Inteligencia Artificial

CC-421

César Lara Avila

Universidad Nacional de Ingeniería

(actualización: 2021-01-17)

Bienvenidos

Conceptos fundamentales de Aprendizaje Profundo

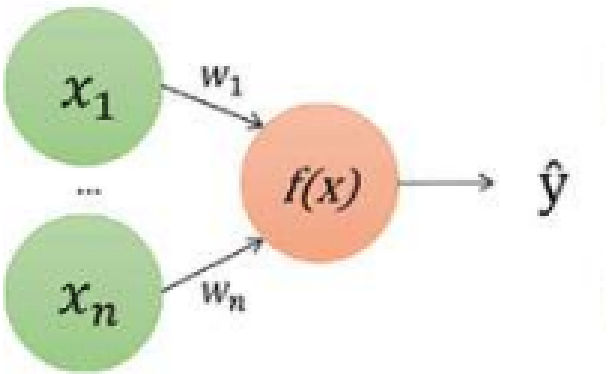
- Introducción
- Separabilidad lineal y no lineal
- Perceptrón multicapa
- Aprendizaje profundo
- Funciones de activación
- Funciones de pérdidas

Introducción

- Uno de los conceptos más comentados en el aprendizaje automático tanto en la comunidad académica como en los medios de comunicación es el campo en evolución del aprendizaje profundo.
- La idea de las redes neuronales y, posteriormente, el aprendizaje profundo, se inspira en la representación biológica del cerebro humano.
- En los últimos 6 a 7 años se ha experimentado un crecimiento exponencial en la popularidad y la aplicación del aprendizaje profundo. Aunque los cimientos de las redes neuronales se remontan a finales de la década de 1960, la arquitectura AlexNet, marcó el comienzo de una explosión de interés en el aprendizaje profundo cuando ganó el concurso de clasificación de imágenes Imagenet 2012 con una red neuronal convolucional de 5 capas.
- Desde entonces, el aprendizaje profundo se ha aplicado a una multitud de dominios y ha logrado un rendimiento de vanguardia en la mayoría de estas áreas.

Explicación del algoritmo de perceptrón

- El aprendizaje profundo en su forma más simple es una evolución del algoritmo del perceptrón, entrenado con un optimizador basado en gradientes.
- El algoritmo de perceptrón es uno de los primeros algoritmos de aprendizaje supervisado, que se remonta a la década de 1950.
- Al igual que una neurona biológica, el algoritmo del perceptrón actúa como una neurona artificial, con múltiples entradas y pesos asociados con cada entrada, cada una de las cuales produce una salida.



Algoritmo del perceptrón

La forma básica del algoritmo de perceptrón para la clasificación binaria es:

$$y(x_1, \ldots, x_n) = f(w_1 x_1 + \cdots + w_n x_n).$$

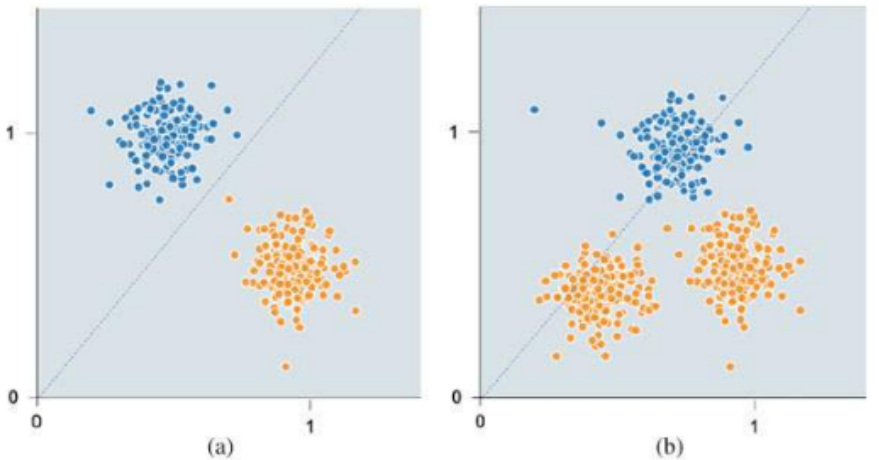
Pesamos individualmente cada x_i por un peso aprendido w_i para asignar la entrada $\mathbf{x} \in R^n$ a un valor de salida y, donde $f(x)$ se define como la función escalonada que se muestra a continuación:

$$f(v) = \begin{cases} 0 & \text{if } v < 0.5 \\ 1 & \text{if } v \geq 0.5 \end{cases}$$

La función de paso toma una entrada de número real y produce un valor binario de 0 o 1, lo que indica una clasificación positiva o negativa si supera el umbral de 0.5.

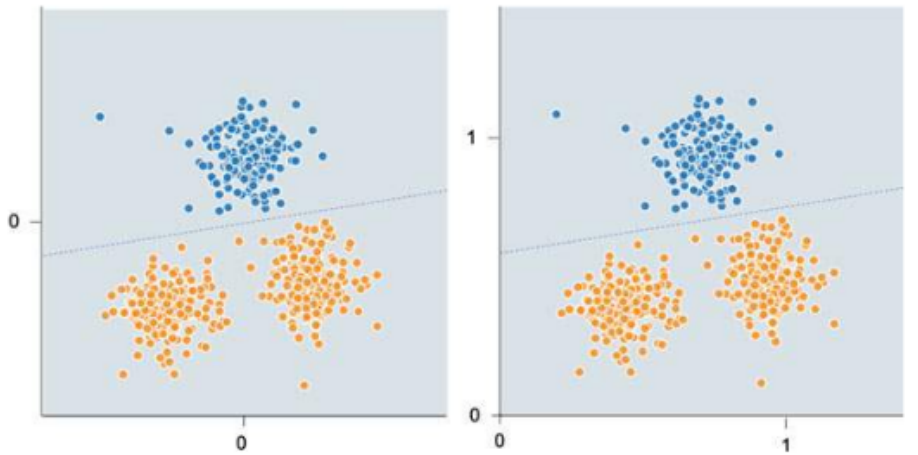
Sesgo

- El algoritmo del perceptrón aprende un hiperplano que separa dos clases.
- Sin embargo, en este punto, el hiperplano separador no puede alejarse del origen, como se muestra en la figura (a). Restringir el hiperplano de esta manera causa problemas, como podemos ver en la figura (b).



Solución al problema

- Una solución es asegurar que nuestros datos se puedan aprender si normalizamos el método para que se centre en el origen como una posible solución o agreguemos un término de sesgo b a la ecuación permitiendo que el hiperplano de clasificación se aleje del origen, como se muestra en la figura:



El perceptron en términos del sesgo

Podemos escribir el perceptrón con un término de sesgo como:

$$y(x_1, \ldots, x_n) = f(w_1 x_1 + \cdots + w_n x_n + b).$$

Alternativamente, podemos tratar b como un peso adicional w_0 ligado a una entrada constante de 1 y se puede escribir como:

$$y(x_1, \ldots, x_n) = f(w_1 x_1 + \cdots + w_n x_n + w_0).$$

Al cambiar a la notación vectorial, podemos reescribir la ecuación anterior como:

$$y(\mathbf{x}) = f(\mathbf{w}\mathbf{x} + b)$$

El término de sesgo es un peso aprendido que elimina la restricción de que el hiperplano separador debe pasar por el origen.

Proceso de aprendizaje

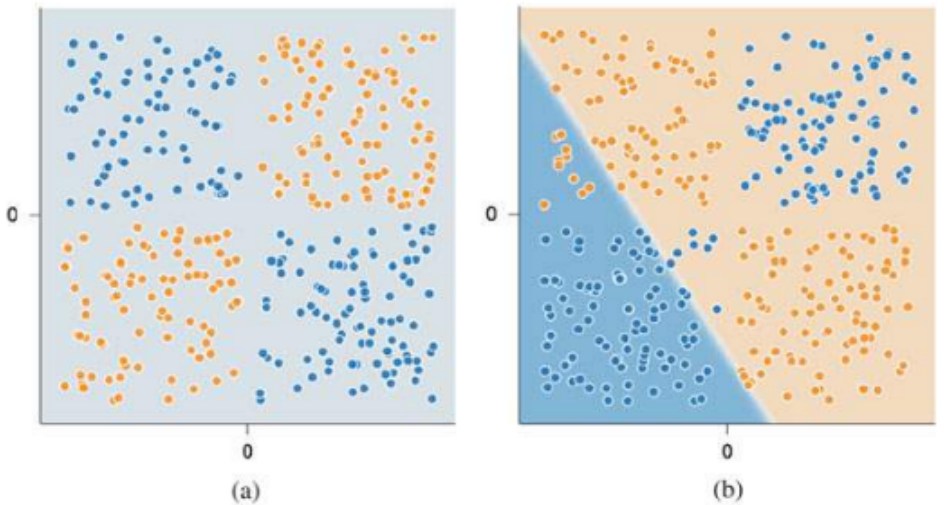
- El proceso de aprendizaje del algoritmo del perceptrón es modificar los pesos $\mathbf{w}$ para lograr un error 0 en el conjunto de entrenamiento.
- Por ejemplo, suponga que necesitamos separar conjuntos de puntos A y B. Comenzando con pesos aleatorios $\mathbf{w}$, mejoramos gradualmente el límite a través de cada iteración con el objetivo de lograr $E(\mathbf{w}, b) = 0$.
- Por lo tanto, minimizaríamos el error de la siguiente función en todo el conjunto de entrenamiento.

$$E(\mathbf{w}) = \sum_{\mathbf{x} \in A} (1 - f(\mathbf{w}\mathbf{x} + b)) + \sum_{\mathbf{x} \in B} f(\mathbf{w}\mathbf{x} + b)$$

Separabilidad lineal y no lineal

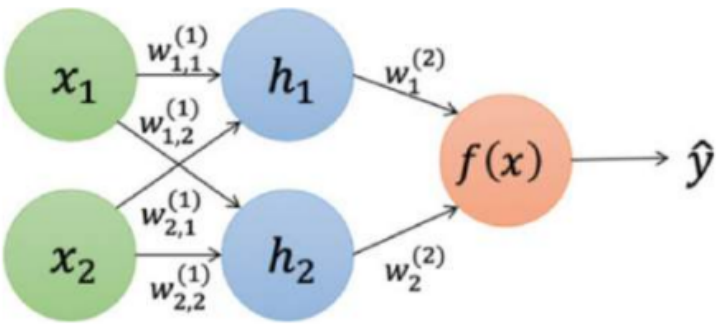
Dos conjuntos de datos son linealmente separables si un solo límite de decisión puede separarlos.

Si aplicamos el perceptrón a un conjunto de datos separable de forma no lineal, como el conjunto de datos que se muestra en la figura (a), no podemos separar los datos como se muestra en la figura (b) ya que solo podemos aprender tres parámetros, w_1, w_2 y b.



Perceptrón multicapa

- El perceptrón multicapa (MLP) vincula varios perceptrones (comúnmente denominados neuronas) en una red. Las neuronas que reciben la misma información se agrupan en una capa de perceptrones. En lugar de usar una función escalonada, se sustituye por una función no lineal diferenciable.
- La aplicación de esta función no lineal, comúnmente conocida como función de activación o no linealidad, permite que el valor de salida sea una combinación ponderada no lineal de sus entradas, lo que crea características no lineales utilizadas por la siguiente capa.



MLP

- El MLP está compuesto por neuronas interconectadas y es, por tanto, una red neuronal. Específicamente, es una red neuronal de retroalimentación, ya que hay una dirección para el flujo de datos a través de la red (sin ciclos, conexiones recurrentes).
- Un MLP debe contener una capa de entrada y salida y al menos una capa oculta. Además, las capas también están *completamente conectadas*, lo que significa que la salida de cada capa está conectada a cada neurona de la siguiente capa. En otras palabras, se aprende un parámetro de peso para cada combinación de neurona de entrada y neurona de salida entre las capas.
- La capa oculta proporciona dos salidas, h_1 y h_2, que pueden ser combinaciones no lineales de sus valores de entrada x_1 y x_2. La capa de salida pesa sus entradas de la capa oculta y con un mapeo no lineal hace su predicción.

Entrenamiento de un MLP

El entrenamiento de los pesos del MLP (y por extensión, una red neuronal) se basa en cuatro componentes principales:

1. Propagación hacia adelante: calcula la salida de red para un ejemplo de entrada.

2. Cálculo de errores: calcula el error de predicción entre la predicción de la red y el objetivo.

3. Retropropagación: Calcula los gradientes en orden inverso con respecto a la entrada y los pesos.

4. Actualización de parámetros: utiliza el descenso de gradiente estocástico para actualizar los pesos de la red para reducir el error de ese ejemplo.

Propagación hacia adelante

- Usamos la función sigmoidea, representada por $\sigma(x)$, como función de activación para el MLP. La definición de la función sigmoidea es:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- El objetivo de este proceso es calcular la salida actual de la red para un ejemplo x en particular, con cada salida conectada como entrada a las neuronas de la siguiente capa.

- Por conveniencia de notación y computación, los pesos de la capa se combinan en una única matriz de pesos, $\mathbf{W}_l$, que representa la colección de pesos en esa capa, donde l es el número de capa.

- La transformación lineal realizada por el cálculo de la capa para cada peso es un cálculo del producto interno entre $\mathbf{x}$ y $\mathbf{W}_l$. Este tipo se conoce habitualmente como capa *totalmente conectada*, *producto interno* o *lineal* porque un peso conecta cada entrada con cada salida.

Propagación hacia adelante

- Calcular la predicción $\hat{y}$ para un ejemplo x donde h_1 y h_2 representan las respectivas salidas de capa se convierte en:

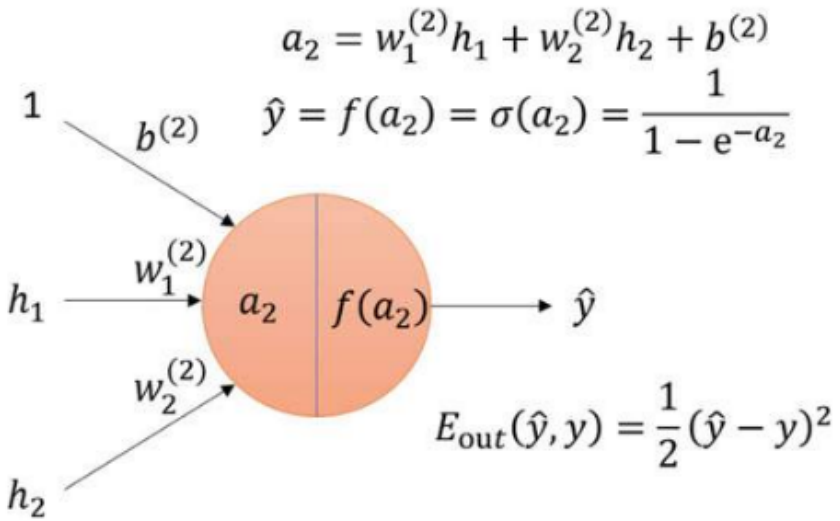
$$
\begin{aligned}
f(v) &= \sigma(v) \\
\mathbf{h_1} &= f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\
h_2 &= f(\mathbf{W}_2\mathbf{h}_1 + b_2) \\
\hat{y} &= h_2
\end{aligned}
$$

- Ten en cuenta que el sesgo $\mathbf{b}_1$ es un vector porque hay un valor de sesgo asociado con cada neurona de la capa. Solo hay una neurona en la capa de salida, por lo que el sesgo b_2 es un escalar.

Al final del paso de propagación hacia adelante, tenemos una predicción de salida para la red. Una vez que se entrena la red, se evalúa un nuevo ejemplo mediante la propagación hacia adelante.

Propagación hacia adelante

Neurona de salida que muestra el cálculo completo de la salida de preactivación y posactivación.



Cálculo de errores

- El paso de cálculo de errores verifica qué tan bien se desempeñó nuestra red en un ejemplo dado. Usamos el error cuadrático medio (MSE) como la función de pérdida utilizada en este ejemplo (tratando el entrenamiento como un problema de regresión). MSE se define como:

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2.$$

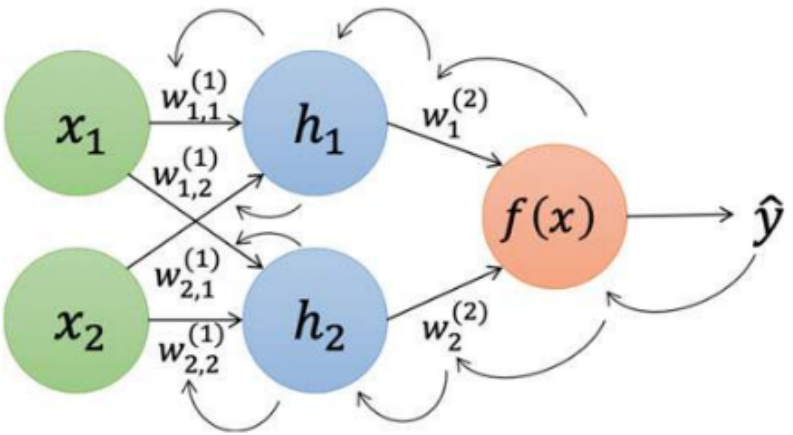
- Esta función de error se usa comúnmente para problemas de regresión, midiendo el promedio de los errores cuadrados para el objetivo.

- El paso de cálculo del error produce un valor de error escalar para el ejemplo de entrenamiento.

Retropropagación

- Durante la propagación hacia adelante, se calcula una predicción de salida $\hat{y}$ para la entrada x y los parámetros de red θ. Para mejorar nuestra predicción, podemos usar SGD para disminuir el error de toda la red. La determinación del error para cada uno de los parámetros se puede realizar mediante la regla de la cadena.

- Podemos usar la regla de la cadena para calcular las derivadas de cada capa (y operación) en el orden inverso de la propagación hacia adelante, como se ve en la figura:



Retropropagación

- En nuestro ejemplo anterior, la predicción $\hat{y}$ dependía de $\mathbf{W}_2$. Podemos calcular el error de predicción con respecto a $\mathbf{W}_2$, usando la regla de la cadena:

$$\frac{\partial E}{\partial \mathbf{W}_2} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{W}_2}$$

- La regla de la cadena nos permite calcular el gradiente del error para cada uno de los parámetros aprendibles θ, lo que nos permite actualizar la red usando el descenso de gradiente estocástico.

- Comenzamos calculando el gradiente en la capa de salida con respecto a la predicción.

$$\nabla_{\hat{y}} E(\hat{y}, y) = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

- Entonces podemos calcular el error con respecto a los parámetros de la capa 2. Actualmente tenemos el gradiente de *post-activación*, por lo que necesitamos calcular el gradiente de *pre-activación*:

$$\begin{aligned}\nabla_{\mathbf{a}_2} E = \frac{\partial E}{\partial \mathbf{a}_2} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \\ &= \frac{\partial E}{\partial \hat{y}} \odot f^{'}(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)\end{aligned}$$

Ahora calculamos el error con respecto a $\mathbf{W}_2$ y $\mathbf{b}_2$.

$$\begin{aligned}\nabla_{\mathbf{W}_2} E = \frac{\partial E}{\partial \mathbf{W}_2} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{W}_2} \\ &= \frac{\partial E}{\partial \mathbf{a}_2} \mathbf{h}_1^T\end{aligned}$$

$$\begin{aligned}\nabla_{\mathbf{b}_2} E = \frac{\partial E}{\partial \mathbf{b}_2} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{b}_2} \\ &= \frac{\partial E}{\partial \mathbf{a}_2}\end{aligned}$$

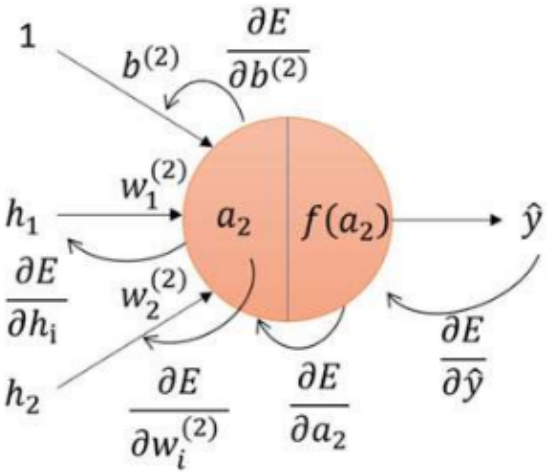
También podemos calcular el error de la entrada a la capa 2 (la salida posterior a la activación de la capa 1).

$$\begin{aligned}\nabla_{\mathbf{h}_1} E = \frac{\partial E}{\partial \mathbf{h}_1} &= \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{a}_2} \cdot \frac{\partial \mathbf{a}_2}{\partial \mathbf{h}_1} \\ &= \mathbf{W}_2^T \frac{\partial E}{\partial \mathbf{a}_2}\end{aligned}$$

Luego repetimos este proceso para calcular el error para los parámetros de la capa 1, $\mathbf{W}_1$ y $\mathbf{b}_1$, propagando así el error hacia atrás por toda la red.

La figura muestra el paso de propagación hacia atrás para la neurona de salida de la red en el ejemplo del perceptrón multicapa.

Retropropagación a través de la neurona de salida:



$$\nabla_{\hat{y}} E = \frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

$$\frac{\partial E}{\partial a_2} = \frac{\partial E}{\partial \hat{y}} f'(a_2)$$

$$\frac{\partial E}{\partial w_i^{(2)}} = \frac{\partial E}{\partial a_2} h_i$$

$$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial a_2}$$

$$\frac{\partial E}{\partial h_i} = \frac{\partial E}{\partial a_2} w_i^{(2)}$$

Actualización de parámetros

- El último paso en el proceso de entrenamiento es la actualización de parámetros. Después de obtener los gradientes con respecto a todos los parámetros aprendibles en la red, podemos completar un solo paso SGD, actualizando los parámetros para cada capa de acuerdo con la tasa de aprendizaje α.

$$\theta = \theta - \alpha \nabla_\theta E$$

- La simplicidad de la regla de actualización SGD presentada aquí tiene un costo. El valor de α es particularmente vital en SGD y afecta la velocidad de convergencia, la calidad de la convergencia e incluso la capacidad de la red para converger.

- La simplicidad de la red que se presenta aquí alivia la naturaleza tediosa de seleccionar una tasa de aprendizaje, pero para redes más profundas, este proceso puede ser mucho más difícil.

- La importancia de elegir una buena tasa de aprendizaje ha llevado a toda un área de investigación en torno a los algoritmos de optimización del descenso de gradientes.

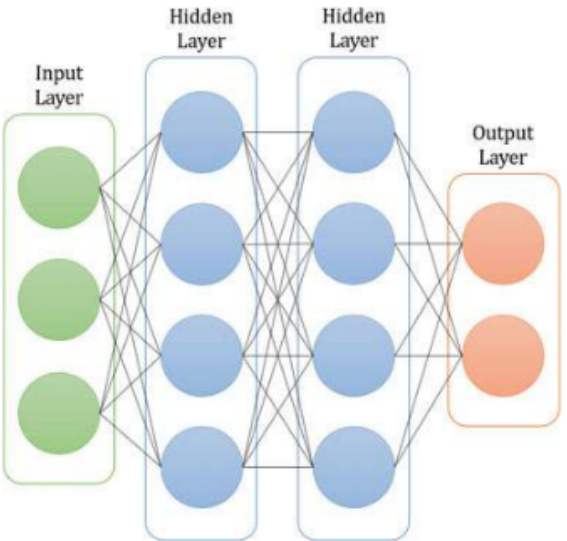
Teorema de aproximación universal

- El teorema de aproximación universal ha demostrado que una red neuronal de alimentación directa con una sola capa puede aproximarse a cualquier función continua con solo restricciones limitadas en el número de neuronas en la capa.
- El teorema de aproximación universal se demostró inicialmente para arquitecturas de redes neuronales que utilizan la función de activación sigmoidea, pero posteriormente se demostró que se aplica a todas las redes completamente conectadas.
- La topografía del espacio de parámetros se vuelve más variada a medida que los problemas de aprendizaje automático se vuelven más complejos. Un enfoque simple de descenso de gradiente puede tener dificultades para aprender la función específica. En cambio, varias capas de neuronas se apilan consecutivamente y se entrenan conjuntamente con la propagación hacia atrás. La red de capas aprende múltiples funciones no lineales para adaptarse al conjunto de datos de entrenamiento.

El aprendizaje profundo se refiere a muchas capas de redes neuronales conectadas en secuencia.

Aprendizaje Profundo

- El término aprendizaje profundo es algo ambiguo. Por ejemplo, ¿la red neuronal que se muestra en la figura se consideraría profunda o superficial?

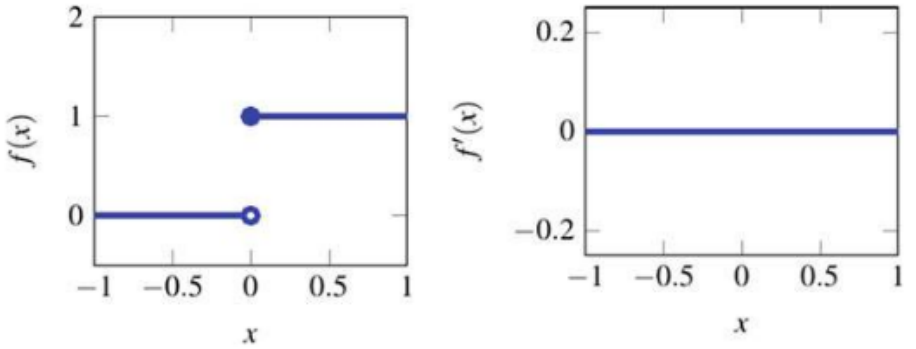


- En general, las redes profundas siguen siendo redes neuronales, pero generalmente con más capas.

- La flexibilidad de las redes neuronales es lo que las hace tan atractivas.
- Las redes neuronales se aplican a muchos tipos de problemas dada la simplicidad y la eficacia de los métodos de propagación hacía atrás y optimización basada en gradiente.
- En muchos círculos, el aprendizaje profundo es un término de cambio de marca para las redes neuronales o se usa para referirse a redes neuronales con muchas capas consecutivas (profundas).

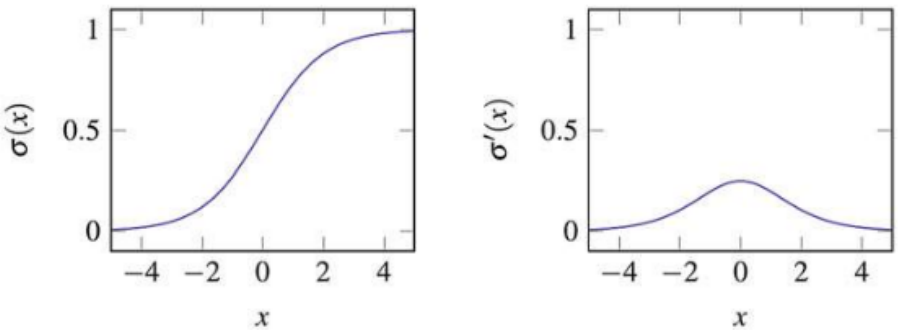
Funciones de activación

- Al calcular el gradiente de la capa de salida, se hace evidente que la función de paso no es exactamente útil cuando se intenta calcular un gradiente. Como se muestra en la figura, la derivada es 0 en todas partes, lo que significa que cualquier descenso de gradiente es inútil.
- Por lo tanto, deseamos utilizar una función de activación no lineal que proporcione una derivada significativa en el proceso de retropropagación.



Sigmoide

- Una mejor función para usar como función de activación es el sigmoide logístico: $\sigma(x) = \frac{1}{1+e^{-x}}$.
- Como podemos ver en el gráfico de la figura, esta función actúa como una función de aplastamiento continuo que limita su salida en el rango $(0, 1)$.



- Tiene una derivada suave y continua y está centrada en cero, creando un límite de decisión simple para tareas de clasificación binaria, y la derivada de la función sigmoidea es matemáticamente conveniente:

$$\sigma^{'}(x) = \sigma(x)(1 - \sigma(x))$$

Inconvenientes

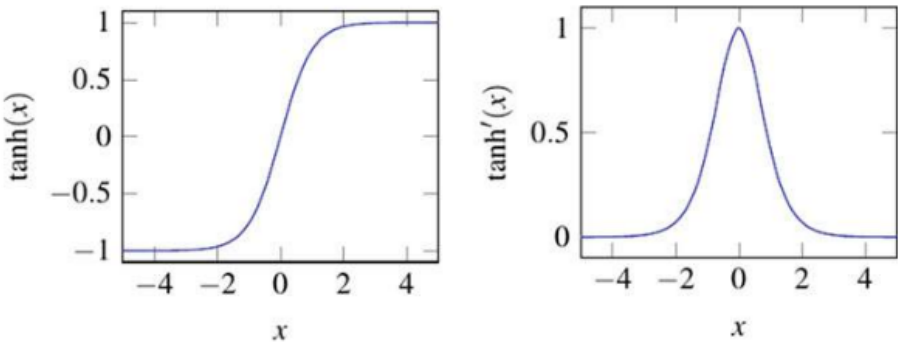
Sin embargo, existen algunas propiedades indeseables de la función sigmoidea.

- La saturación de los gradientes sigmoides en los extremos de la curva (muy cerca de $\sigma(x) \leftarrow 0$ o $\sigma(x) \leftarrow 1$) hará que los gradientes estén muy cerca de 0.Evitar esto puede requerir una inicialización cuidadosa de los pesos de la red u otras estrategias de regularización.

- Las salidas del sigmoide no se centran alrededor de 0, sino alrededor de 0.5. Esto introduce una discrepancia entre las capas porque las salidas no están en un rango consistente. Esto a menudo se denomina *cambio de covariables interno*.

Tanh

- La función $tanh$ es otra función de activación común, definida como: $f(x) = tanh(x)$. También actúa como una función de aplastamiento, delimitando su salida en el rango $(-1, 1)$ como se muestra en la figura:

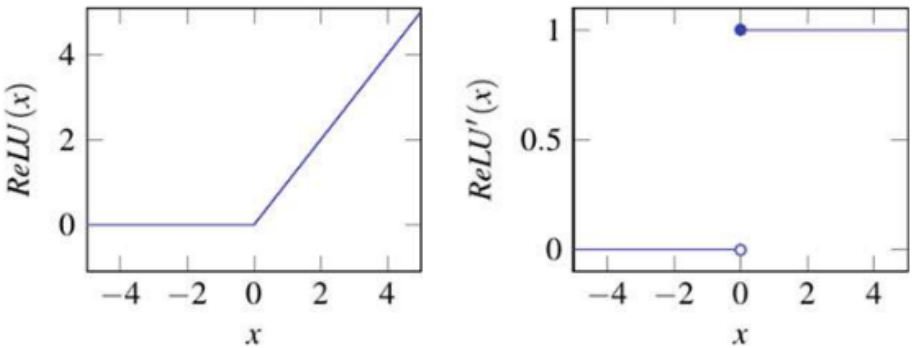


- También puede verse como un sigmoide escalado y desplazado: $tanh(x) = 2 * \sigma(2x) - 1$.

- La función $tanh$ resuelve uno de los problemas con la no linealidad sigmoidea porque está centrada en cero. Sin embargo, todavía tenemos el mismo problema con la saturación del gradiente en los extremos de la función.

ReLU

- La unidad lineal rectificada (ReLU) es una función de activación rápida y simple que se encuentra normalmente en visión por computadora. La función es un umbral lineal, definido como: $f(x) = \max(0, x)$



- Un inconveniente de la simplicidad de las actualizaciones de gradiente siendo 0 o 1 es que puede provocar que las neuronas *mueran* durante el entrenamiento.

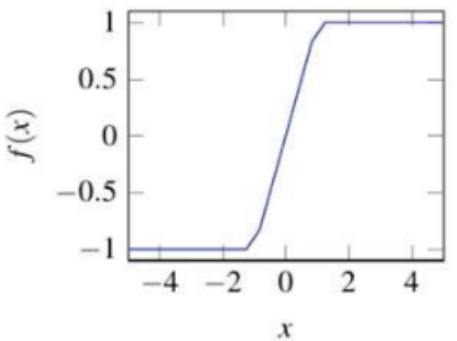
- Algunos han demostrado que hasta el 40% de las neuronas en una red pueden *morir* con la función de activación de ReLU si la tasa de aprendizaje se establece demasiado alta.

Otras funciones de activación

Se han incorporado otras funciones de activación para limitar los efectos de las funciones descritas anteriormente.

Hard Tanh

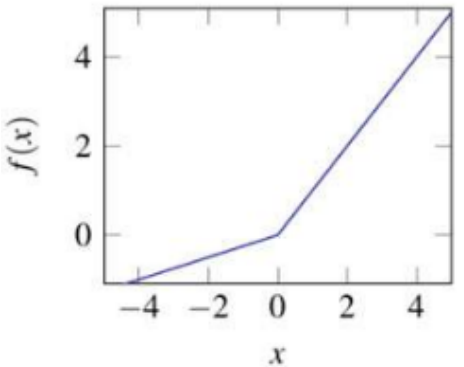
- La función $hard\ tanh$ es computacionalmente más barata que $tanh$. Sin embargo, vuelve a introducir la desventaja de la saturación del gradiente en los extremos: $f(x) = \max(-1, \min(1, x))$.



Leaky ReLU

- El $Leaky\ ReLU$ introduce un parámetro α que permite retropropagar pequeños gradientes cuando la activación no está activa, eliminando así la *muerte* de neuronas durante el entrenamiento.

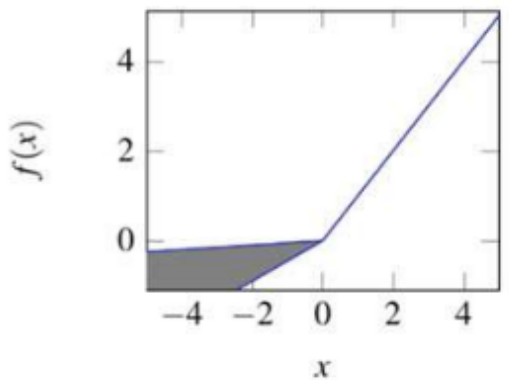
$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}.$$



PRELU

- La unidad lineal rectificada paramétrica, similar a $Leaky\ ReLU$, utiliza un parámetro α para escalar la pendiente de la parte negativa de la entrada, sin embargo, se aprende un parámetro α para cada neurona (duplicando el número de pesos aprendidos).
- Ten en cuenta que cuando el valor de $\alpha = 0$ esta es la función ReLU y cuando α es fijo, es equivalente a $LeakyReLU$.

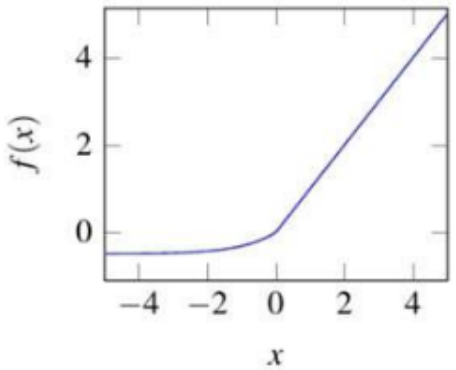
$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}.$$



ELU

El ELU es una modificación del ReLU que permite que la media de activaciones se acerque a 0, lo que potencialmente acelera la convergencia.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}.$$



Maxout

La función maxout adopta un enfoque diferente para las funciones de activación. Aprende dos matrices de peso y toma la salida más alta para cada elemento: $f(x) = \max(w_1 x + b_1, w_2 x + b_2)$.

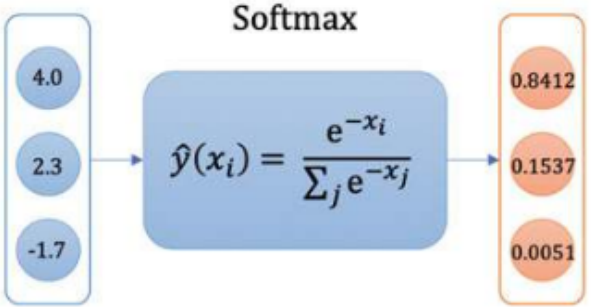
Softmax

- La función softmax nos permite generar una distribución de probabilidad categórica sobre K clases.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

- Podemos usar el softmax para producir un vector de probabilidades de acuerdo con la salida de esa neurona. En el caso de un problema de clasificación que tiene $K = 3$ clases, la capa final de la red será una capa completamente conectada con una salida de tres neuronas.

- Si aplicamos la función softmax a la salida de la última capa, obtenemos una probabilidad para cada clase asignando una clase a cada neurona.



- Las probabilidades de softmax pueden volverse muy pequeñas, especialmente cuando hay muchas clases y las predicciones se vuelven más confiables.
- La mayoría de las veces se utiliza una función softmax basada en log para evitar errores de underflow. La función softmax es un caso particular para las funciones de activación, ya que rara vez se ve como una activación que ocurre entre capas.
- Por lo tanto, el softmax a menudo se trata como la última capa de una red para la clasificación multiclase en lugar de una función de activación.

Softmax jerárquico

- A medida que el número de clases comienza a crecer, como suele ser el caso en las tareas de lenguaje, el cálculo de la función softmax puede ser costoso de calcular.
- El softmax jerárquico aproxima la función softmax al representar la función como un árbol binario con la profundidad que produce activaciones de clase menos probables.
    - El árbol debe estar equilibrado ya que la red está entrenada, pero tendrá una profundidad de $log_2(K)$ donde K es el número de clases, lo que significa que solo los $log_2(K)$ estados deben evaluarse para calcular la probabilidad de salida de una clase.
- En una tarea de modelado de lenguaje, la capa de salida puede estar tratando de predecir qué palabra será la siguiente en la secuencia. Por lo tanto, la salida de la red sería una distribución de probabilidad sobre el número de términos en el vocabulario, que podrían ser miles o cientos de miles.

Funciones de pérdidas

- Otro aspecto importante del entrenamiento de redes neuronales es la elección de funciones de error a las que a menudo se hace referencia como criterio.
- La selección de la función de error depende del tipo de problema que se aborde.

Error cuadrático medio (MSE)

- Calcula el error cuadrático entre la predicción de clasificación y el objetivo. En el entrenamiento minimiza la diferencia de magnitud. Un inconveniente de MSE es que es susceptible a valores atípicos ya que la diferencia es al cuadrado.

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

Error absoluto medio

- El error absoluto medio da una medida de la diferencia absoluta entre el valor objetivo y la predicción. Su uso minimiza la magnitud del error sin considerar la dirección, haciéndolo menos sensible a valores atípicos.

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|.$$

Probabilidad logarítmica negativa

- La probabilidad logarítmica negativa (NLL) es la función de pérdida más común utilizada para problemas de clasificación multiclase.
- El cálculo de la entropía es una probabilidad logarítmica promedio ponderada sobre los posibles eventos o clasificaciones en un problema de clasificación multiclase.

$$E(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)).$$

- Esto hace que la pérdida aumente a medida que la distribución de probabilidad de la predicción diverge de la etiqueta objetivo.

Pérdida Hinge

- La pérdida Hinge es una clasificación de pérdida de margen máximo tomada de la pérdida de SVM. Intenta separar los puntos de datos entre clases maximizando el margen entre ellos.

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{i=1}^{n} \max(0, 1 - y_i \hat{y}_i).$$

- Aunque no es diferenciable, es convexo, lo que lo hace útil para trabajar como función de pérdida.

Pérdida de Kullback – Leibler (KL)

- Podemos optimizar funciones, como la divergencia KL, que mide una métrica de distancia en un espacio continuo. El error de divergencia KL se puede describir mediante:

$$\begin{aligned} E(\hat{\mathbf{y}}, \mathbf{y}) &= \frac{1}{n} \sum_{i=1}^{n} D_{KL}(y_i || \hat{y}_i) \\ &= \frac{1}{n} \sum_{i=1}^{n} (y_i \cdot \log(y_i)) - \frac{1}{n} \sum_{i=1}^{n} (y_i \cdot \log(\hat{y}_i)) \end{aligned}$$

Métodos de optimización

El proceso de entrenamiento de las redes neuronales se basa en el SGD. Sin embargo, SGD puede causar muchas dificultades indeseables durante el proceso de entrenamiento.

Descenso de gradiente estocástico

El SGD es el proceso de realizar actualizaciones a un conjunto de pesos en la dirección del gradiente para reducir el error. La regla de actualización de SGD es de la siguiente forma:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta E$$

donde θ representa los parámetros que se pueden aprender, α es la tasa de aprendizaje y $\nabla_\theta E$ es el gradiente del error con respecto a los parámetros.

Momentum

El momento es una modificación de SGD para mover el objetivo más rápidamente a los mínimos. La ecuación de actualización de parámetros para el momento es:

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta E \\
\theta_{t+1} &= \theta_t - v_t
\end{aligned}
$$

donde θ_t representa un parámetro en la iteración t.

- El momentum, calcula un vector de velocidad que captura la dirección acumulativa que han producido los gradientes anteriores. Este vector de velocidad se escala mediante un hiperparámetro adicional η, que sugiere cuánto puede contribuir la velocidad acumulativa a la actualización.

Adagrad

Adagrad es un método de optimización adaptable basado en gradientes. Adapta la tasa de aprendizaje a cada uno de los parámetros de la red, haciendo actualizaciones más sustanciales a los parámetros poco frecuentes y actualizaciones más pequeñas a los frecuentes.

La ecuación de Adagrad es dado por:

$$g_{t,i} = \nabla_\theta E(\theta_{\theta_{t,i}})$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \circ g_{t,i}$$

donde g_t es el gradiente en el tiempo t a lo largo de cada componente de θ, G_t es la matriz diagonal de la suma de hasta t pasos de gradientes pasados, con respecto a todos los parámetros θ en la diagonal, η es la tasa de aprendizaje general y ϵ es un término de ajuste (generalmente $1e-8$) que evita que la ecuación se divida por cero.

- El principal inconveniente de Adagrad es que la acumulación de los gradientes cuadrados es positiva, lo que hace que la suma crezca, reduce la tasa de aprendizaje y evita que el modelo aprenda más.

RMS-Prop

Se han introducido variantes adicionales, como Adadelta o RMS-Prop , para aliviar este problema.

El RMS-prop, desarrollado por Hinton, se introdujo para resolver las deficiencias de Adagrad. También divide la tasa de aprendizaje por un promedio de gradientes cuadrados, pero también decae esta cantidad exponencialmente.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

donde $\rho = 0.9$ y la tasa de aprendizaje $\eta = 0.001$ se sugiere en las notas presentadas.

ADAM

Adam, es otro método de optimización adaptativa. También calcula las tasas de aprendizaje para cada parámetro, pero además de mantener un promedio exponencialmente decreciente de los gradientes cuadrados anteriores, similar al momentum, también incorpora un promedio de gradientes pasados m_t.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

- Los resultados empíricos muestran que Adam funciona bien en la práctica en comparación con otras técnicas de optimización basadas en gradientes. Si bien Adam ha sido una técnica popular, algunas críticas a la prueba original han surgido mostrando convergencia a mínimos subóptimos en algunas situaciones.

Fin!