Clase 3 - Introducción a Deep Learning

Facundo González

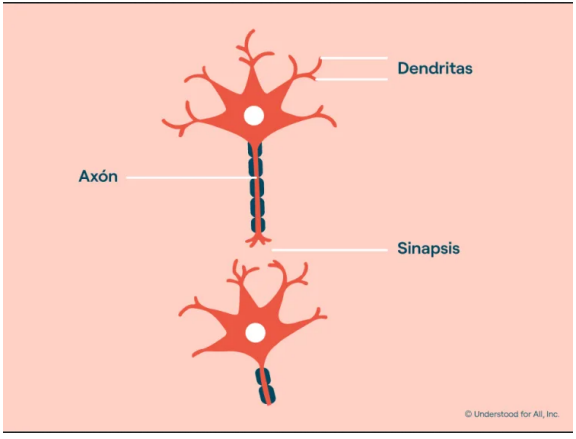
16 de abril de 2024

Outline

Qué son las redes neuronales?

Para mejorar la tarea de crear un modelo que aprenda sobre datos nuevos, se propusieron crear algoritmos que imiten al cerebro humano.

Figura: Neurona Cerebral



Qué son las redes neuronales?

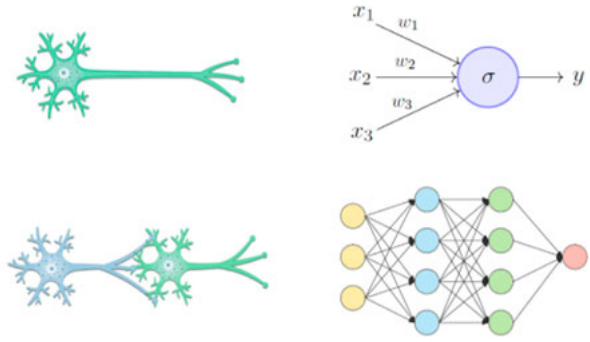
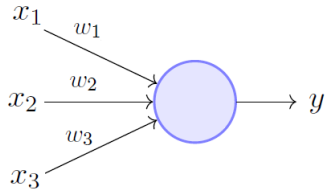


Figura: Modelo de Redes Neuronales

Perceptrón

Primero veamos la unidad básica: el perceptrón.



Perceptron Model (Minsky-Papert in 1969)

Figura: Perceptron

Matemáticamente, el perceptrón recibe como input un vector x, y computa un número y como salida. El valor de y será calculado con los **pesos** $w_1, w_2, \cdots, w_n$.

Perceptrón

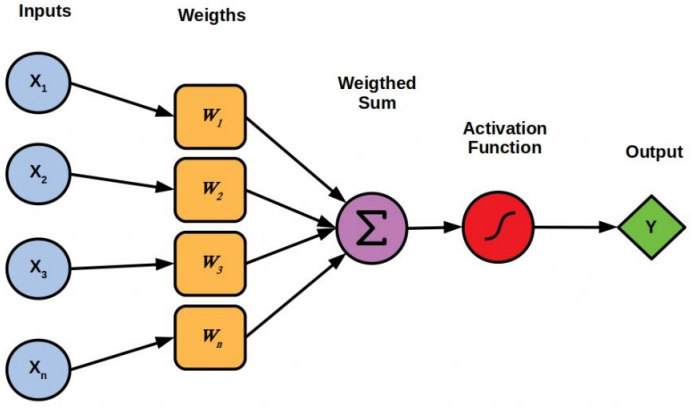
La información que tiene el perceptrón se representa con sus pesos $w_1, w_2, \cdots, w_n$. El calculo que se realiza es:

$$y = \sum_{i=0}^{n} x_i * w_i$$

Es decir, el **producto interno** entre los vectores w y x.

Perceptrón

El problema que tiene este modelo sencillo, es que siempre resulta en transformaciones lineales. Para eso, se le aplica una función **no lineal** al resultado de la neurona. A esta función la llamamos **función de activación**.



Función de activación

Esta función es la que se encarga de que el modelo pueda aprender abstracciones más complejas, sino sería una secuencia de transformaciones lineales. Hay varias funciones que funcionan bien como función de activación:

- ReLU: $f(x) = max(0, x)$

Función de activación

Esta función es la que se encarga de que el modelo pueda aprender abstracciones más complejas, sino sería una secuencia de transformaciones lineales. Hay varias funciones que funcionan bien como función de activación:

- ReLU: $f(x) = max(0, x)$
- Sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$

Función de activación

Esta función es la que se encarga de que el modelo pueda aprender abstracciones más complejas, sino sería una secuencia de transformaciones lineales. Hay varias funciones que funcionan bien como función de activación:

- ReLU: $f(x) = max(0, x)$
- Sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$
- TanH: $f(x) = tanh(x)$

Función de activación

Esta función es la que se encarga de que el modelo pueda aprender abstracciones más complejas, sino sería una secuencia de transformaciones lineales. Hay varias funciones que funcionan bien como función de activación:

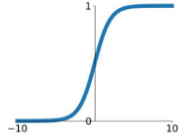
- ReLU: $f(x) = max(0, x)$
- Sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$
- TanH: $f(x) = tanh(x)$
- Softmax: utiliza la sigmoide para devolver un vector de probabilidades. Se utiliza para clasificación.

Función de activación

Activation Functions

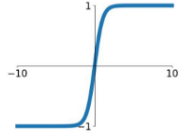
Sigmoid

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



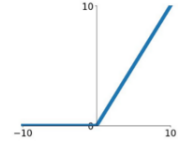
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$

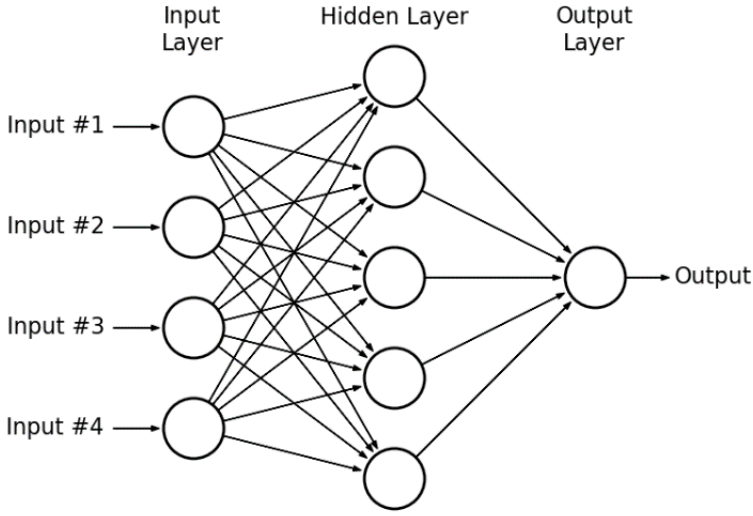


Multi-Layer Perceptron

Con los perceptrones, podemos armas redes más complejas de varias capas donde la entrada una capa de perceptrones es la salida de la capa anterior.

Multi-Layer Perceptron

Con los perceptrones, podemos armas redes más complejas de varias capas donde la entrada una capa de perceptrones es la salida de la capa anterior.



MLP

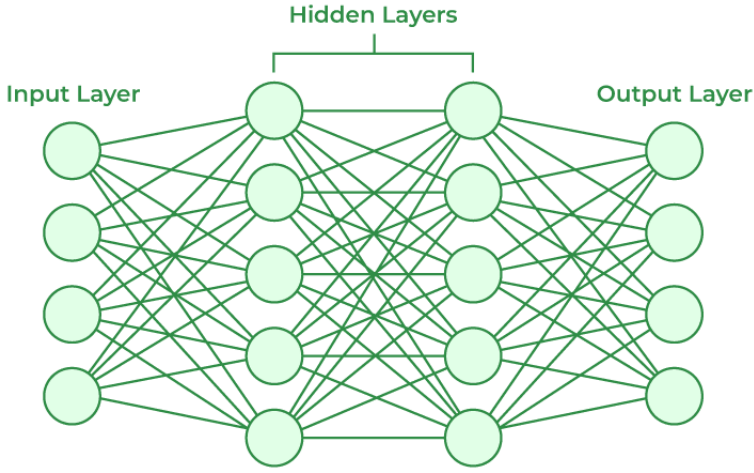


Figura: Red neuronal profunda

Entrenamiento

Una vez definidas las capas, las dimensiones y las funciones de activación tenemos que entrenar a la red con nuestros datos para ajustar los pesos. Para eso vamos a definir un algoritmo que, teniendo en cuenta alguna métrica de pérdida, ajuste los pesos para minimizarla.

Para eso necesitamos definir una **función de pérdida** (*loss function*) para saber cuánto debemos mejorar. Por ejemplo podemos utilizar el *RMSE*:

$$L_w(x, d) = \sqrt{\sum_{i=0}^{n} d_i - f_w(x)_i}$$

Donde $f_w(x)$ es la salida de la red neuronal con los pesos w para el input x.

Descenso de gradiente

Con la función de pérdida definida, vamos a utilizar el algoritmo de **descenso de gradiente** para encontrar los w que minimicen esta pérdida. Recordar que queremos:

$$\hat{w} = \arg\min_{w} \frac{1}{n} \sum_{i=1}^{n} L_w(x_i, d_i)$$

Para lograr esto, el algoritmo propone:

1. Inicializar w aleatoriamente.

Descenso de gradiente

Con la función de pérdida definida, vamos a utilizar el algoritmo de **descenso de gradiente** para encontrar los w que minimicen esta pérdida. Recordar que queremos:

$$\hat{w} = \arg\min_{w} \frac{1}{n} \sum_{i=1}^{n} L_w(x_i, d_i)$$

Para lograr esto, el algoritmo propone:

1. Inicializar w aleatoriamente.
2. Actualizar pesos en contra de la dirección del gradiente: $w = w - \delta \nabla_w L(D)$

Descenso de gradiente

Con la función de pérdida definida, vamos a utilizar el algoritmo de **descenso de gradiente** para encontrar los w que minimicen esta pérdida. Recordar que queremos:

$$\hat{w} = \arg\min_{w} \frac{1}{n} \sum_{i=1}^{n} L_w(x_i, d_i)$$

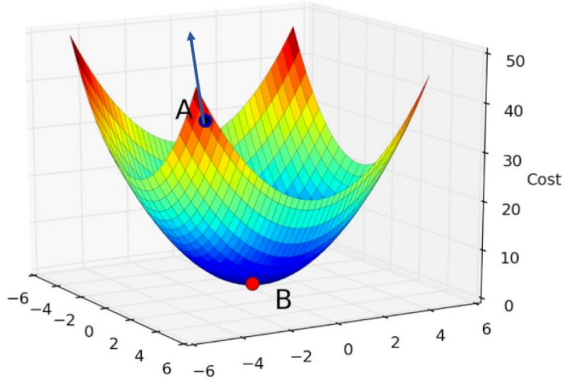
Para lograr esto, el algoritmo propone:

1. Inicializar w aleatoriamente.
2. Actualizar pesos en contra de la dirección del gradiente: $w = w - \delta \nabla_w L(D)$
3. Repetir hasta algún criterio de convergencia.

Gradiente

Veamos visualmente qué representa el gradiente:

$$\nabla f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \right)$$



Backpropagation

El cálculo del gradiente no es trivial, sobre todo para las funciones de pérdida con modelos muy grandes. La técnica de **backpropagation** nos permite calcular las derivadas aplicando la regla de la cadena iterativamente.

Las librerías de ML en Python tienen esto implementado para poder hacer el calculo del gradiente.

Learning Rate

Si se fijan en el paso 2 del descenso de gradiente, la fórmula nos dice:

$$w = w - \delta \nabla_w L(D)$$

El δ es lo que llamamos **learning rate**. Define el tamaño del paso a realizar en cada iteración. Es un hiper parámetro, y es importante definirlo correctamente.

Si el learning rate es muy chico, los pasos serán chicos y tardaremos mucho en mejorar los pesos de nuestro modelo.

Si el learning rate es muy alto los pasos serán muy grandes y podemos oscilar o hasta diverger y perder el mínimo local.

Validation Set

Al utilizar este algoritmo de entrenamiento, es muy común tener un nuevo conjunto de datos para validar el comportamiento en nuevos datos **en cada iteración**. Para eso vamos a tener un **dataset de validación**, el cuál vamos a utilizar en cada iteración para medir cómo va mejorando el modelo.

Esto tiene dos ventajas:

- Podemos detectar problemas en el modelo como overfitting de mejor manera. Incluso podemos utilizar esto como criterio de corte, es decir frenar el algoritmo cuando la función de perdida aumente en el dataset de validación (*early stopping*).

Validation Set

Al utilizar este algoritmo de entrenamiento, es muy común tener un nuevo conjunto de datos para validar el comportamiento en nuevos datos **en cada iteración**. Para eso vamos a tener un **dataset de validación**, el cuál vamos a utilizar en cada iteración para medir cómo va mejorando el modelo.

Esto tiene dos ventajas:

- Podemos detectar problemas en el modelo como overfitting de mejor manera. Incluso podemos utilizar esto como criterio de corte, es decir frenar el algoritmo cuando la función de perdida aumente en el dataset de validación (*early stopping*).
- Es útil para tener una idea rápida de cómo afectan hiper parámetros como el *learning rate* a la performance de la red.

Validation Set

Al utilizar este algoritmo de entrenamiento, es muy común tener un nuevo conjunto de datos para validar el comportamiento en nuevos datos **en cada iteración**. Para eso vamos a tener un **dataset de validación**, el cuál vamos a utilizar en cada iteración para medir cómo va mejorando el modelo.

Esto tiene dos ventajas:

- Podemos detectar problemas en el modelo como overfitting de mejor manera. Incluso podemos utilizar esto como criterio de corte, es decir frenar el algoritmo cuando la función de perdida aumente en el dataset de validación (*early stopping*).
- Es útil para tener una idea rápida de cómo afectan hiper parámetros como el *learning rate* a la performance de la red.

  ¿Por qué un nuevo dataset si ya tenemos el de testeo?

Detección de dígitos

El ejercicio con el que vamos a poner a prueba esto es una red neuronal capaz de clasificar dígitos escritos a mano. Va a ser un modelo de **clasificación** en una red neuronal.

Es un ejemplo común por el famoso dataset de MNIST, con datos etiquetados de miles de dígitos. Cada imagen tiene un tamaño de 28×28, importante para tener en cuenta la capa de entrada.

Vamos a implementar la red neuronal en Tensorflow y Keras

Red neuronal

Para eso vamos a tener que definir varias cosas:

1. Dimensión de la capa de entrada, teniendo en cuenta la dimensión de las imagenes.

Red neuronal

Para eso vamos a tener que definir varias cosas:

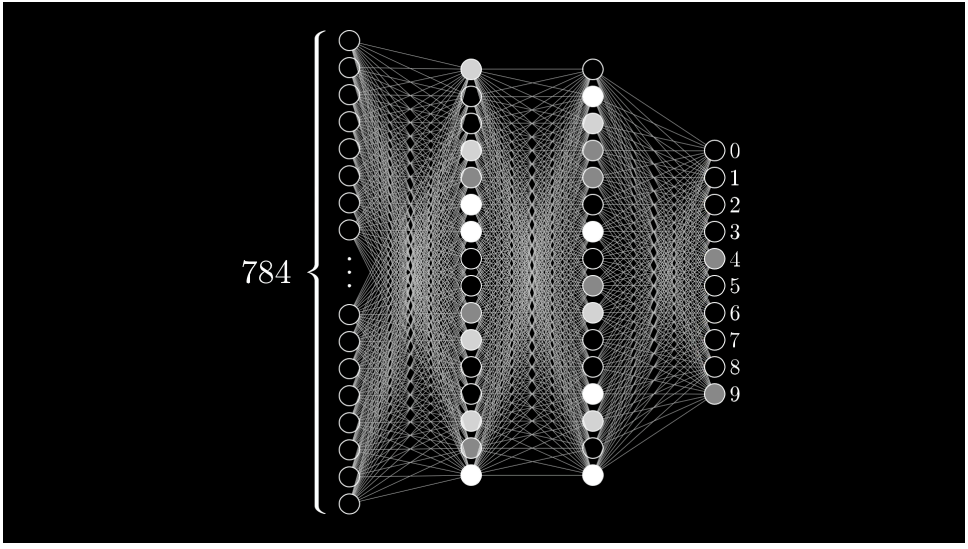
1. Dimensión de la capa de entrada, teniendo en cuenta la dimensión de las imagenes.
2. Cuántas capas ocultas y de qué tamaño. También con qué función de activación.

Red neuronal

Para eso vamos a tener que definir varias cosas:

1. Dimensión de la capa de entrada, teniendo en cuenta la dimensión de las imagenes.
2. Cuántas capas ocultas y de qué tamaño. También con qué función de activación.
3. La capa final va a tener 10 nodos (1 por cada dígito) y una función de activación Softmax por ser un problema de clasificación.

Red neuronal



Referencias

- Deep learning book
- Neural networks
- 3blue1brown