

# Tema 2: Fundamentos de las redes neuronales profundas

---



**Aprendizaje**  
**Profundo**

Grado en Ingeniería y Ciencia de datos (Universidad de Oviedo)

---

Pablo González, Pablo Pérez  
{gonzalezgpablo, pabloperez}@uniovi.es  
Centro de Inteligencia Artificial, Gijón

# El perceptrón

---

# El perceptrón: la pieza clave de una red neuronal

- El perceptrón es la pieza clave de cualquier red neuronal y por supuesto del aprendizaje profundo.
- Fue introducido en 1957 por Frank Rosenblatt.
- Es un modelo motivado por la biología



# El perceptrón: la pieza clave de una red neuronal



## Salida de una perceptrón

Para evaluar la salida  $\hat{y}$  de un perceptrón, multiplicamos las entradas  $X$  por los pesos  $w$  y aplicamos la función de activación  $g$ .

$$\hat{y} = g\left(\sum_{i=1}^n x_i w_i\right)$$

# El perceptrón: la pieza clave de una red neuronal



## Bias

Un perceptrón incluye además un peso extra  $w_0$ , correspondiente al **bias**, que permite ajustar el umbral de activación del perceptrón.

$$\hat{y} = g\left(w_0 + \sum_{i=1}^n x_i w_i\right)$$

# El perceptrón: la pieza clave de una red neuronal



## Vista con operaciones matriciales

La evaluación de un perceptrón puede reescribirse utilizando vectores en lugar del sumatorio.

$$\hat{y} = g(w_0 + X^T W)$$

donde  $W$  es el vector con pesos y  $X$  es un vector con las entradas.

# El perceptrón: Función de activación

La función de activación  $g$  juega un papel clave en el perceptrón

- La función de activación es una función matemática que se utiliza en un perceptrón para determinar si una neurona debe activarse o no.
- La función de activación toma la suma ponderada de las entradas y las pasa a través de una función **no lineal** para producir la salida de la neurona.
- La función de activación introduce **no linealidad** en el modelo, lo que permite al perceptrón aprender **patrones más complejos** en los datos de entrada.
- Hay varias funciones de activación comunes utilizadas en los perceptrones, como la función escalón, la función sigmoide y la función ReLU.

# El perceptrón: Función de activación

## Algunos ejemplos de funciones de activación



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



$$g(z) = \max(0, z)$$



## ¿Cuál es la utilidad de la función de activación?

Qué sucede si no incluimos una función de activación. El perceptrón sería capaz de resolver problemas **linealmente separables**.



## ¿Cuál es la utilidad de la función de activación?

Qué sucede si no incluimos una función de activación. El perceptrón sería capaz de resolver problemas **linealmente separables**.



$$-1.15x_1 - 0.21x_2 + 4.1 = 0$$

# ¿Cuál es la utilidad de la función de activación?

¿Qué sucede si el problema **no es linealmente separable**?



# ¿Cuál es la utilidad de la función de activación?

No es posible encontrar un plano que sea capaz de separar las dos clases de manera adecuada.



# ¿Cuál es la utilidad de la función de activación?

Gracias a las **funciones de activación no lineales** y la unión de **varias capas de perceptrones** es posible aproximar funciones de este tipo.



## Ejemplo con un perceptrón

Dado el ejemplo anterior, con los datos separables, el modelo obtenido sería el siguiente:  $\hat{y} = g(-1.15x_1 - 0.21x_2 + 4.1)$



## Ejemplo con un perceptrón

El perceptrón tendría los siguientes pesos:



$$w_0 = 4.1 \text{ y } W = \begin{bmatrix} -1.15 \\ -0.21 \end{bmatrix}$$

$$\hat{y} = g(z) = g(w_0 + X^T W) = g\left(4.1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} -1.15 \\ -0.21 \end{bmatrix}\right)$$

## Ejemplo con un perceptrón

Asumiendo que  $g$  es la función sigmoide y que el punto que queremos evaluar es el (0,6):

$$\hat{y} = g\left(4.1 + \begin{bmatrix} 0 \\ 6 \end{bmatrix}^T \begin{bmatrix} -1.15 \\ -0.21 \end{bmatrix}\right) = g(4.1 + 6 * (-0.21)) = g(2.84) = 0.94$$

Con la función **sigmoide**, y cuando esta está en la capa de salida de la red, podemos convertir a una probabilidad la salida del perceptrón. Si  $z < 0$ , entonces  $\hat{y} < 0.5$ ; si  $z > 0$ , entonces  $\hat{y} > 0.5$



# Ejemplo con un perceptrón

Valores de las probabilidades para todo el espacio de entrada:



- Punto (0,6),  $\hat{y} = 0.94$
- Punto (6,8),  $\hat{y} = 0.01$  (realiza tú mismo el cálculo)

# Redes totalmente conectadas

---

# Uniendo varios perceptrones

Un solo perceptrón **está muy limitado** a funciones simples y lineales. La verdadera potencia de las redes neuronales viene al **unir varios perceptrones** en una misma red.

Antes de unir varios perceptrones vamos a **simplificar** su representación, para que los diagramas sean legibles:



# Uniendo con varios perceptrones

La manera más común de unir varios perceptrones son con **capas totalmente conectadas**.

- Cada capa puede tener varias neuronas.
- Podemos utilizar varias capas, conectando unas con otras.



# Uniendo con varios perceptrones (1 capa oculta)

La idea es que la salida de cada perceptrón (neurona), sirva como entrada a las de la capa siguiente. De esta manera podríamos calcular la salida de la red (**pasada hacia adelante**), de izquierda a derecha utilizando las reglas explicadas anteriormente.



# Salida en una red totalmente conectada

Para evaluar la salida  $\hat{y}$  a partir de las entradas  $X$  en una red totalmente conectada se procede de la siguiente manera:

- Capa oculta:  $h_i = b_i^{(1)} + \sum_{j=1}^3 x_j w_{j,i}^{(1)}$
- Cada de salida:  $\hat{y} = g(b^{(2)} + \sum_{j=1}^4 h_j w_j^{(2)})$



# Salida en una red totalmente conectada (1 capa oculta)

En general, para cualquier tamaño de red, tendríamos lo siguiente:

- Capa oculta:  $h_i = b_i^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)}$
- Cada de salida:  $\hat{y} = g(b^{(2)} + \sum_{j=1}^n h_j w_j^{(2)})$



## Nota

En el caso de querer aplicar una función de activación en la primera capa podríamos escribir:  $h_i = g(b_i^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)})$

# Salida en una red totalmente conectada (1 capa oculta)

Un ejemplo para la neurona  $h_1$  sería el siguiente:

$$h_1 = b_1^{(1)} + \sum_{j=1}^3 x_j w_{j,1}^{(1)} = b_1^{(1)} + x_1 w_{1,1}^{(1)} + x_2 w_{2,1}^{(1)} + x_3 w_{3,1}^{(1)}$$





# Salida en una red totalmente conectada (1 capa oculta)

La clave de la evaluación de una red es la **multiplicación de matrices**:



# Múltiples capas totalmente conectadas

Podemos seguir añadiendo capas totalmente conectadas para hacer nuestra red más grande.

- El número de pesos crecerá muy rápido.
- La **dimensión** de las matrices a multiplicar será **mayor**.
- Pero el proceso explicado será exactamente el mismo independientemente de la dimensión de la red.



# Evaluación de una red con varias capas

En general, para totalmente conectada con  $k$  capas ocultas, la podríamos evaluar de la siguiente manera:

$$h_i^{(k)} = g(b_i^{(k)} + \sum_{j=1}^{n_{k-1}} h_j^{(k-1)} w_{j,i}^{(k)})$$



# Entrenamiento de una red: Función de pérdida

---

# ¿Cómo entrenar la red?

Hasta el momento hemos aprendido como evaluar la salida de una **red neuronal totalmente conectada**, a partir de los datos de entrada.

¿Cómo ser capaces de aprender los pesos de la red ( $W$  y  $b$ ) para que representen los datos entrenamiento y para predecir nuevos ejemplos?

# Ejemplo práctico: XOR

Supongamos que tenemos un conjunto de datos de entrada para el cual queremos entrenar un modelo con una red neuronal.

Ten en cuenta que este conjunto de datos **no es linealmente separable**.

$x_1$	$x_2$	$y$
0	0	0
0	1	1
1	0	1
1	1	0



## Ejemplo práctico: XOR

Vamos a definir una red neuronal muy sencilla, con dos neuronas en la **capa de entrada**, dos neuronas en la **capa oculta**, y una neurona en la **capa de salida**.



La función de activación  $g$  en la capa oculta y en la salida será la **sigmoide**.

## Ejemplo práctico: XOR

Inicializaremos las matrices de pesos  $w^{(1)}$  y  $w^{(2)}$  a valores aleatorios. Por simplicidad en este ejemplo, no consideraremos el bias.

$$w^{(1)} = \begin{bmatrix} 0.96 & 0.89 \\ 0.28 & 0.73 \end{bmatrix} \quad w^{(2)} = \begin{bmatrix} 0.60 \\ 0.22 \end{bmatrix}$$

Veamos que sucede si evaluamos uno de los puntos del conjunto de datos, por ejemplo el punto (0,0).



## Ejemplo práctico: XOR

$$X = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$h = g(X^T w^{(1)}) = g\left(\begin{bmatrix} 0 & 0 \end{bmatrix} * \begin{bmatrix} 0.96 & 0.89 \\ 0.28 & 0.73 \end{bmatrix}\right) = g(\begin{bmatrix} 0 & 0 \end{bmatrix}) = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

$$\hat{y} = g(hw^{(2)}) = g\left(\begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0.60 \\ 0.22 \end{bmatrix}\right) = g(0.41) = 0.60$$



## Ejemplo práctico: XOR

Si observamos el valor real para el punto  $(0, 0)$  vemos que  $y = 0$ , pero nuestra red está devolviendo  $\hat{y} = 0.6$  (al ser un valor  $> 0.5$  lo evaluaríamos como 1).

Estamos **cometiendo un error**, evidentemente porque nuestra red no está entrenada (sus pesos son aleatorios).

Podemos ver la salida  $\hat{y}$  para el resto de puntos del dataset:

$x_1$	$x_2$	$y$	$\hat{y}$	Acierto
0	0	0	0.60	X
0	1	1	0.62	✓
1	0	1	0.64	✓
1	1	0	0.66	X

## Ejemplo práctico: XOR

Llegados a este punto podríamos calcular el error cometido con una **función de pérdida**. Por ejemplo para problemas de clasificación una de las más utilizadas es la **cross entropy loss**:

$$\mathcal{L}(y, \hat{y}) = - \sum_{i=1}^C y_i \log \hat{y}_i$$

Donde  $C$  es el número de clases de nuestro problema. Si estamos en un problema binario:

$$\mathcal{L}(y, \hat{y}) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

### Funciones de pérdida

Trataremos las funciones de pérdida más adelante en el curso.

## Ejemplo práctico: XOR

Calculo del error para cada ejemplo:

$x_1$	$x_2$	$y$	$\hat{y}$	Correcto	$L(y, \hat{y})$
0	0	0	0.60	X	0.92
0	1	1	0.62	✓	0.48
1	0	1	0.64	✓	0.44
1	1	0	0.66	X	1.07

**¿Cómo ajustar los pesos de nuestra red para ser capaces de minimizar el error cometido?**

# Entrenamiento de una red: descenso del gradiente

---

# ¿Cómo entrenar una red neuronal?

El objetivo del entrenamiento es **minimizar la función de pérdida** alterando para ello los pesos de la red. En general, trataremos de minimizar la función de pérdida sobre los datos de entrenamiento.

Formalmente:  $W^* = \operatorname{argmin} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y_i, f(x_i, W))$

Este problema no se puede resolver de manera analítica ya que la función a minimizar es demasiado compleja. Para ello se utiliza una técnica de minimización conocida como **descenso de gradiente**.

## Evaluación de la red

Es importante recordar la evaluación de la red se representa como  $f(X, W)$ , es decir, una función de los parámetros de la red y la entrada.

# Descenso del gradiente

El descenso del gradiente es un algoritmo que nos permite encontrar el mínimo de una función basándonos en el concepto de **gradiente**. Imaginemos la función  $f(x) = x^2$ . Como sabemos su mínimo está en cero. Dado un  $x$  aleatorio, utilizaremos el descenso de gradiente para encontrar su mínimo.



# Descenso del gradiente

El algoritmo de **descenso de gradiente** se basa en los siguientes pasos:

- ① Partir de un valor aleatorio de  $x$
- ② Calcular el gradiente de la función en el punto  $\frac{\partial f}{\partial x}$ . Esto nos indicará hacia donde crece la función.
- ③ Actualizar  $x = x - \eta \frac{\partial f}{\partial x}$ .  $\eta$  es conocido como el **learning rate**.
- ④ Repetir 2 y 3 hasta la convergencia.



# Descenso del gradiente

El **learning rate** es un parámetro crítico en este algoritmo.



Learning rate muy pequeño



Learning rate demasiado grande

# Descenso del gradiente

Además del learning rate, la **elección de los pesos iniciales** de la red también tiene un efecto importante. Imaginemos la función  $f(x) = x^4 - 2x^2 + x - 1$ . Esta función tiene un **mínimo global** y un **mínimo local**.



$$x_0 = 0.3$$



$$x_0 = 0$$

# Descenso del gradiente

Una técnica para elegir los pesos iniciales suele ser **inicializarlos aleatoriamente** siguiendo una **distribución normal**  $\mathcal{N}(0, \sigma^2)$ .

En cuanto al learning rate, se considera un **hiperparámetro** a ajustar.

- Learning rates muy bajos pueden hacer que la red tarde mucho en converger. Por otra parte, un learning rate muy bajo puede ser más propenso a caer en mínimos locales.
- Learning rates muy altos pueden resultar en la no convergencia de la red en el mínimo global, como hemos visto anteriormente.

## Probar y ganar experiencia

No existe una receta única para establecer el learning rate. En próximas lecciones aprenderemos más sobre este tema y veremos software específico que nos ayudará a elegir el learning rate más adecuado.

# Descenso del gradiente

A continuación se muestra el mismo ejemplo pero en una función con dos variables de entrada  $f(w_1, w_2)$ . Esto podría representar a una red neuronal con dos pesos.



# Descenso del gradiente

## Descenso del gradiente:

- Es un algoritmo usado en varias areas (no solo en la IA), inventado hace más de 150 años.
- La dificultad en las redes neuronales consiste en el computo de las derivadas parciales de cada uno de los parámetros/pesos con respecto a la función de pérdida.
- Recuerda que una red, y más en el aprendizaje profundo, puede tener **miles de millones** de parámetros.
- ¿Cómo **calcular de manera eficiente los gradientes** de cada uno de los parámetros con respecto a la función de pérdida?

# Entrenamiento de una red: Backpropagation

---

# Propagación hacia atrás

El objetivo de la propagación hacia atrás (**backpropagation**) es calcular las derivadas parciales  $\frac{\partial \mathcal{L}}{\partial w}$  y  $\frac{\partial \mathcal{L}}{\partial b}$ , es decir, el gradiente de cada peso ( $w$ ) y bias ( $b$ ) con respecto a la función de pérdida.

Este algoritmo responde a la pregunta de como el **cambio de un parámetro** de la red **afecta a la salida** de la función de pérdida.

El nombre "propagación hacia atrás" viene de que el error (salida de la función de pérdida), **se propaga hacia atrás** a través de la red, desde la salida hasta la capa de entrada.

## Descubrimiento de la propagación hacia atrás

La propagación hacia atrás fue descubierta a mitad de la década de los 80. Aunque el perceptrón había sido descubierto antes, fue este hecho el que permitió empezar a entrenar las redes neuronales de manera eficiente.

# Propagación hacia atrás

Imaginemos la red más sencilla posible, con una neurona en la capa de entrada, una neurona en la capa intermedia y una neurona en la capa de salida.



Recuerda que la salida de la red es una función de los pesos y los datos de entrada.  $\mathcal{L}$  es la función de pérdida utilizada.



# Propagación hacia atrás



¿Cómo podemos calcular el gradiente de los pesos con respecto a la función de pérdida, es decir,  $\frac{\partial \mathcal{L}}{\partial w_1}$  y  $\frac{\partial \mathcal{L}}{\partial w_2}$ ?

La herramienta matemática que utilizaremos será la **regla de la cadena**. Nos permitirá calcular las derivadas parciales cuando la función a derivar es la composición de otras funciones.

# Regla de la cadena

La regla de la cadena se usa para calcular derivadas con la forma:

$$y = f(g(x))$$
$$y' = f'[g(x)]g'(x)$$

donde  $f$  corresponde a la **función externa** y  $g$  a la **función interna**.

La regla de la cadena también se puede escribir de la siguiente forma:

$$y = f(u), \text{ con } u = g(x)$$
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

## Importante

Una red neuronal no es más que una composición de muchas funciones por tanto la regla de la cadena es muy útil para calcular las derivadas parciales con respecto a los parámetros de la misma.

## Regla de la cadena: Ejemplo

Supongamos una función polinómica  $y$ , como por ejemplo:

$$y = (6x^2 + 2)^4$$

# Regla de la cadena: Ejemplo

Supongamos una función polinómica  $y$ , como por ejemplo:

$$y = (6x^2 + 2)^4$$

en este caso, podemos considerar  $y$  como una composición de dos funciones:

- La **función externa**: la potencia a la cuarta
- La **función interna**: lo que hay en el paréntesis.

## Regla de la cadena: Ejemplo

Supongamos una función polinómica  $y$ , como por ejemplo:

$$y = (6x^2 + 2)^4$$

en este caso, podemos considerar  $y$  como una composición de dos funciones:

- La **función externa**: la potencia a la cuarta
- La **función interna**: lo que hay en el paréntesis.

Por tanto aplicaríamos la regla de la cadena de la siguiente forma:

$$y' = 4(6x^2 + 2)(12x)$$

# Propagación hacia atrás



Podemos utilizar la **regla de la cadena** para calcular las derivadas parciales con respecto a los parámetros de la red.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_2} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} \\ \frac{\partial \mathcal{L}}{\partial w_1} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial w_1}\end{aligned}$$

## Importante

La **regla de la cadena** nos permite dividir el cálculo de los gradientes de la función de pérdida con respecto a los pesos de la red en un serie de **cálculos más pequeños y simples**.

# Propagación hacia atrás

La propagación hacia atrás tiene varios problemas. El primero es que necesita **calcular los gradientes para todos los parámetros de la red por cada uno de los ejemplos** del dataset.

Esto significa calcular  $\frac{1}{|d|} \sum_{x_i, y_i \in d} \mathcal{L}(y_i, f(x_i; W))$ . De esta manera promediamos el gradiente en cada parámetro para todos ejemplos del conjunto de datos.

A esta técnica se la conoce comúnmente como **Batch Gradient Descent**. Su principal implicación es que **la complejidad** de la realización de una actualización de los pesos **aumenta linealmente** con respecto al tamaño del conjunto de datos de entrenamiento  $d$ .

# Variaciones del descenso de gradiente

---



# Tipos de descenso de gradiente

Existen varias soluciones para este problema, cada una con sus ventajas e inconvenientes:

- Descenso de gradiente por lotes (**Batch Gradient Descent**). Es la alternativa mencionada anteriormente. El gradiente se calcula para todos los ejemplos antes de actualizar los pesos.
- Descenso de gradiente por mini-lotes (**Mini-Batch Gradient Descent**). Variante del anterior. En lugar de considerar el conjunto de entrenamiento completo se coge un lote de ejemplos (mini-batch).
- Descenso de gradiente estocástico (**Stochastic Gradient Descent**). En este caso se calcula el gradiente y se actualizan utilizando un único ejemplo elegido aleatoriamente.

# Descenso de gradiente por lotes

En el descenso de gradiente por lotes (**Batch Gradient Descent**):

- El conjunto de entrenamiento es **procesado completamente** antes de realizar una actualización de los pesos.
- La actualización de los pesos se basa en la **media de los gradientes** calculados para cada ejemplo.
- Es un algoritmo **determinista** (siempre que los pesos iniciales sean los mismos).
- **Converge lentamente**, especialmente en datasets grandes.
- Es un algoritmo **estable**.
- Tiene a quedarse en **mínimos locales** (especialmente si la función de pérdida no es convexa).

# Descenso de gradiente estocástico

En el descenso de gradiente estocástico (**Stochastic Gradient Descent, SGD**):

- Se selecciona **un único ejemplo aleatorio** en cada iteración para calcular el gradiente.
- Tiene una **convergencia más rápida** porque los pesos se actualizan más frecuentemente.
- Es más **inestable** que las otras versiones ya que los gradientes son calculados para un solo ejemplo.
- Es menos propenso a quedarse atrapado en **mínimos locales**.

# Descenso de gradiente por mini-lotes

El descenso de gradiente por mini-lotes (**Mini-Batch Gradient Descent**):

- Es un **punto intermedio** entre las dos soluciones anteriores.
- Las actualizaciones de los pesos de la red se realiza después de procesar un **subconjunto de los datos** de entrenamiento.
  - El tamaño de este subconjunto se suele llamar **batch size**. Se suelen escoger valores como 32, 64, 128 (potencia de dos). Este valor se considera un hiper-parámetro del proceso de entrenamiento.
- Tiene una **convergencia más rápida** que en la versión por lotes porque actualiza los pesos más frecuentemente.
- Es menos preciso que el descenso de gradiente por lotes ya que **el gradiente calculado es solo una aproximación**.
- Es algoritmo **menos estable** y con más ruido pero esto puede ayudarle a salir de mínimos locales.

# Comparativa entre las diferentes versiones

**Batch Gradient Descent**



**Mini-Batch Gradient Descent**



**Stochastic Gradient Descent**



# ¿Qué algoritmo de optimización elegir?

---

## The Tradeoffs of Large Scale Learning

---

Léon Bottou  
NEC laboratories of America  
Princeton, NJ 08540, USA  
leon@bottou.org

Olivier Bousquet  
Google Zürich  
8002 Zurich, Switzerland  
olivier.bousquet@m4x.org

Un resultado importante descubierto por Bottou and Bousquet (2011) indica lo siguiente:

- SGD no es el algoritmo más efectivo en términos de minimizar el error de entrenamiento.
- Pero, generalmente obtiene **modelos que generalizan mejor**.

### Generalización con respecto a sobreajuste

Aunque estudiaremos este tema en más profundidad, recuerda que lo que nos interesa es ser capaces de generalizar lo aprendido y no simplemente memorizar el conjunto de entrenamiento (sobreajuste).

# ¿Qué algoritmo de optimización elegir?

Generalmente, Mini-Batch Gradient Descent será una buena elección inicial ya que combina las fortalezas de los otros dos enfoques.

Realmente que algoritmo de optimización utilizar depende del problema en concreto y de la arquitectura de la red. De todas maneras, es útil entender los algoritmos de optimización existentes para ser capaces de ajustar hiper-parámetros como por ejemplo el **batch-size** o el **learning rate**.

## Otros algoritmos de optimización

En próximas unidades veremos otros algoritmos de optimización más avanzados, como Adam, AdamW o RMSProp, etc. Estos algoritmos trabajan con conceptos como el **momento** o los **learning rates** adaptativos.

# Desvanecimiento del gradiente y sus soluciones

---



# Desvanecimiento del gradiente

¿Qué es el desvanecimiento del gradiente?

- El **desvanecimiento del gradiente** es un problema que puede ocurrir durante el entrenamiento de redes neuronales profundas.
- Como hemos visto anteriormente, la actualización de los pesos depende de los gradientes de la función de pérdida con respecto a los parámetros de la red neuronal. Estos **se vuelven muy pequeños** a medida que se retropropagan a través de varias capas.
  - Si estos gradientes son muy pequeños puede hacer que los pesos en ciertas capas apenas cambien.
- Este problema puede hacer que la red neuronal **converja muy lentamente** o incluso que **no converja en absoluto**.

# Desvanecimiento del gradiente



Figure: Figura extraída del paper "Understanding the difficulty of training deep feedforward neural networks" de Xavier Glorot y Yoshua Bengio, 2010.

# Causas del desvanecimiento del gradiente

Las principales causas del desvanecimiento del gradiente son:

- Esto puede ocurrir cuando se utilizan **funciones de activación que tienen gradientes muy pequeños**, como la función sigmoide.
- También puede ocurrir cuando se utiliza una **inicialización inapropiada de los pesos** de la red neuronal.



# Desvanecimiento del gradiente: ejemplo

Imaginemos una red neuronal, con dos capas ocultas y funciones de activación ( $\sigma$ ) sigmoides, inicializada con unos pesos de una Gausiana, con media 0 y desviación pequeña, tal que generalmente  $-1 < w_i < 1$ .



En este caso, como ya sabemos, podemos evaluar la red de la siguiente manera:

$$z_1 = w_1 x_1$$

$$h_1 = \sigma(z_1)$$

$$z_2 = w_2 h_1$$

$$h_2 = \sigma(z_2)$$

$$z_3 = w_3 h_2$$

$$\hat{y} = \sigma(z_3)$$

# Desvanecimiento del gradiente: ejemplo



Resumiendo, la evaluación de la red sería:

$$\hat{y} = \sigma(w_3(\sigma(w_2(\sigma(w_1 x_1))))$$

Si ahora queremos calcular la derivada de  $\mathcal{L}$  con respecto al peso  $w_1$ , tendríamos:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3} \frac{\partial z_3}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

# Desvanecimiento del gradiente: ejemplo



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_3} \frac{\partial z_3}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

Dónde:

$$\frac{\partial z_1}{\partial w_1} = x_1, \quad \frac{\partial z_2}{\partial h_1} = w_2, \quad \frac{\partial z_3}{\partial h_2} = w_3$$

y:

$$\frac{\partial h_1}{\partial z_1} = \frac{\partial \sigma(z_1)}{\partial z_1}, \quad \frac{\partial h_2}{\partial z_2} = \frac{\partial \sigma(z_2)}{\partial z_2}, \quad \frac{\partial \hat{y}}{\partial z_3} = \frac{\partial \sigma(z_3)}{\partial z_3}$$

El cálculo del gradiente puede entonces reescribirse como:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \sigma(z_3)}{\partial z_3} w_3 \frac{\partial \sigma(z_2)}{\partial z_2} w_2 \frac{\partial \sigma(z_1)}{\partial z_1} x_1$$

# Desvanecimiento del gradiente: ejemplo



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \sigma(z_3)}{\partial z_3} w_3 \frac{\partial \sigma(z_2)}{\partial z_2} w_2 \frac{\partial \sigma(z_1)}{\partial z_1} x_1$$

De esta ecuación debemos hacer dos observaciones:

- Los pesos  $w_j$  tendrán un valor entre -1 y 1.
- $\sigma'(z_j)$  valdrá como mucho 0.25.

Esto hace que cuanto más productos encadenemos (cuanto más profunda sea la red), mayor será la probabilidad de que el **gradiente sea muy cercano a cero**.

Como podemos ver, **la inicialización de los pesos es muy importante** ya que si tomamos todo ceros por ejemplo, el gradiente siempre valdría cero.

## Otras funciones de activación con el mismo problema





# Función de activación ReLU

La función ReLU tiene las siguientes características:

- Si el valor es mayor que cero, la activación es 1, por tanto **resuelve el problema del desvanecimiento del gradiente**.
- Es más rápida de calcular.

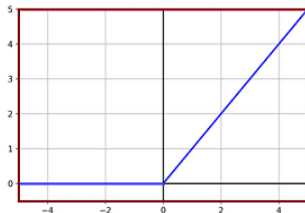
La función ReLU tiene el problema de que cuando el valor es negativo, su derivada es cero, cortando la propagación hacia atrás (**neuronas muertas**).

**ReLU**

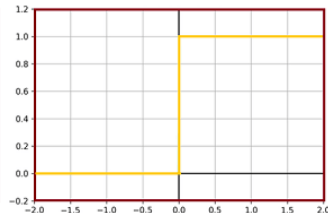
$\max(0, x)$

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{otro caso} \end{cases}$$

Función de activación



Derivada



# Función de activación LeakyReLU

## Leaky ReLU

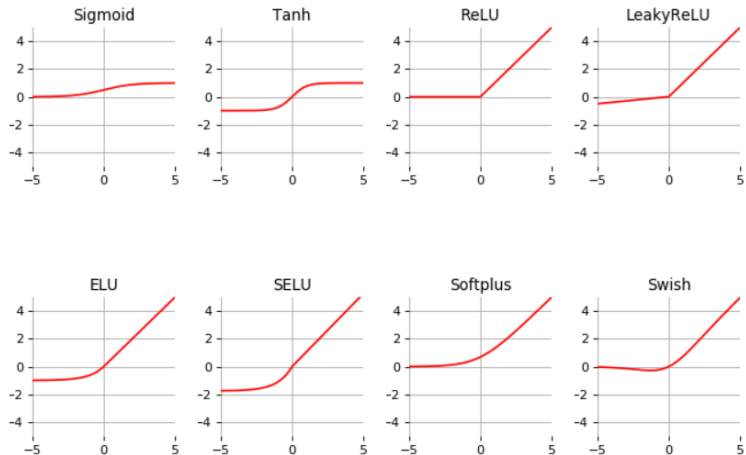
$$\max(0, \alpha x, x)$$

$$f(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0,01 & \text{otro caso} \end{cases}$$



Esta función de activación garantiza que haya un valor distinto de cero para valores negativos, resolviendo parcialmente los problemas de la función ReLU.

# Otras funciones de activación



## Otras soluciones: Redes Residuales

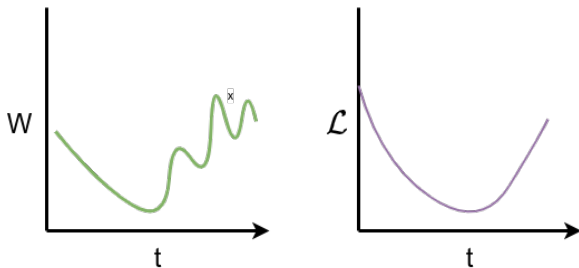


Las **ResNet** (Residual Networks) ofrecen varios beneficios para abordar el problema del desvanecimiento del gradiente:

- Facilitan el entrenamiento de redes más profundas al permitir que **los gradientes se propaguen directamente a través de las conexiones residuales**.
- Evitan la degradación del rendimiento a medida que se aumenta la profundidad de la red.
- Permiten el entrenamiento de arquitecturas extremadamente profundas con **cientos o incluso miles de capas**.

# El problema del Exploding Gradient

El "exploding gradient" es otro desafío común en el entrenamiento de redes neuronales profundas. Ocurre cuando **los gradientes se vuelven cada vez más grandes** a medida que se propagan hacia atrás en las capas más profundas de la red. Esto puede llevar a **inestabilidad en el proceso de optimización** y dificultar el entrenamiento de la red.



# Como detectar y monitorizar estos problemas

Esta parte la veremos en prácticas pero existe software específico que nos permite **monitorizar los pesos de una red neuronal** durante el entrenamiento de la misma.



## Otras funciones de activación importantes: Softmax

La función de activación **softmax** es muy utilizada en redes neuronales cuando tratamos con problemas de clasificación. Tiene como entrada un vector  $\mathbf{x}$  y produce una distribución de probabilidad  $\mathbf{p}$  sobre  $K$  clases.

Está definida como:

$$\text{softmax}(\mathbf{x}_i) = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}} \quad \text{para } i = 1, 2, \dots, K$$

Las principales propiedades de la función softmax son:

- La salida siempre es **positiva y suma uno**.
- Los valores de salida representan las **probabilidades de cada clase**.
- Evidentemente, es una **función diferenciable** (sino no se podría usar en una red).

# Otras funciones de activación importantes: Softmax

Ejemplo de uso de la función softmax:

Última capa de  
la red (logits)

5
6
11
7

$$\frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$$

.00
.01
.97
.01

$$p(y = 0)$$

$$p(y = 1)$$

$$p(y = 2)$$

$$p(y = 3)$$

## Importante

Ten en cuenta que esta función de activación se usa generalmente en la **última capa de la red** para convertir la salida de la red a probabilidades de pertenecer a cada clase. Como veremos más adelante no es adecuada para clasificación binaria (la sigmoide sería más adecuada), solo para clasificación multiclase.



# Diferenciación automática y grafos computacionales

---

# Diferenciación automática

- Como hemos visto en este tema, la **diferenciación** es una pieza clave para el **cálculo de los gradientes** y por tanto para el entrenamiento de una red neuronal.
- Frameworks como **PyTorch** y **TensorFlow** proporcionan herramientas que son capaces de calcular los gradientes para cualquier arquitectura de red propuesta.
- Estos frameworks implementan un mecanismo llamado **grafo computacional** para realizar el cálculo de gradientes de manera eficiente.



# Grafo computacional

- En PyTorch y TensorFlow, las operaciones se representan mediante un **grafo computacional**.
- El grafo está compuesto por **nodos** que representan las **operaciones** y los **artistas** que indican las **dependencias entre las operaciones**.
- El grafo captura la estructura de cómputo de un modelo y permite el cálculo de gradientes eficientemente.
- Durante la **pasada hacia adelante**, el grafo se construye registrando todas las operaciones realizadas.
- Durante el **propagación hacia atrás**, el grafo se utiliza para calcular los gradientes mediante la regla de la cadena.

## Ejemplo

A continuación veremos un ejemplo práctico de un grafo computacional para un problema simple de regresión lineal.

# Grafo computacional: Ejemplo

Supongamos que queremos usar una red neuronal con una sola neurona para construir un **modelo lineal** para aprender un problema de **regresión lineal**.



El modelo a aprender, con una sola neurona, será le siguiente:

$$\hat{y} = wx + b$$

La función de pérdida que utilizaremos será el **error cuadrático**:

$$\mathcal{L} = (\hat{y} - y)^2$$

# Grafo computacional: Ejemplo

Siguiendo la notación que hemos usado en transparencias previas, el modelo ( $\hat{y} = wx + b$ ) se podría representar de la siguiente manera:



## Importante

Ten en cuenta que como la entrada solo tiene una dimensión y solo tenemos un peso, hemos evitado el uso de subíndices. El término bias, también es un parámetro a aprender indicado por la letra  $b$ .

# Grafo computacional: Ejemplo

El **grafo computacional** del modelo sería el siguiente:



## Importante

Ten en cuenta que el grafo computacional representa las operaciones a realizar, divididas en **operaciones básicas** de las cuales **su derivada es conocida**.

# Grafo computacional: Ejemplo



En este grafo computacional, tenemos tres tipos de nodos:

- **Nodos de entrada**, en amarillo, representan los puntos de entrada pertenecientes al conjunto de entrenamiento.
- **Nodos de parámetros**, en violeta, corresponden a los parámetros del modelo.
- **Nodos de computación**, en cajas azules, que representan las operaciones intermedias necesarias para hacer una pasada hacia adelante del modelo.

# Grafo computacional: Ejemplo

Dado este grafo computacional, es muy sencillo hacer una **pasada hacia adelante**:



## Valores iniciales

Ten en cuenta que se ha considerado unos valores para los pesos  $w = 2$ ,  $b = 1$ . También se ha considerado un punto de entrada (3, 9).



## Grafo computacional: Ejemplo

Para realizar una **pasada hacia atrás**, deberíamos de calcular los gradientes de la función de pérdida  $\mathcal{L}$  con respecto a cada parámetro de la red ( $w$  y  $b$  en este caso). Ejemplo de cálculo de  $\frac{\partial \mathcal{L}}{\partial w}$ :

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial e} \frac{\partial e}{\partial w} = \frac{\partial \mathcal{L}}{\partial e} \frac{\partial e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = \frac{\partial \mathcal{L}}{\partial e} \frac{\partial e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w}$$

$$\frac{\partial \mathcal{L}}{\partial e} = 2e; \quad \frac{\partial e}{\partial \hat{y}} = 1; \quad \frac{\partial \hat{y}}{\partial z} = 1; \quad \frac{\partial z}{\partial w} = x = 3$$

$$\frac{\partial \mathcal{L}}{\partial w} = -12$$

# Grafo computacional: Ejemplo

Podemos visualizar la **pasada hacia atrás** gráficamente:



## Grafo computacional: Ejemplo

El mismo cálculo se podría hacer para calcular el gradiente de  $b$ ,  $\frac{\partial \mathcal{L}}{\partial b}$ :

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial e} \frac{\partial e}{\partial b} = \frac{\partial \mathcal{L}}{\partial e} \frac{\partial e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$$

$$\frac{\partial \mathcal{L}}{\partial e} = 2e; \quad \frac{\partial e}{\partial \hat{y}} = 1; \quad \frac{\partial \hat{y}}{\partial b} = 1;$$

$$\frac{\partial \mathcal{L}}{\partial b} = -4$$

### Importante

Como puedes ver en este ejemplo, hay varias derivadas parciales que son **comunes** para diferentes parámetros de la red. Los mecanismos de diferenciación automáticos son capaces de almacenar y **reutilizar** estos valores para evitar calcularlos múltiples veces.

# Grafo computacional: Ejemplo

Ejemplo de este gráfico computacional generado por **PyTorch** y su sistema de diferenciación automática **AutoGrad**:

```
z = w*x  
y_hat = z + b  
e = (y_hat - y_true)  
l = e ** 2  
l.backward()
```



# Diferenciación automática

Para que un sistema de diferenciación automática pueda funcionar:

- Los **bloques** de la red tienen que ser **diferenciables**, incluida la función de pérdida.
- Los **bloques** de la red deben estar **implementados** en el framework correspondiente.
- Los frameworks de aprendizaje profundo proveen maneras de **extenderlos** creando nuevos bloques para los que nosotros mismos indicamos la manera de derivarlos.

## Nota

Los frameworks de aprendizaje profundo proveen los bloques más comunes para la creación de redes neuronales, incluyendo funciones de pérdida, funciones de activación, etc. Esto hace que **la diferenciación sea un aspecto transparente** para los usuarios.

# Redes DAG

---

Hasta ahora todos los conceptos que hemos visto se basan en **redes totalmente conectadas**. Realmente muchas de las arquitecturas de redes neuronales van más allá de esta arquitectura, utilizando por ejemplo:

- **Conexiones no lineales**, es decir, cuando la salida de una capa no se conecta necesariamente con la entrada de la siguiente (un ejemplo son las ResNet vistas anteriormente).
- **Reutilización de pesos** en diferentes partes de la red. Por ejemplo en las redes convolucionales.

## Redes DAG

Las redes DAG (Grafo Dirigido Acíclico) permiten generalizar el concepto de red totalmente conectada para permitir arquitecturas más complejas.

# Redes DAG

Vamos a simplificar el grafo computacional de una red totalmente conectada con dos capas:

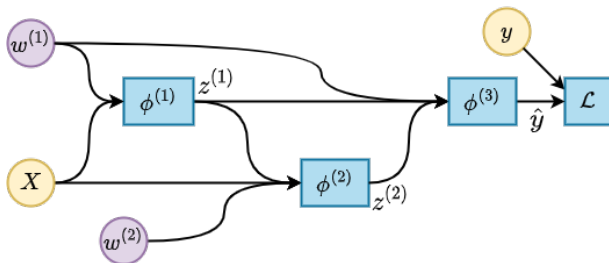


Con las redes DAG, podemos representar otros tipos de arquitecturas:





## Redes DAG: Pasada hacia adelante



Donde:

$$z^{(1)} = \phi^{(1)}(X; w^{(1)}) = w^{(1)}X$$

## Redes DAG: Pasada hacia adelante



Donde:

$$\begin{aligned} z^{(1)} &= \phi^{(1)}(X; w^{(1)}) = w^{(1)}X \\ z^{(2)} &= \phi^{(2)}(X, z^{(1)}; w^{(2)}) = w^{(2)}(z^{(1)} + X) \end{aligned}$$

## Redes DAG: Pasada hacia adelante



Donde:

$$\begin{aligned} z^{(1)} &= \phi^{(1)}(X; w^{(1)}) = w^{(1)}X \\ z^{(2)} &= \phi^{(2)}(X, z^{(1)}; w^{(2)}) = w^{(2)}(z^{(1)} + X) \\ \hat{y} &= \phi^{(3)}(z^{(1)}, z^{(2)}; w^{(1)}) = w^{(1)}(z^{(1)} + z^{(2)}) \end{aligned}$$

### Nota

Ten en cuenta que podríamos haber definido las funciones  $\phi$  de otra manera y obtener un gráfico computacional diferente.

## Redes DAG: Pasada hacia atrás



Como siempre, en la **pasada hacia atrás**, estamos interesados en calcular el gradiente, es decir, la derivada parcial la función de pérdida  $\mathcal{L}$  con respecto a los pesos:

$$\frac{\partial \mathcal{L}}{\partial w^{(1)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^{(1)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \left( \frac{\partial \hat{y}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w^{(1)}} + \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(1)}} \right)$$

## Redes DAG: Pasada hacia atrás



Como siempre, en la **pasada hacia atrás**, estamos interesados en calcular el gradiente, es decir, la derivada parcial la función de pérdida  $\mathcal{L}$  con respecto a los pesos:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^{(1)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \left( \frac{\partial \hat{y}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w^{(1)}} + \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(1)}} \right) \\ \frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(2)}}\end{aligned}$$

# Redes DAG: Reutilización de pesos



En la red anterior podemos ver como el peso  $w^{(1)}$  se **reutiliza** en dos partes de la red. Veremos como esto será un procedimiento común en varias arquitecturas que veremos en temas posteriores.

# Redes DAG: Grafo computacional

A la derecha se muestra el **gráfico computacional** en PyTorch correspondiente a la red anterior, considerando que la función de pérdida es MSE y tenemos una función de activación sigmoide antes de la última capa.

## Nota

Puedes comprobar como los frameworks de aprendizaje profundo permiten definir este tipo de arquitecturas y operar con ellas sin ningún tipo de problema.

