Entrenamiento de redes neuronales. Parte 2

TÉCNICAS ESTADÍSTICAS PARA EL APRENDIZAJE II

Máster Universitario en Estadística Computacional
y Ciencia de Datos para la Toma de Decisiones

Entrenamiento de redes neuronales

Diseño y entrenamiento de una red neuronal

- Topología de la red
- Profundidad de la red
- Tamaño de la red
- Conectividad de la red
- Funciones de activación
- Modos de entrenamiento
- Preprocesamiento de los datos de entrenamiento
- Inicialización de los pesos de la red
- Tasas de aprendizaje
- Ajuste de hiperparámetros

Diseño y entrenamiento de una red neuronal

Desde el punto de vista formal, una red neuronal multicapa es un **aproximador universal** → si disponemos de un **número suficiente** de ejemplos de entrenamiento, una red multicapa de la **capacidad suficiente** será capaz de construir un modelo de los datos con los que la entrenamos. Por desgracia, no existe una definición formal de lo que resulta "suficiente".

El **entrenamiento** de redes neuronales para resolver problemas prácticos es, y tal vez lo sea siempre, **más un arte que una ciencia**.

A lo máximo que podemos aspirar en la práctica es a descubrir una serie de **criterios heurísticos** que nos ayuden a tomar decisiones con respecto al diseño de una red neuronal y de su proceso de entrenamiento. Como todas las heurísticas, normalmente ofrecen **buenos resultados** pero **no garantizan nada desde un punto de vista formal**.

Diseño y entrenamiento de una red neuronal

¿Qué aspectos debemos considerar en el diseño y entrenamiento de una red neuronal artificial?

Topología de la red: ¿Cuántas capas ocultas? ¿Cuántas neuronas por capa? ¿Qué funciones de activación para las neuronas de las distintas capas?

Optimización: ¿Qué modo de entrenamiento empleamos para ajustar los pesos de la red? ¿Cómo ajustamos los pesos de la red? ¿Con qué frecuencia? ¿Mantenemos fijas las tasas de aprendizaje? ¿Cómo inicializamos los pesos de la red?

Generalización: ¿Cómo conseguimos que la red funcione bien con datos distintos a los del conjunto de entrenamiento?

Diseño y entrenamiento de una red neuronal

Para el desarrollo de un proyecto de aprendizaje automático en general, o de *deep learning* en particular, la **estrategia recomendada** suele consistir en:

1. Establecer cuáles son las **métricas** mediante las que evaluaremos el rendimiento del sistema.

2. Poner en marcha un **prototipo** funcional completo del sistema lo antes posible.

3. **Realizar cambios** en el sistema, en función de los análisis que realicemos para determinar dónde se encuentran los principales problemas del sistema y cuáles son sus áreas potenciales de mejora.

Topología de la red

Aunque la mayor parte de las redes neuronales que se utilizan en la práctica son simples redes multicapa, existe una gran variedad de bloques que se pueden utilizar en su construcción, por ejemplo:

- Capas de salida especializadas para determinados tipos de problemas (p.ej., capas *softmax* para problemas de clasificación).
- Capas cuyos patrones de conectividad resulten adecuados para determinados tipos de aplicaciones (p.ej., capas convolutivas utilizadas en procesamiento de imágenes)
- Módulos que doten de memoria a una red neuronal (p.ej., redes recurrentes).

…

Topología de la red

Una vez establecida la **arquitectura general de la red**, el diseñador debe decidir:

- Número de capas de la red.
- Tamaño de cada una de las capas, es decir, el número de neuronas que forman parte de cada capa.
- Patrones de interconexión de las capas.

El diseño modular de los distintos bloques que pueden emplearse en la construcción de una red neuronal artificial nos ayudará a explorar **múltiples opciones de diseño**.

Profundidad de la red

Una red neuronal con **tres capas es suficiente para aproximar cualquier función**, siempre que incluyamos un número suficiente de neuronas en la capa oculta y dispongamos de un conjunto de entrenamiento suficientemente completo.

Aunque, desde los años 80, muchos investigadores se han dado cuenta de que, en general, **con dos capas ocultas una red se entrena mejor**:

Las neuronas de la **primera capa oculta** son capaces de **extraer características locales** (particionan el espacio de entrada en regiones y extraen características de esas regiones).

Las neuronas de la **segunda capa oculta** son capaces de identificar **características globales**, que suelen resultar más útiles para resolver el problema cuando se basan en las características ya extraídas por la primera capa oculta.

Profundidad de la red

Un problema resulta a menudo más fácil de resolver si se utiliza **más de una capa oculta**, donde más fácil significa que **la red aprende más rápido**.

Usando **capas adicionales**, reducimos el número de neuronas necesario para representar la función deseada y mejoramos la capacidad de generalización de nuestro modelo.

Entonces, ¿**cuántas capas ocultas** deberíamos utilizar?

En la práctica, lo que se hace es ir añadiendo capas ocultas hasta que el error sobre el conjunto de validación deje de disminuir.

Tamaño de la red

El **teorema de aproximación universal** establece que una red neuronal multicapa con un número finito de neuronas es capaz de actuar como un aproximador universal, pero no nos dice cuántas neuronas serán suficientes para que la red actúe como tal para un problema dado. Sólo nos indica que la red debe tener un número suficiente de neuronas ocultas.

Habitualmente, se **procura minimizar el número de neuronas utilizado para reducir la carga computacional** que supone el entrenamiento y uso de neuronas adicionales.

En la práctica, por suerte para nosotros, muchas veces se consiguen resultados más que aceptables utilizando un número relativamente pequeño de neuronas.

Tamaño de la red

¿Cómo sabemos si nuestra red neuronal incluye un número suficiente de neuronas ocultas?

Si el entrenamiento de la red no converge, posiblemente no consigamos el rendimiento deseado, por lo que es probable que tengamos que añadir más nodos ocultos a la red.

Si el entrenamiento de la red converge y, para nosotros, resulta fundamental reducir al máximo la carga computacional que requiere su uso (por ejemplo, para integrarla en una aplicación de dispositivos móviles alimentados por baterías), podemos ir simplificando la red a la vez que monitorizamos su rendimiento, con el objetivo de que este último no se degrade demasiado.

Conectividad de la red

¿Cómo establecemos las **conexiones entre las distintas capas** de una red multicapa?

Lo más habitual en la práctica es **conectar las capas formando una cadena**, de forma que la salida de la capa *c* se utilice como entrada de la capa *c+1*, aunque existen muchas otras opciones, por ejemplo:

En vez de que la capa *c+1* reciba únicamente la salida de la capa *c*, se puede construir una red en la que cada capa reciba, como entrada, las salidas de todas las capas que la preceden en la red.

Se puede hacer que cada unidad de una capa se conecte sólo a un subconjunto de unidades de la capa siguiente, como se hace en las redes convolutivas que se emplean para procesar imágenes.

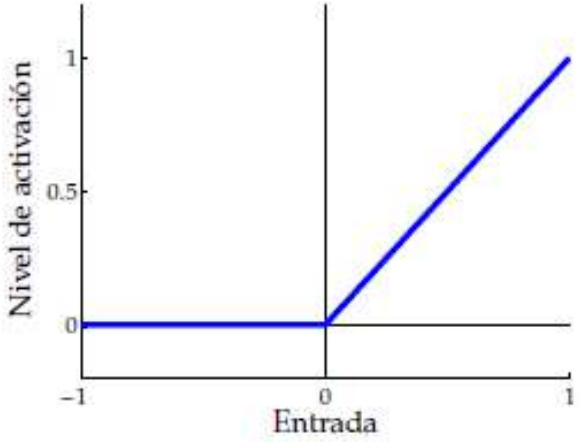
...

Funciones de activación

Como sabemos, dado que para entrenar una red neuronal utilizamos el gradiente del error y en el cálculo de dicho gradiente interviene la derivada de la **función de activación**, se emplean funciones que sean **derivables**.

En *deep learning*, es frecuente el uso de **unidades lineales rectificadas [ReLU: Rectified Linear Units]**.

$$y_j = f_{relu}(z_j) = \max(0, z_j) = \begin{cases} z_j & si \ z_j \geq 0 \\ 0 & si \ z_j < 0 \end{cases}$$



Las neuronas lineales rectificadas se limitan a realizar una combinación lineal de sus pesos y entradas, tras lo que generan una salida no lineal utilizando un umbral.

Funciones de activación

Unidades lineales rectificadas (ReLU)

Al no requerir el uso de funciones como la exponenciación de la función logística o de la tangente hiperbólica, su evaluación es mucho más eficiente y el **entrenamiento** de redes neuronales con unidades ReLU suele ser **mucho más rápido** que con neuronas sigmoidales.

Un **modelo numérico** generalmente es **más sencillo de optimizar** cuando su **comportamiento** es **lineal**, lo que sucede siempre que se utilizan unidades ReLU:

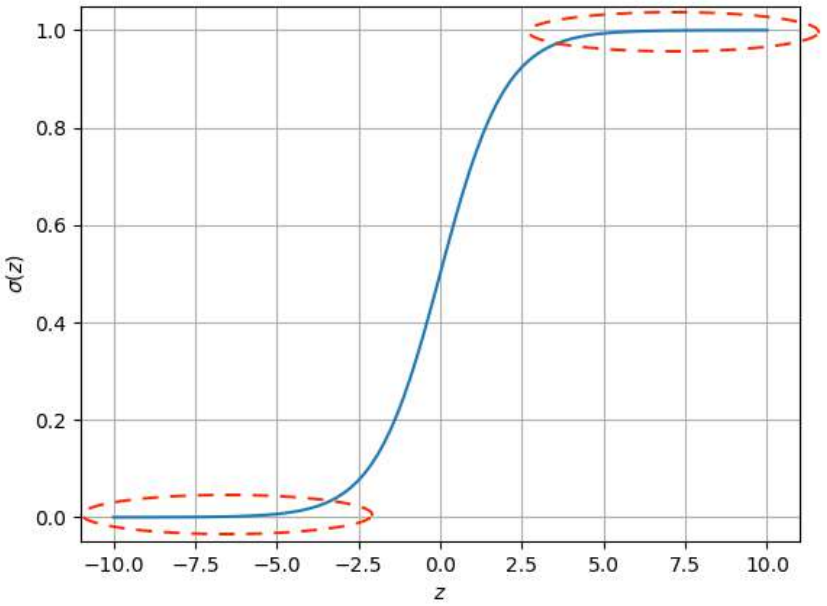
- **Cuando se activa**, una neurona ReLU es equivalente a una simple neurona lineal.
- **Cuando no se activa**, no interfiere en el comportamiento del resto de la red.

Funciones de activación

En general, **las unidades lineales rectificadas tienen una ventaja con respecto a las unidades sigmoidales** cuando forman parte de una red que se entrena con algoritmos basados en *backpropagation*:

Las unidades sigmoidales tienden a saturarse (la función sigmoidal se satura a 1 cuando la entrada (z) es muy alta, y a 0 cuando es muy baja), por lo que la derivada de su función de activación es prácticamente nula.

$$\frac{d}{dz}\sigma(z) = \sigma(z)\,(1-\sigma(z))$$



Funciones de activación

Debido a que una función sigmoidal saturada da lugar a una derivada prácticamente nula y esa derivada interviene en el cálculo del gradiente del error, **la red será incapaz de aprender correctamente los pesos** (los ajustes que realizamos son proporcionales a la derivada de su función de activación). En el mejor de los casos, aprenderá muy lentamente.

Es un problema conocido como **desaparición del gradiente o gradiente evanescente [*vanishing gradient*]**.

Si los pesos de la red aumentan en exceso durante el entrenamiento de la red (p.ej. si utilizamos una tasa de aprendizaje demasiado elevada que hace que el gradiente descendente sea inestable y diverja), los factores involucrados en el cálculo del gradiente tomarán valores mucho mayores que 1. En este caso, hablaríamos de la **explosión del gradiente [*exploding gradient*]**.

Funciones de activación

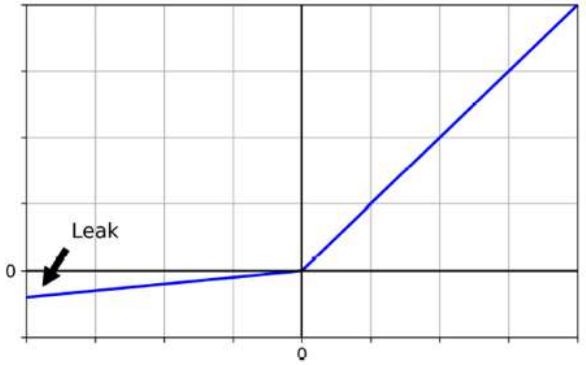
En la práctica, **¿qué tipo de neurona se emplea realmente?**

Muchas bibliotecas están optimizadas para utilizar ReLU.

Pese a esto, si al monitorizar el funcionamiento de la red, se observa que demasiadas neuronas se mantienen completamente inactivas la mayor parte del tiempo, entonces se puede recurrir a alguna de las **generalizaciones de las neuronas ReLU** que se han propuesto con el objetivo de facilitar su entrenamiento. Por ejemplo, **las unidades ReLU con pérdidas [*leaky ReLU*]:**

$\max\{0, z\} \rightarrow \max\{\alpha z, z\}$, p.ej. $\alpha \in [0.01, 0.3]$

Se suele utilizar un pequeño valor de α, sólo para evitar una derivada nula y permitir el ajuste de los pesos durante el entrenamiento de la red.



Funciones de activación

No existe ninguna teoría sólida que nos indique cómo han de escogerse las funciones de activación de las neuronas de una red multicapa. No obstante, **evitar los efectos perjudiciales de la saturación de las neuronas sigmoidales** suele resultar beneficioso.

Se pueden probar múltiples tipos de **funciones de activación** en las **capas ocultas** de una red y muchas de ellas pueden funcionar correctamente.

Muchas funciones de activación distintas a las que hemos visto pueden ser igual de válidas, aunque su uso no se haya generalizado. Hay **tantas posibilidades que ofrecen resultados comparables** a los ya conocidos, que no se suelen considerar interesantes desde el punto de vista formal.

Modos de entrenamiento

Como hemos visto, la esencia del **entrenamiento de redes neuronales** consiste en utilizar una señal de error, analizar cómo fluctúa ese error en función de los parámetros de la red y ajustar dichos parámetros con la intención de reducir el error observado, en dirección del gradiente descendente:

$$\Delta w = -\eta \nabla E$$

Expresión que se traduce en el **cálculo de derivadas parciales del error con respecto a cada uno de los parámetros de la red**.

Una de las decisiones que hemos de tomar a la hora de entrenar una red neuronal es **con qué frecuencia ajustaremos los pesos** durante el entrenamiento de la red o, dicho de otro modo, cómo estimamos el error de la red para ajustar sus parámetros.

Diseño y entrenamiento de una red neuronal

Modos de entrenamiento. Tenemos tres opciones:

Entrenamiento por lotes [*batch learning*]: se ajustan los pesos una vez por cada época. Se recorre el conjunto de entrenamiento completo, acumulando el error para cada uno de los ejemplos de entrenamiento y, al final del recorrido, se realiza una actualización de los pesos de la red.

Entrenamiento online [*online learning*]: se ajustan los pesos de la red cada vez que le mostramos a la red un ejemplo de entrenamiento.

Entrenamiento por minilotes [*mini-batch learning*]: se ajustan los pesos a partir del error estimado para una pequeña muestra del conjunto de entrenamiento elegidos al azar.

Modos de entrenamiento

En el **entrenamiento por lotes**, que sólo se realice una actualización de los pesos en cada época de entrenamiento, puede ser un inconveniente.

Para que el cálculo del gradiente del error sea fiable, se necesitará un **conjunto de entrenamiento grande**.

El **coste computacional del cálculo del gradiente** será proporcional al tamaño del conjunto de entrenamiento. Si nuestro conjunto incluye millones de ejemplos, como suele ser habitual en *big data*, el coste de una simple iteración del algoritmo puede resultar prohibitivo. Dado que, además, se necesitarán **múltiples iteraciones** para conseguir que el algoritmo converja, el coste computacional del entrenamiento de la red puede ser demasiado elevado.

Modos de entrenamiento

El **entrenamiento *online*** suele ser **mucho más rápido** que el aprendizaje por lotes, siendo al menos tan preciso como este último (especialmente, para conjuntos de datos grandes).

La estrategia incremental utilizada por el aprendizaje *online* facilita que la red pueda **adaptarse mejor a cambios** que se puedan producir **en el conjunto de datos de entrenamiento**.

Esto puede resultar de interés, por ejemplo, en el procesamiento de *data streams*, para los que podemos ir actualizando nuestro modelo cada vez que nos llegan datos nuevos.

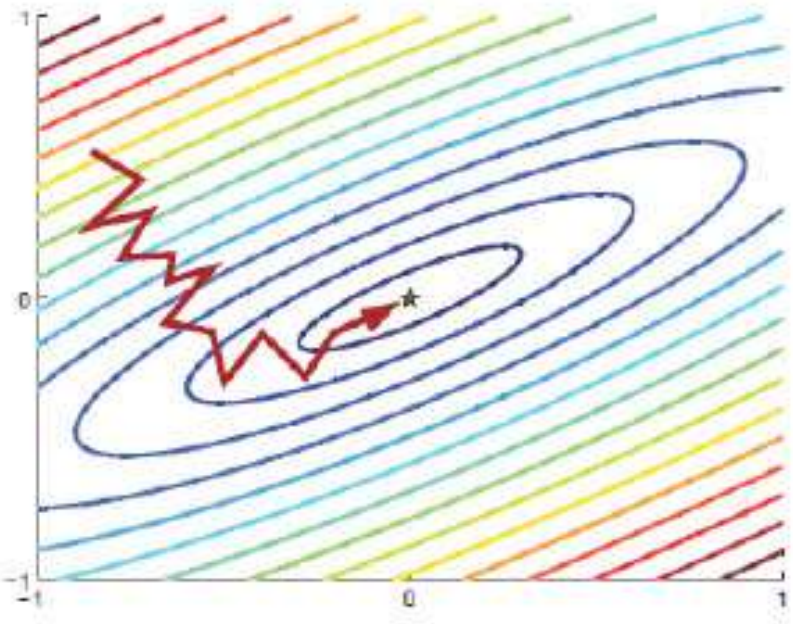
Modos de entrenamiento

Pese a las ventajas del aprendizaje *online*, existen motivos que justifican utilizar el aprendizaje por lotes:

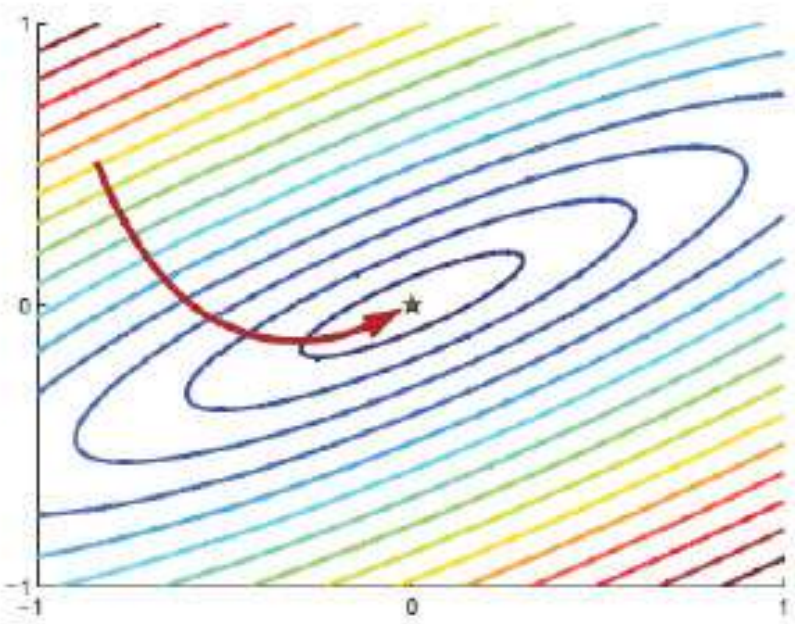
El **entrenamiento por lotes** permite realizar un **análisis formal de las propiedades del proceso de aprendizaje y de su convergencia**, algo que no se puede hacer si utilizamos aprendizaje *online*. Este análisis teórico puede resultar de interés en determinadas aplicaciones, para las que sea necesario **garantizar formalmente determinadas propiedades** del modelo que luego se utiliza en un sistema real.

Algunas técnicas de optimización sólo funcionan cuando se utiliza aprendizaje por lotes y se realiza una **estimación fiable del gradiente del error**.

Modos de entrenamiento



Aprendizaje online



Aprendizaje por lotes

Modos de entrenamiento

En nuestro algoritmo de entrenamiento, lo que hacemos al calcular el gradiente es intentar descubrir una **dirección en la que podamos mover los valores de los parámetros** de la red con la intención de **disminuir su error**.

En realidad, no necesitamos el gradiente real de la función de error. Nos basta realizar una **estimación aproximada del gradiente** para guiarnos en nuestro proceso iterativo de optimización.

Dicha aproximación se puede calcular utilizando sólo un **pequeño conjunto de muestras** (normalmente seleccionadas al azar), sin necesidad de recurrir al conjunto de entrenamiento completo cada vez que queramos darle un pequeño empujón a los parámetros de nuestra red.

Modos de entrenamiento

El **entrenamiento por minilotes** consiste en seleccionar un pequeño número k de ejemplos del conjunto de entrenamiento, lo suficientemente grande como para esperar que la estimación estocástica del gradiente sea similar al gradiente que obtendríamos utilizando todo el conjunto de entrenamiento.

El cálculo simultáneo del gradiente para los ejemplos de un minilote puede paralelizarse fácilmente. Es habitual **ajustar el tamaño k del minilote** en función de la capacidad de cálculo paralelo de la GPU de la que dispongamos, de forma que el tiempo de reloj necesario para realizar una actualización de los pesos de la red resulte prácticamente el mismo usando minilotes y sin usarlos.

En la práctica, el uso de minilotes suele funcionar mejor que el aprendizaje *online* (k=1).

Preprocesamiento de los datos de entrenamiento

Como sucede con cualquier otra técnica de aprendizaje automático, el **rendimiento de una red neuronal** depende de forma crítica del **conjunto de datos de entrenamiento** que se haya utilizado para ajustar sus parámetros.

Es fundamental que el conjunto de entrenamiento sea **completo** y se parezca a los datos con los que realmente se encontrará la red en la práctica, una vez entrenada y puesta en marcha en un entorno real.

En muchas ocasiones, las diferencias que se pueden conseguir ajustando los múltiples parámetros de un algoritmo de aprendizaje automático no son significativas en comparación con las mejoras que se pueden llegar a lograr si invertimos algo más de tiempo en la **recopilación y preparación de un conjunto de datos de entrenamiento adecuado**.

Preprocesamiento de los datos de entrenamiento

Para facilitar el entrenamiento de la red, suele ser recomendable que el **conjunto de entrenamiento** cumpla las siguientes **propiedades**:

- La media de cada variable de entrada en el conjunto de entrenamiento debería ser cercana a cero.
- La escala de las variables de entrada debería ajustarse para que sus varianzas sean similares.
- Si es posible, las variables de entrada deberían estar decorreladas (sin correlación).

Preprocesamiento de los datos de entrenamiento

Para conseguir las dos primeras propiedades, se recomienda **normalizar los datos del conjunto de entrenamiento**.

La normalización de los datos de entrada se puede realizar empleando **variables tipificadas:** $\boldsymbol{z = (x - \mu)/\sigma}$, donde x es el valor original de la variable, μ es su media en el conjunto de entrenamiento y σ su desviación.

De esta forma, cada variable del conjunto de entrenamiento pasa a tener media 0 y varianza 1.

La normalización es, pues, una forma de compensar las diferencias que pueden existir entre las diferentes variables de entrada.

Preprocesamiento de los datos de entrenamiento

Para conseguir que las **variables de entrada no** estén **correlacionadas**, hace falta recurrir a técnicas un poco más sofisticadas, como, por ejemplo, el **análisis de componentes principales** [*PCA: Principal Component Analysis*], que nos permite describir un conjunto de datos en términos de nuevas variables («componentes») no correlacionadas.

El análisis de componentes principales es útil para **reducir la dimensionalidad de los datos** (eliminando componentes con menores *eigenvalues* nos quedamos con aquellos componentes que explican la mayor parte de la variación observada en los datos de entrenamiento).

Inicialización de los pesos de la red

Pese a lo que pueda parecer, una **inicialización correcta de la red** puede suponer la diferencia entre el éxito o el fracaso del algoritmo utilizado para su entrenamiento.

Si, por ejemplo, dentro de una misma capa, dos neuronas tienen exactamente los mismos pesos serán incapaces de aprender características diferentes. Podríamos sustituirlas por una única neurona y obtendríamos el mismo resultado.

Lo recomendable es **inicializar todos los pesos de una red neuronal utilizando valores aleatorios** para romper la simetría de la red.

Inicialización de los pesos de la red

No parece importar demasiado el tipo de **distribución** utilizada para generar los valores iniciales de los pesos (p.ej. distribución uniforme o normal).

Pero sí se recomienda que los pesos se inicialicen con **valores pequeños**, especialmente si utilizamos funciones sigmoidales de activación, con el objetivo de evitar su saturación, que ralentizaría el entrenamiento de la red.

Como sabemos, un algoritmo de optimización como el gradiente descendente realiza pequeños cambios incrementales sobre los valores de los pesos de la red neuronal. Esto hace recomendable que inicialicemos los pesos con **valores cercanos a cero**, **tanto positivos como negativos**, para facilitar que, dada una señal de error, unos se actualicen en un sentido y otros en el sentido opuesto.

Inicialización de los pesos de la red

¿Cuál sería entonces una **buena elección para los valores iniciales de los pesos** de la red?

Lo que se suele hacer es que el **rango** de los pesos de una neurona **dependa de su número de entradas**, también conocido como ***fan-in***.

Si una neurona tiene un número n elevado de conexiones de entrada (un *fan-in* elevado), pequeños cambios en todos sus pesos pueden hacer que nos pasemos, ya que el efecto de un cambio Δw en cada uno de los pesos se traduce en un cambio de magnitud $n\Delta w$ en total cuando consideramos la entrada neta de la neurona.

Es preferible **utilizar pesos más pequeños cuanto mayor** sea n, es decir, **el *fan-in* de una neurona**, por lo que se suelen inicializar los pesos aleatoriamente de forma proporcional a $1/\sqrt{n}$.

Keras dispone de numerosos inicializadores: https://keras.io/api/layers/initializers/

Inicialización de los pesos de la red

Los criterios que hemos visto también son válidos para la inicialización de los **sesgos** de las neuronas, aunque la **estrategia más habitual** consiste en inicializar los sesgos de todas las neuronas de la red con una **constante** elegida de forma heurística e inicializar aleatoriamente sólo los pesos de la red.

La inicialización aleatoria de los pesos ya rompe la simetría de la red, lo que resulta suficiente para que dejemos que sea el gradiente descendente el responsable de establecer un valor adecuado para los sesgos de las distintas neuronas.

También es muy común **inicializar los sesgos a 0**, una inicialización que suele ser compatible con casi todas las heurísticas de inicialización de los pesos.

Tasas de aprendizaje

Recordemos que, una vez inicializados los parámetros de nuestra red neuronal artificial, tenemos que ajustarlos utilizando un algoritmo de entrenamiento. Si usamos el gradiente descendente, la **actualización de los pesos** de la red se hará de acuerdo a una expresión de la forma:

$$\Delta w = -\eta \nabla E$$

O para cada peso de la red:

$$\Delta w_{ij} = -\eta x_i \delta_j$$

donde δ_j representa la derivada parcial del error con respecto a la entrada neta z_j de la neurona j de una capa con entradas x_i. La **magnitud del ajuste** de los pesos dependerá, pues, de la magnitud del **gradiente de la señal de error**, de las **entradas recibidas** y de un parámetro del algoritmo, η , su **tasa de aprendizaje**.

Tasas de aprendizaje

En una red multicapa, los **gradientes del error** que se propagan pueden crecer (explotar) o desvanecerse (desaparecer) relativamente rápido, lo que dificulta el proceso de optimización de los parámetros de la red.

La **tasa de aprendizaje** es el parámetro clave cuyo ajuste puede ayudarnos a conseguir un funcionamiento adecuado del proceso de entrenamiento de la red, ya que determina cuánto se ajustan los pesos en cada iteración del algoritmo.

Para muchos, se trata del **hiperparámetro más importante** del algoritmo de entrenamiento de una red.

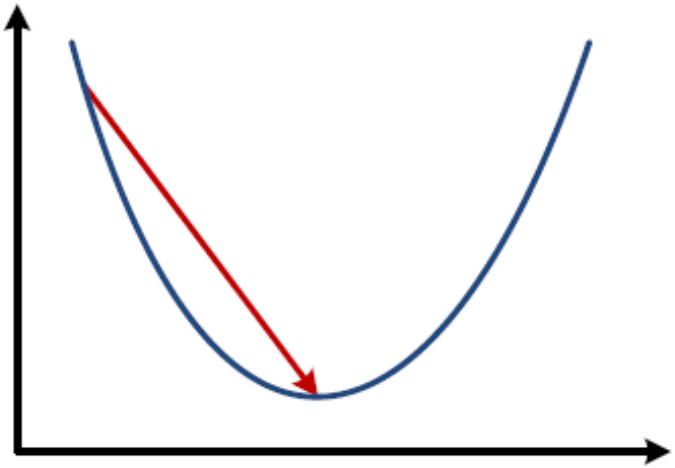
Tasas de aprendizaje

Si la tasa de aprendizaje es **demasiado pequeña**, la convergencia del algoritmo de entrenamiento de la red puede ser innecesariamente lenta al ir modificando los pesos muy lentamente.

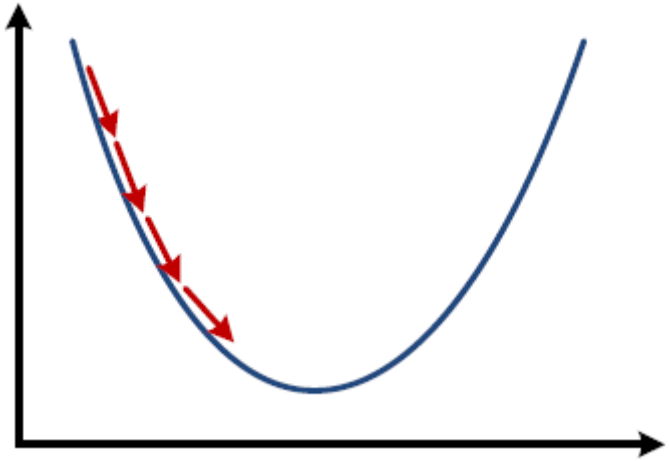
Pero, si es **demasiado elevada**, puede desencadenar oscilaciones y dar lugar a un comportamiento inestable del proceso de aprendizaje, con lo que el error no resultaría aceptable.

Como **criterio heurístico general**, nos gustaría una tasa de aprendizaje que haga que, en cada iteración, las actualizaciones de los parámetros correspondiesen a un cambio más cerca del 1% del valor del parámetro que al 50% o al 0.001%.

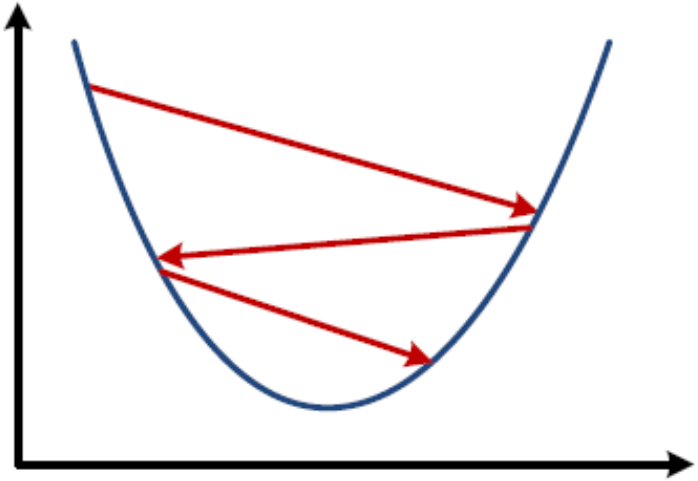
Tasas de aprendizaje. Convergencia del gradiente descendente



La **tasa de aprendizaje óptima** η^* nos lleva al mínimo de la función de error en un solo paso.

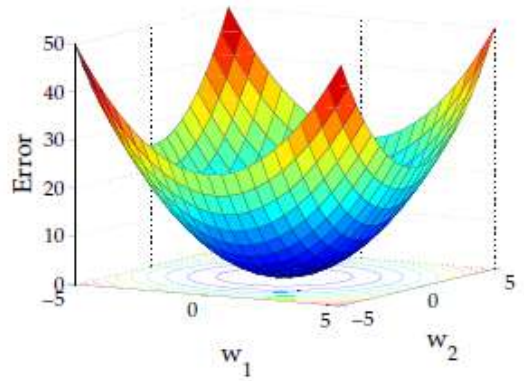


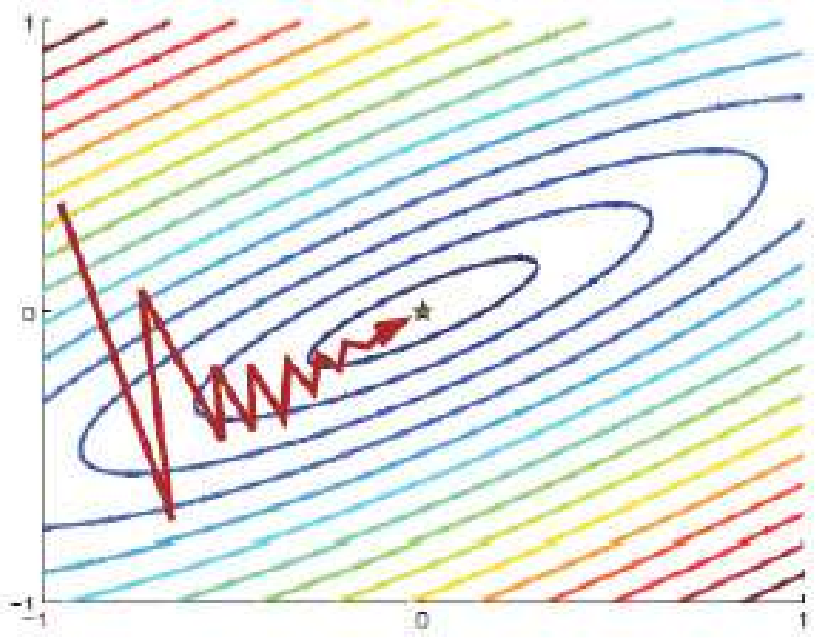
Una **tasa de aprendizaje pequeña**, $\eta < \eta^*$, ralentiza la convergencia del algoritmo.



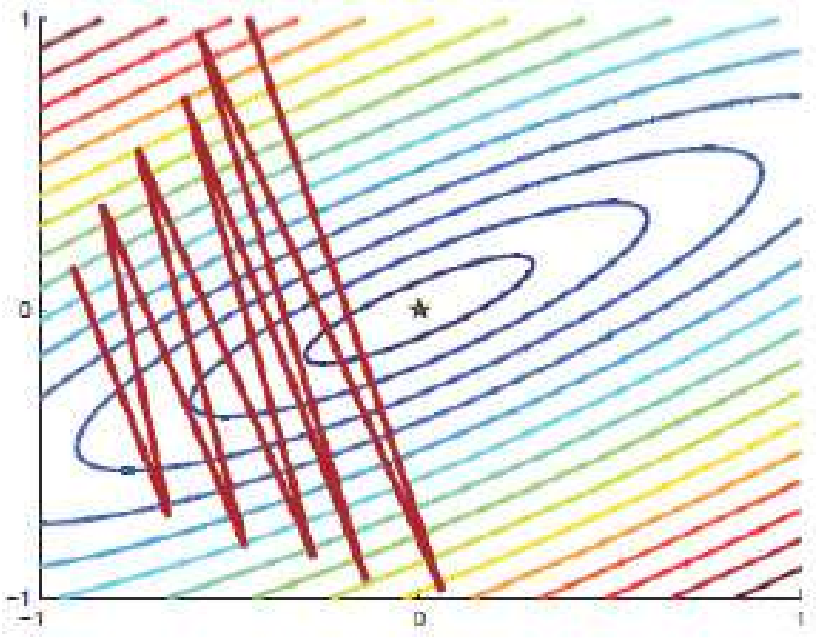
Una **tasa de aprendizaje grande**, $\eta > \eta^*$, provoca oscilaciones

Diseño y entrenamiento de una red neuronal



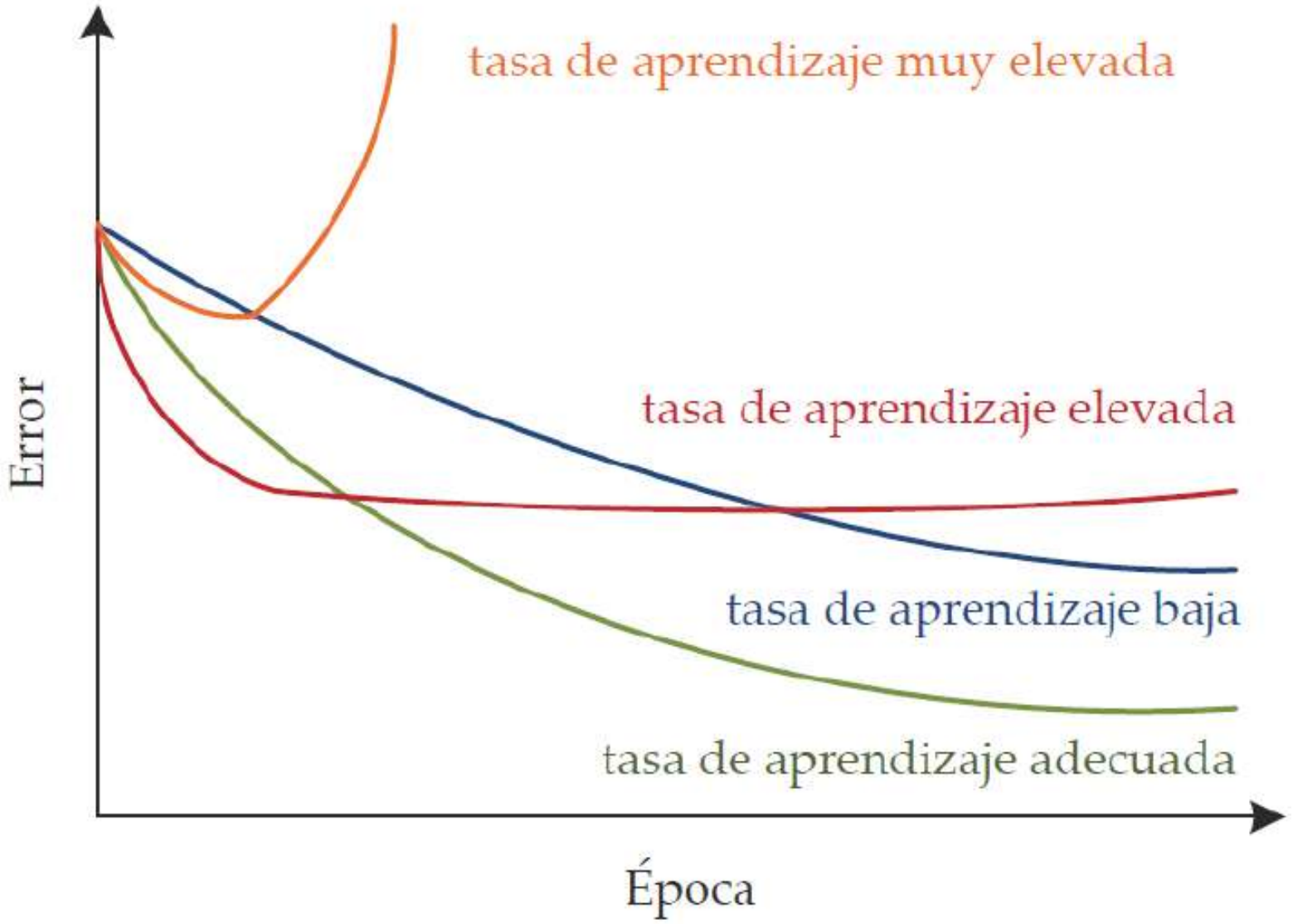


Tasa de aprendizaje adecuada (convergencia)



Tasa de aprendizaje demasiado alta (no convergencia)

Diseño y entrenamiento de una red neuronal



Gradiente descendente con diferentes tasas de aprendizaje

Tasas de aprendizaje

Algunas reglas de tipo heurístico:

Podemos comenzar con un **valor inicial de 0.1** y explorar los efectos que se producen al **doblar y dividir por dos** la tasa de aprendizaje.

Las **neuronas con muchas entradas** deberían tener una **tasa de aprendizaje menor** que las neuronas con pocas entradas, para que la velocidad del proceso de aprendizaje sea similar en todas las capas de la red (si este criterio ya lo hemos utilizado al inicializar los pesos de la red, no es necesario que lo volvamos a aplicar para establecer la tasa de aprendizaje).

La tasa de aprendizaje debería tomar **valores menores en las capas posteriores de la red** donde el gradiente del error suele ser mayor (todas las neuronas deberían aprender al mismo ritmo).

Tasas de aprendizaje

Para **igualar la velocidad del aprendizaje**, además de hacer que las tasas de aprendizaje dependan del número de entradas de cada neurona o de la capa de la red en la que se encuentre, podemos establecer alguna **estrategia que permita adaptar las tasas de aprendizaje**.

Esto es, en vez de establecer una tasa global de aprendizaje, fija para toda la red o común para todas las neuronas de una misma capa, podemos diseñar **heurísticas** que vayan adaptando la tasa de aprendizaje.

Las **tasas de aprendizaje adaptables** pueden establecerse a nivel global o localmente, con tasas específicas para cada capa, para cada neurona o, incluso, para cada peso concreto de la red.

Tasas de aprendizaje

Una estrategia muy común consiste simplemente en hacer que la tasa de aprendizaje vaya **disminuyendo progresivamente**.

Al principio, se emplea una **tasa de aprendizaje más elevada** para acelerar la reducción del error durante las primeras iteraciones del algoritmo. A continuación, **se va reduciendo la tasa de aprendizaje** conforme queremos afinar más en los valores de los parámetros de la red, reduciendo las oscilaciones causadas por una tasa de aprendizaje demasiado elevada.

Como veremos, *Keras* tiene incorporadas diferentes planificaciones para la tasa de aprendizaje basada en el tiempo. Cada planificación altera el ritmo de aprendizaje mediante unas reglas específicas: https://keras.io/api/optimizers/learning_rate_schedules/

Tasas de aprendizaje. Momentos

Además de establecer adecuadamente la escala de los pesos iniciales de la red y ajustar las tasas de aprendizaje, ya sea a nivel local o global, otra **forma común de acelerar la convergencia del algoritmo de entrenamiento** de redes neuronales consiste en el uso de **momentos**.

El uso de momentos añade un hiperparámetro más a la fórmula de **actualización de los pesos** de la red:

$$\Delta w(t) = -\eta \nabla E(w(t)) + \mu \Delta w(t-1)$$

donde el parámetro μ se denomina, incorrectamente, momento. En sentido físico, representa más bien una inercia: cuando se calcula el cambio al que debe someterse un peso de la red, se le añade una fracción del último cambio que sufrió ese parámetro.

Tasas de aprendizaje. Momentos

El uso de momentos ayuda a **mantener las modificaciones de los pesos en la misma dirección**.

Aparte de evitar cambios bruscos en la dirección de búsqueda del mínimo, sorteando muchas oscilaciones, podemos verlo como una **forma de evitar que el gradiente descendente se quede atascado** en el primer mínimo local que encuentre, por poco pronunciado que sea.

Desde el punto de vista físico, es como si dejamos caer una canica por una superficie inclinada. En su descenso llevará una velocidad que se ve afectada por la pendiente local del punto en el que nos encontremos y la fricción con la superficie. Con poca inercia, la canica se quedará en la primera irregularidad que encuentre en su recorrido. Con algo más de inercia, superará algunos baches (mínimos locales) hasta llegar a un mínimo que esperemos que resulte óptimo.

Tasas de aprendizaje. Momentos

$$\Delta w(t) = -\eta \nabla E(w(t)) + \mu \Delta w(t-1)$$

El gradiente del error modifica la velocidad como lo haría la fuerza gravitatoria que hace que una canica se deslice sobre una pendiente. El momento μ introduce fricción en el movimiento, tendiendo a reducir gradualmente su velocidad.

Si **μ = 0** estamos en el **caso típico del gradiente descendente**, que corresponde a un movimiento con fricción absoluta que se queda atascado en el primer mínimo local que nos encontremos.

Cuando **μ** toma un **valor intermedio entre 0 y 1**, la fricción del movimiento nos permitirá conservar nuestra inercia en el movimiento pero, idealmente, no tanto como para pasarnos del mínimo que deseamos encontrar.

Ajuste de hiperparámetros

Las redes neuronales artificiales, como modelos de aprendizaje automático, incluyen un gran número de parámetros que se ajustan durante su entrenamiento. Como hemos visto, el propio proceso de entrenamiento dispone además de **muchos parámetros que** también **hemos de ajustar**: los **hiperparámetros** de la red, siendo el número resultante de **posibles combinaciones** prácticamente infinito.

Para **obtener buenos resultados**, hay que dominar el arte de elegir una buena topología para la red neuronal y establecer de forma adecuada los múltiples hiperparámetros que gobiernan su entrenamiento, pero resulta **extremadamente difícil establecer recomendaciones generales.**

Veamos qué podemos hacer al respecto, manualmente y con ayuda de herramientas que automaticen parcialmente el proceso.

Ajuste de hiperparámetros. Ajuste manual

La estrategia general consiste en atacar el problema (o una versión simplificada) hasta conseguir **una versión inicial de la red** que cumpla mínimamente su función. Al menos, que obtenga mejores resultados que los que podríamos obtener al azar.

Una vez ahí, se puede comenzar un **proceso iterativo de prueba y error** en el que, monitorizando los resultados obtenidos, nos permita ir realizando **mejoras incrementales** sobre el sistema.

Dado que el proceso puede ser costoso, conviene que dispongamos de los mecanismos para **almacenar tanto los resultados que vayamos obteniendo como los parámetros** en sí de la red, para poder reutilizarlos cuando estimemos oportuno sin tener que repetir el proceso completo de entrenamiento.

Ajuste de hiperparámetros. Ajuste automático

Podemos **diseñar algoritmos que automaticen la selección de** todos aquellos **parámetros** que intervienen en el diseño y entrenamiento de una red neuronal pero de los que no se hace cargo el algoritmo de entrenamiento de la red.

Esos algoritmos se denominan técnicas de búsqueda de hiperparámetros [hyperparameter search] o, de forma más concisa, **técnicas de autoajuste [*autotuning*]**.

Desde el punto de vista del aprendizaje automático, se trata también de un **problema de aprendizaje**, salvo que ahora no estamos ajustando los parámetros de un modelo, sino los hiperparámetros que gobiernan el aprendizaje del modelo. De ahí que haya quien lo considere un problema de **meta-aprendizaje**, una **forma de aprender a aprender**.

Ajuste de hiperparámetros. Ajuste automático

Se prueban **diferentes configuraciones de los hiperparámetros**, quedándonos con la mejor configuración encontrada para los mismos, junto con el modelo entrenado usando esa configuración.

El proceso es sencillo desde el punto de vista lógico, aunque **costoso computacionalmente**, ya que cada iteración involucra entrenar una red neuronal completa. Se trata de un problema de optimización dentro de otro problema de optimización.

La optimización de los hiperparámetros es **mucho más difícil que el entrenamiento de la red** porque para el ajuste de los hiperparámetros no existe una fórmula analítica que nos indique de forma explícita en qué dirección deberíamos proseguir nuestra búsqueda, como sí sucede con la función de error que guía el entrenamiento de una red.

Ajuste de hiperparámetros. Ajuste automático

Búsqueda sistemática [*grid search*]: realiza una búsqueda exhaustiva para conjuntos de valores de los hiperparámetros preestablecidos de antemano.

Los valores seleccionados para los distintos hiperparámetros definen una **rejilla** [*grid*] **en el espacio de posibles configuraciones** y el proceso de búsqueda realiza una exploración sistemática en toda la rejilla, de ahí lo de *grid search*.

La búsqueda sistemática es el método de búsqueda de hiperparámetros **más costoso computacionalmente**, si bien es **extremadamente simple** y su **paralelización** es **trivial**.

Ajuste de hiperparámetros. Ajuste automático

Búsqueda aleatoria [*random search*]: consiste en elegir aleatoriamente distintas configuraciones para los valores de los hiperparámetros. Suele ser mucho más eficiente en la optimización de hiperparámetros que la búsqueda sistemática.

Búsqueda inteligente [*smart search*]: se guía el proceso de búsqueda de forma que no resulte completamente aleatorio, recurriendo a técnicas de optimización de hiperparámetros [*hyperparameter optimization*], también conocidas como técnicas de optimización basadas en modelos [*model-based hyperparameter optimization*]. Se construye un modelo del error observado en el conjunto de validación y se utiliza ese modelo para intentar predecir con qué valores de los hiperparámetros podemos obtener mejores resultados.

Bibliografía

•[1] Fernando Berzal. Redes Neuronales & Deep Learning. Edición independiente.

•[2] François Chollet. Deep learning with Python. Manning Shelter Island.

•[3] Ian Goodfellow, Yoshua Bengio & Aaron Courville. Deep Learning . MIT Press.