Entrenamiento de redes neuronales

TÉCNICAS ESTADÍSTICAS PARA EL APRENDIZAJE II

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

Entrenamiento de redes neuronales

Introducción

Entrenamiento de redes multicapa

- Neurona lineal
- La regla delta
- Neurona no lineal
- Entrenamiento de neuronas ocultas

Backpropagation

Introducción

Entrenamiento de redes multicapa

- Neurona lineal
- La regla delta
- Neurona no lineal
- Entrenamiento de neuronas ocultas

Backpropagation

Introducción

Las **redes neuronales con una sola capa de parámetros ajustables** están muy limitadas con respecto a lo que pueden hacer, como es el caso de los perceptrones.

Por tanto, resulta natural ampliar la capacidad de una red neuronal añadiendo **capas adicionales** de neuronas.

Desde el exterior de la red, sólo son visibles la primera y la última capa de una red multicapa: la **capa de entrada** y la **capa de salida**. Todas las demás capas son **capas ocultas**, donde se suelen emplear neuronas con funciones de activación no lineales.

Cada una de las capas recibe sus entradas de la capa inmediatamente anterior en la red (salvo la capa de entrada, que recibe sus entradas del exterior).

Introducción

Las redes neuronales artificiales organizadas por capas, sin realimentación (***feed-forward neural networks***, FFNNs), constituyen la topología de red neuronal más utilizada en la práctica.

En ocasiones, por razones históricas, las FFNNs también se denominan **perceptrones multicapa** (*MultiLayer Perceptrons*), pero esta denominación no es del todo correcta, ya que las redes multicapa no usan el algoritmo de aprendizaje del perceptrón.

Necesitamos un algoritmo eficiente que nos permita **adaptar todos los pesos de una red multicapa**, no sólo los de la capa de salida.

Aprender los pesos correspondientes a las neuronas de las **capas ocultas** equivale a **aprender nuevas características** (no presentes en el conjunto de entrenamiento), lo que resulta especialmente difícil porque nadie nos dice directamente qué es lo que deberíamos aprender en esas unidades ocultas.

Introducción

En vez de fijarnos en los cambios de los pesos, nos fijaremos en los **cambios de las salidas**, que intentaremos acercar a las salidas deseadas (estratégia válida para problemas no convexos).

En la **capa de salida**, el error cometido por las neuronas de salida nos puede servir para ajustar sus pesos.

En las **capas ocultas**, podemos observar cómo varía el error en función de cómo varían sus parámetros, lo que nos servirá para diseñar un algoritmo de entrenamiento de redes multicapa: el algoritmo de propagación de errores hacia atrás, ***backpropagation***.

Técnicamente, el término *backpropagation* se refiere únicamente a la propagación de errores hacia atrás, no al algoritmo de entrenamiento completo.

Introducción

La propagación de errores hacia atrás permite calcular el **gradiente del error con respecto a los diferentes parámetros de la red**, es decir, cómo varía el error conforme varían los parámetros de la red.

Combinado con una técnica de optimización como el **gradiente descendente** se obtiene un algoritmo de entrenamiento para redes multicapa.

El gradiente obtenido por *backpropagation* indica **en qué sentido han de modificarse los parámetros de la red** y la técnica de optimización es la que realiza el ajuste de dichos parámetros.

El gradiente se calcula para una **función de error**, también conocida como función de pérdida [*loss function*], y el método de optimización ajusta los pesos de la red con el objetivo de **minimizar** esa función de error o pérdida.

Introducción

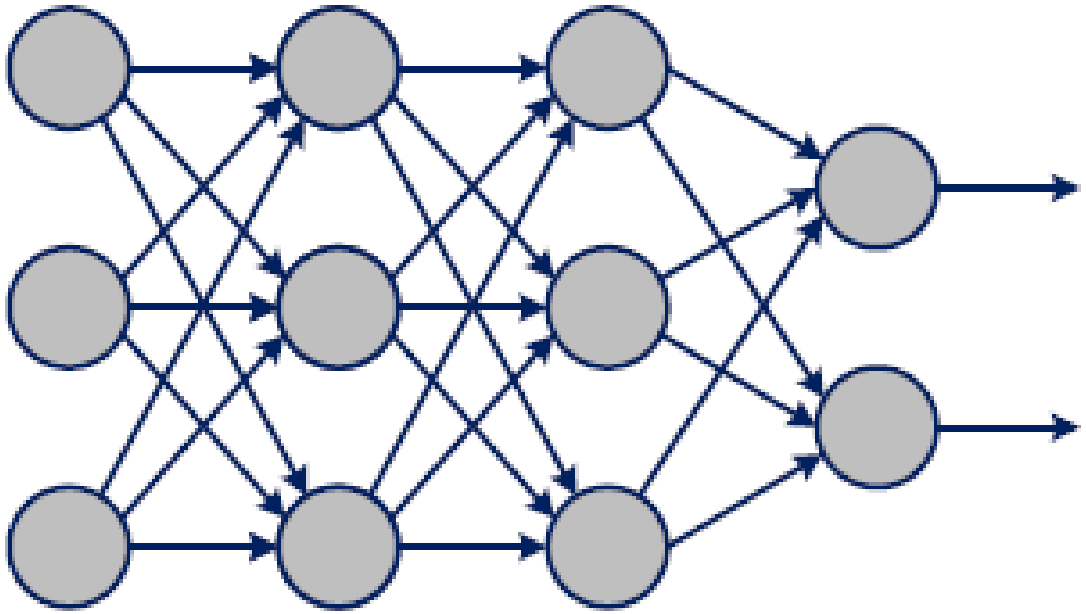
Partiendo de una red con pesos inicializados (normalmente de forma aleatoria) y un conjunto de entrenamiento con valores de entrada y salida esperada, el **proceso de aprendizaje** consta de **dos etapas**:

***Forward* o hacia adelante**: se suministran las muestras de entrenamiento a la red, se obtiene la predicción y se compara esta salida con la salida deseada. Con esto se puede calcular el error (usando una función de error o *loss*) para cada una de las salidas.

***Backward* o hacia atrás**: el error calculado se propaga hacia atrás desde la capa de salida, de forma que en cada neurona se actualicen sus parámetros asociados (pesos) en base a una tasa de aprendizaje y el gradiente calculado para esta neurona (derivada parcial del error total con respecto al peso de la neurona a actualizar).

Una vez finaliza el proceso de entrenamiento ya se puede **usar la red con los pesos aprendidos** para hacer predicciones o clasificaciones. En esta etapa solo será necesario realizar el paso hacia adelante.

Habitualmente, todas las salidas de una capa se distribuyen a todas las neuronas de la siguiente capa, formando capas completamente conectadas (*fully-connected layers*).



No obstante, existen arquitecturas especializadas de redes neuronales artificiales en las que se utilizan otros patrones de conectividad.

Introducción

El uso de **múltiples capas de neuronas ocultas** es lo que permite a las redes neuronales artificiales utilizadas en *deep learning* extraer características más complejas a partir de otras características más simples.

Las capas ocultas le permiten a la red neuronal construir un **modelo interno** de la forma en que los patrones de datos de entrada están relacionados con las salidas deseadas.

Es en el uso de múltiples capas ocultas donde reside el **éxito del *deep learning*** en problemas complejos, como el reconocimiento de voz o la identificación de objetos en imágenes.

Entrenamiento de redes neuronales

Introducción

Entrenamiento de redes multicapa

- Neurona lineal
- La regla delta
- Neurona no lineal
- Entrenamiento de neuronas ocultas

Backpropagation

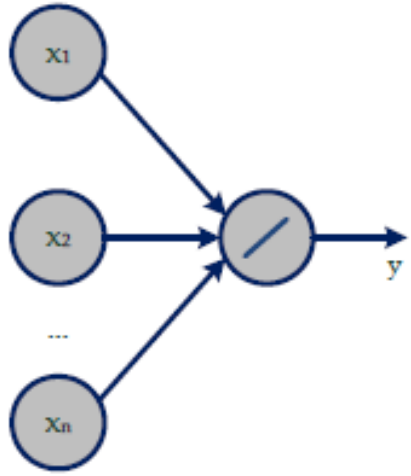
Entrenamiento de redes multicapa

Desde un punto de vista formal, podemos ver las redes multicapa como una función matemática f que, a partir de un vector de entrada x, obtiene un vector de salida $y = f(x)$.

Dado un conjunto de entrenamiento en forma de pares (x, y), el **objetivo del algoritmo de entrenamiento** es ser capaz de aproximar la función f de forma que para cada posible entrada x se obtenga una salida $\hat{y} = f(x)$ lo más similar posible a la observada en el conjunto de entrenamiento.

Además, querremos que la red neuronal proporcione salidas apropiadas para entradas con las que no se hubiese encontrado en el conjunto de entrenamiento. Para que la red neuronal sea capaz de **generalizar** correctamente, tendremos que ajustar su capacidad hasta un nivel que resulte adecuado para la función que pretendemos modelar.

Neurona lineal



Una neurona lineal, se limita a realizar el producto escalar de los vectores de pesos y entradas:

$$y = f(x) = \sum_i w_{ij} x_i = w^T x$$

Si queremos **ajustar los parámetros de una neurona lineal**, su vector de pesos w, tendremos que definir una función de error, coste o pérdida que nos permita decidir qué conjuntos de pesos son mejores y cuáles son peores.

Por ejemplo, podemos recurrir al error cuadrático medio medido sobre el conjunto de entrenamiento:

$MSE = \frac{1}{n}\sum_{j=1}^{n}(t_j - y_j)^2$, donde y_j es la salida proporcionada por la neurona lineal y t_j la salida deseada , nuestro objetivo [*target*].

Neurona lineal

Dado que asumimos que el conjunto de entrenamiento es siempre el mismo (su tamaño no cambia durante el proceso de entrenamiento), minimizar el *MSE* es equivalente a minimizar la **suma de los errores al cuadrado** [*SSE: Sum of Squared Errors*]:

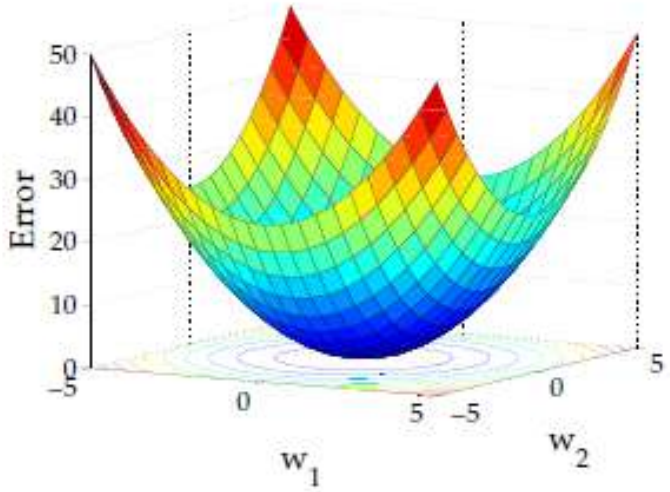
$$SSE = \sum_j (t_j - y_j)^2$$

Dado que $y = w^T x$, **el error depende directamente de los pesos** de entrada de la neurona, que son los que tendremos que ajustar para reducir el error.

Si dibujamos nuestra **función de error** en función de un peso, obtendremos una parábola. En función de dos pesos, obtendremos un paraboloide de revolución (el resultado de rotar una parábola sobre su eje). En general, para n pesos, tendremos un paraboloide n-dimensional.

Neurona lineal

Con ayuda de una función de error, coste o pérdida, hemos reducido nuestro problema de aprendizaje supervisado inicial a un **problema de optimización**.



Superficie de error de una neurona lineal con dos entradas.

Este problema de optimización consiste en determinar el **valor más adecuado para los pesos** de la neurona, aquél que reduce su error al mínimo de acuerdo a la función de coste predefinida.

Aunque para una neurona lineal podríamos resolver el problema analíticamente utilizando cálculo diferencial, la solución sería difícil de generalizar, por lo que diseñaremos un **algoritmo iterativo** menos eficiente pero que luego podamos generalizar.

Entrenamiento de redes multicapa

Los métodos numéricos que nos permitirán resolver problemas de optimización complejos se basan en técnicas como el **gradiente descendente**, donde hemos de tomar nuestras decisiones utilizando **información local única y exclusivamente**.

La información local de la que disponemos es la **pendiente del punto en el que nos encontramos**, así que lo lógico sería seguir la dirección con mayor pendiente, hacia abajo.

Esa **dirección** es precisamente la **opuesta al gradiente de nuestra función de error**, de ahí que el método se denomine gradiente descendente.

Si estuviésemos buscando la cima de la montaña a ciegas, seguiríamos la dirección marcada por el gradiente, hablaríamos de maximización y estaríamos utilizando el gradiente ascendente.

Entrenamiento de redes multicapa

Evaluar la pendiente de la superficie donde nos movemos requiere realizar ciertos **cálculos, que conllevan** algo de **tiempo**.

De forma que, en vez de evaluar instantáneamente el gradiente de la función y determinar en cada punto qué dirección seguir, **iremos dando pequeños saltos** en la dirección del gradiente descendente.

El tamaño de esos saltos vendrá dado por la **tasa de aprendizaje** de nuestro algoritmo de aprendizaje, es decir, la tasa de aprendizaje determina con qué frecuencia evaluamos la pendiente de la superficie de error en nuestra búsqueda de su mínimo.

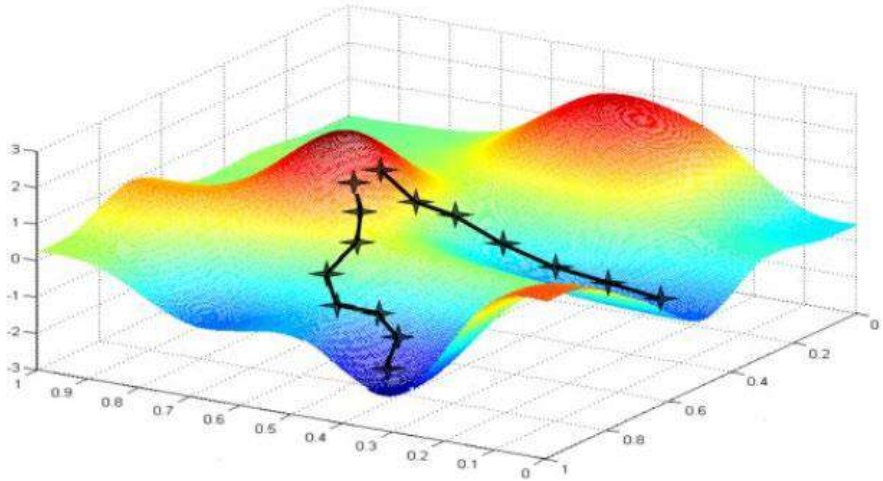
La tasa de aprendizaje deberá ser lo **suficientemente grande** para que no se eternice nuestra búsqueda del mínimo pero, a la vez, lo **suficientemente pequeña** para que no pasemos de largo ese mínimo.

Entrenamiento de redes multicapa

Dado que estamos utilizando sólo información local, que lleguemos o no al **mínimo global** de nuestra función de coste dependerá de la forma que tenga dicha función.

Si la **función** es **convexa**, tendrá un **único mínimo**. Si damos pasos pequeños para ir acercándonos a ese mínimo, pero sin pasarnos, el gradiente descendente nos garantizará llegar al mínimo de la función.

Si la **función no** es **convexa**, como suele ser habitual, el gradiente descendente podría en principio quedarse atascado en un mínimo local.



La regla delta

Entonces, **¿cómo realizamos la actualización de pesos?**

Partimos de una **función de error** que pretendemos minimizar.

Por ejemplo, podríamos utilizar el **error cuadrático medio**, que realiza una media aritmética de los residuos al cuadrado (por convención, añadimos un factor adicional, 1/2, que utilizamos para simplificar los cálculos cuando se derive la función de error):

$$E_{MSE} = \frac{1}{2}\frac{1}{n}\sum_{j}(t_j - y_j)^2$$

Podemos prescindir también del factor $1/n$ y quedarnos directamente con la suma de los residuos al cuadrado:

$$E_{MSE} = \frac{1}{2}\sum_{j}(t_j - y_j)^2$$

La regla delta

Nuestra **medida de error sobre el conjunto de entrenamiento** la podemos reescribir como:

$$E = E_{MSE} = \sum_j E_j$$

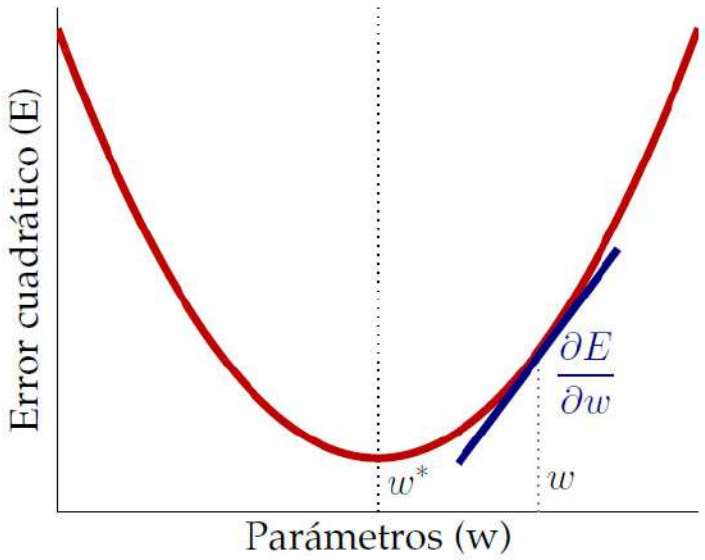
donde $E_j = \frac{1}{2}(t_j - y_j)^2$ es la estimación puntual del error sobre un caso concreto j del conjunto de datos de entrenamiento.

Al usar el error cuadrático como señal de error, obtenemos una **función continua y derivable** que nos indica cómo ir modificando los pesos → para **minimizar el error**, podemos calcular su derivada (gradiente en el caso vectorial) y realizar una corrección de los pesos en el sentido opuesto al indicado por esa derivada.

La regla delta

En el **caso unidimensional**, la derivada del error será positiva cuando nos encontremos en el lado derecho de la parábola definida por la función de error cuadrático.

Por tanto, como deseamos **minimizar** ese **error**, debemos disminuir el valor actual del peso w, con el objetivo de acercarnos al valor óptimo w^*. El movimiento lo hacemos, pues, en sentido contrario al indicado por la derivada del error con respecto al peso.



En el **caso multidimensional**, nuestra superficie de error tendrá forma de paraboloide multidimensional y el gradiente desempeñará el papel de la derivada en el caso unidimensional.

La regla delta

Regla de la cadena: $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$

Las **derivadas parciales del error con respecto a cada uno de los pesos** de la neurona vienen dadas por la expresión:

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y}\frac{\partial y}{\partial w_i}$$

donde hemos utilizado la definición de nuestra función de error como suma de errores instantáneos y la **regla de la cadena** para calcular la derivada del error con respecto a los pesos como un producto de dos derivadas: la del error con respecto a la salida y la de la salida con respecto a los pesos.

Para simplificar las expresiones, utilizaremos **notación vectorial**.

La regla delta

En primer lugar, definimos el **operador gradiente ∇**:

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_n}\right]$$

Entonces, el **vector gradiente de nuestra función de error con respecto a los pesos**, viene dada por:

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

Ese gradiente es el que utilizamos para **ajustar los parámetros de la red** con el objetivo de minimizar su error, moviéndonos en sentido opuesto al indicado por el vector del gradiente:

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

donde con la letra griega $\eta > 0$ (eta) denotamos la **tasa de aprendizaje**.

La regla delta

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y}\frac{\partial y}{\partial w_i}$$

La **derivada de la estimación instantánea del error con respecto a la salida** de la neurona es:

$$E_j = \frac{1}{2}(t_j - y_j)^2 \rightarrow \frac{\partial E_j}{\partial y} = -(t_j - y_j)$$

En el **caso de una neurona lineal**, la derivada de la salida de la neurona con respecto al peso w_i es:

$$y = \sum_i w_{ij} x_i \rightarrow \frac{\partial y}{\partial w_i} = x_i$$

Por lo que el vector gradiente sería:

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}\right] \rightarrow \nabla E = -\sum_j (t_j - y_j) x_j$$

Y ajustaríamos los parámetros de la red con:

$$\Delta w = -\eta \nabla E = \eta \sum_j (t_j - y_j) x_j$$

La regla delta

Resumiendo, en cada iteración del algoritmo, **ajustamos los pesos en proporción al gradiente del error cometido**.

En notación vectorial:

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

Individualmente, para cada peso:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

El **algoritmo del gradiente descendente** es un algoritmo iterativo, que repetidamente calcula el gradiente del error ∇E y realiza ajustes en los parámetros de la red en sentido opuesto a dicho gradiente, lo que nos hace caer por la pendiente de la superficie de error en dirección al mínimo.

Entrenamiento de redes multicapa

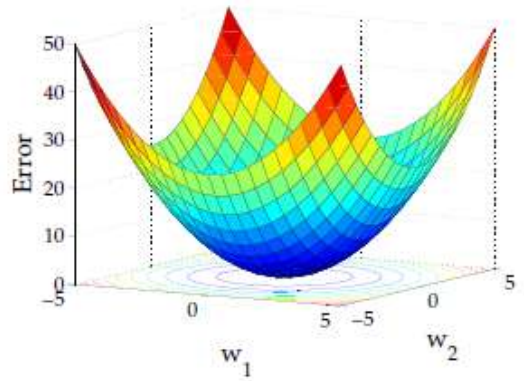
Para conseguir que el método del gradiente descendente funcione correctamente, tendremos que elegir una **tasa de aprendizaje** lo suficientemente pequeña, que nos permita ir **acercándonos a la solución óptima**.

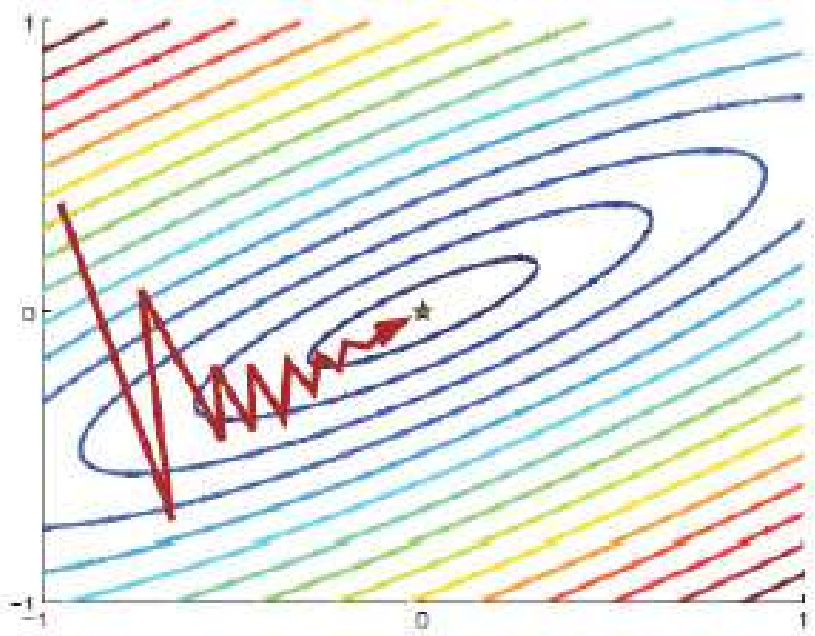
También nos interesa que la **convergencia** al óptimo se realice **lo más rápidamente posible**, lo que implica aumentar el valor de la tasa de aprendizaje.

Por tanto, la elección de un **valor adecuado para la tasa de aprendizaje** es un factor crítico a la hora de aplicar con éxito el método del gradiente descendente para entrenar una red neuronal.

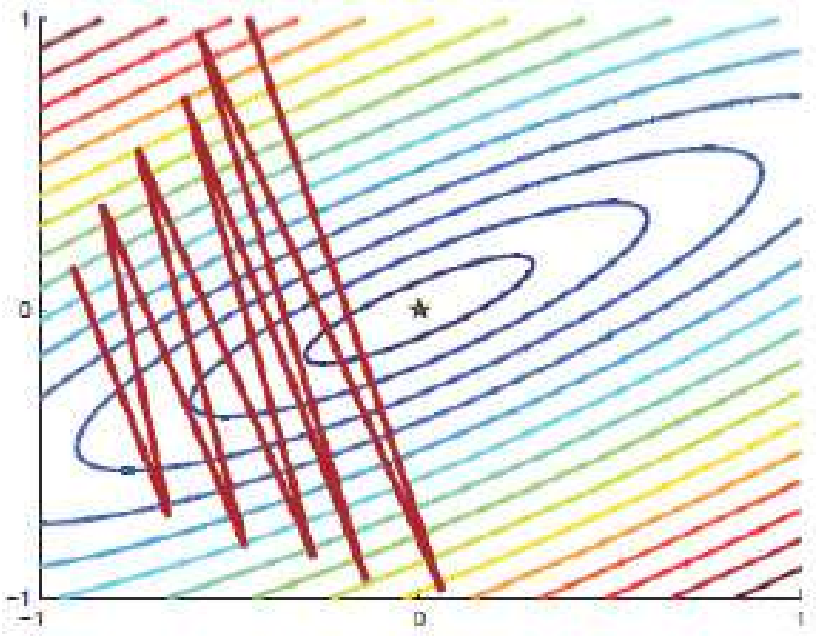
Lo suficientemente pequeña para que el algoritmo converja y lo suficientemente grande para que lo haga rápidamente, sin llegar a divergir.

Entrenamiento de redes multicapa



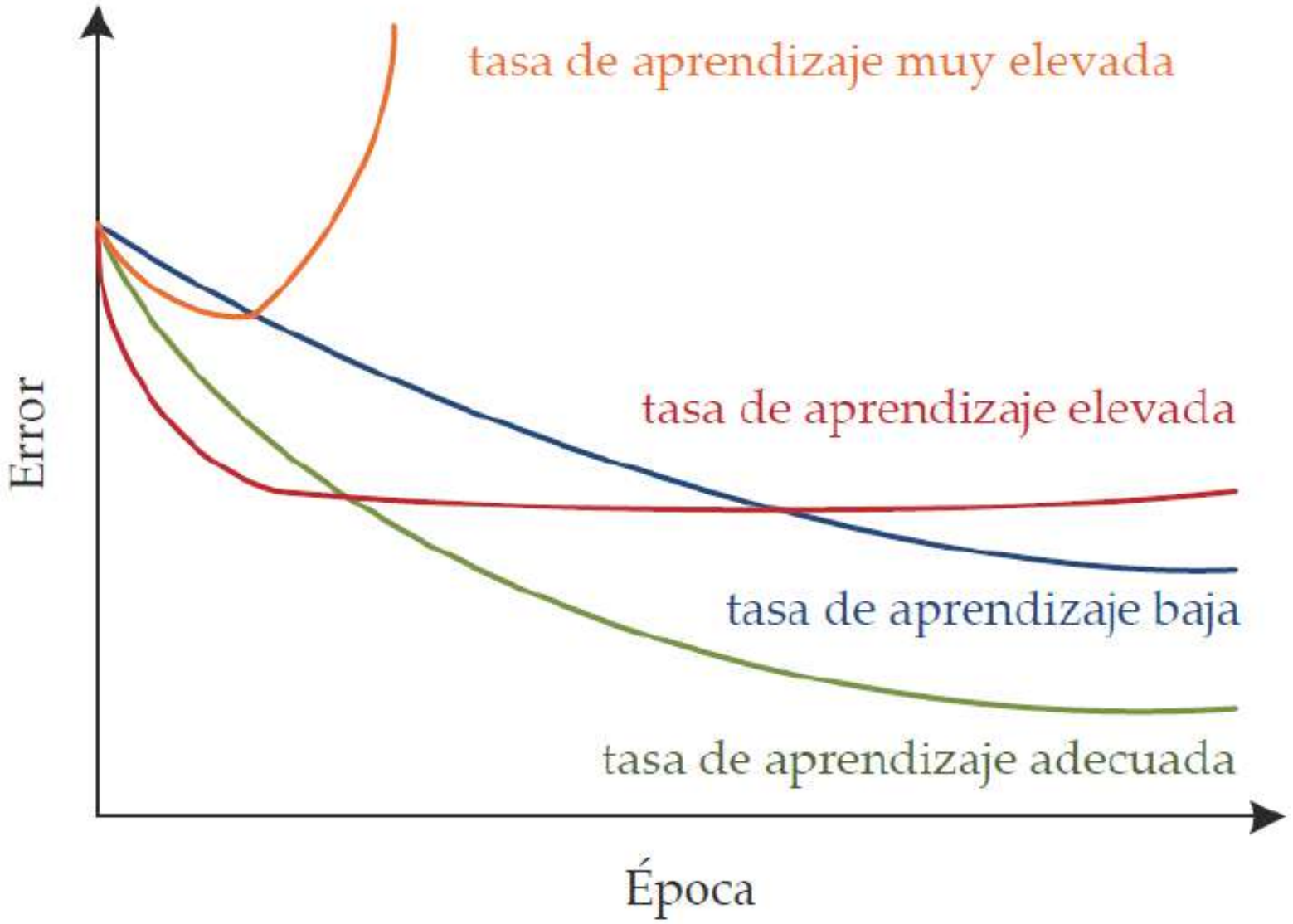


Tasa de aprendizaje adecuada



Tasa de aprendizaje demasiado alta

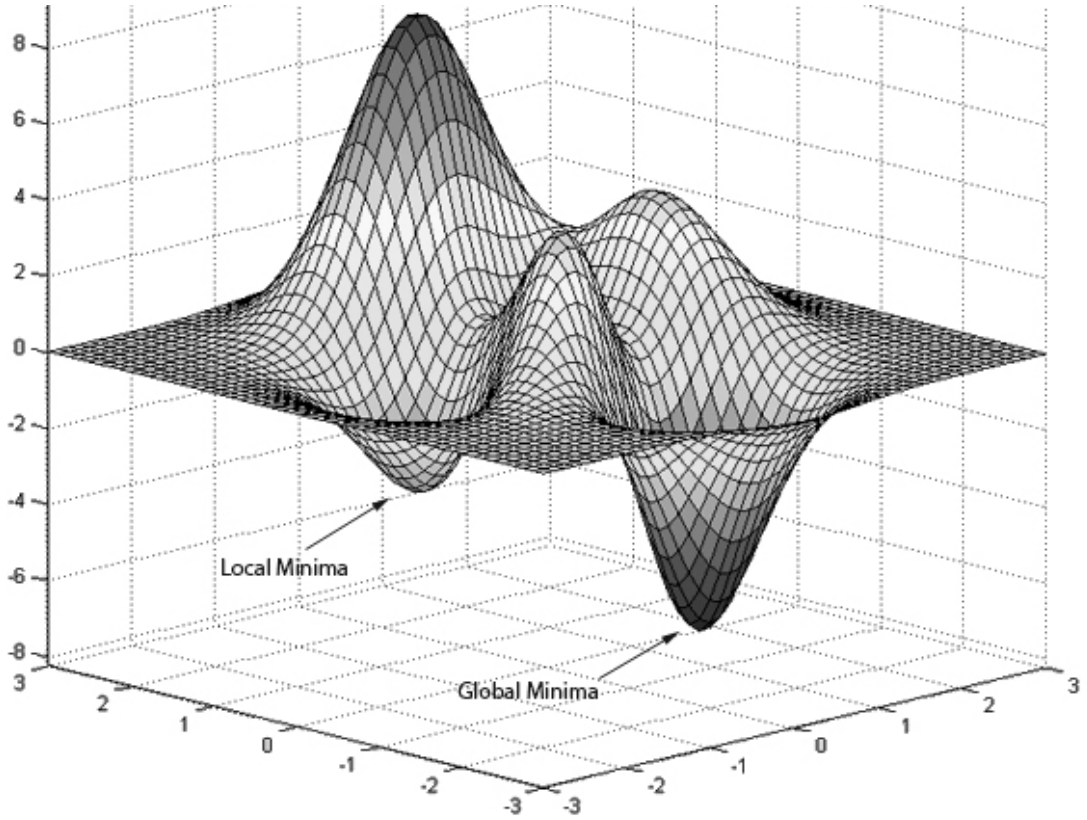
Entrenamiento de redes multicapa



Gradiente descendente con diferentes tasas de aprendizaje

Entrenamiento de redes multicapa

En **redes multicapa con elementos no lineales,** el proceso de **encontrar un mínimo en la superficie de error** formada a partir de los parámetros de la red y las muestras de entrenamiento, suele ser **muy complejo** dado que la superficie normalmente será N-dimensional y contendrá muchos valles, además, en cada momento solo sabemos el error cometido en ese punto, con la configuración de parámetros actual.



Entrenamiento de redes multicapa

En el análisis anterior, hemos asumido que empleamos **aprendizaje por lotes** [*batch learning*], en el que sólo se actualizan los pesos una vez que hemos visto todos los ejemplos del conjunto de entrenamiento y calculado el error E para el conjunto de entrenamiento completo.

Cuando utilizamos **aprendizaje *online***, se realiza una estimación instantánea del error para un único ejemplo del conjunto de entrenamiento y se utiliza esta estimación instantánea para estimar el gradiente y ajustar los parámetros de la red. El gradiente así obtenido dependerá del ejemplo concreto que se haya seleccionado.

Cuando usamos aprendizaje online, generalmente, los datos del conjunto de entrenamiento se van seleccionando al azar. Esta selección al azar del ejemplo con el que se ajustan los pesos es un proceso estocástico, de ahí que se denomine **gradiente descendente estocástico**.

Entrenamiento de redes multicapa

Cuando utilizamos **aprendizaje por lotes y un valor adecuado para la tasa de aprendizaje**, el movimiento hacia el mínimo se realizará de forma progresiva, siguiendo una trayectoria amortiguada.

La **regla delta por lotes** cambia los pesos en proporción a las derivadas del error sumadas para todos los ejemplos del conjunto de entrenamiento.

Cuando utilizamos **aprendizaje online**, la estimación del gradiente fluctuará en función del ejemplo particular que se haya escogido en cada momento.

La dirección de la actualización será menos fiable que la seguida cuando se emplea aprendizaje por lotes pero, como realizamos una actualización de los pesos cada vez que le presentamos un ejemplo a la red, la **convergencia** del algoritmo suele ser **más rápida** en términos de tiempo de CPU.

Neurona no lineal

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y} \frac{\partial y}{\partial w_i}$$

La característica clave de las **funciones de activación no lineales** es que sean **continuas y derivables** para que podamos utilizar el gradiente descendente para ajustar los parámetros de una neurona no lineal.

Aplicamos la **regla de la cadena**, que nos permite representar la derivada de la salida de la neurona con respecto a un peso como el producto de las derivadas de la salida con respecto a su entrada neta y de su entrada neta con respecto al peso:

$$\frac{\partial y}{\partial w_i} = \frac{dy}{dz} \frac{\partial z}{\partial w_i} = f'(z) \frac{\partial z}{\partial w_i}$$

Dado que la entrada neta de la neurona es una simple combinación lineal, su derivada es muy fácil de calcular: $\frac{\partial z}{\partial w_i} = x_i$

Neurona no lineal

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y} \frac{\partial y}{\partial w_i}$$

La **derivada del error con respecto a la salida** la calculamos anteriormente:

$$\frac{\partial E_j}{\partial y} = -(t_j - y_j)$$

Por lo que, en notación vectorial para una función de activación f cualquiera, siempre que sea derivable, el **gradiente del error** se puede expresar como:

$$\nabla E = -\sum_j (t_j - y_j) f'(w \cdot x_j) x_j$$

Como antes, este gradiente lo podemos utilizar para **actualizar iterativamente los parámetros de una neurona no lineal**:

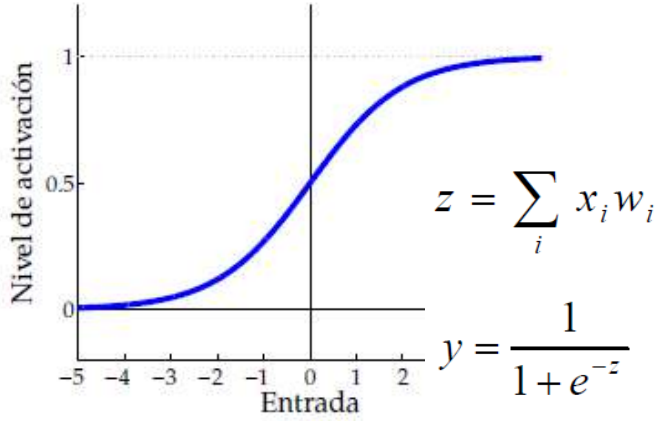
$$\Delta w = -\eta \nabla E$$

$$\frac{\partial E}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial w_i} = \sum_j \frac{\partial E_j}{\partial y} \frac{\partial y}{\partial w_i}$$

Neurona no lineal

Por ejemplo, en el caso de la **función logística** sabemos que:

$$f'(z) = \frac{dy}{dz} = y(1-y)$$



con lo que:

$$\frac{\partial y}{\partial w_i} = \frac{dy}{dz}\frac{\partial z}{\partial w_i} = f'(z)x_i = y(1-y)x_i$$

y con esto podemos obtener cómo varía el error cometido por la neurona no lineal conforme cambian sus parámetros.

En notación vectorial, el **gradiente del error con respecto al vector de pesos** viene dado por:

$$\nabla E = -\sum_j (t_j - y_j) y_j (1-y_j) x_j$$

Entrenamiento de neuronas ocultas

Nuestro objetivo es ser capaces de **entrenar redes neuronales con múltiples capas**, de forma que el entrenamiento de las neuronas de sus capas ocultas les permita ser capaces de encontrar, automáticamente, características que resulten útiles en la tarea particular para la que entrenamos la red.

Utilizando la misma estrategia que antes, se puede comprobar que aplicando la **regla de la cadena dos veces**, podemos describir la derivada parcial del error con respecto a cada parámetro de una red neuronal multicapa como un producto de otras derivadas parciales:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}$$

Entrenamiento de neuronas ocultas

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

De atrás hacia adelante:

- La derivada parcial de la entrada neta z_j de la neurona j con respecto a un peso particular w_{ij} no es más que su **entrada** x_i.

- La derivada parcial de la salida y_j de la neurona j con respecto a su entrada neta z_j no es más que la **derivada de la función de activación** de la neurona evaluada para su entrada neta $f'(z_j)$.
  Por ej., $f'(z_j) = y_j(1 - y_j)$ en el caso de la función de activación logística.

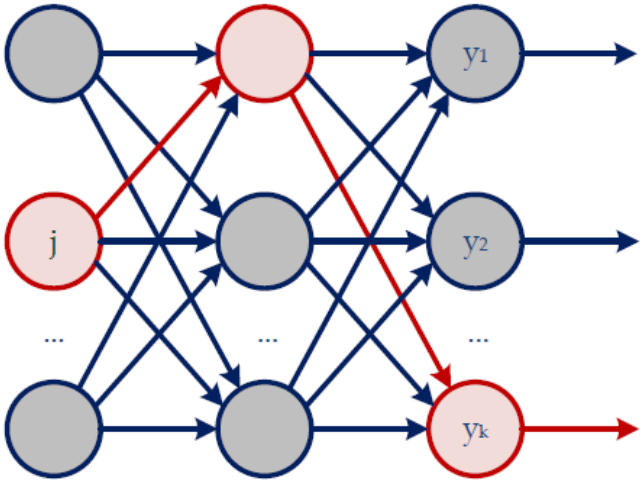
- Referente a la derivada parcial del error E con respecto a la salida y_j:

  - Si la neurona forma parte de la **capa de salida**, es fácil de evaluar (es lo que venimos haciendo hasta ahora).

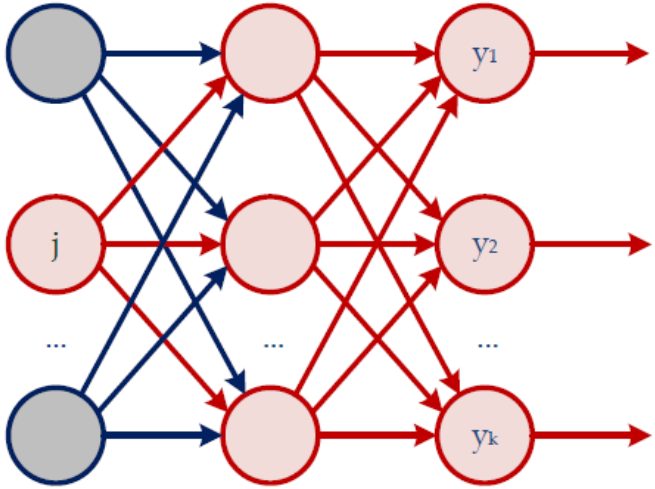
  - Pero si la neurona j está en una **capa oculta** de la red, encontrar esta derivada no resulta evidente.

Entrenamiento de neuronas ocultas

Para **ajustar los pesos de neuronas ocultas** se han de realizar cálculos mucho más complejos, ya que el resultado de cada neurona y, por lo tanto, el error cometido depende del resto de neuronas.



Uno de los caminos a través de los que un cambio en la actividad de la neurona *j* afecta a la salida de la red.



Todos los caminos a través de los que un cambio en la actividad de la neurona *j* puede afectar a la salida de la red multicapa.

Entrenamiento de neuronas ocultas

No sabemos realmente lo que debe hacer cada neurona oculta, pero podemos **calcular cómo cambia el error cuando cambia su actividad**.

Un pequeño **cambio en los pesos** w_{ij} de una neurona oculta j de la capa c generará un cambio en su salida y_j.

Para obtener el efecto conjunto que tiene el cambio del peso w_{ij} sobre el error global hemos de **considerar todos los caminos posibles** que conectan la neurona j con la salida de la red. Por este motivo, **la derivada del error es una suma sobre todos los caminos posibles.**

Para **cada camino individual** desde la neurona j hasta la salida de la red, se van multiplicando las derivadas parciales de los niveles de activación de cada neurona con respecto al nivel de activación de la neurona que la precede en ese camino hacia la salida.

Entrenamiento de neuronas ocultas

El **algoritmo de propagación de errores hacia atrás**, que utilizaremos para ajustar los parámetros de una red multicapa, lo que hace es calcular de una forma eficiente la suma de las contribuciones de todos los caminos desde una neurona hasta la salida.

Para ello, **aprovecha la topología de la red**, organizada en una serie de capas con conexiones única y exclusivamente entre capas adyacentes.

Esa topología le permite **realizar el cálculo del gradiente de forma inteligente**, con lo que se puede determinar eficientemente cómo se propagan a través de la red pequeñas perturbaciones en los pesos de las neuronas ocultas, cómo alcanzan la salida esas perturbaciones y cómo afectan a la función de error.

Entrenamiento de redes neuronales

Introducción

Entrenamiento de redes multicapa

- Neurona lineal
- La regla delta
- Neurona no lineal
- Entrenamiento de neuronas ocultas

Backpropagation

Recordemos que lo que pretendemos es aplicar iterativamente un **método de optimización** basado en el **gradiente descendente** para ir ajustando los pesos de la red neuronal y conseguir entrenarla para que haga lo que queramos:

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

Por regla general, utilizaremos una **función de error, coste o pérdida** que cumpla dos propiedades:

a) Se puede especificar como un promedio de las funciones de coste asociadas a ejemplos concretos del conjunto de entrenamiento.

b) Se puede definir en términos de la salida que se obtiene de la red y_j y del valor que deseamos obtener t_j.

Por ejemplo, el error cuadrático: $E_j = \frac{1}{2}(t_j - y_j)^2$

Recordemos que el **algoritmo de aprendizaje basado en *backpropagation*** y **gradiente descendente** consta de dos fases:

- Fase de propagación **hacia adelante**: suministramos a la red un patrón de entrada y calculamos la salida de la red para dicho patrón.

- Fase de propagación **hacia atrás**: evaluamos el error cometido por la red y propagamos dicho error hacia atrás, capa por capa, de forma que se pueda calcular eficientemente el gradiente del error para las neuronas ocultas de la red.

El **error en la salida de una neurona** puede deberse a los valores de sus pesos, a las entradas que se reciben de las capas anteriores de la red o a una combinación de ambos factores, motivo por el que usaremos la entrada neta z_j de la neurona como base para nuestros cálculos.

Para ello, definimos $\boldsymbol{\delta_j^c}$ **para una neurona** j **de la capa** c como la derivada parcial del error con respecto a la entrada neta z_j^c de la neurona:

$$\delta_j^c = \frac{\partial E}{\partial z_j^c}$$

Intuitivamente, **este "delta" nos indica cómo influye una perturbación en la entrada neta de la neurona** (que puede deberse a un cambio en alguno de sus pesos o en alguna de sus entradas) **en el error de la red**.

Al pasar de las derivadas parciales con respecto a la salida, a las derivadas parciales con respecto a la entrada neta, lo que hemos hecho es **incorporar la función de activación *f* de la neurona en el cálculo de los "deltas"** que reutilizaremos en el cálculo del gradiente del error:

$$\delta_j^c = \frac{\partial E}{\partial z_j^c} = \frac{\partial E}{\partial y_j^c}\frac{dy_j^c}{dz_j^c} = \frac{\partial E}{\partial y_j^c} f'(z_j^c)$$

donde f' es la derivada de la función de activación de la neurona de salida: 1 para una neurona lineal, $y_j\,(1-y_j)$ para la función logística, $1_{z\geq 0}$ para las unidades lineales rectificadas (ReLU), etc.

Para **neuronas de las capas ocultas**, sabemos que tenemos que considerar todos los caminos que conectan la salida de una neurona oculta con las salidas de la red neuronal → en vez de ir enumerando todos esos caminos uno por uno, se aprovecha la **topología de la red** para hacer ese cálculo de forma eficiente.

Dado que las distintas neuronas de la capa *c* comparten los caminos que van desde la capa *c+1* hasta la salida de la red, esto nos permite calcular el gradiente del error para la capa *c+1* y, a partir de ese gradiente, calcular el gradiente para las neuronas de la capa *c*.

Expresamos la **contribución al error de las neuronas de la capa *c*** en términos de la contribución al error de las neuronas de la capa *c+1* :

$$\frac{\partial E}{\partial y_j^c} = \sum_k \frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial y_j^c}$$

Y reescribimos la expresión anterior en términos de las entradas netas de las neuronas, en vez de emplear sus niveles de activación:

Backpropagation

$$\delta_j^c = \frac{\partial E}{\partial z_j^c}$$

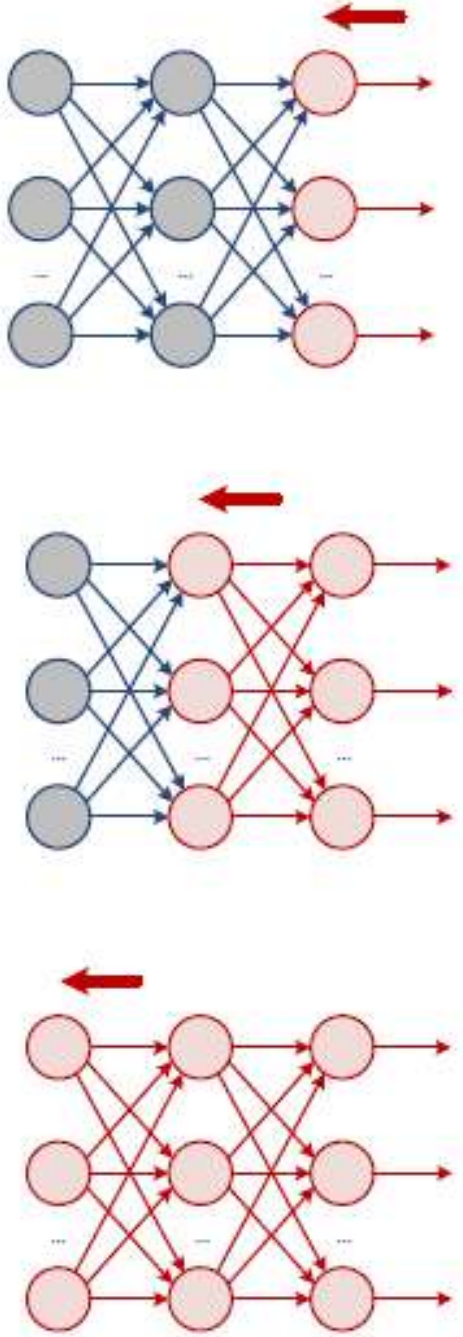
$$= \frac{\partial E}{\partial y_j^c} f'(z_j^c)$$

$$= \left(\sum_k \frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c)$$

$$= \left(\sum_k \left(\frac{\partial E}{\partial y_k^{c+1}} \frac{\partial y_k^{c+1}}{\partial z_j^{c+1}} \right) \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c)$$

$$= \left(\sum_k \frac{\partial E}{\partial z_j^{c+1}} \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c)$$

$$= \left(\sum_k \delta_j^{c+1} \frac{\partial z_k^{c+1}}{\partial y_j^c} \right) f'(z_j^c)$$



Usando que la salida de la capa *c* es la entrada de la capa *c+1*, es decir, $y^c = x^{c+1}$, y calculando las derivadas parciales de la entrada neta z^{c+1} con respecto a las entradas individuales x^{c+1}, obtenemos una expresión que nos permite **calcular los deltas de la capa *c* a partir de los deltas de la capa *c+1*:**

$$\delta_j^c = \left(\sum_k \delta_j^{c+1} \frac{\partial z_k^{c+1}}{\partial x_j^{c+1}} \right) f'(z_j^c)$$

$$= \left(\sum_k \delta_j^{c+1} w_{jk}^{c+1} \right) f'(z_j^c)$$

Una vez que tenemos todos los "deltas", la **derivada del error con respecto a cada peso de una red multicapa** la podemos expresar como:

$$\frac{\partial E}{\partial w_{ij}^c} = \frac{\partial E}{\partial z_j^c} \frac{\partial z_j^c}{\partial w_{ij}^c} = \delta_j^c x_i^c$$

Una vez que tenemos los gradientes del error con respecto a los parámetros de la red, como resultado de la propagación de errores hacia atrás, podemos aplicar una **regla de actualización de los parámetros de la red** similar a la regla delta.

Por este motivo, la regla utilizada en el entrenamiento de redes multicapa se suele denominar **regla delta generalizada**:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

donde η es la tasa de aprendizaje utilizada por el método del gradiente descendente.

Con esto, **modificamos los pesos en la dirección de máxima pendiente dada por el gradiente de la función de error**. Como queremos minimizar el error, lo hacemos en **sentido opuesto** al indicado por el gradiente.

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}^c} = \frac{\partial E}{\partial z_j^c} \frac{\partial z_j^c}{\partial w_{ij}^c} = \delta_j^c x_i^c$$

$$\delta_j^c = \left(\sum_k \delta_j^{c+1} w_{jk}^{c+1} \right) f'(z_j^c)$$

Por tanto, **en el ajuste de un peso asociado a la entrada x_i de una neurona, intervienen tres factores**: el valor de la entrada x_i, la derivada de su función de activación $f'(z_j)$ dada su entrada neta y una señal de error que depende de la capa en la que nos encontremos y que proviene de las siguientes capas de la red.

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.