Clase 2. Introducción a Redes Neuronales (BackPropagation)

1. Hiperparámetros

Un **modelo** es una abstracción de la información (por decirlo con otras palabras, reglas que nos permitan describir un patrón o comportamiento), que nos permita distinguir a partir de los features si aceptamos o no a un aplicante. Este modelo va a aprender de información o de un **training set**.

Hay parámetros que no pueden aprender directamente de la información - estos explican cómo aprende el modelo. Se les llama hiperparámetros.

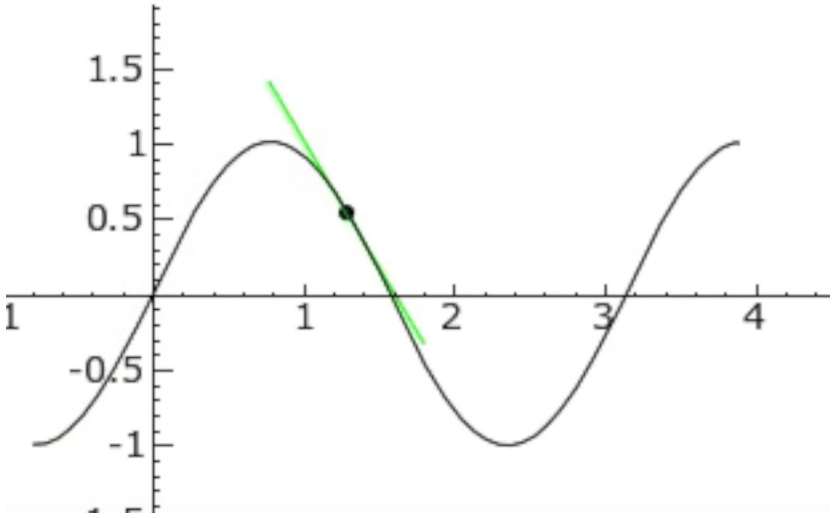
Algunos ejemplos son:

- Optimizador (Gradient Descent)
- Función de pérdida
- Número de hidden layers
- Número de neuronas en capas ocultas
- Epochs
- Learning rate
- Activation function (Sigmoid, ReLU, ...)

2. Repaso de Matemáticas

- Derivada

Es la pendiente de la tangente de una función en un punto dado. La derivada de una función es otra función que describe la pendiente en todos los puntos.



- Derivada Parcial

La derivada parcial es la derivada de una función de varias variables. Cuando se calcula la derivada parcial, se calcula respecto a una de las variables.

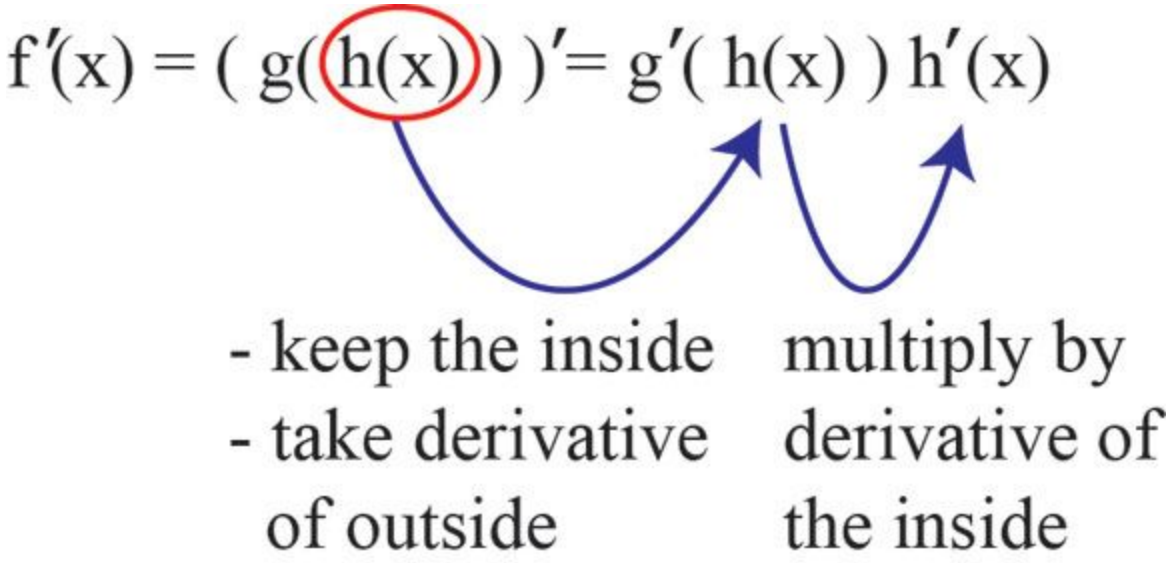
$$f(x,y) = y^4 + 5xy$$

$$\frac{\partial f}{\partial x} = 5y$$

$$\frac{\partial f}{\partial y} = 4y^3 + 5x$$

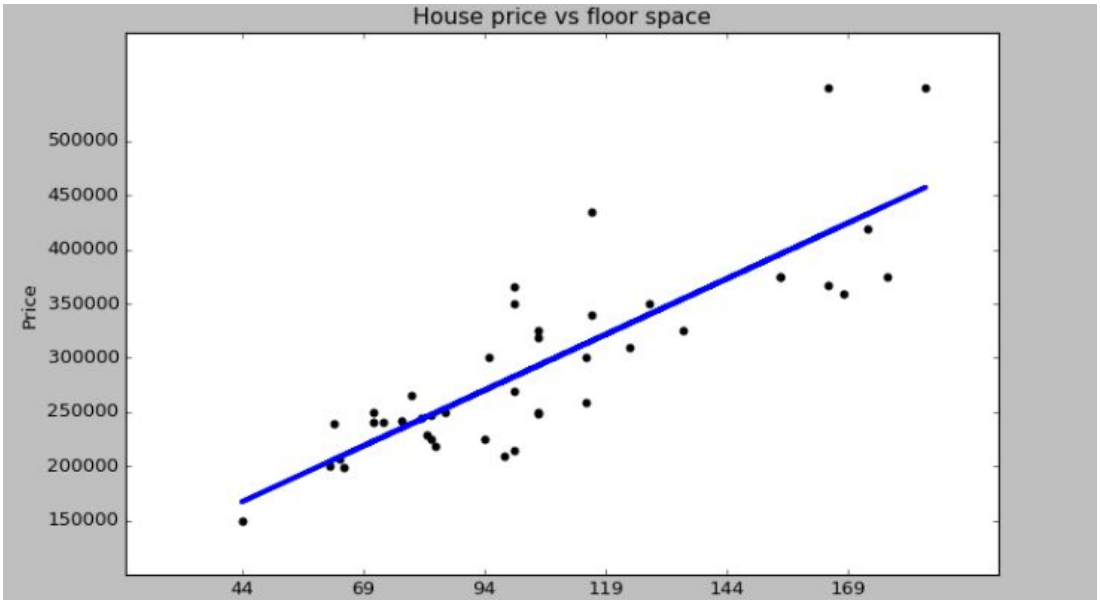
- Regla de la cadena

La regla de la cadena es una técnica para calcular funciones compuestas (función dentro de otra función). Las redes neuronales son funciones compuestas. Cada capa de la red es una función, y se va componiendo el resultado.



3. Regresión

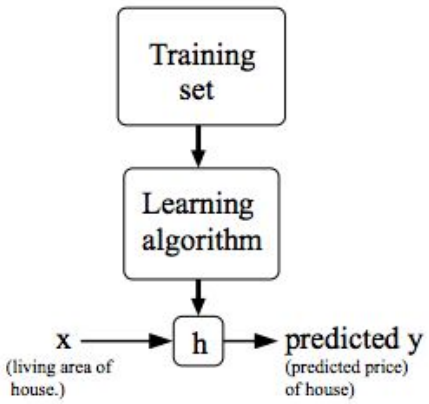
En la siguiente gráfica se ve la relación entre tamaño de la casa y el costo de la casa. La línea azul es nuestra hipótesis, mientras que los puntos negros son nuestro training set.



La regresión linear es una técnica de Aprendizaje Supervisado. El objetivo es, a partir de un conjunto de entradas, predecir un valor continuo. Como dice el nombre, el output es una línea, un valor continuo. Es decir, hay infinitas probabilidades. ¿Cuál será tu salario a partir de datos como dónde y qué estudiaste? ¿Cuánto valdrá tu casa a partir del número de cuartos?.

Training set

- m = # of training samples
- x = variables de entrada o features
- y = variable de salida
- (x, y) un ejemplo de entrenamiento
- $(x^{(i)}, y^{(i)})$ - i^{th} training sample

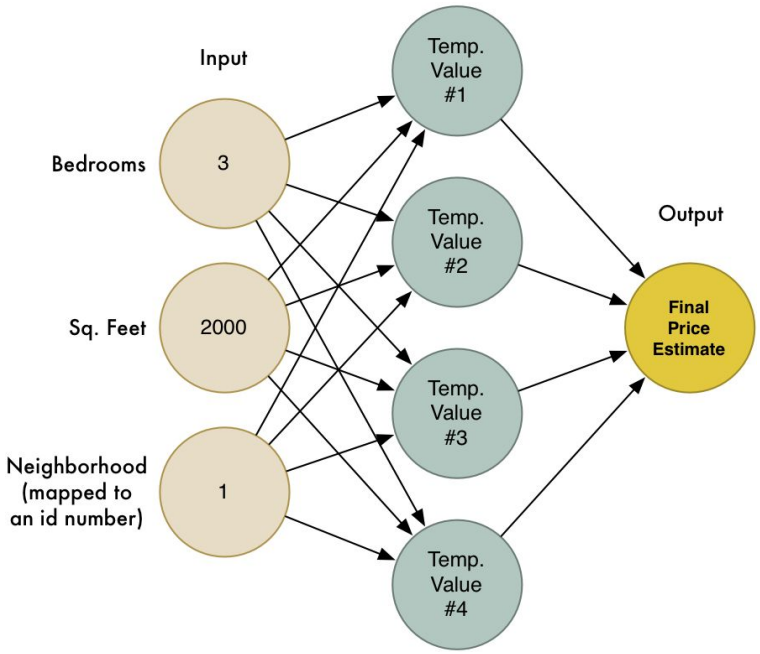


Para este problema, se utiliza un training set. Nosotros definimos un algoritmo de aprendizaje, como redes neuronales, para proponer una hipótesis. La hipótesis, o modelo entrenado, nos sirve para predecir **y** (costo de la cada) para nuevos valores de x (tamaño de la cada).

Para regresión lineal con una variable, nuestra hipótesis se describe por:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Esta función describe una línea (y=mx + b). Aquí tenemos dos parámetros. El primero describe la condición cuando x=0, la segunda explica la importancia, o pendiente, de x. Visto desde el punto de vista de una red neuronal, lo podemos ver como w*x + b (weight, input, bias). Un ejemplo de red neuronal para regresión lineal se podría ver de la siguiente manera:



4. **Función de error**

La pregunta principal es cómo asignar los parámetros (weight y bias). El objetivo es elegir θ_0 y θ_1 de manera que hθ(x) se acerque a la y real en nuestros ejemplos de entrenamiento (x,y). Este es un claro problema de optimización. Recuerden que parámetros e hiperparámetros son diferentes cosas. Los parámetros van a ser aprendidos a partir del training set. Los hiperparámetros los definen ustedes para describir cómo aprende el modelo.

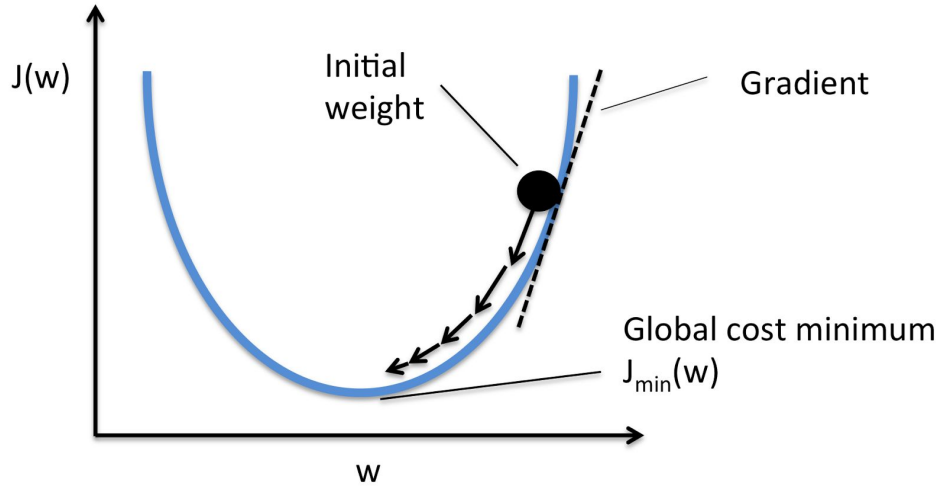
Lo primero que tenemos que hacer es evaluar nuestro modelo. Esto lo podemos hacer con **loss functions**, o funciones de error. Una función de pérdida es Mean Squared Error:

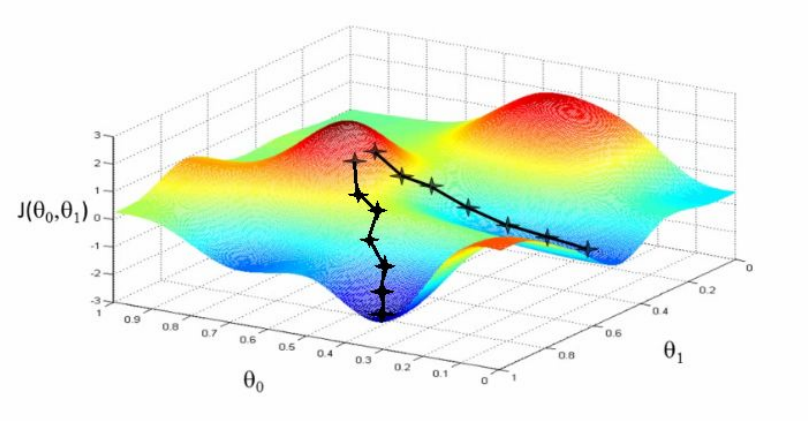
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}_i - y_i \right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i \right)^2$$

Aunque dividir entre dos no es parte de la función, mucha gente lo agrega para que, al calcular su derivada, se cancelen y se simplifique el problema. Optimizar para J es lo mismo que optimizar para 2J.

5. Gradient Descent

Una vez que podemos calcular el error, el reto está en optimizar (minimizar) para ese error. El algoritmo más común para esto es **Gradient Descent**.





En la gráfica anterior se muestra Gradient Descent para regresión linear (dos parámetros). En este caso, como se puede ver, si se inicia en diferentes puntos, se puede llegar a mínimos diferentes, lo cuál es una de las principales fallas de GD.

Algoritmo

- Inicia con parámetros iniciales (se suelen iniciar en 0)
- Se cambian los parámetros para reducir J hasta llegar a un mínimo.

$$\text{Repeat until convergence } \{$$

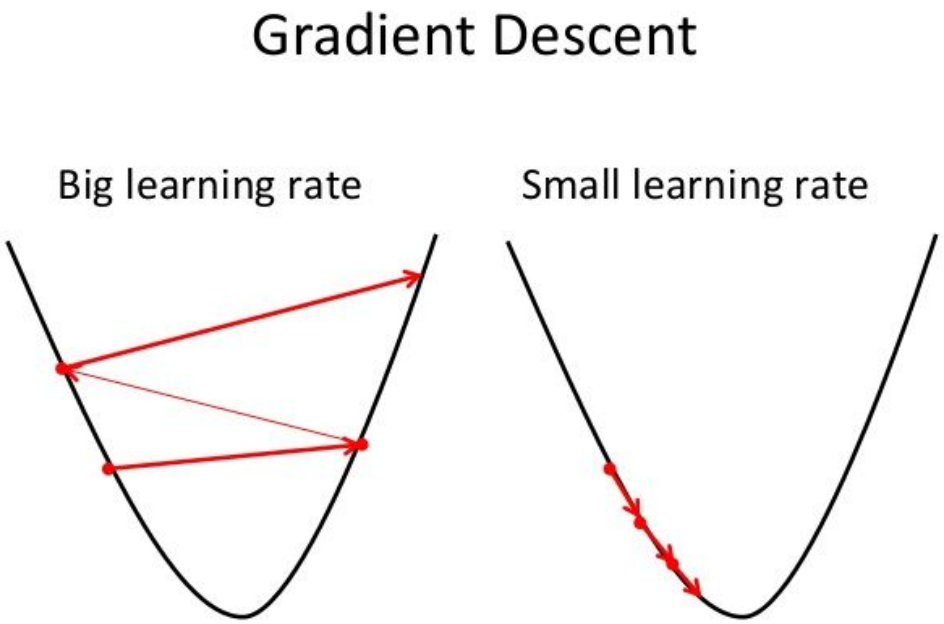
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

6. **Learning Rate**

El learning rate describe qué tan “fuertes” son los pasos que se dan en Gradient Descent. Si es muy pequeño, el algoritmo tardará mucho en converger y llegar al mínimo. Si es muy grande, se puede saltar el mínimo e incluso divergir.



7. Gradient Descent para Regresión Linear

Dado que Gradient Descent busca minimizar el error moviendo el parámetro según la pendiente de la función del error, necesitamos utilizar las derivadas.

Esta es la función de error:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}_i - y_i\right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i\right)^2$$

Estas son las derivadas parciales respecto a weight y bias:

$$\frac{\partial}{\partial \mathrm{m}} = \frac{2}{N} \sum_{i=1}^{N} - x_i(y_i - (mx_i + b))$$

$$\frac{\partial}{\partial \mathrm{b}} = \frac{2}{N} \sum_{i=1}^{N} -(y_i - (mx_i + b))$$

Si tenemos muchos parámetros, como en DL, podemos hacer esto:

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

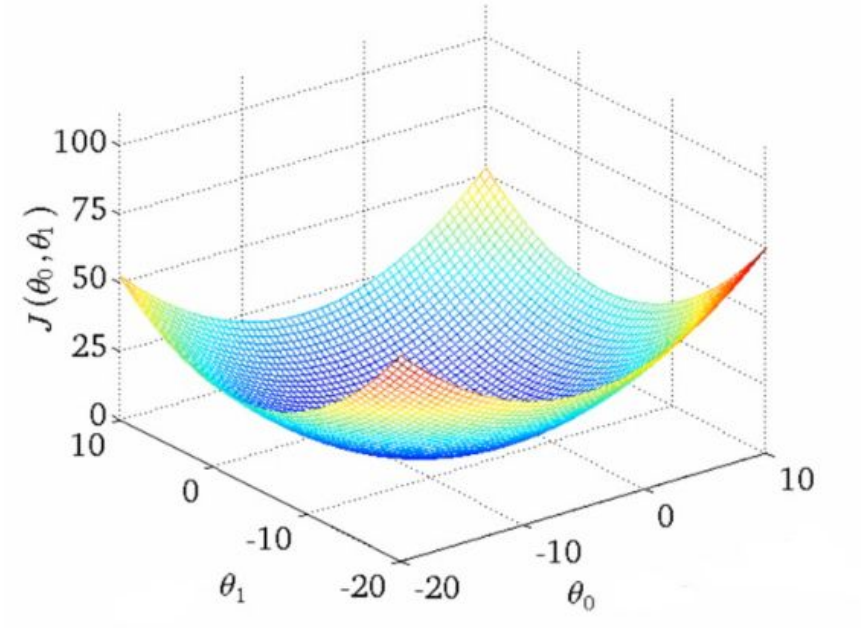
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)}$$

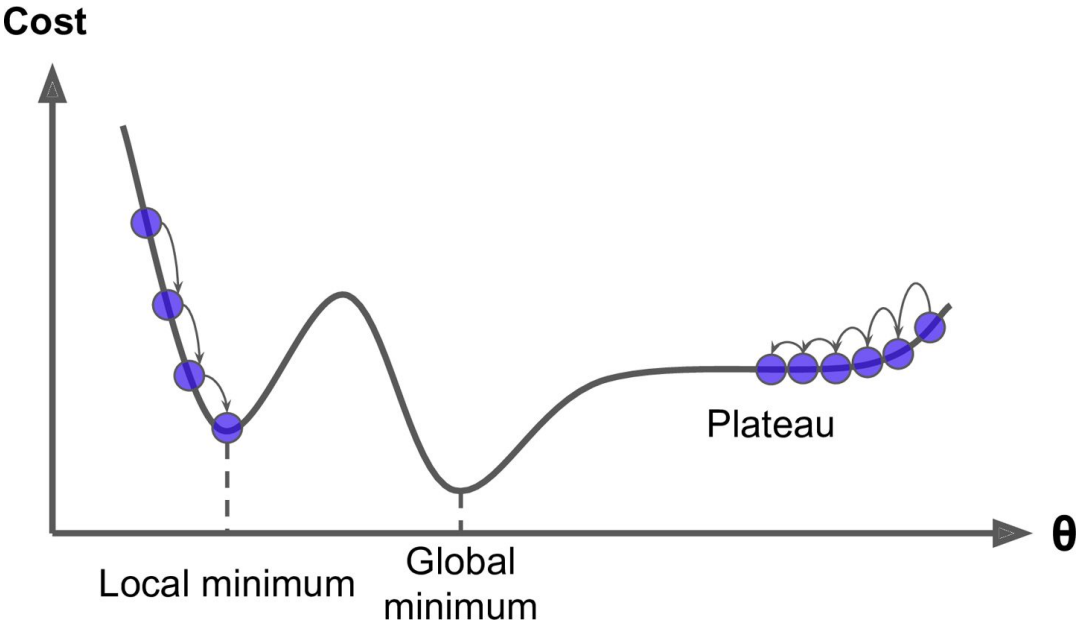
...

}

En el caso de regresión linear, se ha descubierto que su función de error es convexa:



El problema con Gradient Descent es que a veces llegar a mínimos locales en vez de mínimos globales. Hay variantes de GD que permiten prevenir esto. Esta variante, la más común, se llama Batch Gradient Descent. Aquí, para cada step_gradient, utilizamos todo nuestro training set. Hay variantes que permiten enviar sólo una parte del dataset.



8. **Gradient Descent para Redes Neuronales**

a. Definir el problema

En este problema tenemos 4 ejemplos en nuestro training set. Cada ejemplo tiene 3 features. A partir de estas tres entradas, queremos predecir un output. Nota que 4 ejemplos no es nada para una red neuronal. Las redes neuronales suelen necesitar de 10mil a 10 trillones de ejemplos.

Inputs			Output
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

b. Diseñar la arquitectura de la red neuronal

Lo principal es que tendremos 3 neuronas de entrada (por cada feature) y una neurona de salida. Después de eso, nosotros podemos definir cuántas capas ocultas, sus funciones de activación, y su tamaño. Para este caso sólo vamos a tener una capa oculta con 4 neuronas. Por lo tanto, nuestra red tendrá la estructura 3-4-1. Esto es importante para el siguiente paso.

c. Inicializar los pesos

Inicializamos los pesos de manera aleatoria con un promedio de 0. Las matrices de peso deben tener la estructura (3,4) y (4,1).

d. Definimos la función de activación

Hay muchas funciones de activación. En este caso utilizaremos la función sigmoid. También debemos definir su derivada.

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

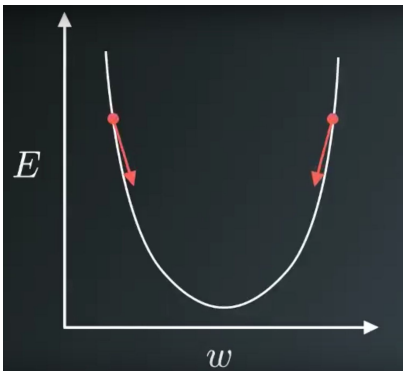
e. FeedForward

Utilizamos multiplicación de matrices para calcular el valor de cada capa. En este ejemplo no tendremos bias. Debemos activar el resultado de cada capa. Por esto se considera que las redes neuronales son funciones compuestas.

```
# Feed forward
l0 = X
l1 = sigmoid(np.dot(l0, weights0))
l2 = sigmoid(np.dot(l1, weights1))
```

f. Calcular el error

En este problema no utilizamos MSE, aquí sólo calculamos la diferencia en **y** y **y** predecida, o la última capa. El objetivo es utilizar el error y el gradient para ir moviendo los pesos.



$$\Delta w = -gradient$$

g. Backpropagate

Se calcula la derivada parcial de la función de error respecto a cada peso individual de la red neuronal. Permite calcular el error en cada neurona.

```
l2_error = y - l2
l2_delta = l2_error*sigmoid_deriv(l2)

l1_error = l2_delta.dot(weights1.T)
l1_delta = l1_error * sigmoid_deriv(l1)

weights1 += l1.T.dot(l2_delta)
weights0 += l0.T.dot(l1_delta)
```

En este problema no tenemos learning_rate (o es igual a 1), pero si lo tuviéramos, lo actualizaremos de esta manera:

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

9. Algoritmos de Redes Neuronales

1. Forward Propagation: Calcular suma de entrada y activación de cada neurona aplicando multiplicaciones de matrices iterativamente.
2. Calcular la función de error/pérdida en la última capa L. Depende de la función de pérdida y el training sample específico.
3. Backpropagation: Calcula el error en cada neurona de cada capa. Se utiliza multiplicación de matrices con derivadas para hacer esto.
4. Calcula la derivada del costo respecto a los pesos y la derivada del costo respecto a los biases. Esto es parte de Gradient Descent.
5. Actualiza los pesos y bias.

10. Otras funciones de pérdida

Puedes revisar otras funciones de pérdida en este link. Si quieres aprender las diferencias de cada uno, revisa este link.

11. Training vs Testing set

El training set es el que se da al algoritmo para entrenar y que aprenda. El testing set es el que usamos para evaluar el modelo. Cuando tenemos un dataset, con 100k de ejemplos, normalmente nos quedamos con 80k, entrenamos el modelo con esos. Usamos los 20k restantes para evaluar el modelo y ver qué tan bien lo hace.

Overfitting es cuando se entrena mucho al modelo. En este caso el modelo no generaliza bien, sino que aprendió de memoria la información.

Underfitting es cuando no se entrena suficiente un modelo. En este caso, el modelo no es capaz de generalizar para nuevos casos.