

FASTBOOK 02

El proceso de aprendizaje de una red

Introducción al Deep Learning



02. El proceso de aprendizaje de una red

Puesto que ya hemos estudiado los principales componentes del aprendizaje profundo y, concretamente, los elementos de la arquitectura *feed forward network*, en este fastbook, nos centraremos en el proceso de aprendizaje de una red de deep learning.

Cubriremos los siguientes temas.

- Los pasos de preparación de datos necesaria que son específicos a deep learning.
- Los elementos importantes en el proceso de entrenamiento para las redes neuronales.
- Los tipos de funciones de coste y cómo se usan en el proceso de aprendizaje.
- Cómo se usa la optimización y el descenso del gradiente para mejorar el aprendizaje.
- La evaluación de los resultados del entrenamiento de una red neuronal.
- Las técnicas para mejorar los resultados de un modelo.

Además, a lo largo del documento se irán compartiendo referencias a la librería de deep learning, la llamada PyTorch, para conectar cada componente con su referencia de implementación práctica.

- Preparación de los datos
- Entrenamiento
- Funciones de coste
- Optimizador y descenso del gradiente (gradient descent)
- Evaluación
- Mejorar los resultados
- Resumen y recursos de interés

Preparación de los datos



La fase de la preparación de los datos es un componente muy importante en el proceso de aprendizaje para deep learning.

Si queremos que un algoritmo de deep learning pueda aprender bien, los datos deben estar en el formato correcto y ser de buena calidad. Deep learning suele mejorar con más datos, por lo que **cuantos más datos de buena calidad estén disponibles, mejor**.

Análisis de los datos

El primer paso en esta fase es **analizar los datos para entender mejor las características**, la cantidad y la calidad de los datos que dispones. Este análisis previo es fundamental en el machine learning, a nivel general, y, por supuesto, también es un componente importante para deep learning. De hecho, podemos emplear las mismas técnicas que aplicamos en el machine learning para un problema de deep learning.

Limpieza de los datos

Un punto para tener en cuenta es que los datos del mundo real pueden contener errores, valores ausentes, ruido, datos irrelevantes... Por tanto, la fase de limpieza de datos (lo mismo que en machine learning, en general) es **muy útil para identificar esos problemas** y aplicar técnicas distintas para abordarlos.

La calidad de los datos puede tener un gran efecto en los resultados, ya que datos de poca calidad pueden conducir a resultados sin sentido.

Como se ha mencionado previamente, podemos aplicar las técnicas sobre limpieza de datos en machine learning también para un problema de deep learning.

Preprocesamiento de los datos

El paso de preprocesamiento de los datos se centra en transformar los datos al formato que necesita una red neuronal.

A diferencia de otras técnicas de machine learning, aquí deep learning aprende las características de los datos directamente de los mismos, por lo que no requiere mucha ingeniería de características.

El preprocessamiento en deep learning suele incluir estos **tres puntos destacables**:

Convertir valores categóricos

Tenemos que convertir variables que contienen valores no numéricos en valores numéricos, utilizando las técnicas que aprendimos la asignatura de machine learning.

Escalado de datos

Convertir todas las variables a la misma escala.

Aumento de datos

Aumentar artificialmente el conjunto de datos mediante la creación de nuevos ejemplos usando los datos existentes en el conjunto de datos.

Escalado de datos

Los conjuntos de datos pueden tener **escalas muy diferentes entre las distintas variables**: algunos valores pueden ser muy pequeños (0,008) y otras grandes (12.900). Pero las redes neuronales funcionan mejor cuando todos los datos están en la misma escala, por lo tanto, siempre debemos escalar los datos. Para ello, disponemos de **dos métodos**: la estandarización y la normalización.

El **método de estandarización** convierte los datos para tener una media de cero y una desviación estándar de uno.

Este proceso usa la media y la desviación estándar para escalar los datos y opera bajo el supuesto de que **los datos tienen una distribución gaussiana**. Los valores de los datos no son restringidos a un rango fijo, aunque la mayoría de los valores deben estar entre 1 y -1.

Así se realiza la estandarización:

$$x'_i = \frac{(x_i - \mu)}{\sigma}$$

x'_i : valor reescalado para el punto i en la característica x

x_i : valor original para el punto i en la característica x

μ : valor medio de la característica

σ : desviación estándar de la característica

En cuanto al **método de la normalización**, este convierte los datos para tener valores entre 0 y 1 o -1 y 1.

La normalización usa los valores mínimo y máximo de la variable para rescalar los datos. Si la distribución de los datos no es gaussiana o la desviación estándar es muy pequeña, la normalización funcionará mejor que la estandarización. Sin embargo, esta técnica es sensible a valores atípicos, así que en el caso de que haya valores atípicos grandes, elegiremos la estandarización.

Así se realiza la normalización:

$$x'_i = \frac{(x_i - \min(x))}{\max(x) - \min(x)}$$

x'_i : valor reescalado para el punto i en la característica x
 x_i : valor original para el punto i en la característica x
 $\max(x)$: valor máximo de la característica
 $\min(x)$: valor mínimo de la característica

Recuerda: la librería de Python, sklearn, tiene funciones para realizar la [normalización](#) y la [estandarización](#) de los datos.

¿Y cómo aumentamos los datos?

Ya sabemos que el rendimiento de deep learning, generalmente, mejora cuanto más dato tiene. Sin embargo, para la mayoría de los problemas, no hay una cantidad ilimitada de datos disponible. Una manera de aumentar el tamaño del conjunto de datos es crear nuevos datos utilizando los datos del conjunto de datos.

Aquí, vemos algunos ejemplos para **aplicar la técnica del aumento de datos**:

Imágenes

Recortar, voltear, rotar, agregar ruido o variación de luz a las imágenes. Esto ayuda a la red neuronal a aprender de una variedad más amplia de datos, lo que le permite generalizar mejor.

Procesamiento del lenguaje natural

Generar texto nuevo reemplazando palabras con sus sinónimos o realizando cambios en la estructura de la oración. Esto ayuda a la red neuronal a tener una mejor comprensión del idioma.

Audio

Agregar ruido o sonidos de fondo al audio o cambiar el tono o la velocidad del audio. Esto ayuda a que la red neuronal se adapte mejor a situaciones del mundo real en las que el audio puede no estar en un entorno controlado.

Fuente: Saulo Barreto. <https://www.baeldung.com/cs/ml-data-augmentation>.

Original



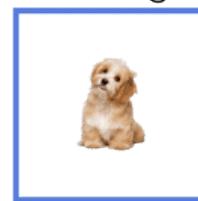
Rotation



Flip



Scaling



Brightness



PyTorch tiene varias funciones para aumentar los datos de imágenes.



Dispones de más información en la página: [transforming and augmenting images.](#)

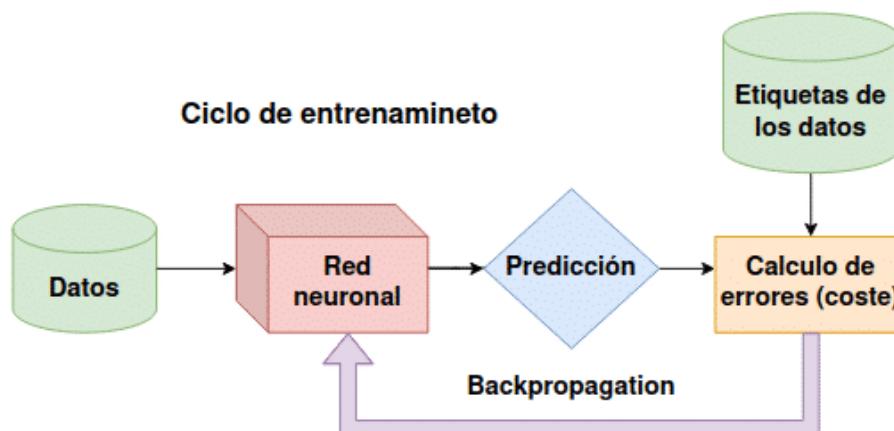
Entrenamiento



Si recuerda en qué consiste el *feed forward network*, sabrás que **la red neuronal aprende pasando los datos a través de la red**, calculando el error y luego, actualizando los pesos de la red para reducir el error. Pues bien, en este apartado, descubriremos las diferentes fases del proceso para entrenar a la red neuronal.

Resumen del entrenamiento

Las primeras veces que la red neuronal vea los datos cometerá muchos errores. ¿Qué supone esto? Que esta debe entrenarse en un proceso iterativo. Esto significa que **necesita ver los datos varias veces** para poder aprender gradualmente.



Recuerda: para que la red neuronal aprenda, tiene que ver los datos muchas veces.

En este proceso iterativo, los datos pasan a través de la red y se hacen predicciones para cada ejemplo de datos. Luego, estas predicciones **se comparan con el valor real para calcular el error**. Por ejemplo, estamos entrenando una red neuronal para predecir si una imagen es de un gato o de un perro. Si pasamos una imagen de un gato a través de la red neuronal y la predicción es ‘perro’, entonces, se genera un error. Si esta predicción es ‘gato’, entonces, no se genera ningún error.

Una vez calculados los errores, la red neuronal promedia los errores de todos los ejemplos de los datos que se han pasado y actualiza los pesos y bias mediante retropropagación para intentar reducir el error total cometido por la red neuronal. Una vez actualizados los pesos y bias, los datos se pasan otra vez, se vuelve a calcular el error y los pesos y bias se actualizan de nuevo.

La idea es que cada vez que los datos pasan a través de la red neuronal, el error que cometa esta sea más pequeño.

De esta forma, la red neuronal va mejorando progresivamente hasta llegar a un punto en el que más o menos deja de mejorar.

A continuación, explicamos algunos términos utilizados con frecuencia en este proceso de aprendizaje iterativo. Estos son hiperparámetros que solemos especificar, al entrenar una red neuronal, para indicar al algoritmo cuántos datos debe ver al mismo tiempo y cuántas veces deber ver el conjunto de datos de entrenamiento completo.

Estos enlaces tienen más información sobre el proceso de entrenamiento y construcción de modelos en PyTorch:

- Proceso de entrenamiento:
<https://PyTorch.org/tutorials/beginner/introyt/trainingyt.html>.
- Construcción de modelos:
https://PyTorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html.

Hiperparámetros para los datos y el entrenamiento

Los hiperparámetros son parámetros que se deben establecer antes de entrenar al modelo. Pueden afectar la rapidez con la que el modelo aprende y su rendimiento.

A continuación, vamos a **revisar algunos de los más importantes** durante el proceso de entrenamiento, ¡apunta!

Épocas

Una época significa que todos los datos han pasado a través de la red neuronal. Es un pase hacia adelante y hacia atrás de todos los datos de entrenamiento. Por ejemplo, si tenemos 10.000 ejemplos de datos, cada vez que todos estos 10.000 ejemplos de datos pasan a través de la red neuronal, lo llamamos época.

Batches

El deep learning requiere muchos datos, por lo tanto, si introdujéramos todos los datos a la vez en la red neuronal, consumiría una enorme cantidad de memoria. Por esta razón, normalmente dividimos los datos en *batches* más pequeños y los alimentamos por separado a la red neuronal. Por ejemplo, para un conjunto de datos de 10.000 imágenes, se podría dividir en batches de 100 imágenes. Cada *batch* se introduce en la red neuronal por separado, de modo que la red neuronal solo procesa 100 imágenes a la vez en lugar del conjunto de datos completo de 10.000 imágenes.

Batch size

Este atributo hace referencia a la cantidad de ejemplos de datos en un solo *batch*. Por ejemplo, si el tamaño del *batch* es 128, habrá 128 ejemplos de datos en cada *batch*.

Iteraciones

Este hiperparámetro recoge la cantidad de *batches* que se necesitan para completar una época. Por ejemplo, si tenemos 6.400 ejemplos de datos, en el conjunto de datos de entrenamiento, y un tamaño de *batch* de 64, entonces el número de iteraciones sería 100.

PyTorch

La biblioteca PyTorch usa funciones de [Datasets and Dataloaders](#) para dividir los datos en batches.

Funciones de coste



Una vez que hemos pasado un *batch* de datos a través de la red neuronal y hemos hecho predicciones sobre los datos, ¿cómo se calcula el error?

Hay **dos funciones importantes** relacionadas con el cálculo del error:

LA FUNCIÓN DE PÉRDIDA

LA FUNCIÓN DE COSTE

Que calcula el error para un único ejemplo de datos de entrenamiento.

LA FUNCIÓN DE PÉRDIDA

LA FUNCIÓN DE COSTE

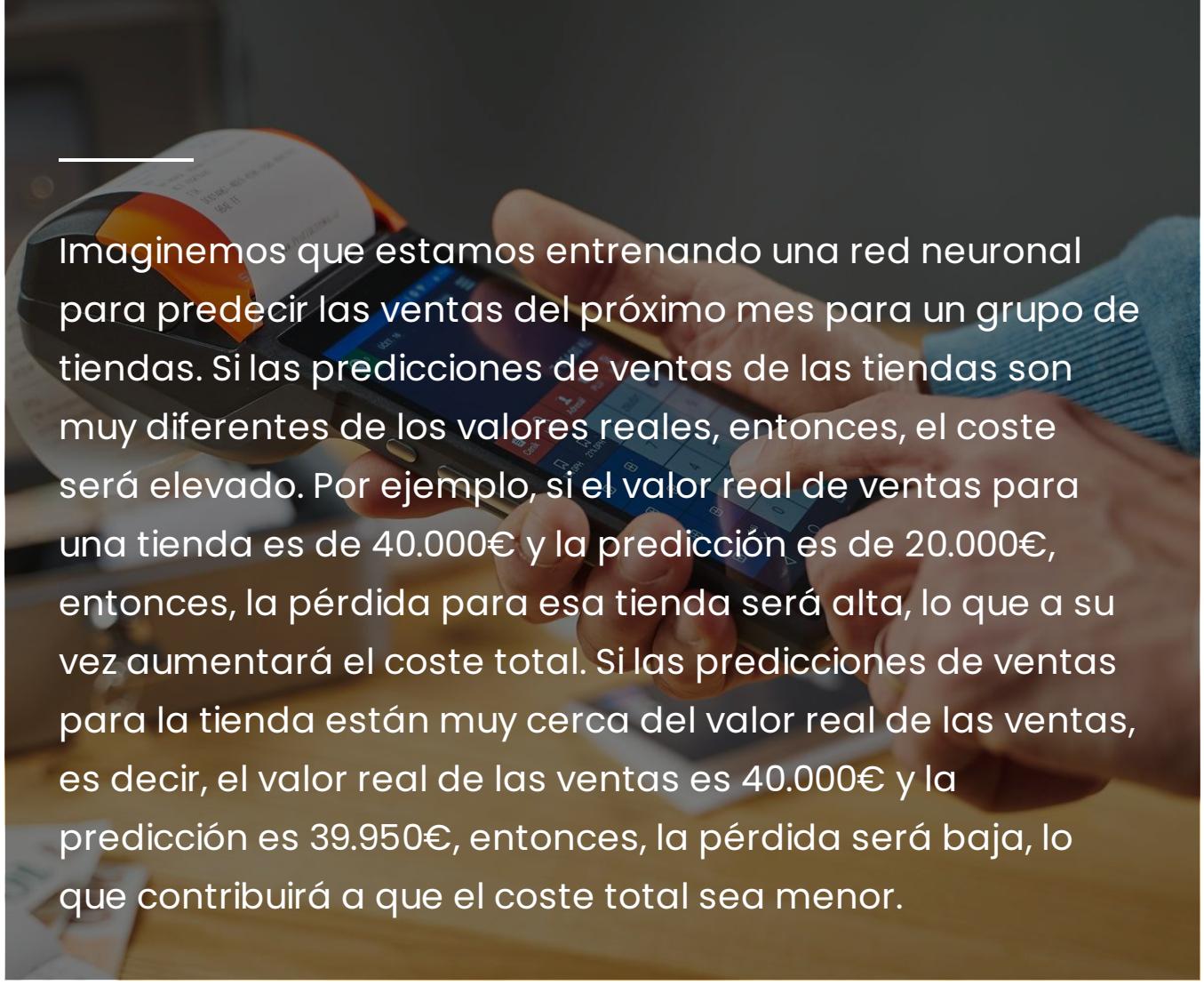
Que calcula el promedio de las funciones de pérdida de los datos de entrenamiento.

¿Qué debes recordar sobre estas funciones?

Que muchas veces los términos ‘pérdida’ y ‘coste’ se usan indistintamente para referirse a lo mismo y que dichas funciones cuantifican la diferencia entre los valores predichos y los valores reales.

Durante el entrenamiento, buscamos **minimizar la diferencia entre los valores**

predichos y reales. Por tanto, la idea es minimizar la función de coste, ya que un valor más pequeño significa menos error, lo que significa que los valores predichos se parecen más a los valores reales.



Imaginemos que estamos entrenando una red neuronal para predecir las ventas del próximo mes para un grupo de tiendas. Si las predicciones de ventas de las tiendas son muy diferentes de los valores reales, entonces, el coste será elevado. Por ejemplo, si el valor real de ventas para una tienda es de 40.000€ y la predicción es de 20.000€, entonces, la pérdida para esa tienda será alta, lo que a su vez aumentará el coste total. Si las predicciones de ventas para la tienda están muy cerca del valor real de las ventas, es decir, el valor real de las ventas es 40.000€ y la predicción es 39.950€, entonces, la pérdida será baja, lo que contribuirá a que el coste total sea menor.

Interpretación de la función de coste

La red neuronal utiliza la información de la función de coste para actualizar sus pesos y sesgos para mejorar su rendimiento.

Además, la información del coste nos permite entender el comportamiento de la red neuronal durante el entrenamiento. Por ejemplo, si el coste disminuye rápidamente, significa que la red neuronal está aprendiendo rápidamente, y si el coste disminuye muy lentamente o no disminuye en absoluto, significa que la red neuronal se ha estancado en su aprendizaje o no está aprendiendo.

Los tipos de funciones de coste

Como sabemos, la función de coste que se utilizará en un problema concreto dependerá del tipo de problema con el que nos encontramos. Un problema de clasificación, por ejemplo, tendrá una forma diferente de calcular el error que un problema de regresión. Dicho esto, primero debemos identificar qué tipo de problema es y, en base al tipo de problema, seleccionaremos la función de costes que queremos utilizar.

La siguiente tabla enumera algunas **funciones de costes comunes** para cada tipo de problema.

| Funciones de coste de regresión | Funciones de coste de clasificación |
|---------------------------------|-------------------------------------|
| Error medio cuadrado (MSE) | Entropía cruzada binaria (BCE) |
| Error absoluto medio (MAE) | Entropía cruzada categórica (CCE) |

A continuación, describimos brevemente cada una de las funciones de costes mencionadas en la tabla anterior, pero antes consulta PyTorch, ya que cuenta con varias funciones de coste que podemos usar.

Funciones de coste de regresión

El **error cuadrático medio** (MSE) es una de las métricas más populares para los problemas de regresión y se calcula tomando la diferencia entre los valores reales y los valores predichos y elevando al cuadrado esa diferencia para cada punto de datos.

Luego, se toma la suma de estas diferencias al cuadrado y se dividen por el número total de puntos de datos para calcular la media.

 Consulta la función en PyTorch para aplicar el error cuadrático medio [MSE](#).

Y aquí vemos la fórmula para calcular el error cuadrático medio:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

y_i : valor real del punto de datos

\hat{y}_i : valor predicho del punto de datos

El **error absoluto medio** (MAE) se calcula tomando el valor absoluto de la diferencia entre el valor real y el valor predicho para cada punto de datos.

Luego, se calcula la suma de todas las diferencias y esa suma se divide por el número total de puntos de datos para obtener la media. Tomamos el valor absoluto, ya que algunos puntos de datos pueden tener diferencias positivas (si los valores reales son mayores que el valor predicho) y algunos pueden tener diferencias negativas (si el valor real es menor que el valor predicho). Sin embargo, lo que nos interesa es la magnitud de la diferencia entre ellos, no si el valor de la diferencia es positivo o negativo.

 Consulta la función en PyTorch para aplicar el error absoluto medio [MAE](#).

La fórmula para calcular el error absoluto medio es:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n (|y_i - \hat{y}_i|)$$

y_i : valor real del punto de datos

\hat{y}_i : valor predicho del punto de datos

Funciones de coste de clasificación

La **entropía cruzada binaria**, también llamada ‘pérdida logarítmica’, se utiliza para calcular el coste en problemas de clasificación binaria (problemas con solo dos clases).

 Consulta la función en PyTorch para aplicar la entropía cruzada binaria [BCELoss](#).

Ya continuación, se muestra la fórmula utilizada para calcular la entropía cruzada binaria.

$$\text{Log loss} = \frac{1}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

p_i : probabilidad de que el punto de datos pertenezca a la clase 1

$1 - p_i$: probabilidad de que el punto de datos pertenezca a la clase 0

y_i : valor real de la clase (0 or 1)

Como puedes ver en la fórmula, si el valor de la clase real es uno, la segunda parte de la ecuación desaparece porque $1-y_i = 0$; y si el valor de clase real es cero, entonces la primera parte de la ecuación desaparece. Para cada punto de datos, se calcula la distancia entre el valor de clase real y la probabilidad prevista de que el punto de datos pertenezca a esa clase. Por ejemplo, para la clase 1, un punto de datos con una probabilidad de 0,94 de pertenecer a la clase 1 tendrá una distancia menor que un punto de datos con una probabilidad de solo 0,32.

La **entropía cruzada categórica** (CCE) es similar a la entropía cruzada binaria, pero se utiliza para calcular el coste en problemas de clasificación de clases múltiples.

La pérdida logarítmica tiene en cuenta M clases en lugar de solo 2 clases. Al igual que la entropía cruzada binaria, solo se calcula la distancia entre la clase real y la probabilidad predicha del punto de datos que pertenece a esa clase.



Consulta la función en PyTorch para aplicar la entropía cruzada categórica [CrossEntropyLoss](#).

Optimizador y descenso del gradiente (gradient descent)



Qualentum Lab

Llegados a este punto, pasaremos el conjunto de datos de entrenamiento a través de la red neuronal, realizaremos predicciones para cada ejemplo de datos y, luego, calcularemos el coste en función de la diferencia entre las predicciones y los valores reales. ¿Y el siguiente paso?

Que la red neuronal actualice sus pesos y bias de manera que cometa menos errores la próxima vez que haga predicciones. Ahora bien...

¿Cómo sabe la red neuronal cómo debe ajustar los pesos y sesgos para minimizar el coste?

1

Optimizadores

La red neuronal usa un optimizador para aprender los pesos y sesgos que minimiza la función de coste.

Estos optimizadores son algoritmos de optimización iterativos. Para cada paso de los datos a través de la red neuronal, el optimizador le dice a la red neuronal cómo actualizar los pesos y sesgos para reducir el coste. Entonces, para cada época de entrenamiento, el coste debería disminuir. Recuerda que la red neuronal actualiza sus pesos y bias iterativamente hasta que el coste sea cercano o igual a cero.

Por tanto, los optimizadores influyen en la rapidez con la que aprende una red neuronal y en el rendimiento final de la red neuronal.

2

Descenso del gradiente

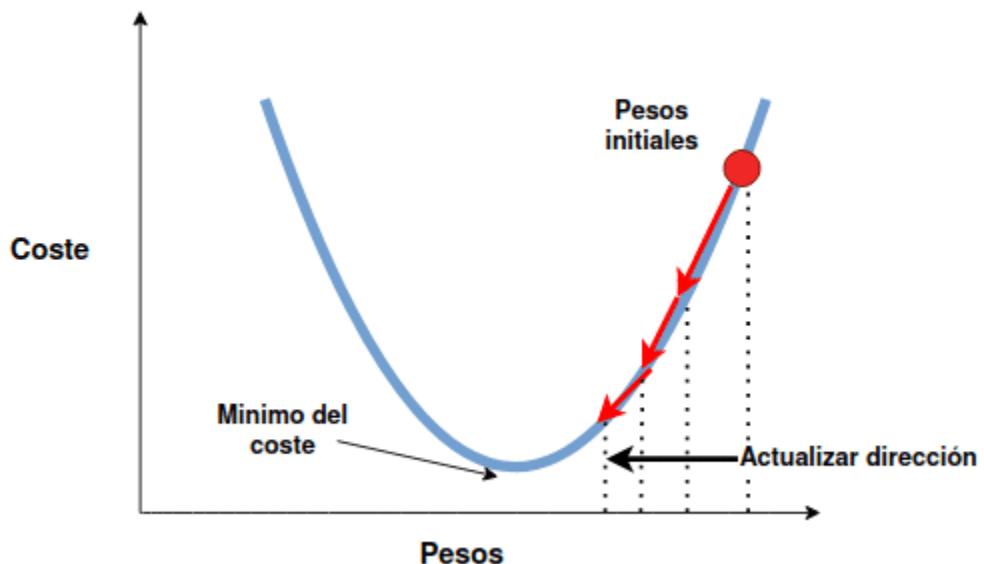
El descenso de gradiente es uno de los optimizadores más populares y es la base de otros optimizadores del deep learning.

El gradiente describe la pendiente de una función y proporciona la dirección y magnitud del ascenso más pronunciado. Al entrenar una red neuronal, el algoritmo de descenso de gradiente calcula el gradiente de la función de coste.

¿Cómo ayuda el cálculo del gradiente de la función de costes al algoritmo a saber cómo actualizar las ponderaciones y los sesgos?

Dado que el gradiente da la dirección del ascenso más pronunciado, ya que queremos minimizar la función de coste (no maximizarla), el descenso del gradiente se mueve en la dirección opuesta al gradiente, en la dirección del descenso más pronunciado, como se puede ver en la siguiente figura.

En cada iteración, el algoritmo de descenso de gradiente utiliza esta dirección de descenso más pronunciado (gradiente negativo) para **actualizar los pesos de la red neuronal**. De modo que para cada iteración el algoritmo de descenso de gradiente se acerque al conjunto de pesos que proporciona el coste más bajo.



Al inicio del entrenamiento el gradiente será mayor, ya que la pendiente será más pronunciada. Luego, a medida que los pesos se actualizan mediante iteraciones del algoritmo de descenso de gradiente, la pendiente se volverá menos pronunciada y el gradiente más pequeño hasta que alcance un mínimo local, donde se vuelve cero o muy cercano a cero. Este proceso lo vemos en la imagen mostrada, donde los pasos comienzan siendo más grandes cerca del 'peso inicial' y luego, a medida que los pasos se acercan al coste mínimo, se hacen cada vez más pequeños.

Este es el proceso de descenso de gradiente de una red neuronal:

- 1 Elige los pesos iniciales.
- 2 Calcula el gradiente.

3

Actualiza los pesos de la red neuronal en la dirección opuesta al gradiente. Da un paso en la dirección opuesta al gradiente.

4

Repite 2 y 3 hasta que el gradiente sea cercano o igual a cero.

Los **pesos** se actualizan utilizando la siguiente ecuación:

$$w = w - \alpha \frac{\partial}{\partial w} J(\Theta)$$

w : pesos en la red neural

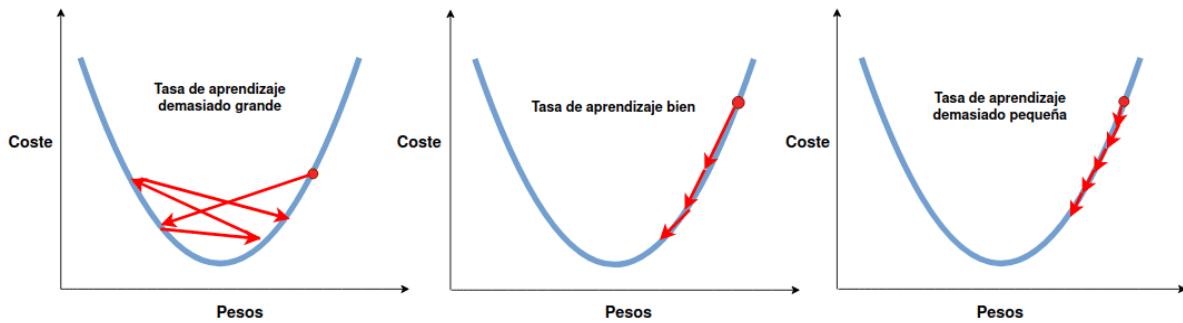
$\frac{\partial}{\partial w} J(\Theta)$: gradiente de la función de coste

α : tasa de aprendizaje

El término alfa corresponde con la **tasa de aprendizaje** y define el tamaño de los pasos dados en cada iteración para alcanzar el mínimo (magnitud de la actualización a realizar en los pesos).

La tasa de aprendizaje suele ser un valor pequeño (por ejemplo, 0,01 o 0,001).

Como se puede ver en la imagen a continuación, la tasa de aprendizaje es un parámetro importante. Si es demasiado alto, existe el riesgo de que el descenso del gradiente nunca alcance el mínimo. Sin embargo, si la tasa de aprendizaje es demasiado baja, **el descenso del gradiente puede tardar mucho tiempo** en alcanzar el mínimo, lo que hace que el entrenamiento sea muy ineficiente.



Además del descenso de gradiente, existen **varios optimizadores diferentes** que se pueden utilizar en deep learning, como el descenso de gradiente estocástico, el descenso de gradiente estocástico con impulso, Adam y RMSProp.

i Puedes encontrar más información sobre los optimizadores en PyTorch a través de este [enlace](#). Y en cuanto al ciclo de entrenamiento entero, puedes encontrar más información [aquí](#).

Evaluación



Al igual que en otros tipos de machine learning, debemos dividir el **conjunto de datos en un conjunto de entrenamiento, validación y prueba** para evaluar con mayor precisión el rendimiento del modelo.

Podemos utilizar las **curvas de aprendizaje** para trazar el progreso del rendimiento de una red neuronal a lo largo del tiempo. Te permiten ver cómo está aprendiendo la red neuronal y su estado en cada paso del entrenamiento, por ejemplo, al final de cada época. Por lo tanto, las curvas de aprendizaje suelen incluir métricas tanto para los datos de entrenamiento como para los datos de validación.

La curva de aprendizaje de los datos de entrenamiento muestra si está aprendiendo bien la red neuronal, y la curva de aprendizaje de los datos de validación muestra si el modelo es capaz de generalizar datos que no ha visto previamente.

Hay dos tipos de curvas de aprendizaje que se utilizan con frecuencia: las de optimización y las de rendimiento.

LAS CURVAS DE APRENDIZAJE DE OPTIMIZACIÓN

LAS CURVAS DE APRENDIZAJE DE RENDIMIENTO

Muestran el comportamiento de la función de coste durante el entrenamiento, por ejemplo, entropía cruzada binaria o error cuadrático medio.

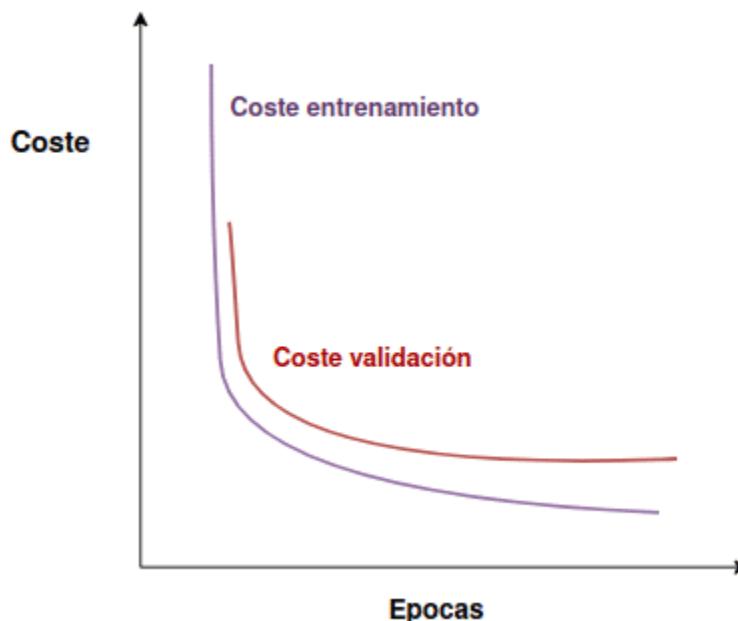
LAS CURVAS DE APRENDIZAJE DE OPTIMIZACIÓN

LAS CURVAS DE APRENDIZAJE DE RENDIMIENTO

Muestran el comportamiento de la métrica para la cual se evaluará el modelo, por ejemplo, la precisión o puntuación F1.

A continuación, vemos un ejemplo de una curva de aprendizaje que muestra el comportamiento de la función de coste para los datos de entrenamiento y validación con respecto a las épocas entrenadas.

El gráfico muestra el coste a lo largo del tiempo (número de épocas).



Si te has fijado bien, las curvas son suaves, lo que significa que el modelo está aprendiendo de manera gradual y estable durante todo el entrenamiento. Si la curva fuese muy irregular significaría que algo anda mal. Además, la curva se estabiliza después de varias épocas. En este punto, el entrenamiento adicional no conduce a mejoras significativas en el rendimiento, como se puede ver en el gráfico.

Esto significa que la red neuronal ha aprendido tanto como va a aprender de los datos de entrenamiento y el entrenamiento del modelo se puede detener sin perder ninguna mejora significativa en el rendimiento.

Mejorar los resultados



Ahora has entrenado tu primer modelo y la curva de aprendizaje de los datos de entrenamiento ha llegado a un punto en el que no muestra ninguna mejora significativa. Sin embargo, los resultados de tu modelo no son los que esperabas. Necesitas mejorar de alguna manera los resultados de este. Pues bien, en este apartado hablaremos de algunas técnicas que puedes utilizar para mejorar el rendimiento de tu red neuronal.

Ajuste de hiperparámetros

Los hiperparámetros en los algoritmos de deep learning pueden tener un gran impacto en el rendimiento del modelo, la capacidad de generalizar a nuevos datos y el tiempo de entrenamiento.

Estos hiperparámetros son la tasa de aprendizaje, la cantidad de capas y unidades ocultas, el tamaño del *batch* y las épocas, entre otros.

Una de las formas de mejorar los algoritmos de aprendizaje profundo es ajustar los hiperparámetros. En ese ajuste, se establecen diferentes valores para cada hiperparámetro y luego, el modelo se entrena utilizando estos mismos.

Por ejemplo, puedes probar tasas de aprendizaje de 0,1, 0,01 y 0,001 y entrenar al modelo con ellas. Si la tasa de aprendizaje 0,1 tiene un coste final de 2,3, la tasa de aprendizaje 0,01 tiene un coste final de 0,7 y la tasa de aprendizaje 0,001 tiene un coste final de 1,3, entonces se elegiría una tasa de aprendizaje de 0,01 para el modelo final, porque tiene el coste más bajo.

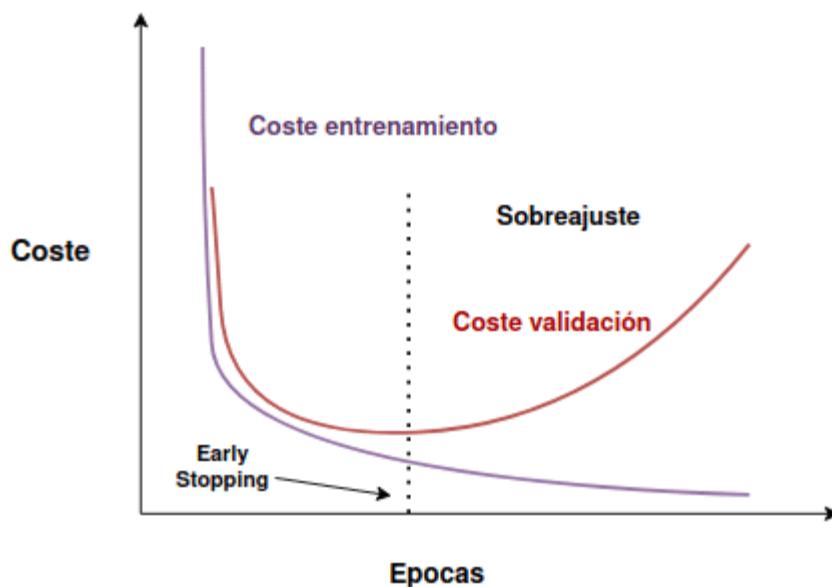
Puedes elegir los hiperparámetros de forma manual o automática, utilizando un algoritmo de optimización de hiperparámetros. La idea es probar varios valores para cada hiperparámetro que quieras ajustar y elegir el mejor.

Uno de los hiperparámetros más importantes para ajustar es la tasa de aprendizaje. La tasa de aprendizaje puede determinar si el modelo es capaz de aprender bien o no. Si la tasa de aprendizaje es demasiado grande o pequeña, es posible que el modelo nunca pueda lograr un error de entrenamiento bajo.

La clave es encontrar la mejor tasa de aprendizaje para cada problema.

Sobreajuste (*overfitting*)

En la curva de aprendizaje que hemos visto en la gráfica del apartado anterior ('Evaluación'), la pérdida disminuye gradualmente tanto para los datos de entrenamiento como para los de validación y sus pérdidas son bastante similares. Este no es siempre el caso. En algunos casos, la curva de aprendizaje puede parecerse a la siguiente.



El coste comienza a disminuir tanto para el conjunto de datos de entrenamiento como para el de validación, pero luego comienza a aumentar nuevamente para el conjunto de validación. Este tipo de fenómeno se conoce como **sobreajuste**.

El sobreajuste ocurre cuando la red neuronal aprende demasiado bien los datos de entrenamiento y no puede generalizar a los datos de validación, por lo que tiene un mal desempeño con los datos de validación.

¿Cómo se puede prevenir el sobreajuste?

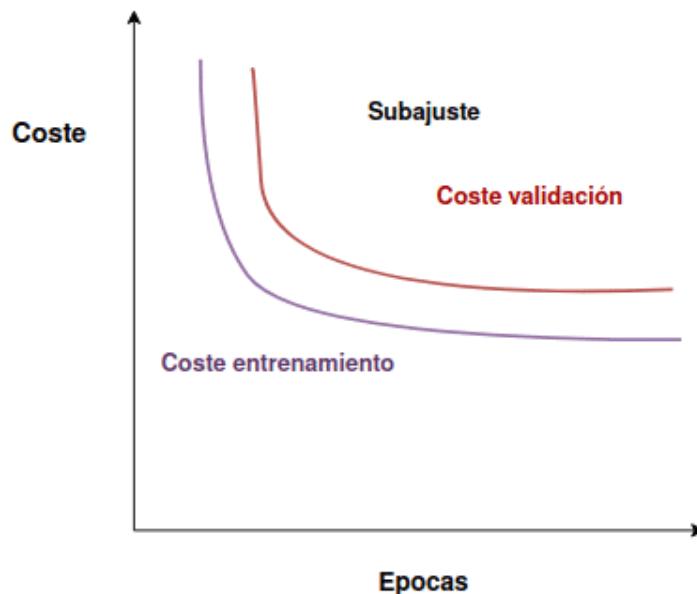
Una forma de **evitar el sobreajuste es utilizar un modelo más simple**. El modelo actual puede ser demasiado complejo para el problema.

- Una solución es **utilizar la detención anticipada** (*early stopping*) para detener el entrenamiento del modelo en un punto óptimo, como se puede ver en el gráfico anterior, en el que el entrenamiento se detiene donde la pérdida de validación es menor.
- Otra causa del sobreajuste podría ser que el conjunto de datos de entrenamiento sea demasiado pequeño. Puede que no sea representativo de los conjuntos de validación y prueba. En este caso, aumentar el conjunto de datos de entrenamiento con una variedad más amplia de ejemplos podría mejorar el rendimiento del modelo. Puedes utilizar el **aumento de datos** para aumentar el tamaño y la complejidad del conjunto de datos.

Además, puedes aplicar [dropout](#) a la red neuronal para evitar el sobreajuste. Dropout es una técnica donde se ignoran temporalmente algunos de los nodos durante el entrenamiento. Esto tiene el efecto de entrenar varias arquitecturas en paralelo.

Desajuste (*underfitting*)

Hablamos de **desajuste** cuando la red neuronal tiene un rendimiento deficiente en los datos de entrenamiento, lo que también da como resultado que el modelo sufra un rendimiento deficiente en los conjuntos de validación y prueba. Como se puede ver en la siguiente imagen, el coste del entrenamiento nunca disminuye mucho.



¿Cómo se puede prevenir el desajuste?

- El desajuste puede deberse a un conjunto de datos de entrenamiento que es demasiado pequeño o que no contiene suficiente información necesaria para resolver el problema, de modo que **aumentar el conjunto de datos** podría mejorar el rendimiento del modelo.
- También el desajuste puede deberse a un modelo demasiado simple para el problema. En este caso, **aumentar la complejidad del modelo** ayudará al rendimiento del modelo. Por ejemplo, incluir más capas o unidades ocultas.



Finalmente, **entrenar los datos** durante más tiempo puede resolver el desajuste en el caso de que la causa de dicho desajuste resida en que el modelo no ha visto los datos suficientes veces para aprender. Ampliar el tiempo de entrenamiento puede resultar útil en aquellas situaciones en las que se interrumpe el aprendizaje mientras el coste sigue disminuyendo.

Resumen y recursos de interés



A lo largo de este fastbook, hemos analizamos el proceso de aprendizaje de una red de deep learning. Comenzábamos con el preprocesamiento de datos específico de las redes neuronales. Luego, hemos cubierto el proceso de entrenamiento, donde hemos visto algunos hiperparámetros importantes que deben establecerse antes del entrenamiento. Hemos analizado la función de coste y cómo mide el rendimiento del entrenamiento y cómo el optimizador ayuda a entrenar la red neuronal. Finalmente, hemos aprendido el proceso de evaluación del rendimiento de una red neuronal y algunas técnicas para mejorar los resultados del entrenamiento con una red neuronal.

Ahora, ya sabemos que es importante escalar los datos antes de entrenar con redes neuronales y que debemos entrenar para un número de épocas suficiente para dejar que la red neuronal vea los datos suficientes veces si queremos que aprenda bien; también que debemos contar una tasa de aprendizaje que no sea demasiado grande ni excesivamente pequeña para que el optimizador (en este caso, el descenso de gradiente) pueda minimizar la función de costo. Y como aprendizaje relevante: estamos preparados para usar las curvas de aprendizaje y detectar si nuestro modelo ha entrenado bien o si sufre del desajuste o el sobreajuste.

Por último, compartimos aquí las referencias bibliográficas sobre este tema, ambas pueden resultar de tu interés.

- Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*,
<https://www.deeplearningbook.org/>.
- Sanket Doshi. *Various Optimization Algorithms For Training Neural Network*,
<https://towardsdatascience.com/optimizers-for-training-neural-network-59450d7lcaf6>.

¡Enhora buena! Fastbook superado



Qualentum.com