

Redes de Neuronas Artificiales

Aprendentatge Automàtic

APA/GEI/FIB/UPC - 2025/2026 1Q

 / Javier Béjar



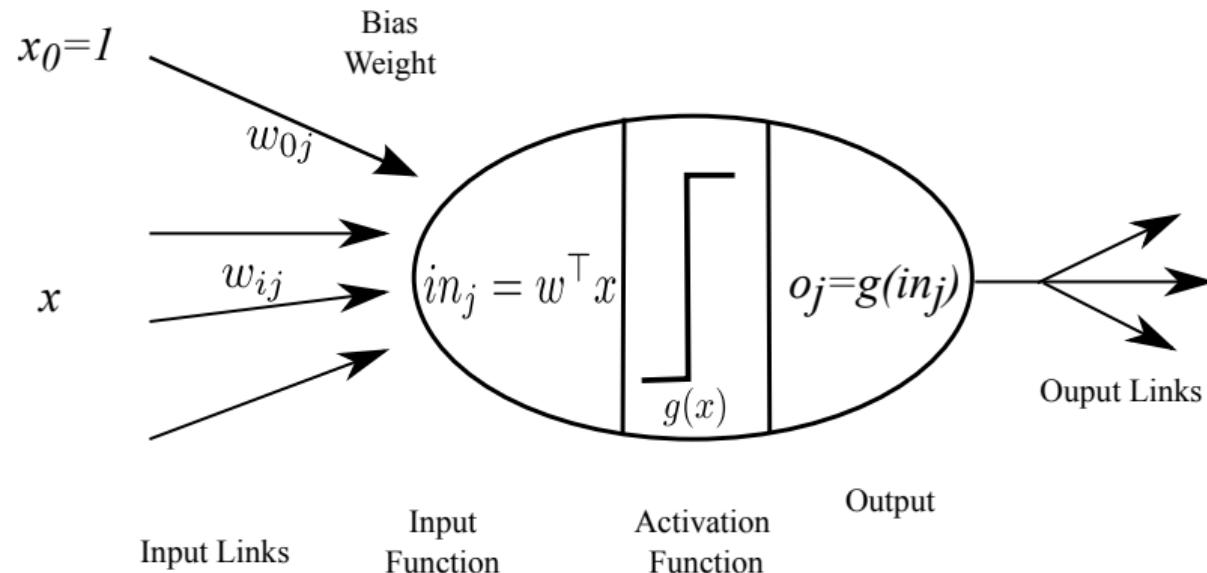
Introducción

- Las neuronas son los componentes básicos del cerebro
- Su conectividad define la programación que nos permite resolver todas nuestras tareas cotidianas
- Son capaces de realizar computación paralela y tolerante a fallos
- Podemos **inspirarnos** en su funcionamiento para definir modelos de aprendizaje
- Muchos de estos modelos son realmente simples (pero potentes)
- Los que se usan habitualmente no se parecen al funcionamiento real de las neuronas del cerebro



La tarea que realiza una sola neurona es muy simple

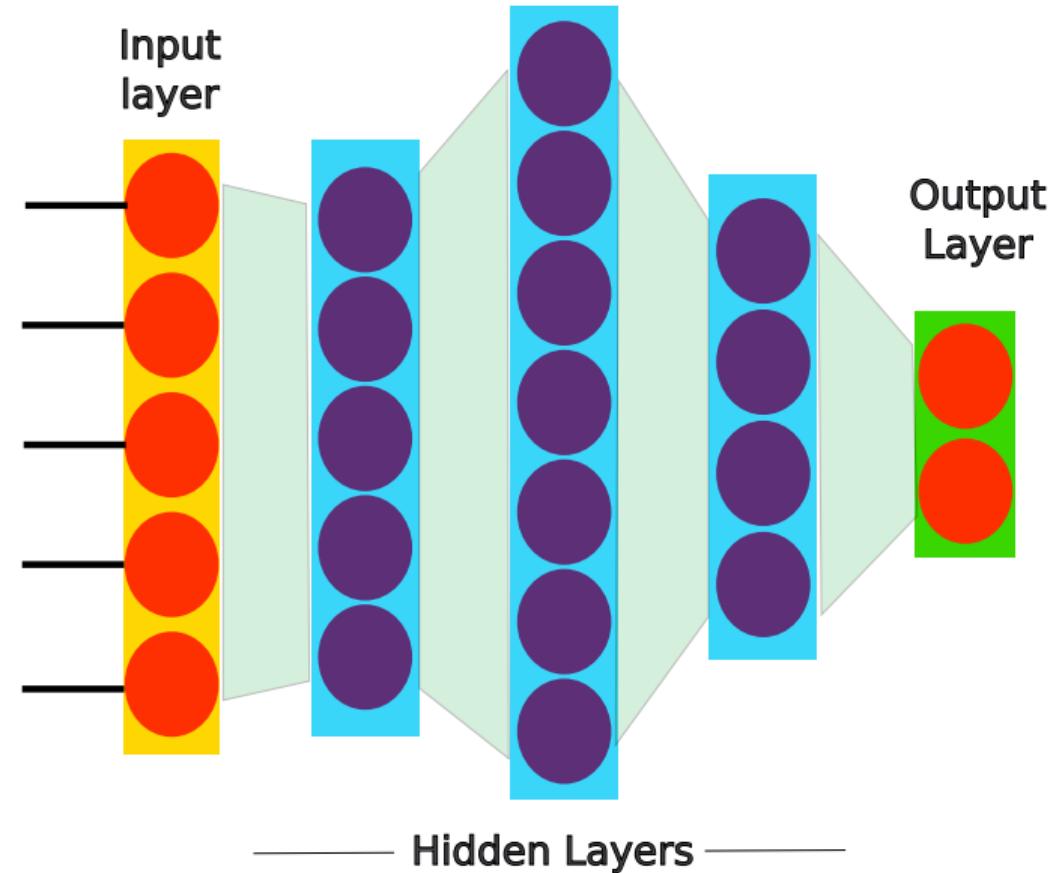
- Dado un conjunto de entradas, calcular una salida usando su valor combinado y una transformación posiblemente no lineal



- Ⓐ Una neurona tiene una entrada de sesgo (x_0) con valor 1
- Ⓑ Los pesos de la neurona (w_i) son los parámetros del modelo
- Ⓒ La entrada se calcula como una suma ponderada de las entradas (combinación lineal)
- Ⓓ La salida \hat{y} se obtiene aplicando la función de activación $g(x)$ a la entrada

$$\hat{y} = f(x, w) = g \left(\sum_{i=0}^I w_i x_i \right)$$

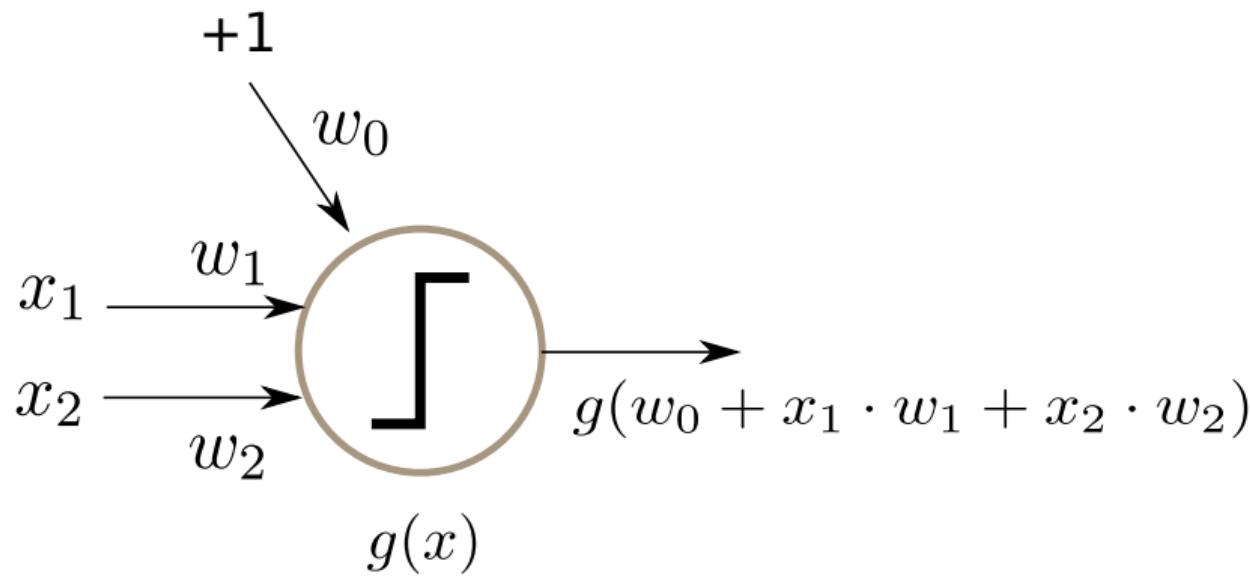
- Las redes *feed-forward* están organizadas en **capas**, cada una completamente conectada a la siguiente, formando un DAG (*Directed Acyclic Graph*)
 - Redes neuronales de una sola capa (perceptrón): capa de entrada, capa de salida
 - Redes neuronales multicapa (MLP): capa de entrada, capas ocultas, capa de salida
- La capa de entrada tiene tantas unidades como entradas en el problema
- La capa de salida tiene tantas unidades como sea necesario para la tarea
- Cada capa oculta tiene varias unidades (posiblemente un número diferente por capa)
- La entrada de cada unidad son todas las salidas de las unidades de la capa anterior

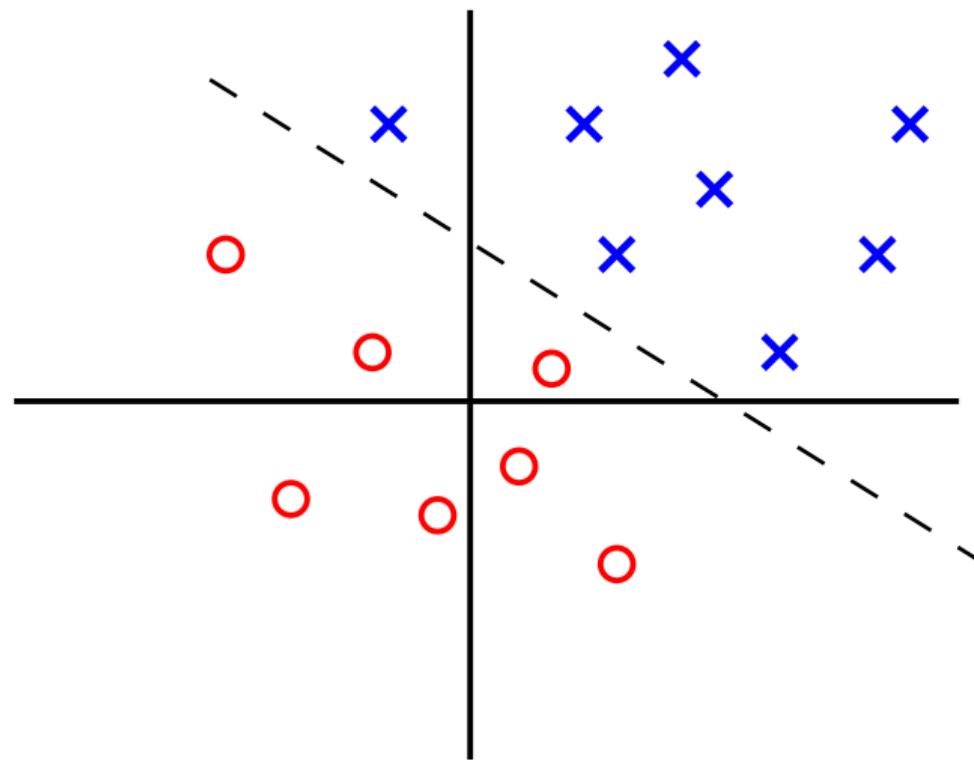


Redes neuronales de una sola capa

- Un perceptrón con una unidad de salida permite la clasificación binaria y regresión (se pueden usar varias unidades cuando hay más clases/salidas)
- El modelo con **una sola unidad lineal de salida** para la clasificación es una **función discriminante lineal** que divide los ejemplos en dos clases
- Podemos usar diferentes funciones de activación en la capa de salida para clasificación
 - Salida lineal = Discriminante lineal (el signo es la clase)
 - Salida sigmoide o tangente hiperbólica \approx Regresión logística (probabilidad de la clase)

Unidad de perceptrón para dos entradas

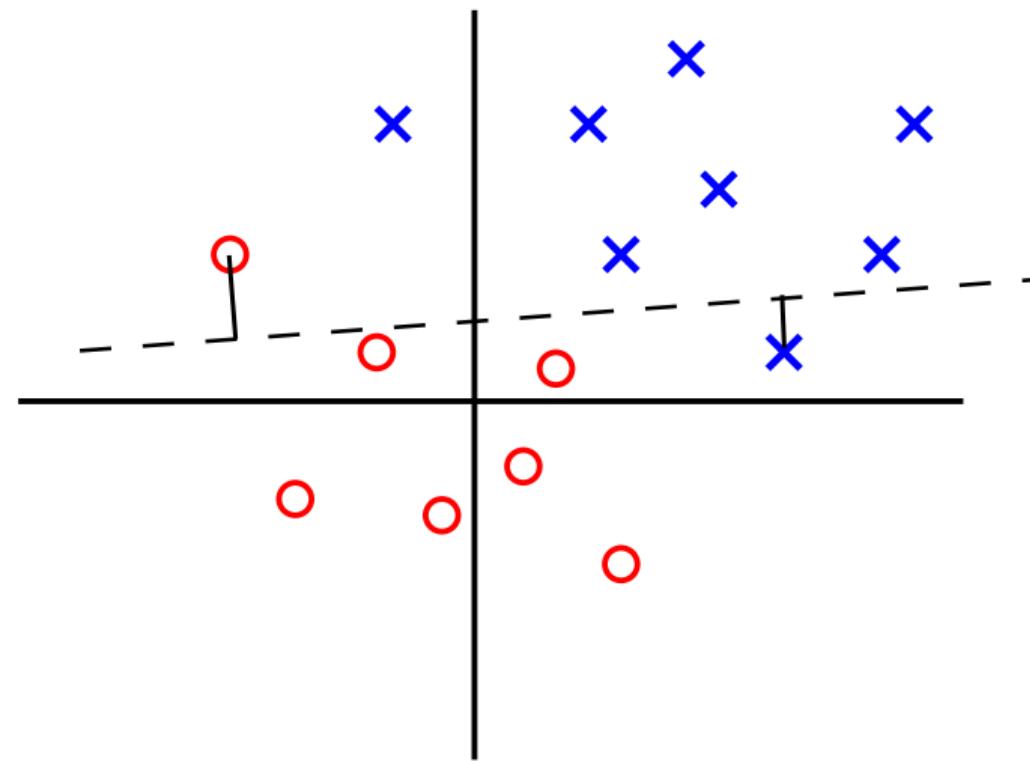




- Para aprender un discriminante lineal con el perceptrón usamos como etiquetas los valores $y_n \in \{-1, +1\}$
- Minimizando el error cuadrático como función de pérdida reducimos la distancia de los ejemplos mal clasificados a la función discriminante lineal que define el perceptrón
- Optimizamos:

$$\ell(y_n, \hat{y}_n) = \frac{1}{2} \sum_{n=1}^N (y_n - \hat{y}_n)^2$$

donde y_n es la etiqueta del ejemplo x_n y \hat{y}_n es la salida obtenida por el perceptrón $f(x_n, w) = w^\top x + w_0$



- El discriminante lineal se puede obtener mediante descenso de gradiente
- La actualización de los parámetros la obtenemos con la derivada de la función de pérdida respecto a los parámetros $\nabla \ell(x, w)$
- La regla de actualización será:

$$w \leftarrow w - \alpha \nabla \ell(w)$$

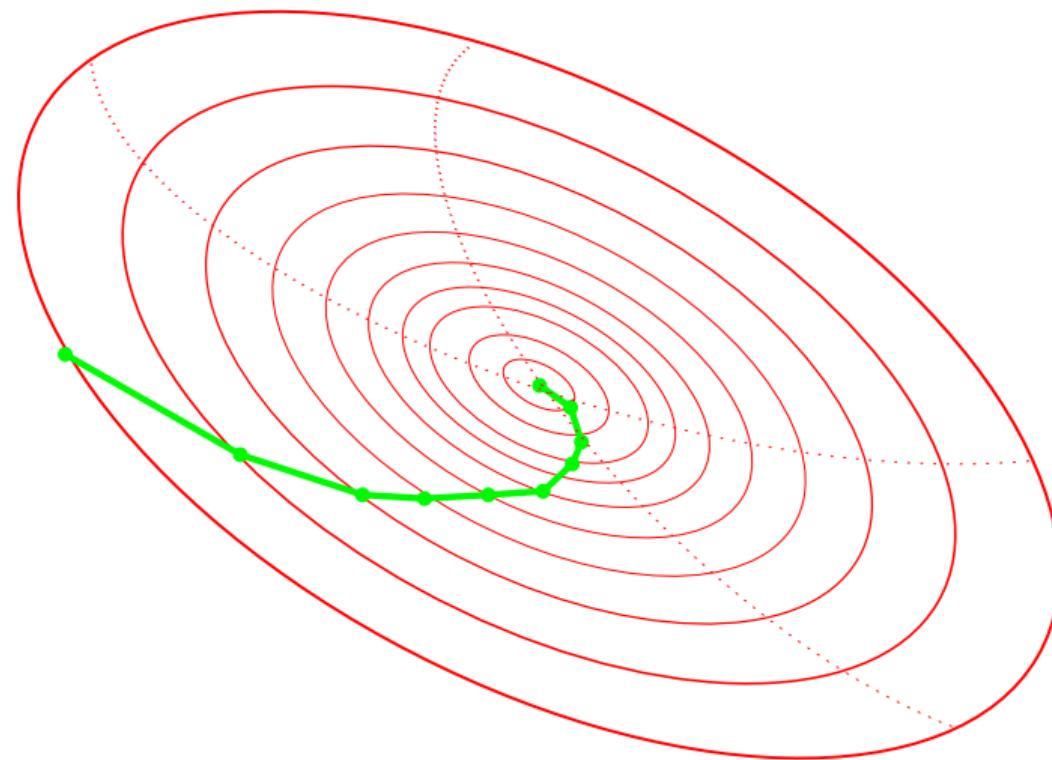
- Esta actualización se repite hasta converger (cada iteración se llama **epoch**)
- La actualización se puede también realizar con el promedio del gradiente de un grupo de ejemplos

$$\begin{aligned}\frac{\partial \ell}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{n=1}^N (y_n - \hat{y}_n)^2 \\&= \frac{1}{2} \sum_{n=1}^N \frac{\partial}{\partial w_i} (y_n - \hat{y}_n)^2 \\&= \frac{1}{2} \sum_{n=1}^N 2(y_n - \hat{y}_n) \frac{\partial}{\partial w_i} (y_n - \hat{y}_n) \quad (\text{regla de la cadena}) \\&= \sum_{n=1}^N (y_n - \hat{y}_n) \frac{\partial}{\partial w_i} (y_n - w^\top x_n) \\&= - \sum_{n=1}^N (y_n - \hat{y}_n) (x_{ni})\end{aligned}$$

- La regla de actualización es:

$$\Delta w_i = \alpha \cdot \sum_{n=1}^N (y_n - \hat{y}_n) x_{ni}$$

- El espacio de parámetros para unidades lineales está formado por una superficie hiperparaboloide, esto significa que existe un mínimo global
- El parámetro α ([tasa de aprendizaje](#)) tiene que ser lo suficientemente pequeño para no sobrepasar el mínimo, pero no demasiado pequeño para que tarde una eternidad en alcanzarlo.
- Por lo general, este parámetro α disminuye con las iteraciones ([decay](#))



Algoritmo: Perceptrón lineal descenso de gradiente

Input : $(\mathcal{X}, \mathcal{Y}) = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ (conjunto de entrenamiento, con etiquetas binarias $y_i \in \{+1, -1\}$)

Output: w (vector de pesos)

$t \leftarrow 0$

$w(t) \leftarrow$ pesos aleatorios

repetir

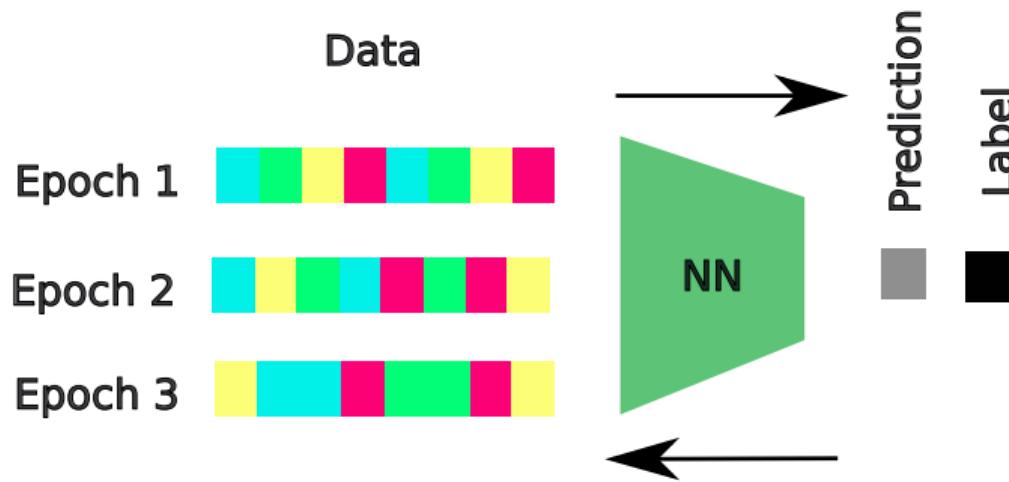
$$\Delta w(t) = \alpha \cdot \sum_{n \in \mathcal{X}} (y_n - \hat{y}_n) x_n$$

$$w(t + 1) = w(t) + \Delta w(t)$$

$t ++$

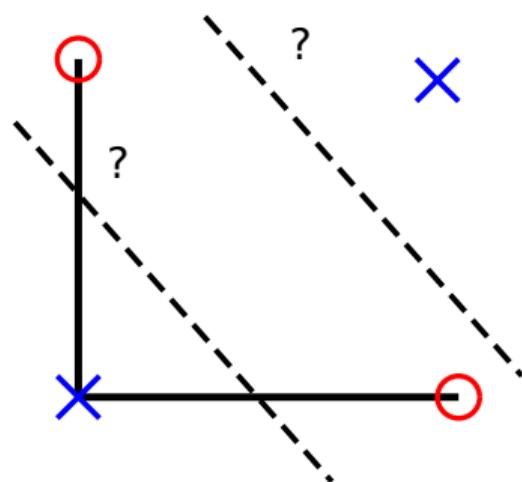
hasta sin cambios en w

- Si tenemos un conjunto de datos grande, calcular la actualización con todos los datos puede ser costoso
- El aprendizaje se puede realizar en línea, actualizando los pesos usando un ejemplo a la vez, esto se conoce como **Descenso de gradiente estocástico**
- Los ejemplos se barajan aleatoriamente en cada iteración (esto evita ciclos)



- ④ Usar solo un ejemplo podría significar tener una gran variación en el gradiente que usamos en las actualizaciones
- ④ Un compromiso es agrupar los ejemplos en submuestras (lotes, batch) aleatorias y promediar sus gradientes, esto se conoce como **Descenso de Gradiente Estocástico por Lotes (Minibatch SGD)**
- ④ La idea es que el **gradiente** de los ejemplos es una **variable aleatoria** que tiene como expectativa el gradiente real, por lo que se pueden usar **grupos de ejemplos aleatorios** para **estimar esa expectativa**

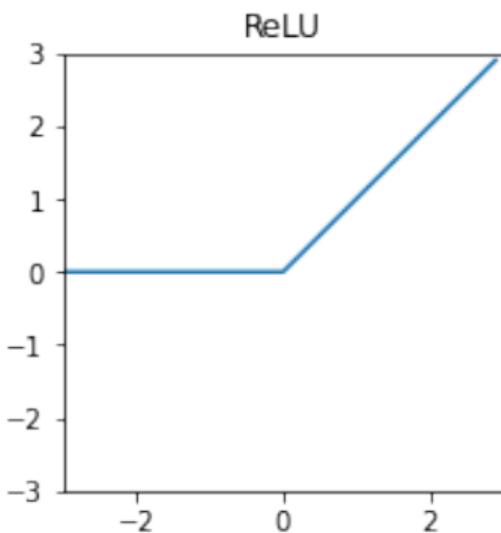
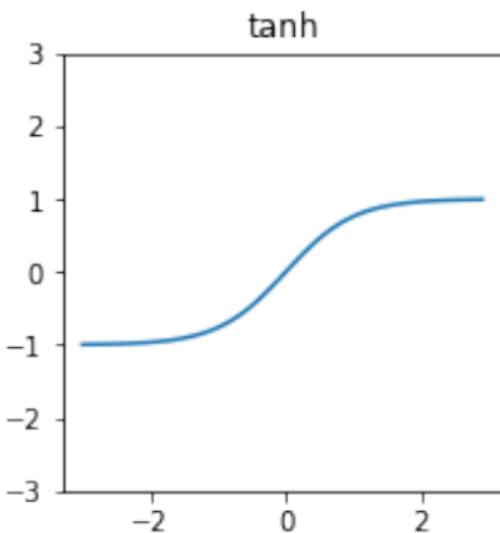
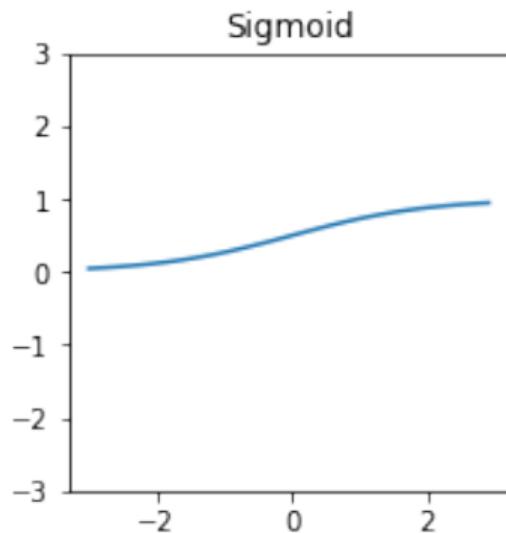
- Con los perceptrones lineales solo podemos clasificar correctamente problemas linealmente separables, de lo contrario no convergen
- El espacio de hipótesis no es lo suficientemente expresivo para problemas reales
- Ejemplo, la función XOR:



x_1	x_2	$f(x)$
0	0	1
0	1	0
1	0	0
1	1	1

Redes neuronales multicapa

- Las unidades del Perceptrón se pueden organizar en capas formando redes *feed forward*
- Cada capa realiza una transformación que se pasa a la siguiente capa como entrada
- La combinación de unidades lineales solo permite aprender funciones lineales
- Aprender funciones no lineales necesita de funciones de activación no lineales



- Usamos funciones no lineales que se ajustan durante el aprendizaje

$$\phi_i(x) = f_i(f_j(x, w_j), w_i) = g_i(w_{i0} + \sum_{m=1}^M w_{im}f_j(x, w_{jm}))$$

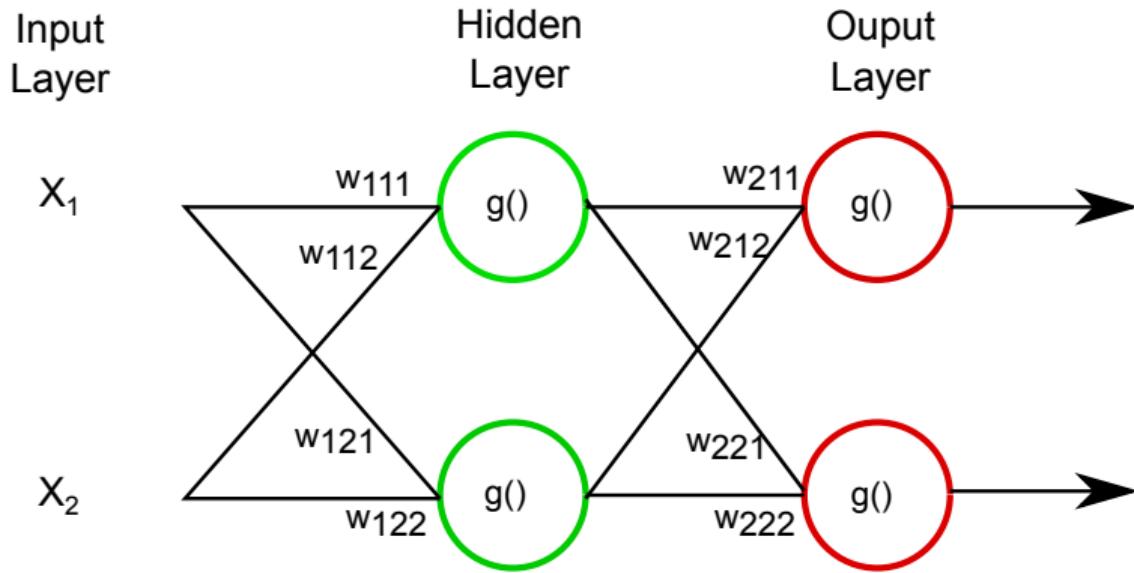
i es una unidad de perceptrón en la capa actual, f_j es la transformación que se obtiene de todas las capas anteriores, que son la entrada de i

- Las funciones base (ϕ) que transforman la entrada ahora no son fijas, las aprendemos durante el entrenamiento
- Aprendemos las características que mejor se adaptan al problema
- Ahora también tenemos funciones que **no son lineales** con respecto a los parámetros

- ⑤ Las redes multicapa permiten aproximar cualquier función dadas suficientes unidades por capa y suficientes capas ([Funciones de aproximación universal](#))
 - Cualquier función booleana es representable por un MLP de dos capas (incluido XOR) (el número de unidades ocultas crece exponencialmente con el número de entradas)
 - Cualquier función continua se puede aproximar mediante una red de dos capas (unidades ocultas sigmoideas, unidades de salida lineal)
 - Cualquier función arbitraria se puede aproximar mediante una red de tres capas (unidades ocultas sigmoideas, unidades de salida lineal)
- ⑥ La arquitectura de la red define las funciones que se pueden aproximar, pero no sabemos cuáles son

- ④ Regresión univariante: una neurona de salida con una función de activación lineal
- ④ Regresión multivariante: tantas neuronas de salida como funciones de salida con función de activación lineal
- ④ Clasificación binaria: una neurona de salida con una función de activación sigmoidea
- ④ Clasificación de clases múltiples: tantas salidas como clases con una activación softmax

- ④ Para redes de **una sola capa** cuando tenemos múltiples salidas, podemos aprender **cada salida por separado**
- ⑤ En una **red multicapa**:
 - Tenemos un conjunto de parámetros para cada unidad de la capa y cada capa está completamente conectada a la siguiente capa
 - Todas las salidas de la red están interconectadas por la interdependencia generada por las capas ocultas
 - La actualización de los pesos de una salida afecta las conexiones con otras salidas



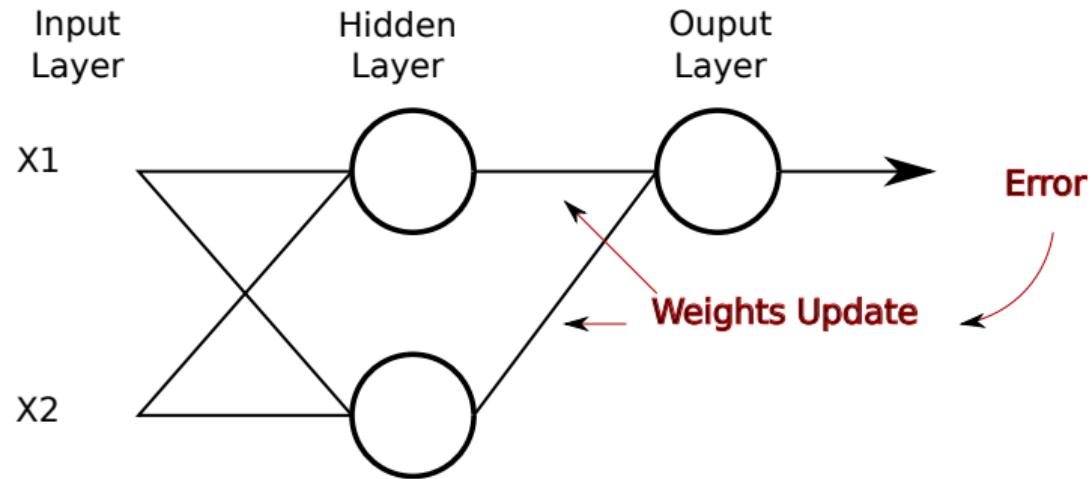
- Toda red neuronal es un **grafo dirigido acíclico** (DAG) y los cálculos para el entrenamiento deben seguir ese grafo
- El algoritmo utilizado para el entrenamiento se llama **Backpropagation**
- Es un algoritmo de **programación dinámica** que utiliza el **orden topológico del grafo** para **calcular los gradientes** de la función dado un conjunto de ejemplos

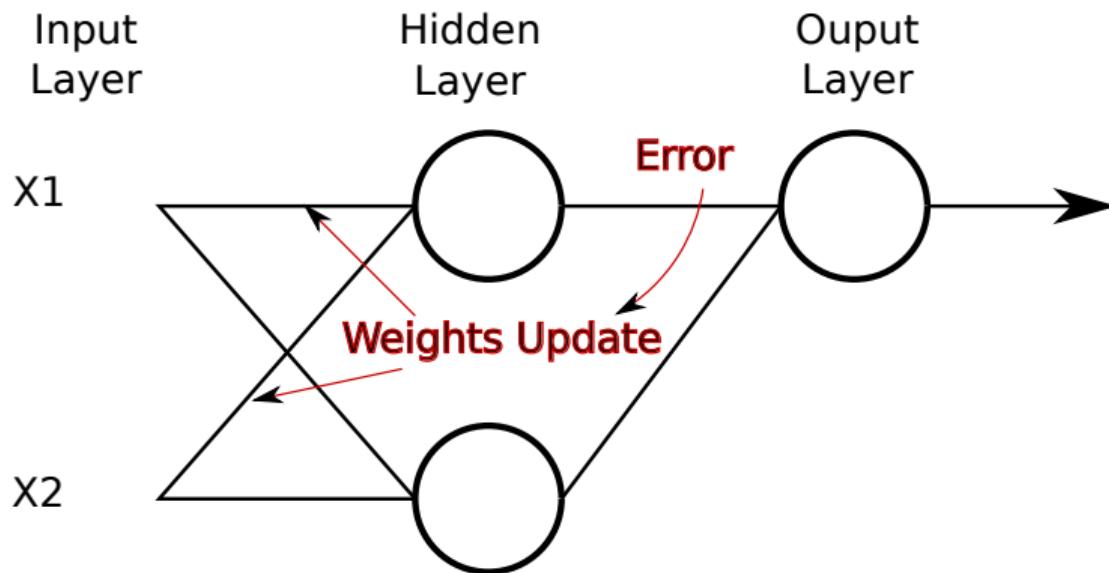
- ⑤ La programación dinámica es necesaria para evitar cálculos repetidos dado que algunas operaciones son compartidas por varios caminos en el grafo
- ⑥ Los parámetros de cada capa se calculan y actualizan al mismo tiempo
- ⑦ Combina dos pasos
 1. El **paso hacia adelante** evalúa la función para los ejemplos para calcular el error
 2. El **paso hacia atrás** calcula el gradiente de la función y la actualización de los pesos

1. Propagar los ejemplos a través de la red para obtener la salida ([propagación hacia adelante](#))
 - o Para cada capa, cada unidad calcula las combinaciones lineales de las entradas y los pesos
 - o Cada unidad calcula las funciones de activación y pasa la salida a todas las unidades de la siguiente capa
2. Propagar el error de salida capa por capa actualizando los pesos de las neuronas ([propagación hacia atrás](#))
 - o Cada unidad calcula la diferencia entre el valor estimado obtenido por la siguiente capa y el valor real de la unidad
 - o La diferencia se propaga a cada unidad de la capa anterior proporcionalmente a su peso y los pesos se actualizan

- El error del **perceptrón de una sola capa** vincula directamente la transformación de la entrada en La salida
- En el caso de **múltiples capas** cada capa tiene su propio error
- El error de la capa de salida es directamente el error calculado a partir de los valores verdaderos
- El error de las capas ocultas es más difícil de definir, ya que no es la diferencia con el valor final

- La idea es utilizar el error de la siguiente capa para influir en los pesos de la capa anterior
- Estamos propagando hacia atrás el error de la salida, de ahí el nombre de **backpropagation**
- El error de las unidades en la capa salida se distribuye a la capa anterior proporcionalmente a los pesos de las conexiones
- El proceso se repite hasta llegar a la capa de entrada



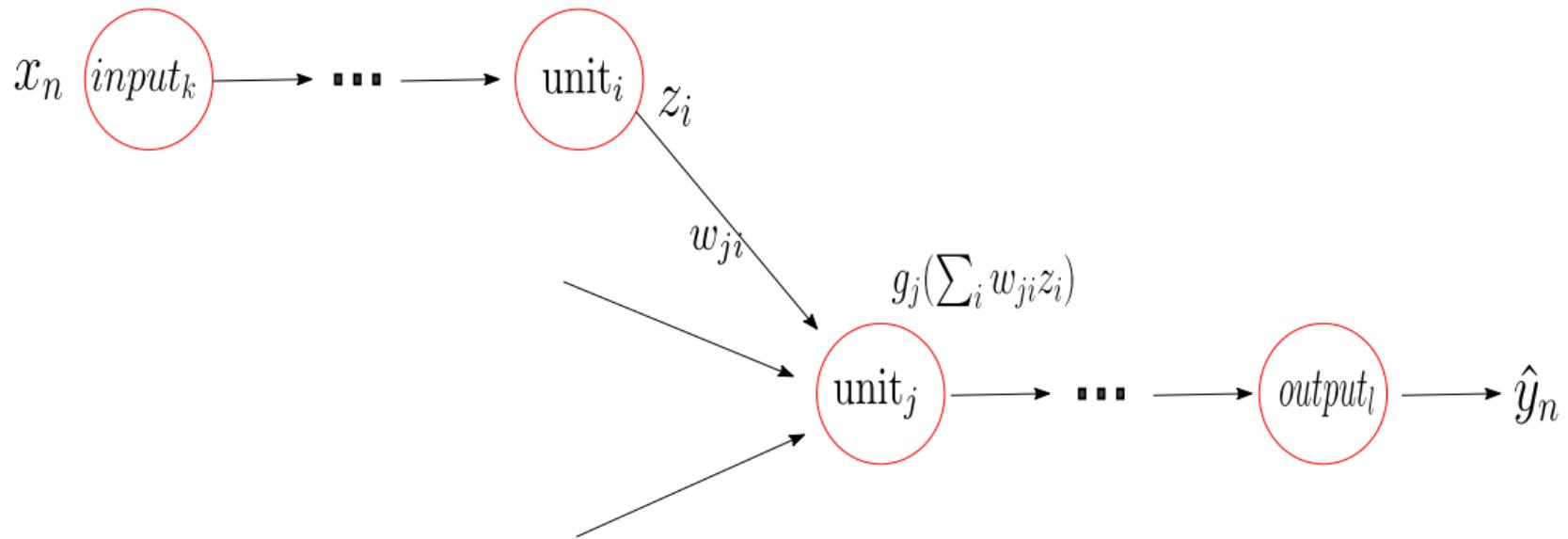


- Un MLP calcula en una unidad la función:

$$a_j = \sum_i w_{ji} z_i$$

donde z_i es la salida calculada por la unidad i de la capa anterior y w_{ji} es el peso de la conexión entre la unidad i y la unidad j

- La salida de la unidad j se calcula como $z_j = g(a_j)$ donde g es una función de activación
- El paso hacia adelante calcula en orden todos estos valores hasta que llega a la salida como $\hat{y}_n = f(x_n, w)$ para que se pueda calcular el error de la capa de salida



- Para recalcular los pesos w_{ji} necesitamos obtener la derivada del error respecto a los pesos para cada ejemplo

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}}$$

- El error depende de los pesos solo en la suma de a_j , por lo que aplicando la regla de la cadena:

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = \frac{\partial \ell(x_n, w)}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

dónde dado que $a_j = \sum_i w_{ji} z_i$

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

- ④ La derivada del error respecto a la unidad j se llama **término de error**

$$\delta_j = \frac{\partial \ell(x_n, w)}{\partial a_j}$$

- ④ Este término depende de la función de error que estemos usando
- ④ Entonces la derivada del error respecto a los pesos se puede expresar como:

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = \delta_j z_i$$

- Si usamos como función de pérdida el error cuadrático como vimos con la regla delta (ver diapositiva 14) tenemos que el error se puede calcular como:

$$\frac{\partial \ell(x_n, w)}{\partial a_j} = \frac{\partial \frac{1}{2}(y_n - a_j)^2}{\partial a_j} = -(y_n - a_j) \quad (1)$$

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = -(y_n - a_j) \frac{\partial a_j}{\partial w_{ji}} = -(y_n - a_j) z_i \quad (2)$$

- Si asumimos una red con una sola capa y la función identidad como función de activación ($a_j = w_{ji}x_{ni}$) obtenemos la regla delta

$$\frac{\partial \ell(x_n, w)}{\partial w_{ji}} = -(y_n - a_j)x_{ni}$$

- ④ El error solo depende de la salida real, ya que no hay vínculo con otras unidades
- ④ El gradiente depende de la derivada de la función de activación de la salida y la función de error
- ④ Esto es lo que estamos optimizando en los métodos lineales

- Para las unidades de las capas ocultas, cada unidad acumula las derivadas de las unidades a las que está conectada, aplicando la regla de la cadena:

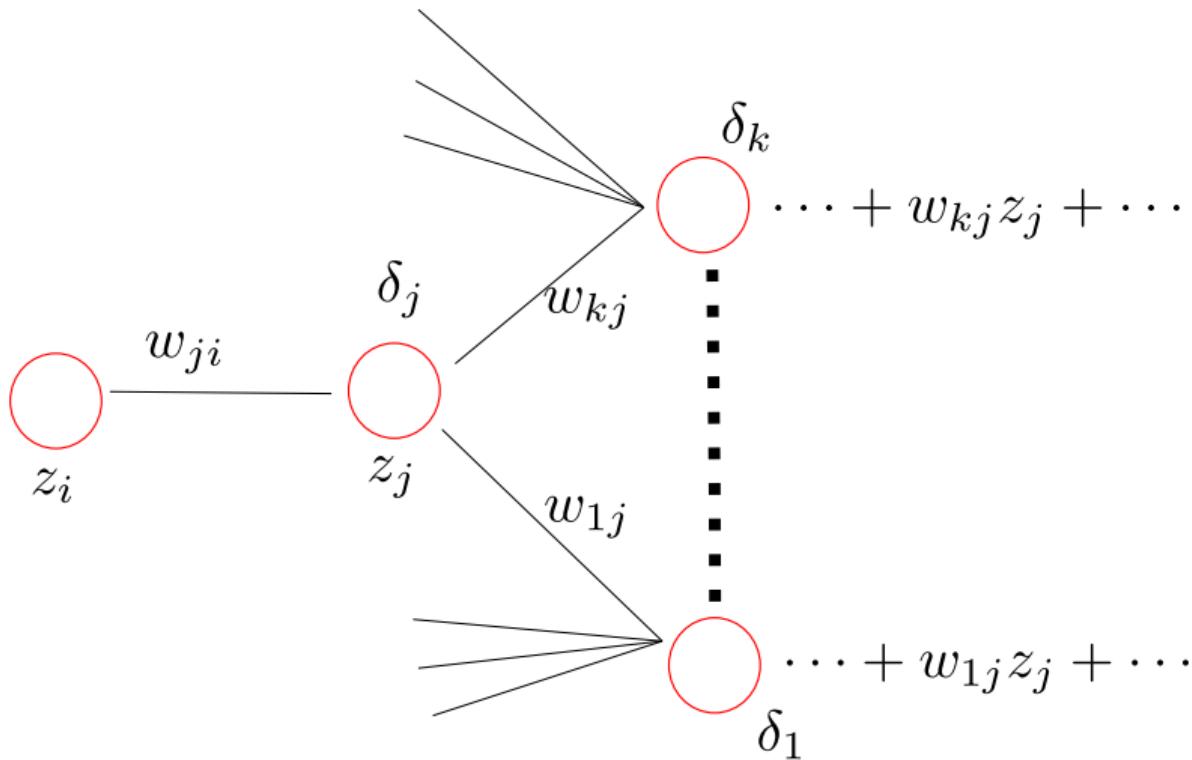
$$\delta_j = \frac{\partial \ell(x_n, w)}{\partial a_j} = \sum_k \frac{\partial \ell(x_n, w)}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j}$$

sumamos todas las unidades k de la siguiente capa con la que j está conectada

- Como $a_k = w_{kj}z_j$ y $z_j = g(a_j)$, obtenemos

$$\delta_j = \sum_k \delta_k \frac{\partial w_{kj}z_j}{\partial a_j} = \sum_k \delta_k w_{kj} \frac{\partial g(a_j)}{\partial a_j} = g'(a_j) \sum_k w_{kj} \delta_k$$

Así que simplemente propagamos hacia atrás el δ_k desde la siguiente capa proporcionalmente a los pesos de conexión



- Con los gradientes para cada variable podemos recalcular los pesos usando el descenso de gradiente (todo/estocástico/mini lote estocástico)
- Para la capa de salida:

$$w_{ji}^{t+1} = w_{ji}^t - \alpha \delta_j z_i$$

- Para las capas ocultas

$$w_{ji}^{t+1} = w_{ji}^t - \alpha \left(g'(a_j) \sum_k w_{kj} \delta_k \right) z_i$$

Usando funciones de activación sigmoideas, de modo que $g(a_j) = z_j = \sigma(a_j)$:¹

$$w_{ji}^{t+1} = w_{ji}^t - \alpha \left(z_j (1 - z_j) \sum_k w_{kj} \delta_k \right) z_i$$

¹ $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

- ⑤ Actualmente, las redes neuronales se definen directamente como grafos computacionales
- ⑤ Una red neuronal puede tener en sus nodos muchos tipos de cómputos interconectados con otros nodos, cada uno necesita una derivada y una forma de calcular el paso adelante y atrás
- ⑤ Todos estos cálculos se definen **programáticamente** y se combinan en tiempo de ejecución usando **diferenciación automática**
- ⑤ Esto evita tener que definir los cálculos a realizar si se define la red con capas estándar y permite añadir nuevas capas definiendo como se calcula el paso hacia adelante y atrás

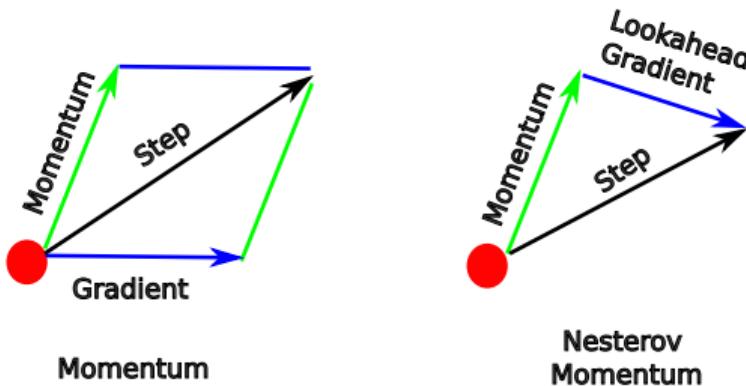
- Para redes neuronales multicapa, el espacio de hipótesis es el conjunto de todos los valores de pesos posibles que las unidades pueden tener
- La superficie determinada por estos parámetros y la función de error pueden tener muchos óptimos locales
- Esto significa que la solución puede cambiar dependiendo de la inicialización
- La convergencia también puede ser muy lenta para ciertos problemas

- Una mejora introducida para obtener una convergencia más rápida es el uso del método llamado **momento**
- Este método altera la actualización de los pesos incluyendo una parte ponderada de la anterior actualización Δw_{ji}^{t-1} :

$$w_{ji}^{t+1} \leftarrow w_{ji}^t + \alpha \Delta w_{ji}^t + \mu \Delta w_{ji}^{t-1} \text{ con } 0 \leq \mu < 1$$

- El efecto es mantener la exploración en la misma dirección, permitiendo saltar y realizar pasos más largos, lo que ayuda a evitar mínimos locales obteniendo a una convergencia más rápida

- **Momento de Nesterov** es una mejora que usa una mirada un paso hacia adelante y cambia la dirección del gradiente actual de acuerdo con un gradiente futuro

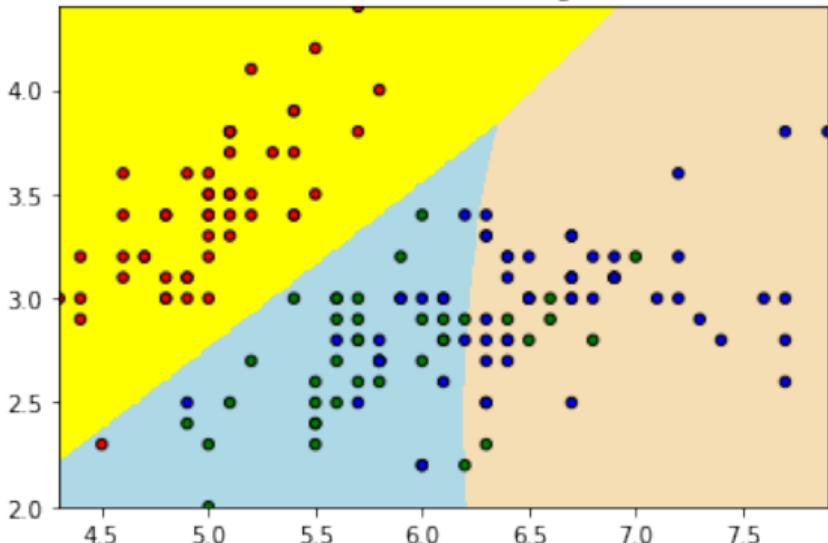


- Se pueden usar variaciones más sofisticadas de Gradient Descent para una convergencia más rápida usando reglas de actualización que adaptan la velocidad de aprendizaje o realizan actualizaciones de diferente paso para cada peso (Adam, Adamax, RMSProp, Adagrad...)

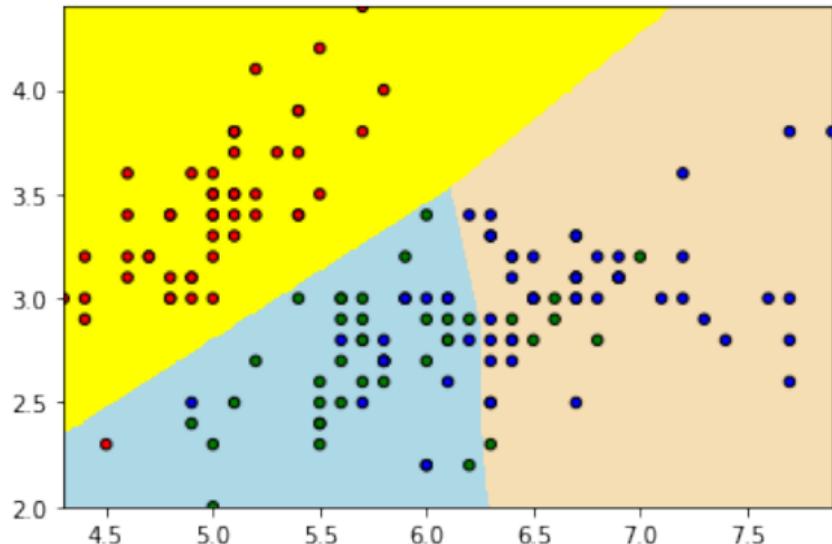
- La condición más obvia para terminar la optimización es cuando el error no se puede reducir más, por lo general esto conduce a **sobre ajuste**
- Este sobreajuste ocurre durante las últimas iteraciones del algoritmo
- En este punto los pesos intentan minimizar el error ajustando características específicas de los datos (generalmente el ruido)
- Una posibilidad es usar regularización, por ejemplo L_2 o L_1 (decaimiento de pesos, **decay**)
- También se puede limitar el número de iteraciones del algoritmo, esto se llama **terminación temprana**

- Optimizar una red neuronal no es tarea fácil
 - Decidir el número de capas
 - Decidir las unidades por capa
 - Decidir la función de error
 - Decidir la función de activación (por capa)
 - Decidir el algoritmo de optimización y sus parámetros
 - ¿Usar terminación temprana?
 - ¿Momento?
 - ¿Regularización?
 - ...
- El coste computacional dificulta el uso de la validación cruzada
- El tamaño del conjunto de datos importa (muchos parámetros a ajustar)

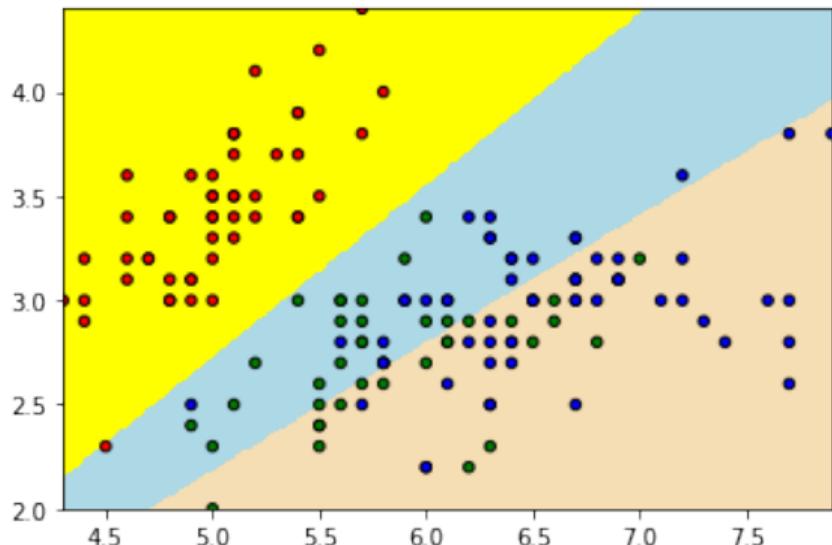
3-Class classification MLP - act=logistic - neu = 100



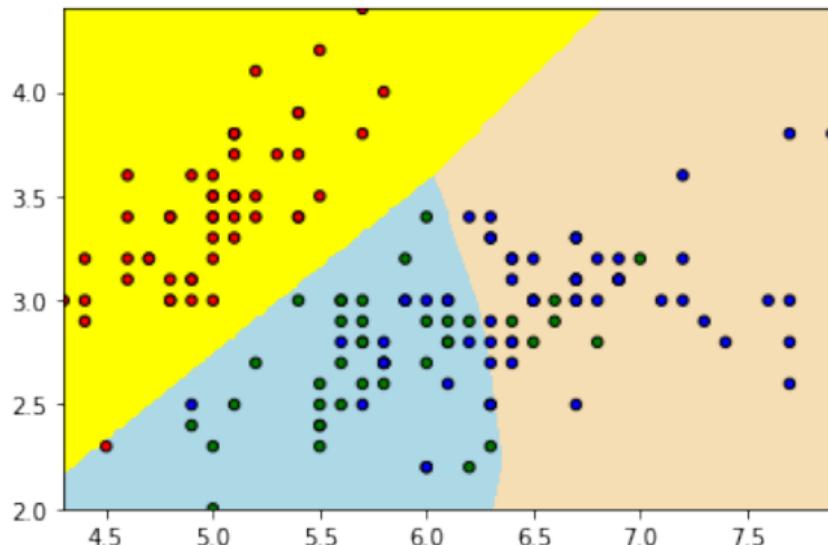
3-Class classification MLP - act=relu - neu = 100



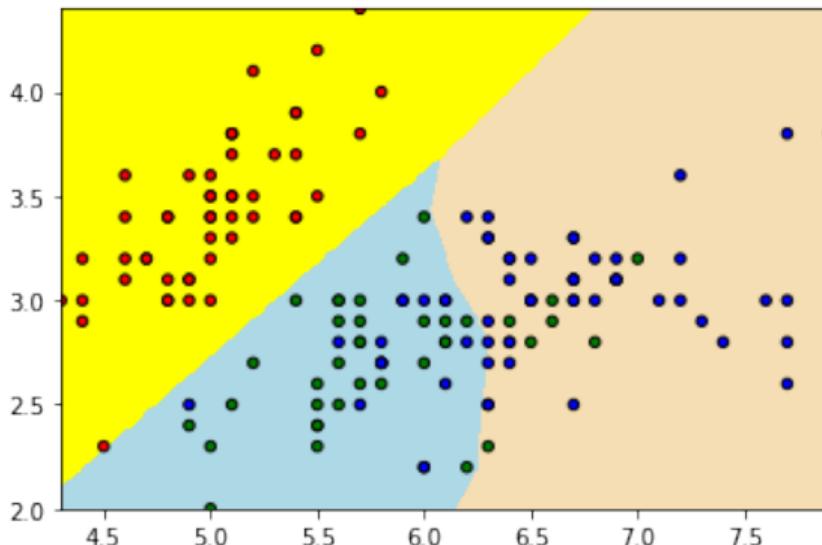
3-Class classification MLP - act=relu - neu = 10



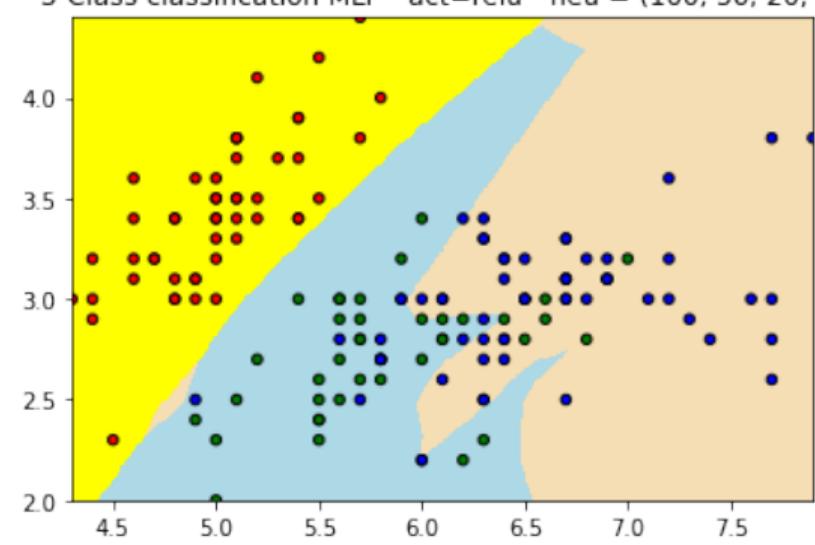
3-Class classification MLP - act=relu - neu = 1000



3-Class classification MLP - act=relu - neu = (50, 50)



3-Class classification MLP - act=relu - neu = (100, 50, 20, 5)





Este [notebook](#) muestra la diferencia entre entrenar redes neuronales usando CPU (scikit-learn) y GPU (con tensorflow)

También podéis ver un [video](#) que explica el contenido del notebook

Este [notebook](#) implementa un MLP con una sola capa oculta desde cero en python para mostrar cómo funciona backpropagation

Capas especializadas

- El MLP es una función aproximadora universal, pero hay aplicaciones donde capas especializadas son más eficientes
- Las redes neuronales permiten aplicaciones que van más allá de los modelos que hemos visto
- La potencia de las redes viene de su capacidad para aprender características sobre los datos
- El extraer esas características necesita de cálculos especializados

- ④ Una de las aplicaciones más extendidas de las redes neuronales es la visión artificial
- ④ El objetivo es procesar matrices de píxeles sobre la que está definida una tarea, por ejemplo
 - Clasificación de objetos, localización de objetos, segmentación
- ④ Estas tareas son demasiado complejas de definir para otros modelos de aprendizaje
- ④ El número de parámetros que tendría un MLP para procesar imágenes lo hace impracticable
- ④ Podemos definir modelos especializados de neuronas para estos propósitos

- ① Las redes convolucionales son la arquitectura más extendida en visión artificial
- ① Se inspiran en el funcionamiento del córtex visual
- ① Son redes que tienen una conectividad local a diferencia del MLP donde todo está conectado
- ① Esa conectividad local reduce el coste computacional
- ① Se pueden definir para el tratamiento de datos de entrada con múltiples dimensiones (1D, 2D, 3D...)

- La neurona convolucional se corresponde con una matriz de pesos con un tamaño fijo
- Esa neurona se traslada a través de la matriz de datos de entrada
- Para cada elemento de la matriz se calcula la combinación lineal de los elementos que lo rodean con la matriz de pesos
- La salida de una neurona convolucional es una nueva matriz de datos
- Una capa convolucional se define por:
 - El tamaño de la matriz de pesos (kernel)
 - El paso con el que se aplica (stride)
 - El número de neuronas (filtros)
 - Qué se hace cuando se aplica fuera de la matriz de datos (padding)

0	0	0	0	0	0
0	1	3	2	0	0
0	2	3	4	1	0
0	0	2	3	1	0
0	5	2	1	3	0
0	0	0	0	0	0

Input

3x3 kernel

2	2	0
3	1	1
0	1	0

6	9	15	

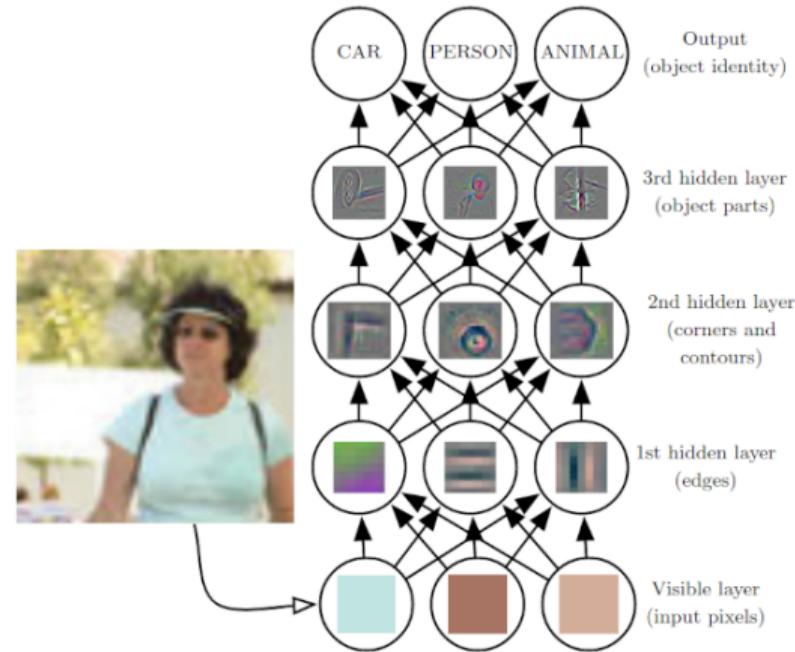
Feature Map

- Cada neurona convolucional genera un **mapa de características** (feature map)
- Estos corresponden a los atributos extraídos por la neurona (filtro)
- La propagación hacia atrás ajusta los valores de la matriz del filtro convolucional
- Por lo general el resultado de los mapas de características se agrupa de alguna manera disminuyendo el tamaño de la entrada
 - Calculando el máximo, mínimo, media de grupos de posiciones
- Esto se puede repetir varias veces hasta llegar al conjunto de capas específicas para la tarea, por lo general un MLP

Aprendizaje profundo

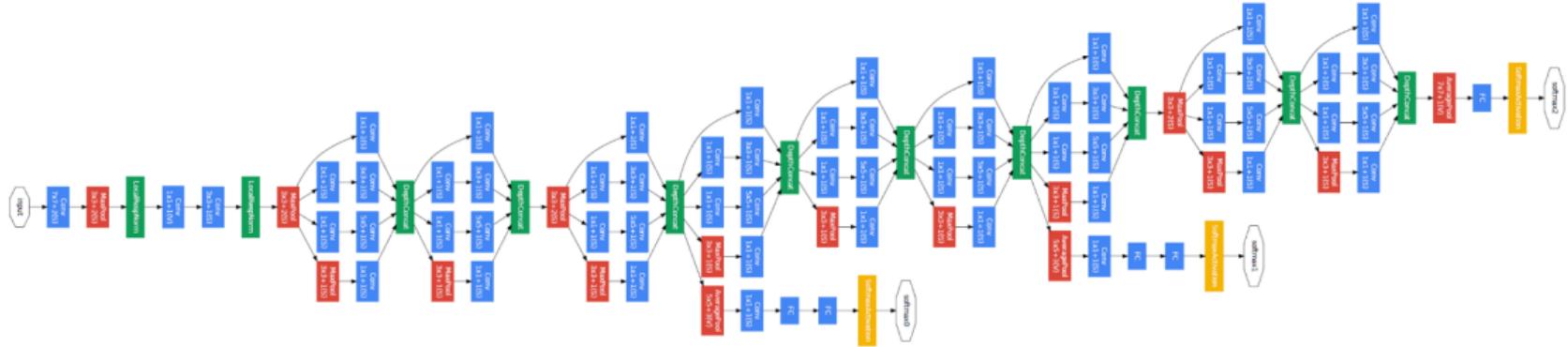
- ⑤ La mayor limitación de la aplicación a problemas reales es su gran coste computacional
- ⑥ El año 2012 marca un punto de inflexión al empezar a utilizarse hardware especializado (GPUs)
- ⑦ Ese año una red de 8 capas ganó la competición Imagenet reduciendo el error de clasificación top-5 al 15%
 - Imagenet es un conjunto de datos de 1.2M de imágenes con 1000 clases
- ⑧ Tres años después una red de 152 capas (ResNet) lo redujo al 3.5%

- La ventaja de la profundidad es obtener mejores características para la tarea a resolver
- Algunas redes muestran la capacidad de obtener características con sentido semántico
- La semántica de estas características se hace más compleja con la profundidad

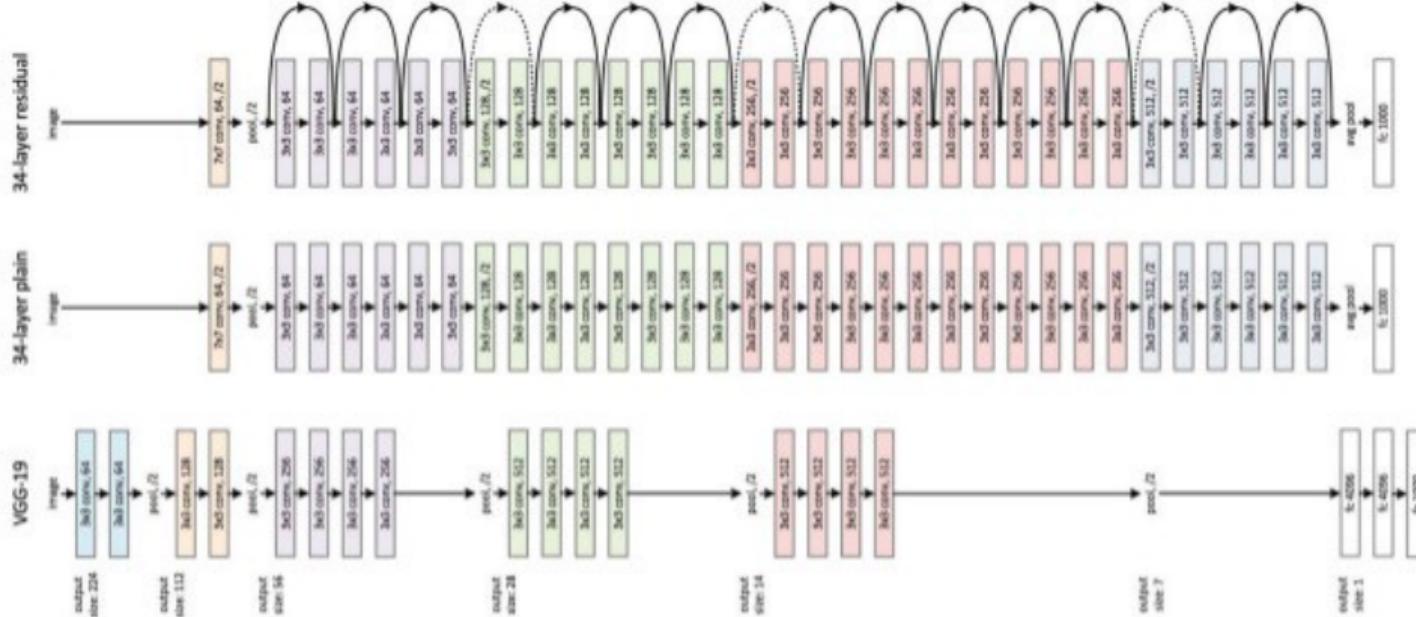


- ⑤ Visión artificial ha sido un área muy beneficiada
- ⑤ Ha impulsado del desarrollo de nuevas capas, organización de la arquitectura y funciones de activación
- ⑤ Se han desarrollado una gran variedad de arquitecturas para diferentes problemas de visión
 - Clasificación: Alexnet, VGG, Inception, ResNet, ResNeXt, EfficientNet...
 - Detección de objetos: R-CNN, Yolo, Retina-net...
 - Segmentación de imágenes: SegNet, FastFCN, U-net,

Inception network



ResNet



- Deep learning ha permitido mejorar tareas de procesamiento de imagen e inventar nuevas aplicaciones
 - Coloreado de imágenes
 - Eliminación de ruido
 - Completado de imágenes
 - Superresolución
 - Transferencia de estilo
 - Traducción de imagen a imagen (por ejemplo deep fakes)
 - ...

bicubic
(21.59dB/0.6423)



SRResNet
(23.53dB/0.7832)



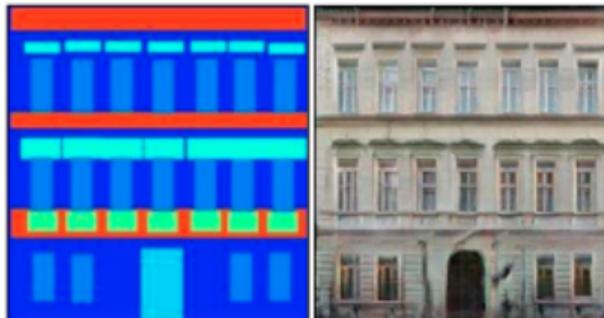
SRGAN
(21.15dB/0.6868)



original



Labels to Facade



input

output



input

BW to Color



output

Day to Night



input



output

Edges to Photo

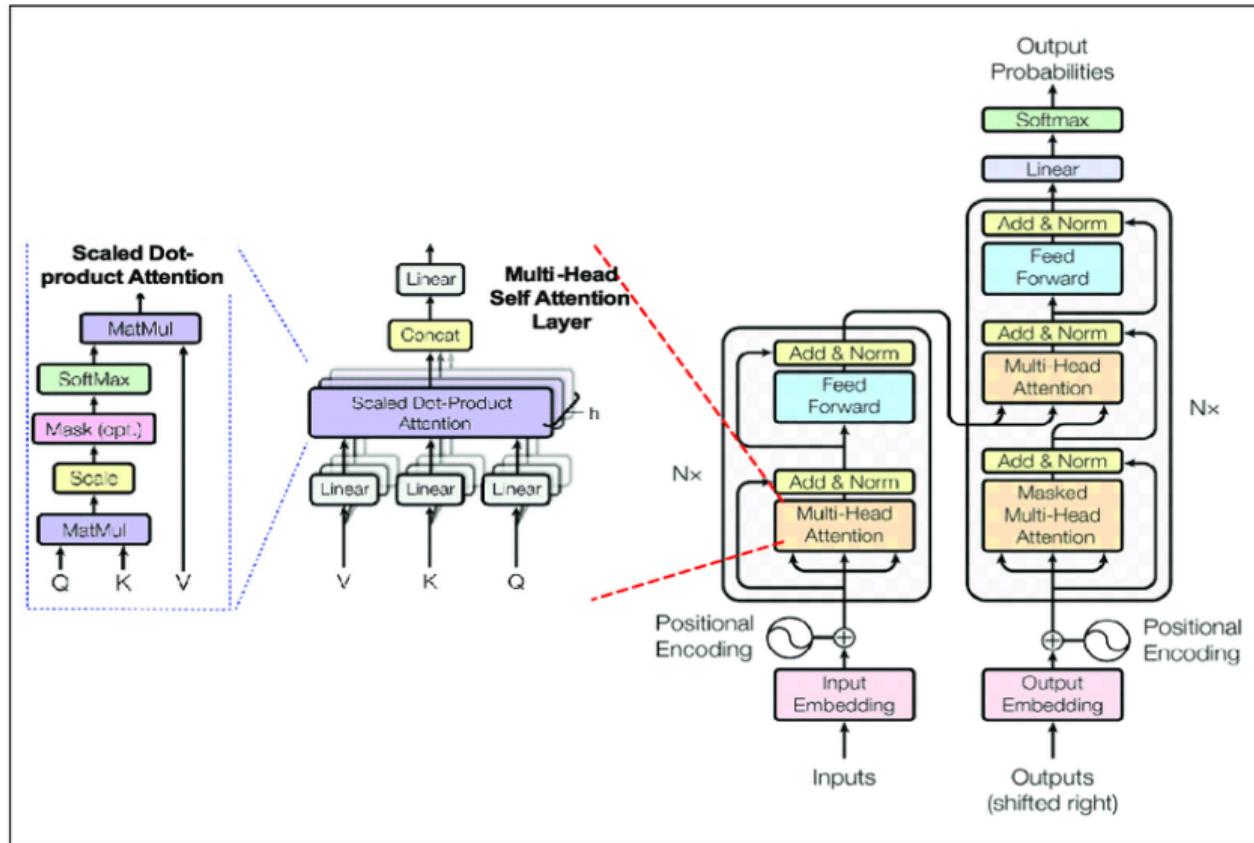


input



output

- El deep learning supone un antes y un después en PLN
- Es ahora el estado del arte de todas las tareas de PLN
- Una arquitectura especializada en aprendizaje sobre secuencias destaca sobre todas: Arquitectura Transformer
- A partir de ella se desarrollan los modelos que se aplican en las diferentes tareas de PLN
- Su ventaja es que un modelo puede ser entrenado para hacer múltiples tareas
- Modelos: BERT (Google), GPT (Open AI)



- Los modelos de deep learning **no supervisados** permiten la generación de datos nuevos
- Estos pueden aplicarse por ejemplo a la generación de imágenes o textos que no están en el conjunto de datos de entrenamiento
- Se pueden combinar varias modalidades de datos (texto, video, audio) para “**guiar**” la generación





Teddy bears mixing sparkling chemicals as mad scientist as digital art



A bald eagle made of chocolate powder, mango, and whipped cream.

Imagen

Explaining a joke

Prompt

Explain this joke:

Joke: Did you see that Google just hired an eloquent whale for their TPU team? It showed them how to communicate between two different pods!

Model Response

Prediction: TPUs are a type of computer chip that Google uses for deep learning. A "pod" is a group of TPUs. A "pod" is also a group of whales. The joke is that the whale is able to communicate between two groups of whales, but the speaker is pretending that the whale is able to communicate between two groups of TPUs.

- Una década de deep learning ha llevado a redes con billones de parámetros
- Eso ha obligado a:
 - El desarrollo hardware especializado (GPUs, TPUs)
 - La necesidad del uso de clústeres HPC
- El entrenamiento de las redes que se usan en aplicaciones comerciales se cuenta en años de GPU/TPU
- El éxito de las aplicaciones necesita de grandes conjuntos de datos especializados para cada tarea (imagen, video, texto, audio)
- El crear/recolectar/etiquetar esos conjuntos de datos supone el mayor cuello de botella

- El gran potencial del aprendizaje profundo ha movilizado a la industria
- Ahora las RRNN se encuentran en el centro de muchas aplicaciones comerciales
- Se están construyendo nuevas herramientas que definirán el futuro del desarrollo en IA
 - Herramientas de software para desarrollar Deep Learning: Tensorflow, PyTorch, JAX, MXNet, ...
 - Pero también todas las herramientas/infraestructura para entrenar y desplegar modelos
- Pero entrenar modelos de vanguardia está lejos del programador común (se mide en años de GPU)
- No obstante hay muchos modelos preentrenados que se pueden utilizar y reajustar

Interpretabilidad/Explicabilidad

- Básicamente, esta es la más negra de las cajas negras
- Estos modelos no se pueden interpretar excepto cuando son equivalentes a modelos explicables (regresión lineal/logística)
- Se pueden aplicar métodos independientes del modelo para evaluar la relevancia de características o explicación de ejemplos
- Se están desarrollando métodos específicos para explicar la predicción de la red para un ejemplo para algunas tareas, especialmente para visión por computadora
 - Visualización de las características que se aprenden
 - Atribución de píxeles propagando hacia atrás hasta los píxeles de entrada (Grad-CAM, SmoothGrad)



Este **notebook** muestra un ejemplo de MLP con una explicación de los modelos