones aleatorias a la imagen de entrada, produciendo otras similares, pero creíbles. Hay que tener en cuenta el contexto del conjunto de datos para no generar imágenes que nunca podrían encontrarse en realidad. Estas transformaciones las realizará la clase **ImageGenerator** al vuelo, sin necesidad de almacenarlas en disco. Veamos como se generan varias diferentes con la clase.

```
for fn in os.listdir(train_cats_dir):
    path = os.path.join(train_cats_dir, fn)
    img = load_img(path)
    data = img_to_array(img)
    samples = expand_dims(data, 0)

    # example of "rotation_range"
    datagen = ImageDataGenerator(rotation_range=45)

    it = datagen.flow(samples, batch_size=1)
    for i in range(6):
        plt.subplot(230 + 1 + i)
        batch = it.next()
        image = batch[0].astype('uint8')
        plt.imshow(image)
    plt.show()
```



Pretendemos transformar la imagen que se va a pasar a la red cada epoch de tal manera que la red nunca reciba la misma imagen, el resultado es que multiplicaremos el número de imágenes por el número de épocas, generando al vuelo gran cantidad de datos. Hay que recordar que estos datos no se guardan, con lo que con cada ejecución variarán los datos de entrada.

```
modelDA = Sequential()
modelDA.add(Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
modelDA.add(MaxPooling2D(2, 2))
modelDA.add(Conv2D(64, (3, 3), activation='relu'))
modelDA.add(MaxPooling2D(2, 2))
modelDA.add(Conv2D(128, (3, 3), activation='relu'))
modelDA.add(MaxPooling2D(2, 2))
modelDA.add(Conv2D(128, (3, 3), activation='relu'))
modelDA.add(MaxPooling2D(2, 2))
modelDA.add(Flatten())
modelDA.add(Dense(512, activation='relu'))
modelDA.add(Dense(1, activation='sigmoid'))

modelDA.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])
```

A continuación, exponemos los generadores de imágenes para los tres conjuntos

```
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

validation_datagen = ImageDataGenerator(rescale=1.0 / 255.)
test_datagen = ImageDataGenerator(rescale=1.0 / 255.)
```

Solo se crean nuevas imágenes en las de **train**, nunca en los otros conjuntos, en los que simplemente se escalan y normalizan como siempre con las imágenes.

Al trabajar con directorios y no con datos cargados, tenemos que crear una estructura de datos que vaya proporcionándolos a la red según los mini-batch elegidos, en Python crearemos generadores sobre el contenido de los directorios (**flow_from_directory**) a partir de los objetos generadores de imágenes.

```
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))
validation_generator = validation_datagen.flow_from_directory(
    validation_dir, batch_size=20,
    class_mode='binary', target_size=(150, 150))
test_generator = test_datagen.flow_from_directory(test_dir,
                                                 batch_size=20, class_mode='binary',
                                                 target_size=(150, 150))
```

Por último, ya estamos en condiciones de entrenar y validar el modelo

```
steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

historyDA = modelDA.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=100,
    validation_data=validation_generator,
    validation_steps=validation_steps,
    verbose=2)

print(steps_per_epoch)
print(validation_steps)
test_lost, test_acc = modelDA.evaluate(test_generator)
print("Test Accuracy:", test_acc)

acc = historyDA.history['acc']
val_acc = historyDA.history['val_acc']
loss = historyDA.history['loss']
val_loss = historyDA.history['val_loss']

epochs = range(1, len(acc) + 1, 1) # obtener número de epochs

plt.plot(epochs, acc, 'r--', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.ylabel('acc')
plt.xlabel('epochs')

plt.legend()
plt.figure()

plt.plot(epochs, loss, 'r--')
plt.plot(epochs, val_loss, 'b')
plt.title('Training and validation loss')
plt.ylabel('loss')
plt.xlabel('epochs')

plt.legend()
plt.figure()
```

Hay que recordar que están entrenados sobre trece imágenes en vez del conjunto completo.

Uso de modelos ya entrenados

En lugar de entrenar una red completamente, podemos usar una entrenada para nuestros fines.

Hay dos aproximaciones usarlo directamente, o modificar las capas de la red entrenada.

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
images_resized = tf.image.resize(images, [224, 224])
inputs = keras.applications.resnet50.preprocess_input(
    images_resized * 255)
y_proba = model.predict(inputs)
```

Como vemos en el ejemplo anterior, es muy simple hacer uso de un modelo ya entrenado, simplemente lo cargamos, preparamos las imágenes al tamaño que espera el modelo, procesamos dichas imágenes y realizamos la predicción.

Evidentemente puede dar un buen resultado, pero podremos mejorarlo si modificamos el modelo ya entrenado con algunas capas más y lo entramos de nuevo.

Transferencia del aprendizaje

Ya hablamos en el capítulo de redes neuronales de la transferencia de aprendizaje. Es una técnica por la que, en un modelo ya entrenado fijamos algunas capas y reentrenamos con nuestros ejemplos de problema en particular que queremos resolver mejorando los tiempos de entrenamiento y el rendimiento final. Este es el caso que vamos a ver ahora.

Hay que recordar que las CNN aprender de lo más simple a lo más complejo, con lo que, si vamos eliminando capas desde la salida, nos quedaremos con los datos que generalizan bien para todas las clases, en nuestro ejemplo de animales.

Extracción de características

En esta aproximación vamos a liberar el modelo del discriminante final. Hay que recordar que la estructura de la red son bloques de capas convolucionales, MaxPooling de forma repetida, pero al final se aplana el resultado y se pasa a otro conjunto de capas densas. Este último conjunto es el que se elimina para poner el nuestro propio en función del problema y se reentrena con muchos menos ejemplos, fijando las capas del modelo importado y ajustando solo las capas que añadimos nosotros. Cargamos el modelo entrenado y eliminamos la última capa, el clasificador.

```
pre_trained_model = VGG16(input_shape=(150, 150, 3),
                           include_top=False,
                           weights='imagenet')
pre_trained_model.summary()
```

En el siguiente paso, congelamos todas las capas del modelo entrenado, creamos nuestro clasificador y creamos el modelo nuevo compuesto de ambos, para pasar a entrenarlo después.

```
for layer in pre_trained_model.layers:
    layer.trainable = False

pre_trained_model.summary()
modelFE = Sequential()
modelFE.add(pre_trained_model)
modelFE.add(Flatten())
modelFE.add(Dense(256, activation='relu'))
modelFE.add(Dense(1, activation='sigmoid'))
```

```

modelFE.summary()

modelFE.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])

batch_size = 20
steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

historyFE = modelFE.fit(
    train_generator,
    validation_data=validation_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=100,
    validation_steps=validation_steps,
    verbose=2)

```

El último paso es evaluar, como hicimos en el punto de transformación de imágenes.

Afinado del modelo

Un ajuste más acertado generalmente es en el que además de eliminar el clasificador, no se fijan todas las capas del modelo que quedan, solo las más altas, para que se ajusten también a nuestro problema las últimas capas convolucionales. Para que funcione correctamente, deberemos establecer ritmos de aprendizajes lentos para evitar que cambien mucho los parámetros ya aprendidos en las capas no fijas convolucionales.

```

pre_trained_model = VGG16(input_shape=(150, 150, 3),
                           include_top=False,
                           weights='imagenet')

pre_trained_model.trainable = True

set_trainable = False

for layer in pre_trained_model.layers:
    if layer.name == 'block5_conv1': # solo el block_5 se entrenará
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False

pre_trained_model.summary()

modelFT = Sequential()
modelFT.add(pre_trained_model)
modelFT.add(Flatten())
modelFT.add(Dense(256, activation='relu'))
modelFT.add(Dense(1, activation='sigmoid'))

modelFT.summary()

modelFT.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])
historyFT = modelFT.fit(
    train_generator,

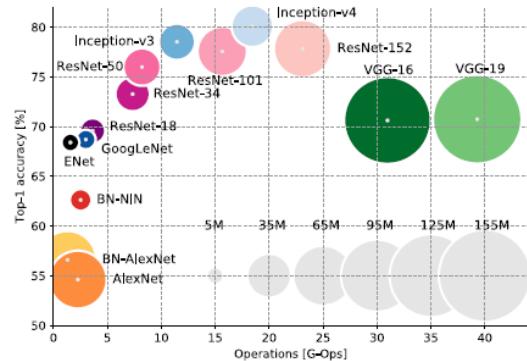
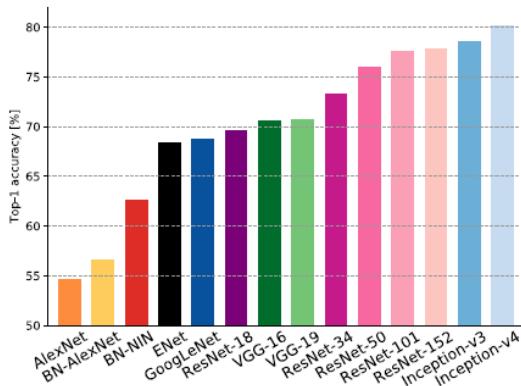
```

```
validation_data=validation_generator,
steps_per_epoch=steps_per_epoch,
epochs=100,
validation_steps=validation_steps,
verbose=2)
```

Modelos implementados en Keras

https://www.tensorflow.org/api_docs/python/tf/keras/applications

- densenet module: DenseNet models for Keras.
- efficientnet module: EfficientNet models for Keras.
- inception_resnet_v2 module: Inception-ResNet V2 model for Keras.
- inception_v3 module: Inception V3 model for Keras.
- mobilenet module: MobileNet v1 models for Keras.
- mobilenet_v2 module: MobileNet v2 models for Keras.
- mobilenet_v3 module: MobileNet v3 models for Keras.
- nasnet module: NASNet-A models for Keras.
- resnet module: ResNet models for Keras.
- resnet_v2 module: ResNet v2 models for Keras.
- vgg16 module: VGG16 model for Keras.
- vgg19 module: VGG19 model for Keras.
- xception module: Xception V1 model for Keras.



Clasificación y localización de objetos

La localización de objetos en una imagen puede entenderse como un problema de regresión. Para predecir el marco que encierra el objeto, la aproximación más general es intentar añadir el centro del objeto y la anchura y altura de la caja que lo encierra. Esta aproximación hace que haya que predecir cuatro números más, pero que no implica un cambio importante, simplemente hay que añadir una capa Densa con cuatro unidades entrenadas con MSE.

```
base_model = keras.applications.xception.Xception(
    weights="imagenet", include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(
    n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
optimizer = keras.optimizers.SGD(Learning_rate=0.01, momentum=0.9,
nesterov=True, decay=0.001)
```

```
model = keras.Model(inputs=base_model.input,
                     outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy"])
```

El problema se plantea a la hora de crear los datos de entrenamiento: tenemos muchos datos con etiquetas, pero hay muy pocos datos con límites, con lo que tendremos que crearlos nosotros a mano, o pedir ayuda con las imágenes para que nos las etiqueten diciendo los bordes del objeto a buscar (la tupla que se pasará a fit será: [images, (etiquetas, bordes)]).

Redes Convolucionales Completas

Las redes FCN se introdujeron para tareas de segmentación semántica. La idea es cambiar la última capa densa por una convolucional, de tal manera que el tamaño del filtro tiene que ser igual al tamaño de la entrada de los mapas de características y usar **Valid** para el padding, además el parámetro stride tiene que ser a 1.

La característica principal es que las FCN contiene solo capas convolucionales y permite que la red sea entrenada en imágenes de cualquier tamaño.

La elección de la arquitectura de detección dependerá del problema.

- YOLO. Esta arquitectura se propuso para la detección de objetos en las imágenes y en sus diferentes versiones se han ido mejorando paulatinamente.
- SSD
- Faster-RCNN

Hacer un trabajo sobre este punto para reconocer objetos.

Segmentación semántica

La segmentación semántica intenta clasificar los pixeles de una imagen de acuerdo a una clase, detectando la clase, no la individualidad, con lo que si hay dos coches en la imagen juntos aparecerán los dos como coches y no como dos objetos separados.



Existen varias arquitecturas para este problema y las más sencillas se basan en una nueva capa **Conv2DTranspose()**.

Ejercicios

Aplicar las técnicas cuatro técnicas anteriores al conjunto CelebA

Aplicar las técnicas cuatro técnicas anteriores al conjunto CIFAR-10

exe_0016.py

Gatos y perros con transferencia de conocimiento

Código 039.py

Reconocimiento de objetos

Código 040_a.py

Código 040_b.py

Código 040_c.py

UT-7

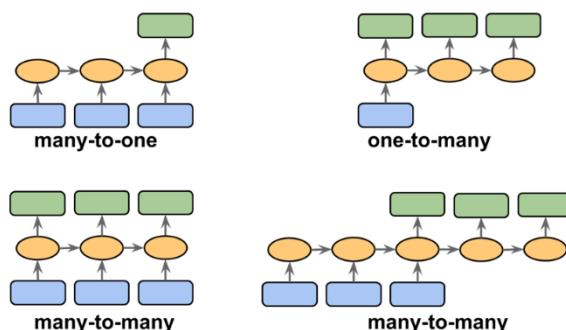
Cap 16.-Tratamiento del lenguaje natural con RNN

Las RNN se desarrollaron para modelizar secuencias, y las más usadas actualmente es el lenguaje natural, pero con aplicaciones a múltiples áreas de mercado.

Cuando trabajamos con el lenguaje natural, las características de los datos son intrínsecamente diferentes a las vistas hasta ahora, tiene un orden y son secuenciales. Lo que hace que los elementos de una secuencia sean únicos es el orden en el que aparecen y son dependientes de los anteriores, que es completamente opuesto a que sean independientes y uniformemente distribuidos, como esperaban todos los algoritmos vistos hasta ahora. Hay muchos tipos de datos con este formato (precios de un producto, valores de la bolsa, etc.) y las series temporales son un tipo de datos en secuencia en la que cada ejemplo está asociado a un tiempo concreto. Este tipo de secuencias presentan unas características propias de tratamiento.

Como las secuencias dependen generalmente de los datos anteriores, tenemos que usar algoritmos de redes neuronales que sean capaces de mantener memoria de los datos ya procesados, y ese tipo de redes se llaman Redes Neuronales Recurrentes (RNN).

Hay gran cantidad de aplicaciones de las RNN como el tratamiento del lenguaje natural, traducciones, etiquetado de imágenes, generación de texto, etc. Pero para resolver correctamente el problema que se nos plantea debemos entender las diferentes arquitecturas de RNN existentes.

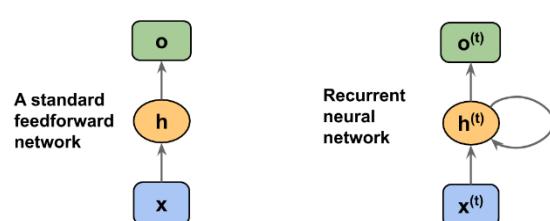


En la figura vemos las tres posibles estructuras de red. En la primera (many to one) la entrada es una secuencia y la salida un único escalar. En la estructura one to many la entrada es una única entrada en formato tradicional y la salida una secuencia, y por último, en la tercera estructura la entrada es una secuencia y la salida también (entendemos secuencia como conjunto de datos con un orden específico).

En la figura vemos las tres posibles estructuras de red. En la primera (many to one) la entrada es una secuencia y la salida un único escalar. En la estructura one to many la entrada es una única entrada en formato tradicional y la salida una secuencia, y por último, en la tercera estructura la entrada es una secuencia y la salida también (entendemos secuencia como conjunto de datos con un orden específico).

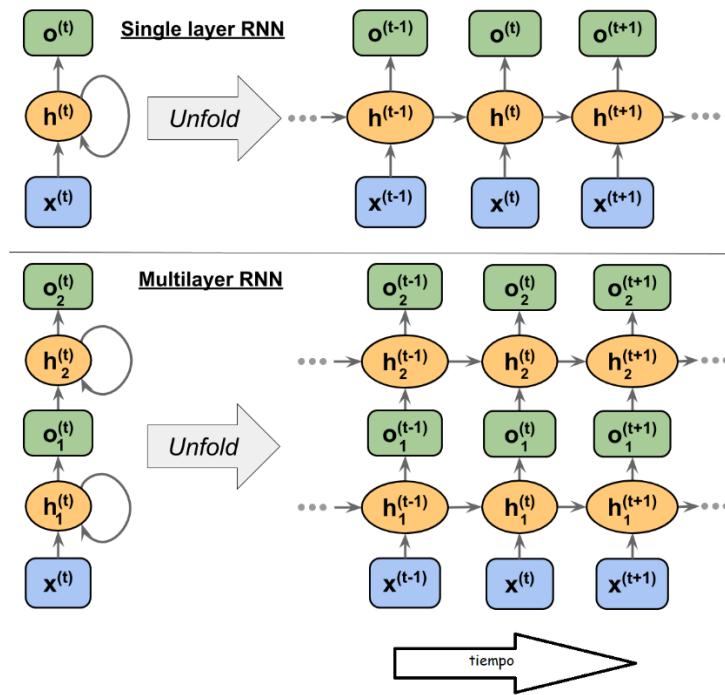
Estructura de las RNN

Las características específicas de los datos hacen que la estructura de las redes Recurrentes es específica para ser capaces de mantener la memoria de los datos ya tratados y gestionar los datos que entran nuevos.



Una RNN tendrá entradas y salidas como una red normal, pero en las capas internas, una capa recogerá datos de la capa anterior y de ella misma con los datos del paso anterior de cálculo (frame). Este bucle es el que permite la memoria en la red sobre los eventos pasados. Cada paso de tiempo se denomina frame y como en el primer frame no hay datos de entrada se inicializan éstos a cero generalmente. Cada

neurona tiene dos conjuntos de pesos, uno para las entradas actuales y otro para las entradas del paso anterior.

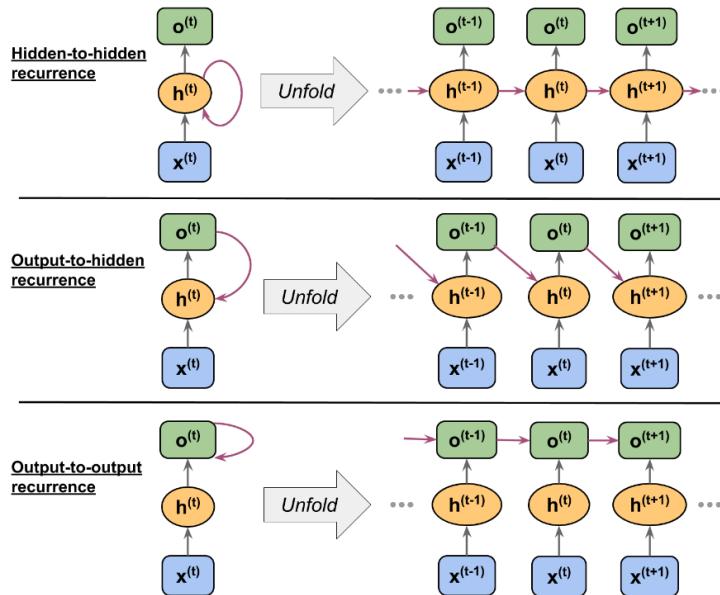


Con estas estructuras, la función de activación en una RNN es muy similar a las RN generales, pero teniendo en cuenta la entrada del tiempo. El algoritmo de aprendizaje también es similar, pero el cálculo del gradiente se hace muy complejo pudiendo aparecer problemas de desvanecimiento del gradiente o explosión del gradiente, para solucionar estos problemas se usan tres técnicas: Recorte de gradiente, TBPTT o LSTM, siendo esta última la más utilizada.

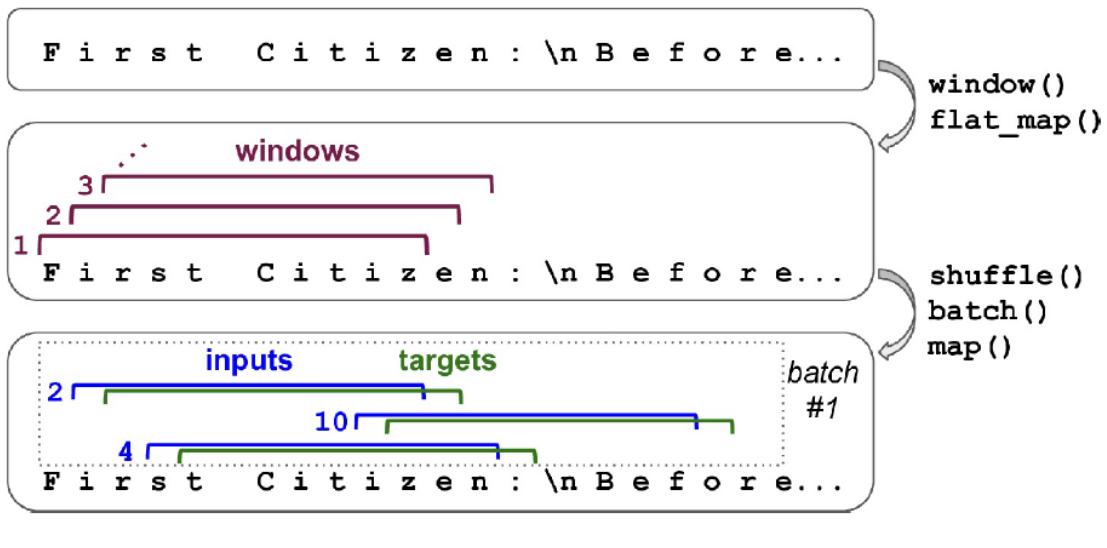
Las capas LSTM son una extensión de las RNN que básicamente amplían la memoria para aprender de experiencias importantes que han pasado anteriormente y permiten recordar sus entradas durante un largo periodo de tiempo. Esta memoria se puede ver como una celda bloqueada, donde bloqueada significa que la célula decide si almacenar o eliminar información dentro en función de la importancia que asigna a la información que está recibiendo. Estas capas son similares a la función sigmoide que van de 0 a 1 y una característica principal es que mantienen los gradientes empinados para que la precisión sea alta y el entrenamiento corto evitando los problemas principales de los gradientes.

Han aparecido también capas GRU últimamente que simplifican las LSTM y su rendimiento está a la par, pero computacionalmente son más eficientes.

Las RNN están evolucionando constantemente, y aparecen estructuras con relativa frecuencia, la explicada anteriormente es una, pero a continuación la compararemos con otra en la que el bucle de memoria se encuentra en las salidas en vez de las capas internas.



Prediciendo sentimientos con IMDB



Código 041.py

Generación de nuevo texto

Para generar texto debemos codificar el texto en forma numérica. Para esta labor podemos usar una codificación OneHot, pero cuando el espacio de entrada sea muy elevado (el número de tokens) generará Arrays dispersos de gran dimensionalidad, con lo que es poco eficiente. Otra aproximación es dar un índice a cada token y codificarlo como una array de índices cada frase, pero esta aproximación no captura las relaciones entre las palabras. La codificación que se usa en Word Embedding.

La codificación Word Embedding usa una codificación más eficiente que OneHot y densa en la que palabras similares semánticamente tienen una codificación similar a través de una capa llamada Embedding. Esta codificación tiene un hiperparámetro que se debe ajustar según el problema. Y los valores que codificarán no serán enteros si no valores entre cero y uno. Lo más interesante de esta codificación es que captura la relación entre la palabras y sus relaciones semánticas. De tal manera que podremos hacer operaciones algebraicas del siguiente tipo:

`Word_embedding(reina) = Word_embedding(rey) - Word_embedding(home) + Word_embedding(mujer).` Estas relaciones y parámetros se aprenderán por la red como lo hemos hecho hasta ahora.

No debemos olvidar que podemos usar la técnica de transferencia de aprendizaje ya que se están creando multitud de modelos pre-entranados que podremos usar en nuestros proyectos.

Modelo a nivel de carácter

El ejemplo que hemos desarrollado se basa en un nivel de carácter, en el que al modelo le daremos una secuencia de caracteres y nos predecirá cuál es el siguiente. Hay que notar la potencia del método que con caracteres va a ser capaz de crear textos.

Para entenderlo veamos un ejemplo simple. El texto de entrada será: "hola", que tendrá un conjunto de 4 caracteres ['h', 'o', 'l', 'a']. La secuencia se pasará a la red tres veces, primero h, después ho y por último hol.

Una vez entrenado, la salida de la red será una probabilidad sobre el conjunto de caracteres, del que cogeremos el que nos interese lo uniremos a la entrada original y lo volveremos a introducir a la red para generar más caracteres, hasta el número total de texto generado que nos interese.

Código 041_b.py

Predicción de las temperaturas

En este caso vamos a predecir un valor único a partir de series temporales de datos, se utiliza para modelos climáticos, predicciones de bolsa, etc.

```
import os
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

data_dir = ''
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')
# Son datos que se graban cada 10 minutos, son 144 registros por día
f = open(fname)
data = f.read()
f.close()
lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]
print(header)
print(len(lines))

# preparación de los datos
float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',') [1:]]
    float_data[i, :] = values
print(float_data)

# Unos gráficos para ver la distribución de la temperatura a lo largo
# de año y del día
temp = float_data[:, 1] # temperatura en °C
plt.plot(range(len(temp)), temp)
plt.show()
```

```

plt.plot(range(1440), temp[:1440]) # vemos 10 días
plt.show()

# normalización de los datos, recordar que la normalización se hace
sobre los datos de aprendizaje, se aplica a todos
tamanio_datos_aprendizaje = 200000
tamanio_datos_validacion = 100000
tamanio_datos_test = len(lines) - tamanio_datos_aprendizaje -
                     tamanio_datos_validacion

mean = float_data[:tamanio_datos_aprendizaje].mean(axis=0)
float_data -= mean
std = float_data[:tamanio_datos_aprendizaje].std(axis=0)
float_data /= std

# hay gran cantidad de datos crearemos un generador para reducir
memoria
def generator(data, lookback, delay, min_index, max_index = None,
shuffle=False, batch_size=128, step=6):
    """
    data. Array de datos normalizado.
    lookback. Número de entradas a tener en cuenta hacia atrás.
    delay. Número de mediciones a predecir.
    min_index and max_index. Índices para realizar el particionado.
    shuffle. Mezclar las muestras.
    batch_size. Números de muestras por mini-batch.
    step. Número de muestra a tener en cuenta conjuntas, en principio
    son 6, una hora
    """
    if max_index is None:
        max_index = len(data) - delay - 1

    i = min_index + lookback
    while True:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)
        samples = np.zeros((len(rows), lookback // step,
                           data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets

lookback = 1440
step = 6
delay = 144
batch_size = 128
# Particionado de datos en tres conjuntos a través de generadores
train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,

```

```

        min_index=0,
        max_index=tamanio_datos_aprendizaje,
        shuffle=True,
        step=step,
        batch_size=batch_size)
val_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=tamanio_datos_aprendizaje + 1,
                     max_index=tamanio_datos_aprendizaje +
tamanio_datos_validacion,
                     step=step,
                     batch_size=batch_size)
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=tamanio_datos_aprendizaje +
tamanio_datos_validacion + 1,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

# Número de pasos para tratar el generador de validación completamente
val_steps = (tamanio_datos_validacion + 1 - lookback)
# 300000 - 200001 - lookback
# Número de pasos para tratar el generador de test completamente
test_steps = (len(float_data) - tamanio_datos_validacion + 1 -
              lookback)
# len(float_data) - 300001 - lookback

# Un método exclusivamente estadístico
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
        print("#", end=' ')
        # print(np.mean(batch_maes))

evaluate_naive_method()
# Este proceso da un MAE of 0.29, al normalizar los datos, no son
# interpretables, los pasamos a valores reales
celsius_mae = 0.29 * std[1]
print(celsius_mae)  # 2.57°C predichos

# vamos a usar una NN básica
model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step,
float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=20,
                               validation_data=val_gen,
                               validation_steps=val_steps)

```

```

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()

# Una RNN
# El principal problema de Dropout es cómo aplicarlo
# se decubrió que la manera apropiada era utilizar el mismo patrón de
# dropuyt a cada paso en vez que de
# forma aleatoria
# Cada capa RNN tiene dos argumentos dropout: tasa de neuronas a hacer
# dropout y recurrent_dropout tasa para cada
# celda recurrente
model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()

# Una RNN bidireccional
model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                               steps_per_epoch=500,
                               epochs=40,
                               validation_data=val_gen,
                               validation_steps=val_steps)

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)

```

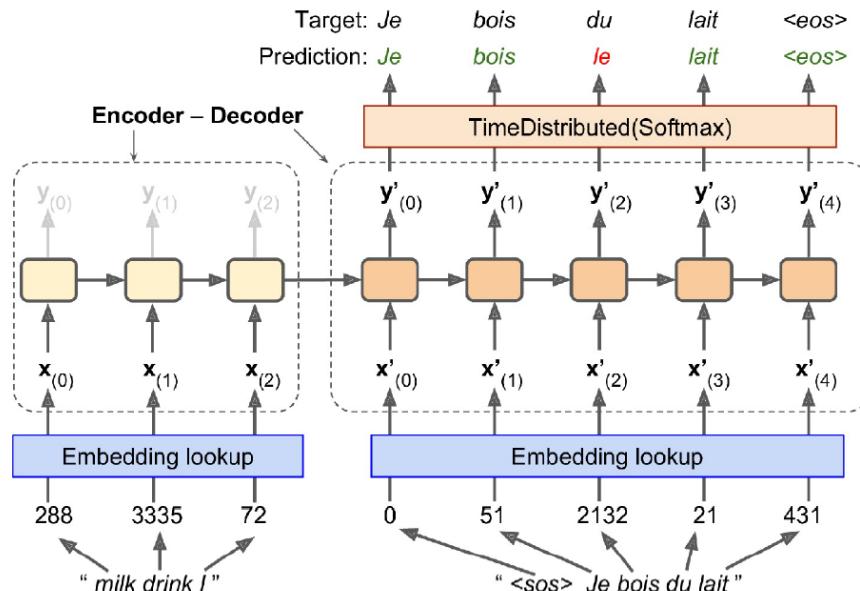
```
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

Código 041_c.py

Mejoras a realizar

- Ajustar el número de celdas en cada capa recurrente
- Ajustar la tasa de aprendizaje
- Usar LSTM en vez de GRU
- Usar una capa Densa más grande

Estructura codificador – decodificador



Una estructura de este tipo se usa para realizar transformaciones de un tipo en otro, por ejemplo, una traducción del inglés al castellano. En la fase codificación aprende la estructura del inglés que en la parte de decodificación será traducida a segundo idioma.

Ejercicios

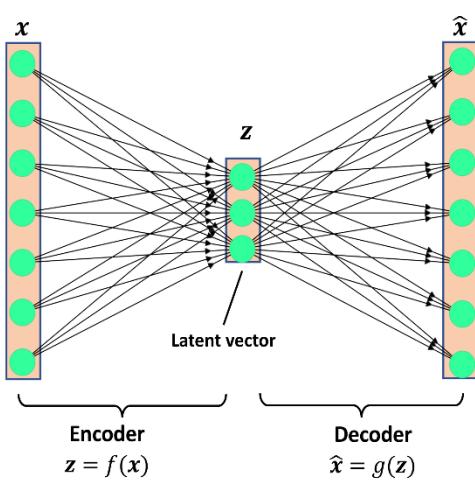
- Tratar de predecir valores en bolsa del fichero **historicos.csv** de cierre del día siguiente.
- Usar otro fichero de texto para entrenar el generador de sentencias (El quijote por ejemplo).

Cap 17.- Generación con nuevo contenido GANs

Las Redes generativas adversarias (GANs) se desarrollaron para ser capaces de generar datos nuevos a partir de un vector aleatorio de entrada. El objetivo principal es crear nuevos datos con la misma distribución que los datos con los que se hayan entrenado. La arquitectura inicial se basaba en un conjunto de redes completamente conectadas, pero sirvieron como demostración de que los conceptos funcionaban.

Hoy en día las redes GANs se utilizan para múltiples propósitos entre los que se encuentran transformaciones de imagen a imagen, aumentando de la resolución de las imágenes, reconstrucción de imágenes y muchas más.

Autoencoders



Para entender el funcionamiento de las GANs empecemos con otros elementos más simples, los autocodificadores, que pueden comprimir y descomprimir los datos. Estas estructuras están compuestas por dos redes concatenadas entre sí a través de un vector. La primera parte se llama codificador y la segunda decodificador. Generalmente el codificador recoge la entrada y la comprime en un vector latente de una dimensionalidad menor generalmente, actúa como compresor de la información (de hecho, se puede usar como técnica de reducción de la dimensionalidad). A continuación, el decodificador

recoge la información del vector latente y la vuelve a convertir a la original, descomprimiéndola. La estructura mostrada es de una única capa, pero se pueden añadir tantas capas ocultas internas como deseemos siempre que el número de valores que genera la última capa del codificador coincida con el número de valores esperados con la primera del decodificador. Una mejora muy apreciable en estos autocodificadores es usar capas convolucionales para el tratamiento de imágenes.

Modelos generativos para crear nuevos datos

Los autocodificadores son capaces de reconstruir los datos de entrada a partir de los datos comprimidos con una menor dimensionalidad, pero son incapaces de generar nuevos datos. Un modelo generativo, es capaz de generar nuevos datos a partir de un vector de entrada con una dimensionalidad mucho menor que los datos generados. Generalmente los datos de entrada estarán dentro de un rango (-1..1) para usar tangente hiperbólica y deberán ajustarse a una distribución (por ejemplo la normal).

Para crear esta posibilidad podemos generalizar el modelo de autocodificador a una estructura denominada VAE. En esta estructura se recibe un ejemplo de entrada y el codificador es modificado para que calcule la media y la varianza de dicho ejemplo. Durante el entrenamiento la red es forzada para que encaje con los ejemplos de entrada según la distribución normal (media cero, varianza uno). Una vez entrenado, se desecha el codificador y se usa el decodificador solo, proporcionándole el vector latente de forma aleatoria.

Entendiendo la estructura de una GAN

Para entender la estructura de una red GANs vamos a asumir primero que la red recibe un vector de entrada aleatorio (z) de una dimensionalidad mucho menor que los datos de salida (x), ajustándose a una distribución conocida (La normal generalmente). Para centrar los ejemplos asumiremos también que vamos a trabajar con imágenes.

Inicializaremos la red con valores aleatorios en los pesos de la misma. Las primeras imágenes que se generen serán solo ruido, y a medida que se entrena la red veremos cómo se van aclarando las mismas. Una vez entrenada la entrada, necesitamos una función que nos compare la salida de esta fase con la imagen real correspondiente y sea capaz de evaluar la diferencia

(pérdida) para poder usar esos valores y cambiar los pesos de la red. Necesitamos un mecanismo que nos determine si dos imágenes son iguales o no, se denomina función asesora. La función asesora no es más que una red neuronal clasificatoria, que recoge como entrada los datos del generador y la

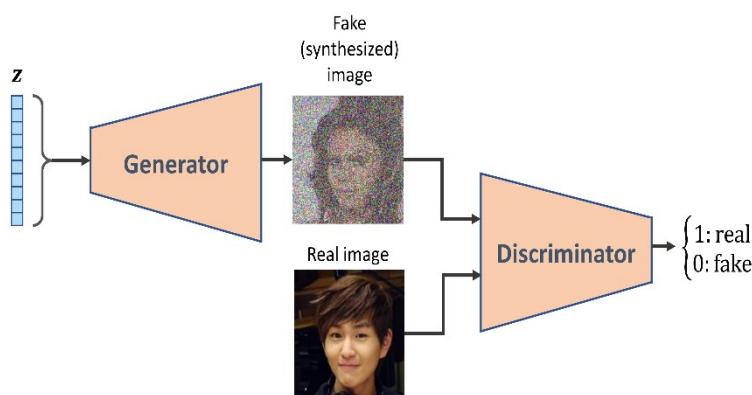
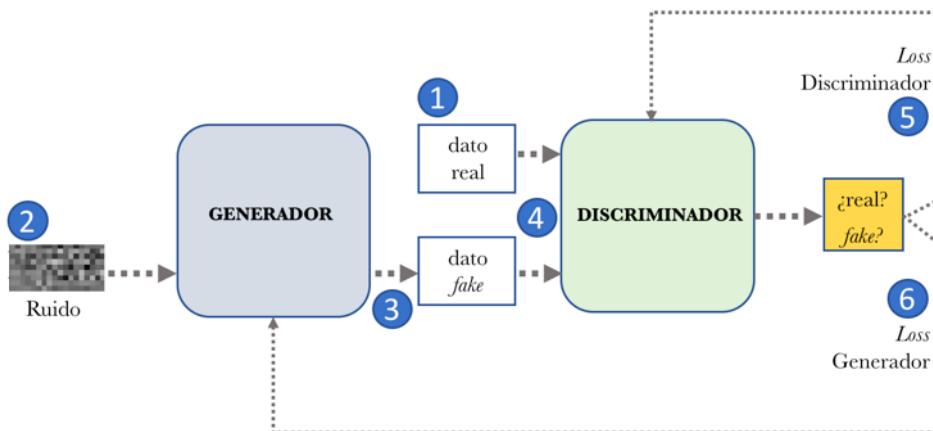


imagen real, determinando si es correcta o falsa.

Las dos redes se entrenan a la vez, de hecho, hay una lucha entre ambas redes (de ahí el nombre de adversarias) en el que el generador mejora su salida para engañar al discriminador y viceversa.

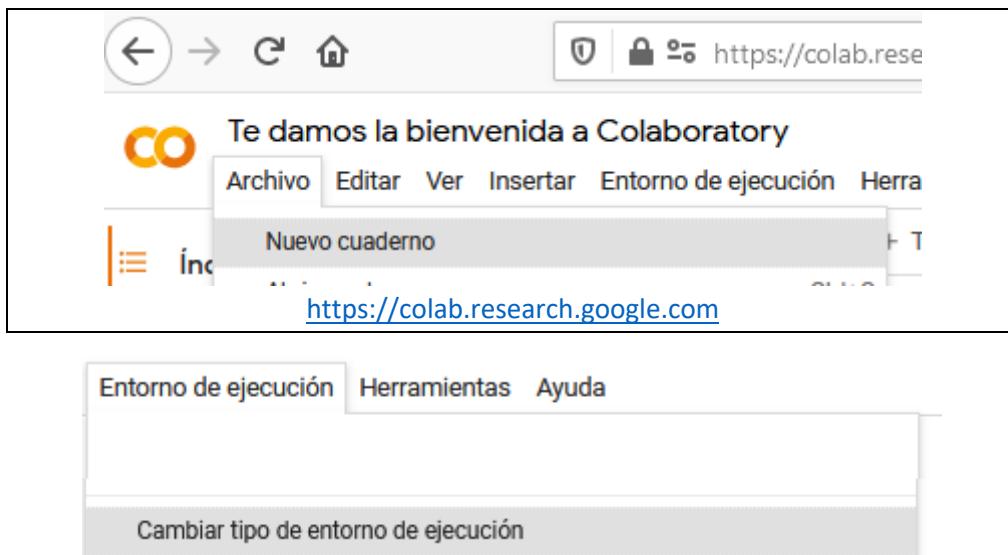


Como vemos tenemos dos conjuntos de datos, la imagen real y la imagen fake que alimentarán el discriminador, a continuación con la predicción se les pasará a dos funciones de pérdida, una para el discriminador y otra para el generador y con el resultado se realizará la retropropagación de las dos redes neuronales.

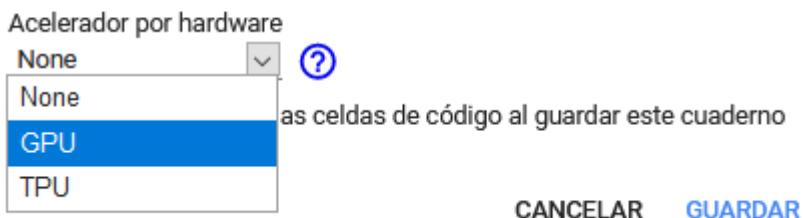
Uso de Google colab

Antes de implementar una GANs tenemos que tener algunos elementos en cuenta que son relevantes. La cantidad de cálculo para entrenar una GAN se hace muy grande, de tal manera que es difícil hacerlo en una computadora sin GPU, por lo que si es el caso, debemos utilizar

entornos virtuales que nos faciliten dicha labor. Uno de ellos es Google Colab que proporciona tiempo de cálculo gratis para Machine Learning.



Configuración del cuaderno

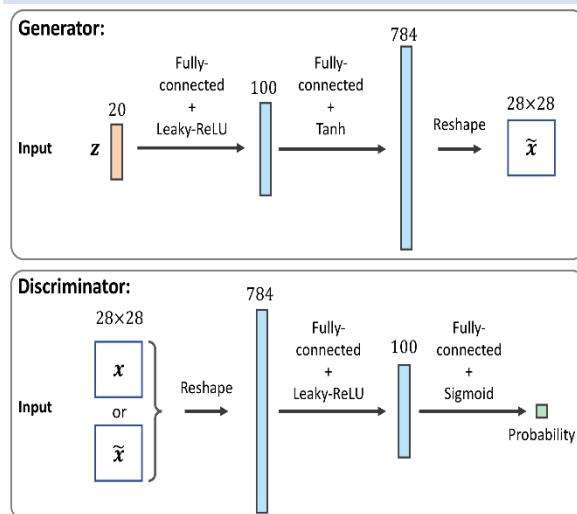


Jupyter Notebook

En los entornos colaborativos en red se usa entornos Python juntos con editores de texto, o lo que es lo mismo bloc Jupyter. Es un mecanismo que nos proporciona una interfaz gráfica simple para la programación del ML y de la escritura de texto.

<https://jupyter.org/documentation>

Ejemplo de una GANs: Caracteres MNIST



En la imagen de la izquierda vemos la red que vamos a implementar. Podemos comprobar que el tamaño del vector de entrada es de 20 y que tiene dos capas completamente conectadas ocultas, la primera de 100 y la última de 784 (28x28) el tamaño de las imágenes de la base de datos. A continuación, en el discriminador se recogen ambas imágenes y a través de dos capas se genera la probabilidad de que sea o no real, se puede usar capas Dropout pero después de las completamente conectadas y como función de la capa final se puede omitir la función sigmoide para recoger solo verdadero o falso.

El rendimiento de esta red neuronal después de 100 épocas no es muy bueno, pero se puede mejorar si usamos redes convolucionales.

```

# -*- coding: utf-8 -*-
"""cj20.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1epBiU1w5lyd3wmYizcXRTySwSGZDw
rla
"""

import tensorflow as tf
print("Versión de TensorFlow:", tf.__version__)

print("GPU Available:", tf.config.list_physical_devices('GPU'))

if tf.config.list_physical_devices('GPU'):
    device_name = tf.test.gpu_device_name()
else:
    device_name = 'cpu:0'

print("GPU device name:", device_name)
  
```

Si queremos tener acceso a nuestro drive debemos montarlo antes

```
#from google.colab import drive
#drive.mount('/content/drive/')
```

Una vez montado tendremos acceso a través de **/content/drive/My Drive**

Funciones de ayuda para generar las dos partes del modelo, el Generador y el discriminador.

```
# Commented out IPython magic to ensure Python compatibility.
import tensorflow as tf
```

```

import tensorflow_Datasets as tfds
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline

## define a function for the generator:
def make_generator_network(
    num_hidden_layers=1,
    num_hidden_units=100,
    num_output_units=784):
    model = tf.keras.Sequential()
    # conjunto de capas ocultas, la inicial al crear el modelo
    for i in range(num_hidden_layers):
        model.add(
            tf.keras.layers.Dense(
                units=num_hidden_units,
                use_bias=False)
        )
    model.add(tf.keras.layers.LeakyReLU())
    # se añade la función como capa porque solo existe de
    # esta manera en keras, no se le puede pasar como parámetro
    # obligatoriamente tiene que ser así

    # capa final, con tanh ya que las pruebas a demostrado que
    # Funciona mejor
    model.add(tf.keras.layers.Dense(
        units=num_output_units, activation='tanh'))
    return model

## define a function for the discriminator:
def make_discriminator_network(
    num_hidden_layers=1,
    num_hidden_units=100,
    num_output_units=1):
    model = tf.keras.Sequential()
    for i in range(num_hidden_layers):
        model.add(tf.keras.layers.Dense(units=num_hidden_units))
        model.add(tf.keras.layers.LeakyReLU())
        model.add(tf.keras.layers.Dropout(rate=0.5))
    model.add(tf.keras.layers.Dense(
        units=num_output_units,
        activation=None))
    return model

```

Generamos el generador (**gen_model**) y el discriminador (**disc_model**).

```

image_size = (28, 28) # Tamaño de la base de datos
z_size = 20
mode_z = 'uniform' # 'uniform' vs. 'normal'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)

gen_model = make_generator_network(
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))

```

```
gen_model.build(input_shape=(None, z_size))
gen_model.summary()

disc_model = make_discriminator_network(
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)

disc_model.build(input_shape=(None, np.prod(image_size)))
disc_model.summary()
```

El siguiente paso es usar la BBDD del MNIST para entrenamiento. Como hemos visto la capa de salida del generador usar la tangente hiperbólica (rango entre -1 y 1) mientras que las imágenes están en el rango 0 a 255, por lo que cambiaremos cada imágenes a ese rango: primero a float, a continuación al rango 0..1 y por último se desplazará al -1..1.

Además generaremos el vector z para cada imagen que se pasará al generador.

```
mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
mnist = mnist_bldr.as_dataset(shuffle_files=False)

def preprocess(ex, mode='uniform'):
    # normaliza la imagen
    image = ex['image']
    # además de convertir a float32 lo cambia al rango 0..1
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.reshape(image, [-1])
    image = image * 2 - 1.0 # convierte al rango -1..1
    # genera el vector de salida
    if mode == 'uniform': # a valores entre -1 y 1
        input_z = tf.random.uniform(
            shape=(z_size,), minval=-1.0, maxval=1.0)
    elif mode == 'normal': # de forma aleatoria
        input_z = tf.random.normal(shape=(z_size,))
    return input_z, image

# No hace falta las etiquetas ni los de prueba
mnist_trainset = mnist['train']

print('Before preprocessing: ')
example = next(iter(mnist_trainset))['image']
print('dtype: ', example.dtype, ' Min: {} Max: {}'.
      format(np.min(example), np.max(example)))

mnist_trainset = mnist_trainset.map(preprocess)

print('After preprocessing: ')
example = next(iter(mnist_trainset))[0]
print('dtype: ', example.dtype, ' Min: {} Max: {}'.
      format(np.min(example), np.max(example)))

# preparamos los lotes de 32 en 32
mnist_trainset = mnist_trainset.batch(32, drop_remainder=True)

# vamos a ver el primer ejemplo
# conseguimos el ruido generado para la imagen y la imagen real
input_z, input_real = next(iter(mnist_trainset))
print('input-z -- shape:', input_z.shape)
print('input-real -- shape:', input_real.shape)
```

```

g_output = gen_model(input_z) # Imagen de salida del generador
print('Output of G -- shape:', g_output.shape)
# calculamos las salidas para las dos imágenes
d_logits_real = disc_model(input_real)
d_logits_fake = disc_model(g_output)
print('Disc. (real) -- shape:', d_logits_real.shape)
print('Disc. (fake) -- shape:', d_logits_fake.shape)

```

La función de pérdida que usaremos se **BinaryCrossentropy** ya que el resultado que esperamos es binario: verdadero o falso.

Los vectores **d_logits_fake**, **d_logits_real** contendrán los valores predichos para la imagen generada (fake) y la imagen real

```
# programamos la red ahora, lo anterior era preparación y pruebas
import time
```

```

num_epochs = 100
batch_size = 64
image_size = (28, 28)
z_size = 20
mode_z = 'uniform'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)
np.random.seed(1)

if mode_z == 'uniform':
    fixed_z = tf.random.uniform(
        shape=(batch_size, z_size),
        minval=-1, maxval=1)
elif mode_z == 'normal':
    fixed_z = tf.random.normal(
        shape=(batch_size, z_size))

# esta función guardará la salida del generador para poder visualizarla
def create_samples(g_model, input_z):
    g_output = g_model(input_z, training=False)
    images = tf.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0

## Recogemos los datos y los preparamos
mnist_trainset = mnist['train']
mnist_trainset = mnist_trainset.map(
    lambda ex: preprocess(ex, mode=mode_z))

mnist_trainset = mnist_trainset.shuffle(10000)
mnist_trainset = mnist_trainset.batch(
    batch_size, drop_remainder=True)

## Entrenamos el modelo
with tf.device(device_name):
    gen_model = make_generator_network(
        num_hidden_layers=gen_hidden_layers,
        num_hidden_units=gen_hidden_size,
        num_output_units=np.prod(image_size))

```

```

gen_model.build(input_shape=(None, z_size))

disc_model = make_discriminator_network(
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)
disc_model.build(input_shape=(None, np.prod(image_size)))

## Funciones de pérdida, hacen falta dos, una por red:
# en este caso usaremos la misma para los dos
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
# Optimizadores, hacen falta dos, uno por red
g_optimizer = tf.keras.optimizers.Adam()
d_optimizer = tf.keras.optimizers.Adam()

all_losses = []
all_d_vals = []
epoch_samples = []

start_time = time.time()
for epoch in range(1, num_epochs+1): # tartar todas las épocas
    epoch_losses, epoch_d_vals = [], []
    # tartar todos los ejemplos
    for i,(input_z,input_real) in enumerate(mnist_trainset):
        ## Calculamos la pérdida del generador
        with tf.GradientTape() as g_tape:
            g_output = gen_model(input_z) # imagen fake
            d_logits_fake = disc_model(g_output, training=True)
            labels_real = tf.ones_like(d_logits_fake)
            g_loss = loss_fn(y_true=labels_real, y_pred=d_logits_fake)
        # calculamos el gradiente del discriminador
        g_grads = g_tape.gradient(g_loss,
                                   gen_model.trainable_variables)
        # aplicamos el gradiente al generador
        g_optimizer.apply_gradients(
            grads_and_vars=zip(g_grads,
                               gen_model.trainable_variables))

        ## Calcularmos la pérdida del discriminador
        with tf.GradientTape() as d_tape:
            d_logits_real = disc_model(input_real, training=True)
            d_labels_real = tf.ones_like(d_logits_real)
            d_loss_real = loss_fn(
                y_true=d_labels_real, y_pred=d_logits_real)
            d_logits_fake = disc_model(g_output, training=True)
            d_labels_fake = tf.zeros_like(d_logits_fake)
            d_loss_fake = loss_fn(
                y_true=d_labels_fake, y_pred=d_logits_fake)

            d_loss = d_loss_real + d_loss_fake

        ## Calculamos el gradiente del discriminador
        d_grads = d_tape.gradient(d_loss,
                                   disc_model.trainable_variables)

        ## Aplicamos los gradients al discriminador
        d_optimizer.apply_gradients(
            grads_and_vars=zip(d_grads,
                               disc_model.trainable_variables))
    # guardamos datos estadísticos para ver la mejoría
    epoch_losses.append(
        (g_loss.numpy(), d_loss.numpy(),

```

```

        d_loss_real.numpy(), d_loss_fake.numpy()))

d_probs_real = tf.reduce_mean(tf.sigmoid(d_logits_real))
d_probs_fake = tf.reduce_mean(tf.sigmoid(d_logits_fake))
epoch_d_vals.append((d_probs_real.numpy(),
                      d_probs_fake.numpy()))
all_losses.append(epoch_losses)
all_d_vals.append(epoch_d_vals)
print(
    'Epoch {:03d} | ET {:.2f} min | Avg Losses >>'
    ' G/D {:.4f}/{:.4f} [D-Real: {:.4f} D-Fake: {:.4f}]'
    .format(
        epoch, (time.time() - start_time)/60,
        *list(np.mean(all_losses[-1], axis=0))))
epoch_samples.append(
    # guardamos una imagene generada
    create_samples(gen_model, fixed_z).numpy())

```

Hemos usado `tf.GradientTape` para guardarnos los datos de gradiente y poder usarlos después y dos optimizadores Adams diferentes, uno por cada red.

El tiempo de compilación estará en torno a los 40 minutos (23 de enero de 2021). El máximo tiempo que nos permite este entorno es de 12 horas de ejecución continuada.

Desempeño del modelo

```

import pickle
#pickle.dump({'all_losses':all_losses,
#             'all_d_vals':all_d_vals,
#             'samples':epoch_samples},
#open(
#    '/content/drive/My Drive/Colab Notebooks/ch20-vanila-Learning.pkl',
#    'wb'))

#gen_model.save(
#    '/content/drive/My Drive/Colab Notebooks/ch20-vanila-gan_gen.h5')
#disc_model.save(
#    '/content/drive/My Drive/Colab Notebooks/ch20-vanila-gan_disc.h5')

import itertools

fig = plt.figure(figsize=(16, 6))

## Plotting the losses
ax = fig.add_subplot(1, 2, 1)
g_losses = [item[0] for item in itertools.chain(*all_losses)]
d_losses = [item[1]/2.0 for item in itertools.chain(*all_losses)]
plt.plot(g_losses, label='Generator loss', alpha=0.95)
plt.plot(d_losses, label='Discriminator loss', alpha=0.95)
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

epochs = np.arange(1, 101)
epoch2iter = lambda e: e*len(all_losses[-1])
epoch_ticks = [1, 20, 40, 60, 80, 100]
newpos = [epoch2iter(e) for e in epoch_ticks]
ax2 = ax.twiny()
ax2.set_xticks(newpos)

```

```

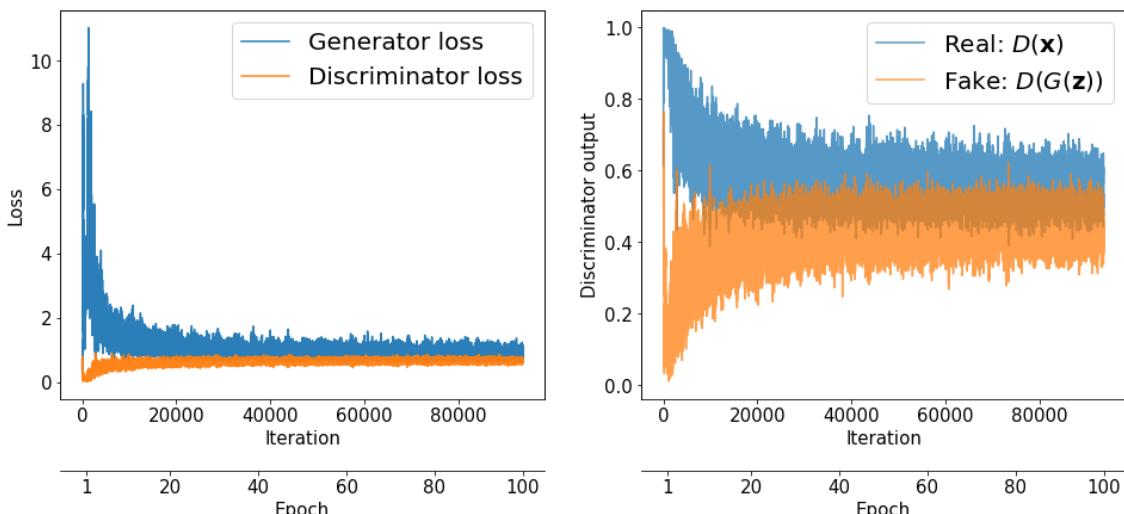
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)

## Plotting the outputs of the discriminator
ax = fig.add_subplot(1, 2, 2)
d_vals_real = [item[0] for item in itertools.chain(*all_d_vals)]
d_vals_fake = [item[1] for item in itertools.chain(*all_d_vals)]
plt.plot(d_vals_real, alpha=0.75, label=r'Real: $D(\mathbf{x})$')
plt.plot(d_vals_fake, alpha=0.75, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)

plt.show()

```



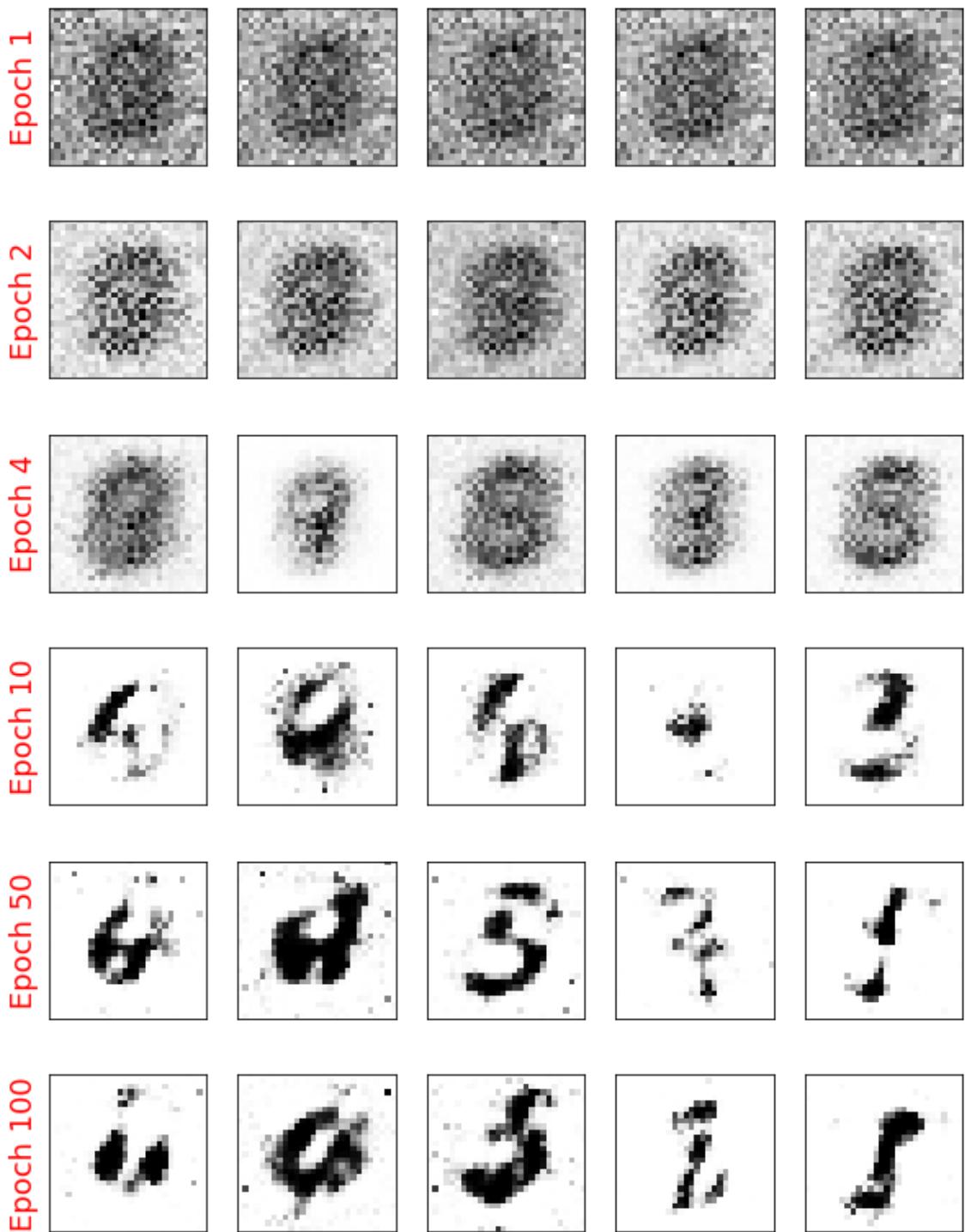
Hemos ido guardando mediante **create_samples** imágenes de muestra de las épocas para poder ver el proceso de generación.

```

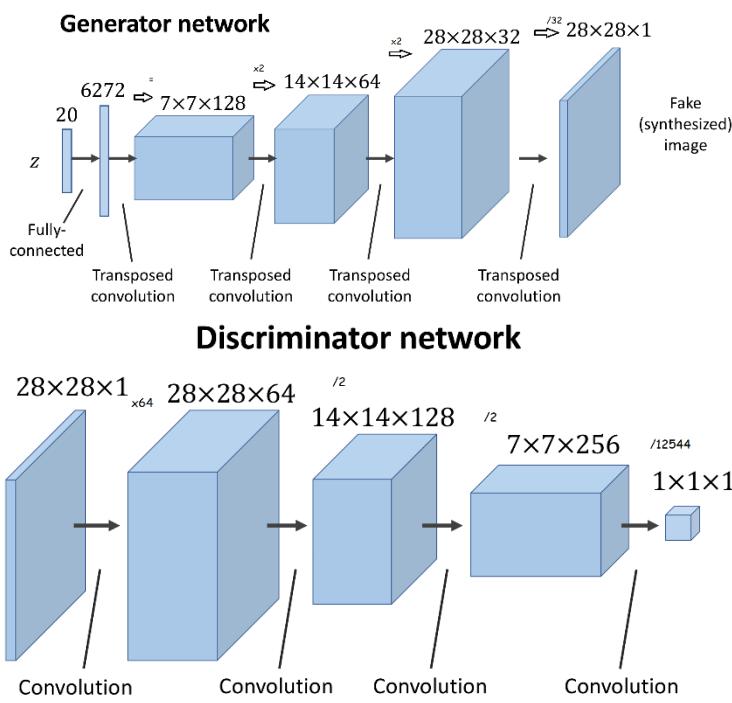
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])

```

```
if j == 0:  
    ax.text(  
        -0.06, 0.5, 'Epoch {}'.format(e),  
        rotation=90, size=18, color='red',  
        horizontalalignment='right',  
        verticalalignment='center',  
        transform=ax.transAxes)  
  
image = epoch_samples[e-1][j]  
ax.imshow(image, cmap='gray_r')  
  
plt.show()
```



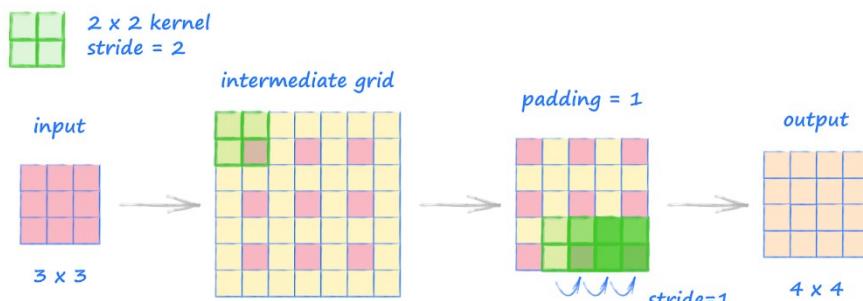
WGANS



Como vimos en el apartado anterior, la calidad de las imágenes generadas no es muy buena incluso después de 100 épocas. Para mejorar la calidad cambiaremos las capas internas por convolucionales y añadiremos unas técnicas que aumentarán el rendimiento.

El primer paso es comenzar con el vector z de la misma manera, pero añadir una capa convolucional transpuesta que aumentará la dimensionalidad a la que necesitemos, posteriormente con bloques de capas convolucionales transpuestas iremos añadiendo

más dimensionalidad y canales hasta la última en la que se reducirán el número de canales solo a uno para obtener la imagen generada. Este tipo de capas también se les llama deconvolucionales y se usan para aumentar la calidad o cantidad de los datos de entrada. Funcionan insertando ceros entre los datos para aumentar la dimensionalidad y después aplicando la operación de convolución de forma general.



La capa deconvolucional añade Padding alrededor de cada elemento para aumentar el tamaño final, en vez del parámetro Padding normal. La operación de convolución se hace de igual.

En la parte del discriminador se realizará el proceso contrario, hasta llegar a un único valor interpretable como la calidad de la imagen usando ahora capas convolucionales normales.

Este tipo de redes hacen que aparezcan diversos problemas y se usan varias técnicas para minimizarlos. La primera es la BatchNormalization que previene cambios en la distribución entre los datos de entrada y de salida de la capa haciendo mejorar la convergencia.

```
# -*- coding: utf-8 -*-
```

```
#from google.colab import drive
#drive.mount('/content/drive/')
```

```

import tensorflow as tf

print("Tensorflow version:", tf.__version__)
print("GPU Available:", tf.config.list_physical_devices('GPU'))
if tf.config.list_physical_devices('GPU'):
    device_name = tf.test.gpu_device_name()

else:
    device_name = 'CPU:0'

print("GPU device:", device_name)

# Commented out IPython magic to ensure Python compatibility.
# import tensorflow_Datasets as tfds
# import numpy as np
# import matplotlib.pyplot as plt
# %matplotlib inline

def make_dcgan_generator(
    z_size=20,
    output_size=(28, 28, 1),
    n_filters=128,
    n_blocks=2):
    size_factor = 2**n_blocks
    hidden_size = (
        output_size[0]//size_factor,
        output_size[1]//size_factor
    )

    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(z_size,)),

        tf.keras.layers.Dense(
            units=n_filters*np.prod(hidden_size),
            use_bias=False),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LeakyReLU(),
        tf.keras.layers.Reshape(
            (hidden_size[0], hidden_size[1], n_filters)),

        tf.keras.layers.Conv2DTranspose(
            filters=n_filters, kernel_size=(5, 5), strides=(1, 1),
            padding='same', use_bias=False), # con la capa siguiente
        tf.keras.layers.BatchNormalization(), # la bias es redundante
        tf.keras.layers.LeakyReLU()
    ])

    nf = n_filters
    for i in range(n_blocks):
        nf = nf // 2
        model.add(
            tf.keras.layers.Conv2DTranspose(
                filters=nf, kernel_size=(5, 5), strides=(2, 2),
                padding='same', use_bias=False))
        model.add(tf.keras.layers.BatchNormalization())
        model.add(tf.keras.layers.LeakyReLU())

    model.add(
        tf.keras.layers.Conv2DTranspose(
            filters=output_size[2], kernel_size=(5, 5),

```

```

        strides=(1, 1), padding='same', use_bias=False,
        activation='tanh'))

    return model

def make_dcgan_discriminator(
    input_size=(28, 28, 1),
    n_filters=64,
    n_blocks=2):
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=input_size),
        tf.keras.layers.Conv2D(
            filters=n_filters, kernel_size=5,
            strides=(1, 1), padding='same'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LeakyReLU()
    ])

    nf = n_filters
    for i in range(n_blocks):
        nf = nf*2
        model.add(
            tf.keras.layers.Conv2D(
                filters=nf, kernel_size=(5, 5),
                strides=(2, 2), padding='same'))
        model.add(tf.keras.layers.BatchNormalization())
        model.add(tf.keras.layers.LeakyReLU())
        model.add(tf.keras.layers.Dropout(0.3))

    model.add(tf.keras.layers.Conv2D(
        filters=1, kernel_size=(7, 7), padding='valid'))

    model.add(tf.keras.layers.Reshape((1,))) # devuelve un valor no
                                              # una probabilidad
    return model

gen_model = make_dcgan_generator()
gen_model.summary()

disc_model = make_dcgan_discriminator()
disc_model.summary()

mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
mnist = mnist_bldr.as_Dataset(shuffle_files=False)

def preprocess(ex, mode='uniform'):
    image = ex['image']
    image = tf.image.convert_image_dtype(image, tf.float32)

    image = image*2 - 1.0
    if mode == 'uniform':
        input_z = tf.random.uniform(
            shape=(z_size,), minval=-1.0, maxval=1.0)
    elif mode == 'normal':
        input_z = tf.random.normal(shape=(z_size,))
    return input_z, image

num_epochs = 100
batch_size = 128
image_size = (28, 28)

```

```

z_size = 20
mode_z = 'uniform'
lambda_gp = 10.0

tf.random.set_seed(1)
np.random.seed(1)

## Set-up the Dataset
mnist_trainset = mnist['train']
mnist_trainset = mnist_trainset.map(preprocess)

mnist_trainset = mnist_trainset.shuffle(10000)
mnist_trainset = mnist_trainset.batch(
    batch_size, drop_remainder=True)

## Set-up the model
with tf.device(device_name):
    gen_model = make_dcgan_generator()
    gen_model.build(input_shape=(None, z_size))
    gen_model.summary()

    disc_model = make_dcgan_discriminator()
    disc_model.build(input_shape=(None, np.prod(image_size)))
    disc_model.summary()

import time

## optimizers:
g_optimizer = tf.keras.optimizers.Adam(0.0002)
d_optimizer = tf.keras.optimizers.Adam(0.0002)

if mode_z == 'uniform':
    fixed_z = tf.random.uniform(
        shape=(batch_size, z_size),
        minval=-1, maxval=1)
elif mode_z == 'normal':
    fixed_z = tf.random.normal(
        shape=(batch_size, z_size))

def create_samples(g_model, input_z):
    g_output = g_model(input_z, training=False)
    images = tf.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0

all_losses = []
epoch_samples = []

start_time = time.time()

for epoch in range(1, num_epochs+1):
    epoch_losses = []
    for i,(input_z,input_real) in enumerate(mnist_trainset):

        ## Compute discriminator's loss and gradients:
        with tf.GradientTape() as d_tape, tf.GradientTape() as g_tape:
            g_output = gen_model(input_z, training=True)

            d_critics_real = disc_model(input_real, training=True)
            d_critics_fake = disc_model(g_output, training=True)

```

```

## Compute generator's loss:
g_loss = -tf.math.reduce_mean(d_critics_fake)

## Compute discriminator's losses
d_loss_real = -tf.math.reduce_mean(d_critics_real)
d_loss_fake = tf.math.reduce_mean(d_critics_fake)
d_loss = d_loss_real + d_loss_fake

## Gradient penalty:
with tf.GradientTape() as gp_tape:
    alpha = tf.random.uniform(
        shape=[d_critics_real.shape[0], 1, 1, 1],
        minval=0.0, maxval=1.0)
    interpolated = (
        alpha*input_real + (1-alpha)*g_output)
    gp_tape.watch(interpolated)
    d_critics_intp = disc_model(interpolated)

    grads_intp = gp_tape.gradient(
        d_critics_intp, [interpolated])[0]
    grads_intp_12 = tf.sqrt(
        tf.reduce_sum(tf.square(grads_intp), axis=[1, 2, 3]))
    grad_penalty = tf.reduce_mean(tf.square(
        grads_intp_12 - 1.0))

    d_loss = d_loss + lambda_gp*grad_penalty

## Optimization: Compute the gradients apply them
d_grads = d_tape.gradient(d_loss,
                           disc_model.trainable_variables)
d_optimizer.apply_gradients(
    grads_and_vars=zip(d_grads,
                       disc_model.trainable_variables))

g_grads = g_tape.gradient(g_loss,
                           gen_model.trainable_variables)
g_optimizer.apply_gradients(
    grads_and_vars=zip(g_grads,
                       gen_model.trainable_variables))

epoch_losses.append(
    (g_loss.numpy(), d_loss.numpy(),
     d_loss_real.numpy(), d_loss_fake.numpy()))

all_losses.append(epoch_losses)

print('Epoch {:3d} | ET {:.2f} min | Avg Losses >>'
      ' G/D {:.2f}/{:.2f} [D-Real: {:.2f} D-Fake: {:.2f}]'
      .format(epoch, (time.time() - start_time)/60,
              *list(np.mean(all_losses[-1], axis=0)))
)

epoch_samples.append(
    create_samples(gen_model, fixed_z).numpy()
)

# import pickle
#pickle.dump({'all_losses':all_losses,
#            'samples':epoch_samples},
#open(
# '/content/drive/My Drive/Colab Notebooks/ch20-WDCGAN-Learning.pkl',

```

```
# 'wb')))

#gen_model.save(
#/content/drive/My Drive/Colab Notebooks/ch20-WDCGAN-gan_gen.h5')
#disc_model.save(
#/content/drive/My Drive/Colab Notebooks/ch20-WDCGAN-gan_disc.h5')

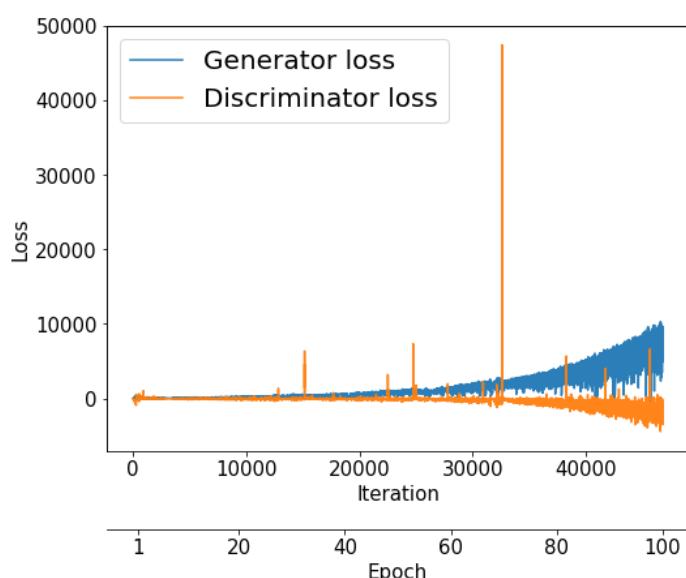
import itertools

fig = plt.figure(figsize=(8, 6))

## Plotting the losses
ax = fig.add_subplot(1, 1, 1)
g_losses = [item[0] for item in itertools.chain(*all_losses)]
d_losses = [item[1] for item in itertools.chain(*all_losses)]
plt.plot(g_losses, label='Generator loss', alpha=0.95)
plt.plot(d_losses, label='Discriminator loss', alpha=0.95)
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

epochs = np.arange(1, 101)
epoch2iter = lambda e: e*len(all_losses[-1])
epoch_ticks = [1, 20, 40, 60, 80, 100]
newpos = [epoch2iter(e) for e in epoch_ticks]
ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)

plt.show()
```



```
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
```

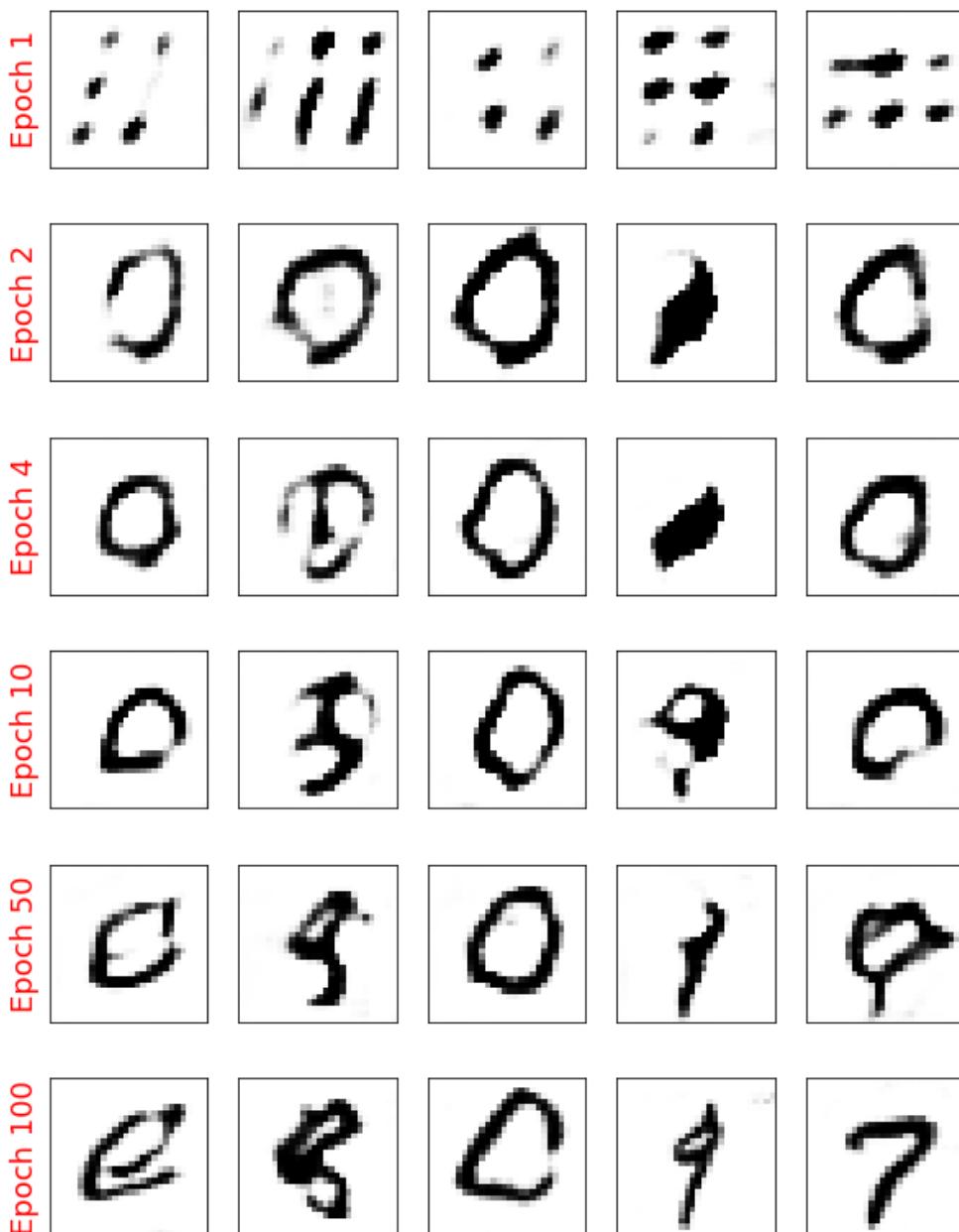
```

for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, 'Epoch {}'.format(e),
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)

        image = epoch_samples[e-1][j]
        ax.imshow(image, cmap='gray_r')

plt.show()

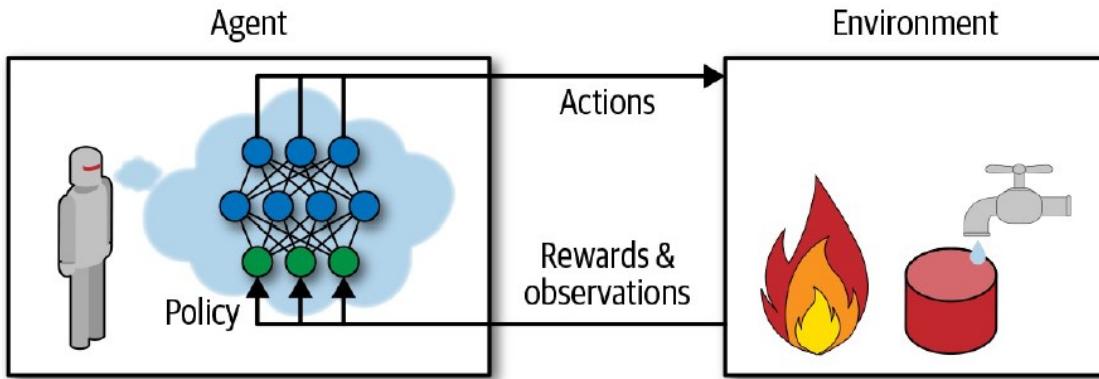
```

Imágenes generadas en los pasos intermedios

Cap 18.- Aprendizaje reforzado

En el aprendizaje reforzado un agente hace observaciones en un entorno y realiza acciones para recibir recompensa. Su objetivo es aprender a actuar de tal manera que se maximice la cantidad de recompensas a lo largo del tiempo. Dicho de otra manera, el agente actúa en el entorno y aprende por medio de prueba y error a maximizar la satisfacción y minimizar el error. Este tipo de machine Learning se puede aplicar a controlar un robot, a aprender cualquier juego como PacMan, ajedrez, etc. A hacer predicciones sobre el mercado de valores y mucho más.

El algoritmo que usa el agente para determinar la acción a realizar se denomina política, y puede ser una red neuronal o cualquier otro algoritmo que se nos ocurra.



OpenAI Gym

OpenAI Gym es una librería que se utiliza como entornos para programar a agentes y sus políticas de forma sencilla. Esta librería expone un conjunto de entornos simulados y todos los métodos para realizar observaciones, acciones e incluso visualizarlos.

```
pip install gym
```

Veamos un ejemplo.

```
env = gym.make("CartPole-v1")

# Primer intento
obs = env.reset()      # inicialización y primera observación
print(obs)   # obs tiene la primera observación de entorno
print(env.action_space)  # vemos el número de acciones posibles
env.render()    # visualizamos el entorno
action = 1
for _ in range(100):
    obs, reward, done, info = env.step(action)  # realizamos la acción
    #obs es la nueva observación reward el valor de la recompensa
    #done igual a true cuando la política no consigue el problema
    #info información extra
    env.render()
    action = 0 if action == 1 else 1  # política
```

Segundo intento.

```

def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500): # serán 500 intentos
    episode_reward = 0
    obs = env.reset()
    for step in range(200): # 200 veces o pasos por intento
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        env.render()
        episode_reward += reward
        if done:
            break
    totals.append(episode_reward)

print(np.mean(totals), np.std(totals), np.min(totals), np.max(totals))

env.close() # liberamos datos

```

Esta segunda aproximación parece que funciona mucho mejor, aunque no es perfecta.

Tercer intento.

En este ejemplo utilizaremos una red neuronal como agente. En la red solo necesitamos una neurona al final para que indique la probabilidad de la acción, por lo que usaremos la función Sigmoid como activación. Si necesitamos más de dos acciones, podemos aumentar el número de neuronas de la capa final y usar Softmax como activación.

```

import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]
model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
        # solo una neurona, probabilidad de izquierda o derecha
])

"""
El RL solo usa acciones y recompensas para aprender el modelo, y las
recompensas generalmente se dan bastante después
de tomar la acción, por lo que es difícil conocer qué acciones han dado
lugar a la recompensa y priorizarlas. Para
resolver este problema generalmente se evalúa cada acción en base a la
suma de todas las recompensas que se dan
después, aplicando un factor de descuento para primar las más cercanas.
Este factor se llama gamma.
"""

"""
Uno de los mecanismos de actualizar el modelo es a través de los
algoritmos de gradientes. El algoritmo es el siguiente
1º la red neuronal juega varias veces y en cada caso calcula el gradiente
pero no los utiliza todavía
2º Después de varias vueltas se calcula la ventaja de la acción

```

3º Si la ventaja es positiva se aplican los gradientes calculados, si la ventaja es negativa se aplican los gradientes opuestos.

4º se calcula la media de todos los gradientes y se aplica el descenso de gradiente

"""

```

def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))

    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    env.render()
    return obs, reward, done, grads

def play_multiple_episodes(env, n_episodes, n_max_steps,
                           model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads

def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(
        rewards, discount_factor)
        for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]

n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200

```

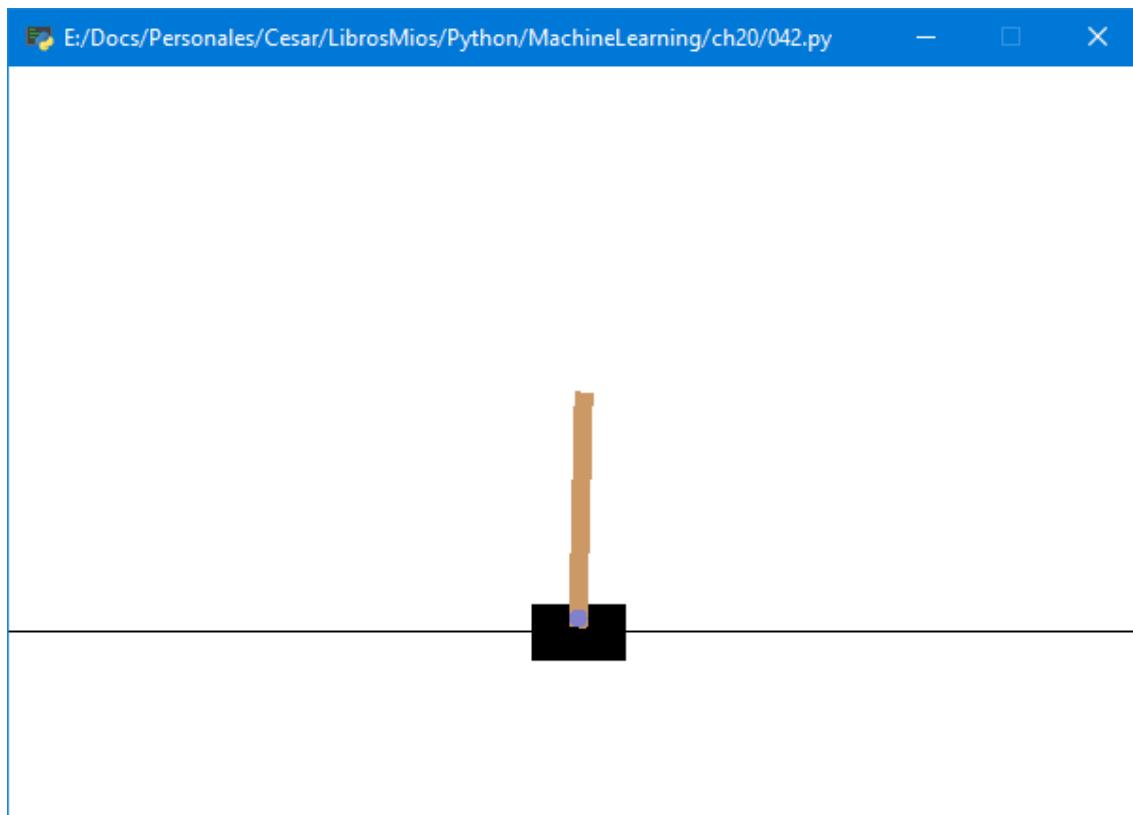
```
discount_factor = 0.95

optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)

    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean([
            final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards in enumerate(
                all_final_rewards)
            for step, final_reward in enumerate(final_rewards)])
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads,
                                  model.trainable_variables))

env.close()
```



Anexo I: Algoritmo simplificado de trabajo (Sin Redes neuronales)

- Determinar si es un problema de clasificación o regresión
- Clasificación
 - Determinar si es binaria o multiclas
 - Ver el balanceo de las clases a buscar. Ver porcentaje de cada etiqueta a clasificar y comprobar cómo están.
 - Eliminar ejemplos
 - Añadir ejemplos
 - Gestionar los datos que faltan
 - Encontrar los nulos
 - Determinar qué hacer
 - Eliminar la columna
 - Eliminar los ejemplos con nulos
 - Sustituir los valores nulos por otros
 - Ceros
 - Media
 - SimpleImputer()
 - Gestionar los extremos
 - Ver si los valores son etiquetas
 - Si lo son convertir a números
 - Codificación manual
 - LabelEncoder()
 - OneHotEncoder()
 - Analizar y entender los datos
 - Histograma
 - Matriz de covarianzas
 - Eliminar características si es necesario
 - Dividir en dos el conjunto de datos
 - train_test_split
 - Escalar los datos
 - Estandarizar y Normalizar
 - StandardScaler()
 - MinMaxScaler()
 - RobustScaler()
 - Normalizer()
 - Regularizar: L1, L2
 - Opción 1
 - Elegir la métrica y el algoritmo
 - Afinar los hiperparámetros
 - Evaluar, usar la curva de validación, ver las métricas
 - Comprobar sobreajuste → reducción de características
 - PCA
 - LDA
 - Bosques aleatorios
 - repetir
 - Repetir con otro algoritmo
 - Opción 2

- Configurar un Grid con varios algoritmos y varios hiperparámetros
- Determinar el mejor
- Entrenar el algoritmo
- Evaluar, usar la curva de validación, ver las métricas
- Comprobar sobreajuste → reducción de características
 - PCA
 - LDA
 - Bosques aleatorios
 - Repetir
- Regresión
 - Similar a la clasificación cambiando la elección métrica y algoritmos.

Anexo II: Algoritmo simplificado de trabajo (Redes neuronales)

- Determinar el problema.
 - Cuáles son los datos de entrada y qué se está intentando predecir
 - Qué tipo de problema: clasificación binaria, multiclasa, regresión escalar o vectorial, etc.
- Elegir una métrica correcta
 - Qué es una precisión adecuada
- Decidir el protocolo de evaluación
 - Holdout
 - K-Fold
- Preparar los datos
 - Deben ser tensores
 - Los valores entre [-1, 1] o [0,1]
 - Si las características no son comparables normalizar
 - Crear características nuevas a partir de las existentes si fuera necesario
- Desarrollar un modelo
 - Decidir la función de activación de la última capa
 - Decidir la función de pérdidas, debe ajustarse al problema.
 - Decidir la configuración del optimizador
- Escalar el modelo hasta el principio del overfit
 - Añadir capas
 - Añadir neuronas
 - Añadir número de épocas
- Monitorizar: loss, validation_loss para las métricas
- Afinar los hiperparámetros. En un bucle
 - Modificar el modelo
 - Entrenar con los datos de entrenamiento y validación
 - Evaluar
 - Volver al primer punto
 - Qué hacer
 - Añadir dropout
 - Cambiar la arquitectura
 - Añadir regularización
 - Cambiar los hiperparámetros
- Entrenar el modelo con todos los datos de entrenamiento
- Evaluar con los datos de test

Anexo III: Recursos

Online

- <https://scikit-learn.org/stable/tutorial/index.html>
- https://scikit-learn.org/stable/user_guide.html
- <https://www.tensorflow.org/guide/>
- https://www.tensorflow.org/api_docs/python/tf
- <https://keras.io/guides/>
- <https://keras.io/api/>
- <https://opencv.org>
- <https://towardsdatascience.com/real-time-face-recognition-an-end-to-end-project-b738bb0f7348>
- <https://towardsdatascience.com/training-a-neural-network-to-detect-gestures-with-opencv-in-python-e09b0a12bdf1>
- <https://nanonets.com/blog/how-to-do-semantic-segmentation-using-deep-Learning/>
- <https://divamgupta.com/image-segmentation/2019/06/06/deep-Learning-semantic-segmentation-keras.html>

Libros

- Python Deep Learning. Introducción práctica con Keras y Tensorflow 2 de Jordi Torres.
<https://torres.ai/python-deep-Learning/>
- Python Machine Learning 3rd Edition de Sebastian Raschka y Vahid Mirjalili
<https://github.com/rasbt/python-machine-Learning-book-3rd-edition/tree/master/>
- Deep Learning with Python de Francois Chollet
<https://github.com/fchollet/deep-Learning-with-python-notebooks>
- Hands on Machine Learning with Scikit Learn, Keras and TensorFlow 2nd Edition de Aurélien Géron
<https://github.com/ageron/handson-ml2/>
- Inteligencia artificial fácil (Machine Learning y Deep Learning prácticos) de Aurélien Vannieuwenhuyze
<https://www.ediciones-eni.com/libro/inteligencia-artificial-facil-machine-Learning-y-deep-Learning-practicos-9782409025327>

Anexo IV: Índice completo

Cap 0.- Time Line	2
Cap 1.- Introducción	3
Consideraciones ético - morales	4
IA débil vs fuerte	5
Cap 2.- Conceptos Generales y terminología	9
Qué es el Machine Learning	9
La revolución del paradigma de programación.....	11
Aplicaciones del ML.....	11
Tipos de ML según el tipo de aprendizaje.....	11
Otras clasificaciones del ML	13
Terminología y nomenclatura	14
Representación de los Datos en el ML.....	15
Los desafíos de la ML	15
Pasos para el ML.....	16
Preprocesado	16
Selección y entrenamiento del modelo predictivo	16
Evaluación del modelo y predicción.....	17
Instalación de Python	17
Cap 3.-Entrenamiento de una ML para clasificación	18
Definición formal de neurona	18
Programando un Perceptrón	19
Convergencia del aprendizaje (Adeline)	20
Descenso del gradiente: Minimizar la función de coste	20
Mejorar el descenso de gradiente con el escalado de las características	21
Descenso de gradiente estocástico (SGD).....	22
Ejercicios.....	23
Cap 4.- Explorando el scikit-learn: Clasificación.....	24
Eligiendo un algoritmo de clasificación.....	24
Entrenando un Perceptrón con scikit-learn	24
Clasificación por Regresión Logística	25
Encontrando los pesos en la función de coste.....	26
Parámetros importantes en la RL.....	27
Controlar los problemas de los modelos con la regularización	27

Máquinas de vector soporte	29
Maximizando el margen.....	29
Variable débil (ξ)	30
Parámetros importantes en las MVS	30
Resolviendo problemas no lineales con el Kernel MVS	30
Implementaciones alternativas.....	32
Órdenes de complejidad	32
Árboles de decisión	33
Ejemplo de clasificación con error de clasificación.....	35
Bosques aleatorios	35
K-vecinos (KNN).....	36
Ejercicios.....	37
Cap 5.- Preprocesado de datos	38
Problemas comunes de los datos.....	38
Procedimientos comunes.....	39
Ausencia de valores.....	42
Entendiendo scikit-learn estimator.....	43
Valores extremos	44
Outliers Buenos vs Outliers Malos	44
Outliers en 1 dimensión	44
Outliers en 2 Dimensiones	45
Outliers en N-dimensiones.....	47
Outliers con una librería.....	47
Otras estrategias para delimitar outliers.	48
Una vez detectados, ¿qué hago?	48
Ampliación.....	48
Datos categóricos.....	48
Mapeado manual	49
Mapeo con LabelEncoder.....	49
Codificación OneHot	49
Count Encoding	51
Target Encoding.....	51
CatBoost Encoding	51
Separando el conjunto de datos en datos de entrenamiento y datos de prueba	51
Normalización	51
Ajustes de problemas del modelo.....	52

Regularización L1 y L2	53
Algoritmos de selección de características	56
Eliminando características con baja varianza.....	56
Selección de características invariables	56
Evaluando la importancia de las características con bosques aleatorios	57
Cap 6.- Reducción de la dimensionalidad: Extracción.....	59
PCA (Análisis de componentes principales)	59
Compresión supervisada mediante análisis discriminante lineal (LDA).....	60
Uso del análisis de componentes principal con Kernel (mapeo no lineal)	61
Otros algoritmos de reducción de la dimensionalidad	61
Cap 7.- Buenas prácticas para la evaluación del modelo y afinado de hiperparámetros	62
Herramientas: Tuberías.....	62
Ajustado de los Hiperparámetros	62
Holdout.....	62
K-Fold.....	63
Ajustado de los Hiperparámetros con búsqueda en Grid	64
Búsqueda en Grid avanzada.....	65
Evaluar el modelo: Curvas de aprendizaje y validación	67
Curva de aprendizaje.....	67
Curva de validación	69
Evaluar el modelo: Uso de diferentes métricas	69
Matriz de confusión	69
Optimización del Recall y Precisión de un modelo	70
Evaluación del modelo: Dibujando las curvas ROC	71
Evaluar el modelo: Métricas para clasificación de múltiples clases.....	73
Evaluar el modelo: Gestión del problema de desequilibrio	73
Evaluar el modelo: Otras métricas	73
Cap 8.- Elección del algoritmo.....	74
Ejercicios.....	75
Clasificación de setas.....	75
Clasificación de asteroides	75
Cap 9.- Combinación de algoritmos en uno solo	76
Voto	76
Bagging	76
AdaBoost	77
GradientBoost y HistGradientBoostingClassifier	77

Stacking	77
Ejercicios.....	78
Más ejercicios.....	78
Cap 10.- Explorando el scikit-learn: Regresión.....	79
Regresión lineal.....	79
Métodos regularizados.....	79
Regresión polinomial.....	80
Regresión usando Bosques aleatorios	80
Regresión usando SVM	80
Ejercicios.....	80
Cap 11.- Explorando el scikit-learn: Aprendizaje no supervisado.....	81
Algoritmo K-Means	81
Algoritmo k-Means++.....	81
Algoritmo FCM	81
Elección del número de clústeres	82
Algoritmo de Árbol Jerárquico	84
Algoritmo DBSCAN	84
Otros algoritmos	85
Cap 12.- Redes neuronales y Aprendizaje profundo: Introducción	87
Conceptos anteriores	87
Preprocesado de datos.....	87
Reducción de la dimensionalidad.....	87
Evaluación del modelo	88
Ajuste de los hiperparámetros.....	88
Problemas en el modelo	88
Introducción	88
Primer ejemplo de NN.....	91
Hola mundo de las NN.....	93
Representación de los datos en NN	94
Caso Especial: Tratamiento de las imágenes	94
Anatomía de una red neuronal	95
Clasificación con NN	95
Implementación de un Perceptron multicapa (MLP) con Keras: Clasificación	95
Resumen.....	96
Referencia	97
Regresión con NN.....	97

Implementación de un Perceptrón multicapa (MLP) con Keras: Regresión	98
Resumen.....	98
Almacenamiento del modelo	99
Uso de llamadas externas (callbacks).....	99
Tensorboard	100
El API Funcional Keras	101
Modelado dinámico con el API	104
Afinado de los hiperparámetros	104
Parada temprana.....	105
Búsqueda manual.....	106
Cap 13.- Redes neuronales y Aprendizaje profundo: Ajuste de la red	109
Reducir el tamaño de la red	109
Desajustes en los gradientes.....	110
Inicialización Glorot y He	110
Funciones de activación que no saturan.....	110
Normalización Batch (Batch Normalización).....	111
Recorte del gradiente.....	112
Uso de capas ya entrenadas.....	112
Optimizadores rápidos	113
Comparación de los optimizadores.....	114
Momento.....	114
Gestión de la tasa de aprendizaje	115
Regularización	116
Dropout	117
Regularización Max-Norm.....	118
Resumen.....	118
Ejercicio	118
Ejercicios.....	119
Clasificación de setas.....	119
Clasificación de asteroides	119
Kaggle	119
Cap 14.- Introducción a Tensorflow	120
Cómo funciona	120
Manejo inicial de TensorFlow	121
Personalizando Keras con Tensorflow	125
Personalización de la métrica.....	126

Personalización de capas	126
Creación de bloques de capas.....	128
Un ejemplo completo: XOR	128
Manejo de datos con Tensorflow.....	129
Acceso a varios ficheros	130
Datasets existentes en Tensorflow	131
Funciones con Tensorflow.....	132
Cálculo de gradientes bajo Tensorflow	133
Personalizando la evaluación del modelo.....	134
Estimadores (tf.estimator)	135
Ejemplo completo con ML con Tensorflow.....	137
Procesamiento con hardware avanzado con Tensorflow	138
Uso de GPU.....	138
Uso de TPUs.....	138
Uso de la nube.....	138
Cap 15.-Tratamiento de imágenes: Redes Convolucionales (CNN)	139
Un ejemplo básico.....	140
La operación de Convolución.....	141
Resumen de funcionamiento	144
Conceptos matemáticos (en 1D por simplicidad)	144
Submuestreo (Pooling).....	146
Imágenes a color	147
Extendiendo el modelo	147
Gestión de los Hiperparámetros	148
Tamaño y número de filtros.....	148
Padding.....	148
Stride	149
Regularización: Dropout y L1, L2.....	149
Regularización: BatchNormalization	149
Regularización: Decaimiento del ratio de aprendizaje	149
Funciones de pérdida a usar	150
Resumen de parámetros en cada bloque	151
Ejemplos	152
MNIST Caracteres numéricos.....	152
CelebA (Fotos de personas).....	152
Gatos y perros	152

Visualización de filtros.....	153
Transferencia del aprendizaje	156
Transformación de imágenes.....	157
Uso de modelos ya entrenados.....	160
Transferencia del aprendizaje	160
Modelos implementados en Keras.....	162
Clasificación y localización de objetos	162
Redes Convolucionales Completas.....	163
Segmentación semántica	163
Ejercicios.....	164
Gatos y perros con transferencia de conocimiento.....	164
Reconocimiento de objetos	164
Cap 16.-Tratamiento del lenguaje natural con RNN	165
Estructura de las RNN	165
Prediciendo sentimientos con IMDB.....	167
Generación de nuevo texto.....	167
Modelo a nivel de carácter.....	168
Predicción de las temperaturas	168
Estructura codificador – decodificador	172
Ejercicios.....	173
Cap 17.- Generación con nuevo contenido GANs.....	174
Autoencoders.....	174
Modelos generativos para crear nuevos datos.....	174
Entendiendo la estructura de una GAN	175
Uso de Google colab	175
Jupyter Notebook.....	176
Ejemplo de una GANs: Caracteres MNIST.....	177
WGANS.....	186
Cap 18.- Aprendizaje reforzado.....	193
OpenAI Gym	193
Anexo I: Algoritmo simplificado de trabajo (Sin Redes neuronales).....	197
Anexo II: Algoritmo simplificado de trabajo (Redes neuronales)	199
Anexo III: Recursos	200
Online	200
Libros	200
Anexo IV: Índice completo	201

