Artificial Neural Networks (ANN)

- Una gran parte de la sección será basada casi solo en teoría,
- Luego las ideas serán implementadas en código.

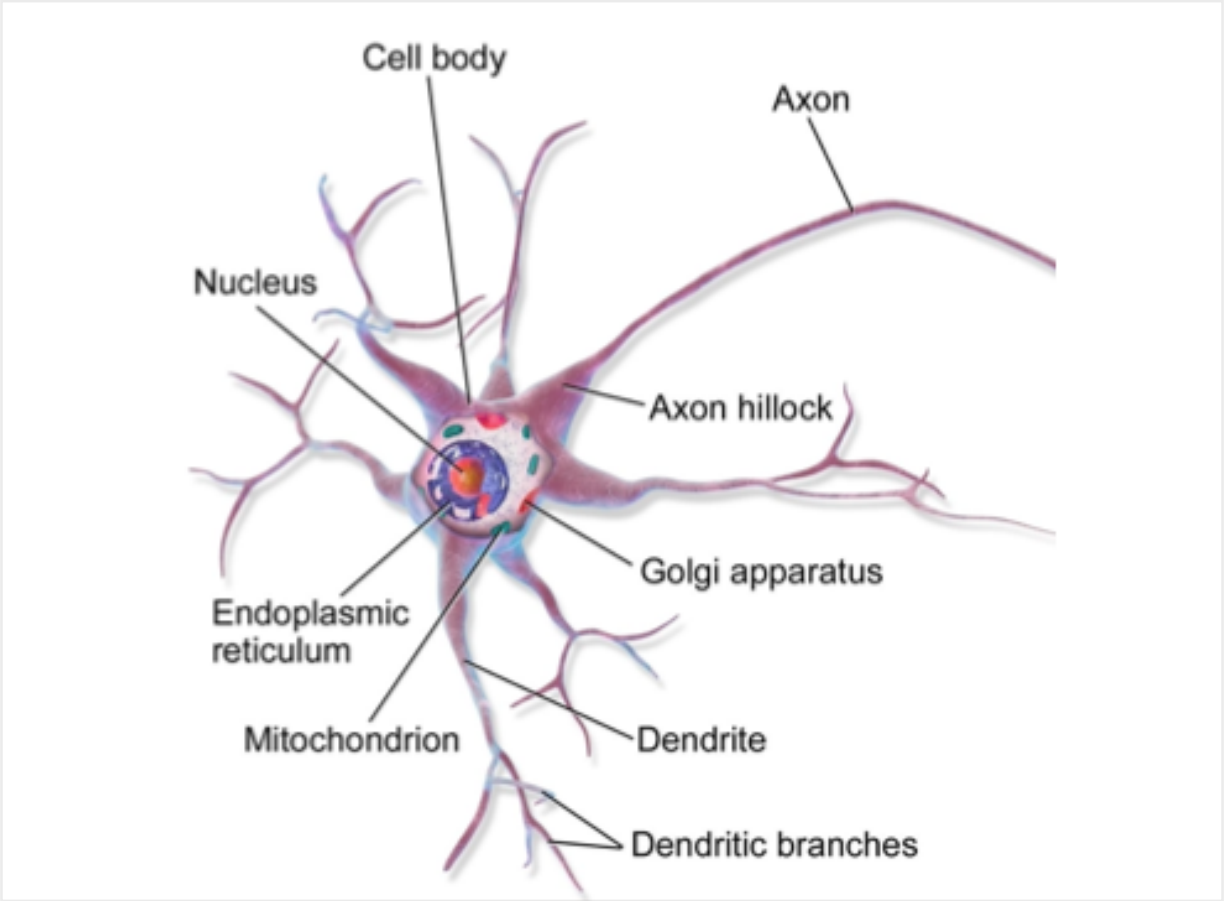
Tópicos:
- Modelo de perceptrón a redes neuronales,
- Funciones de activación,
- Functiones de costo,
- Feed Forward networkds,
- BackPropagation.

Códing topics:
- Tensorflow 2.0 Keras Syntax,
- ANN w/ Keras:
  - Regression,
  - Classification.
- Ejercicios para Keras ANN,
- Tensorboard Visualizations.

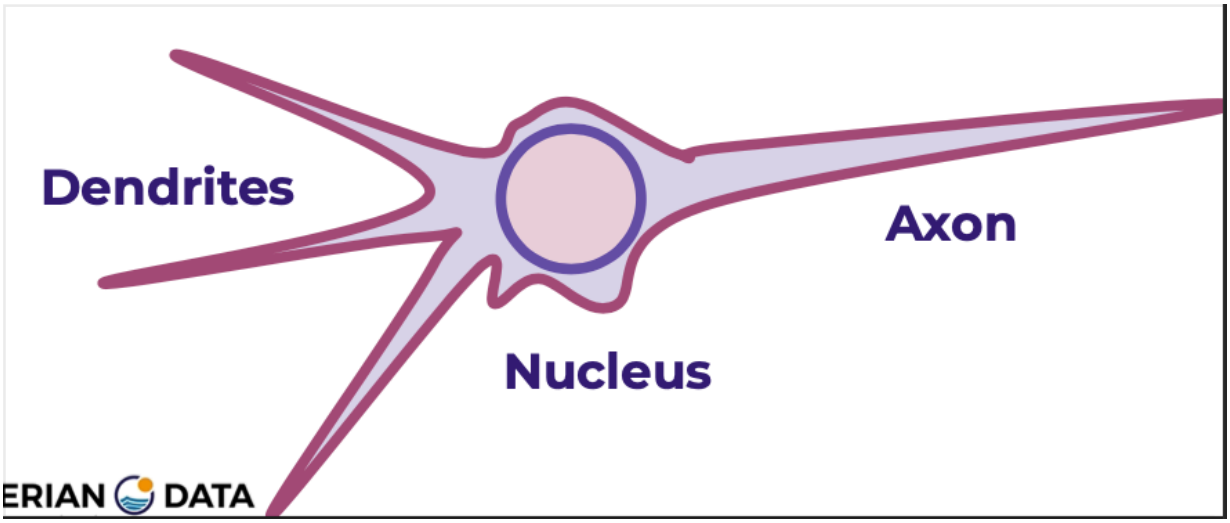
Módelo Perceptrón

para entender DL, hay que entender lo siguiente primero para construir la intuición:
- Neurona biológica,
- Perceptrón,
- Multi-Layer modelo perceptrón,
- Red neuronal de DL.

Ilustración de neurona:



Una versión simplificada para entender el funcionamiento macro se puede ver como:



Las dendritas pueden pensarse como inputs que van al nucleo y el axon se puede entender como un output/salida. El nucleo hace algún tipo de “cálculo” y da una única salida.

En 1958 Fran Rosenblatt creó el modelo matematico qu representa esto. En ese

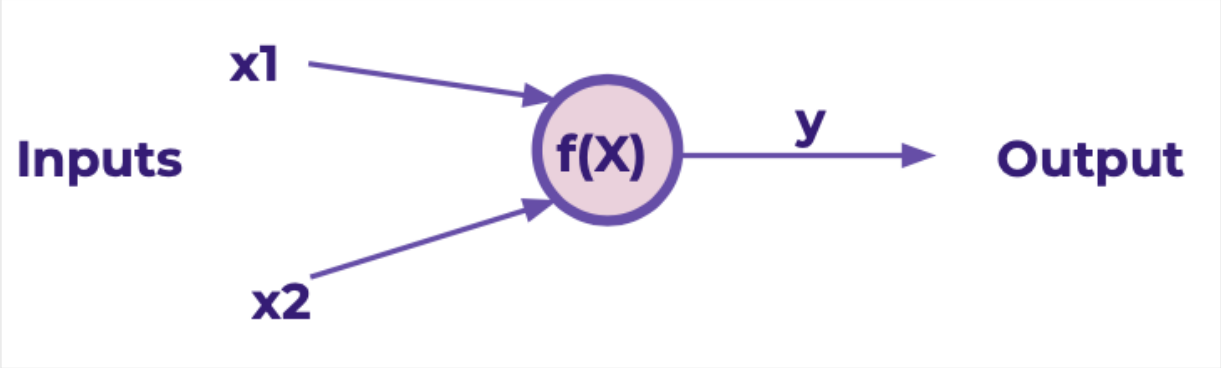
entonces se le vio gran potencial, asegurnado que “El perceptrón podrá eventualmente ser capaz de aprender, tomar decisiones y traducir lenguas”.

Sin embargo, casi 2 décadas después, en 1969, Marvin Minsky y Seymor Papert publicaron el libro “Perceptron”, en donde mataron la idea de un modelo matemático útil del perceptrón. Este libro marcó el inicio del “invierno de la IA”, que se vio representado como una época con muy poca inversión en proyectos de IA.

De todas formas, hoy en día conocemos lo poderosas que son las redes neuronales, las cuales partieron del modelo simple del perceptrón, así que se continuará estudiando el modelo biológico para luego llevarlo a código.

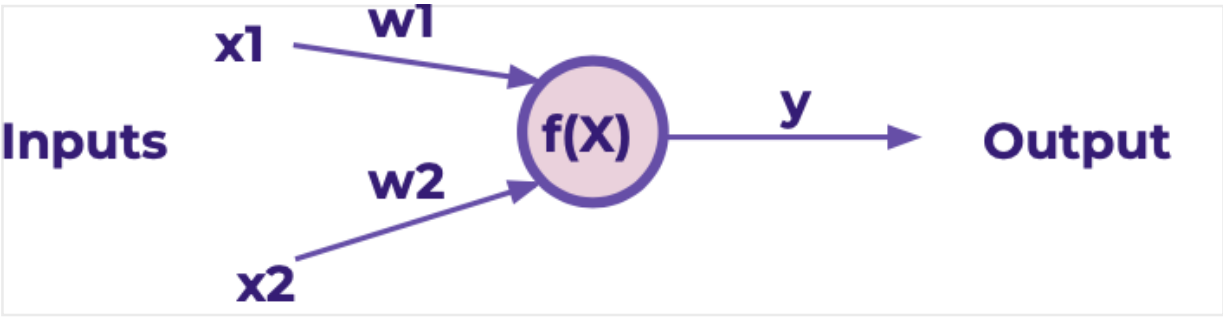
1. Representación de las unidades biológicas por matemáticas:

Se les asigna a las entradas respectivamente x1 & x2,
Existe una función dentro del nucleo que transforma las entradas,
termina en una salida única llamada ´y´.



Si definieramos f(X) como una suma, entonces tendríamos que:
y = x1+x2.

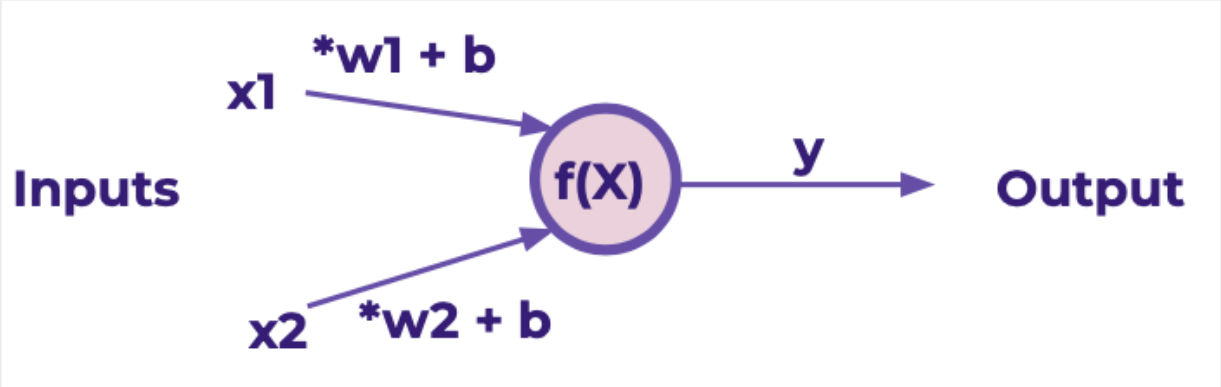
Realistamente, nosotros querríamos ser capaces de ajustar algún parámetro con el fin de “aprender”, para ello se agregan pesos (weights /w) con el fin de ponderarlos a cada entrada.



ahora:

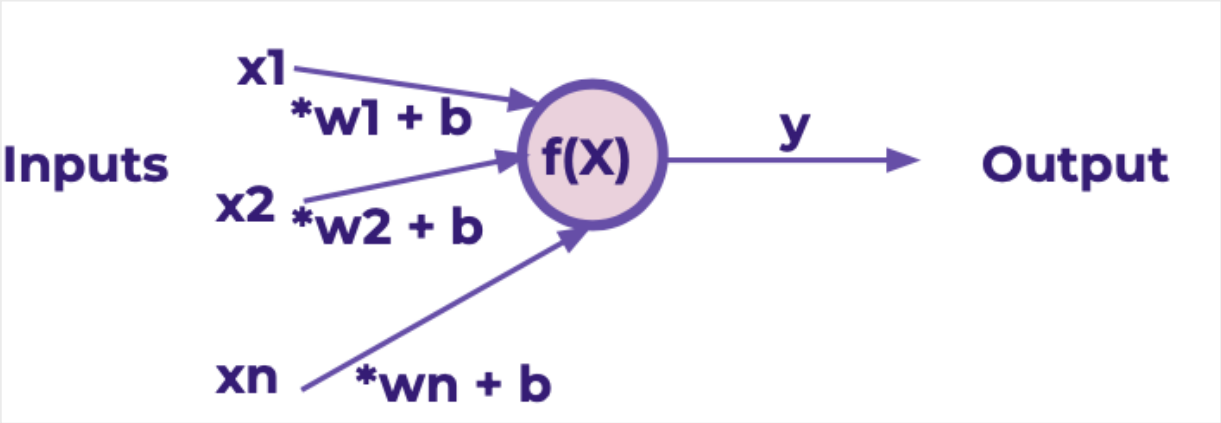
y = x1w1 + x2w2

Pero, ¿qué pasa si los input son 0?, el peso no cambiaria nada.
Es para lo anterior que se ocupa un sesgo /bias, como un escalar que se suma a los pesos,



quedando
y = (x1w1+b) + (x2w2+b)

y finalmente se generaliza a:



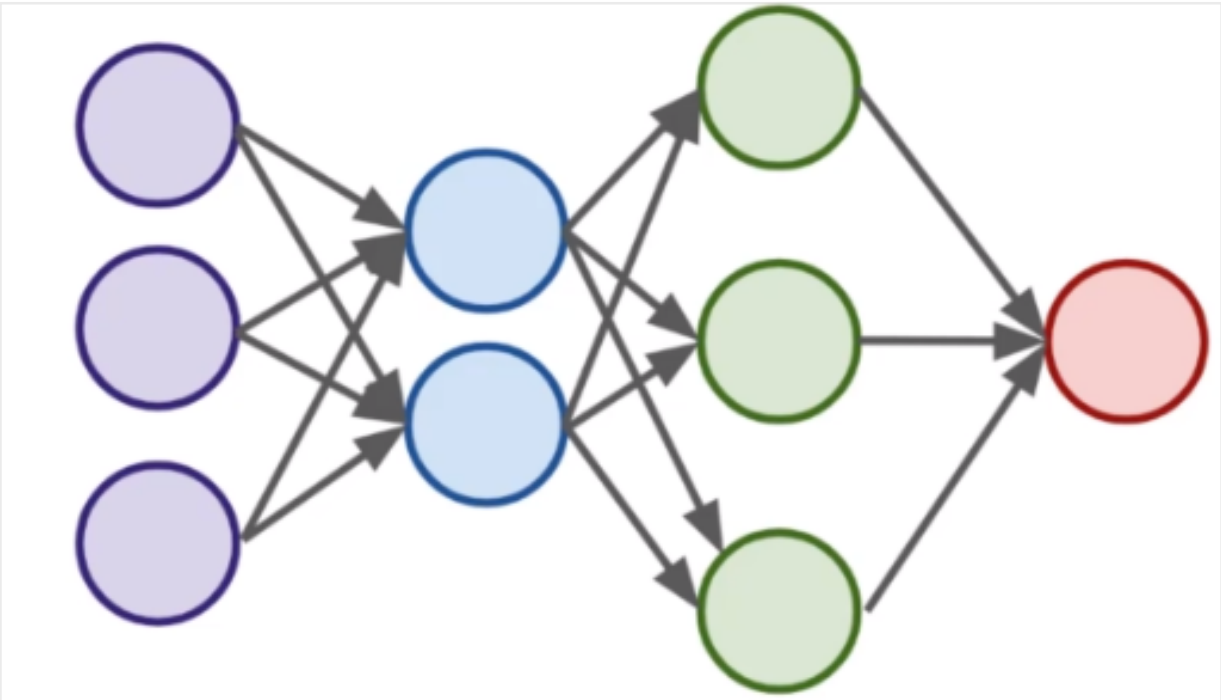
Expresado matematicamente tenemos:

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

Redes Neuronales / Neural Networks

Aprender el funcionamiento de un simple perceptron no será suficiente para aprender sistemas complicados,
Pero podemos expandir la idea de un perceptrón simple para crear un modelo de perceptrón multi capa, también llamado como una red neuronal básica.

¿Cómo construimos una red neuronal de la idea de un perceptrón?

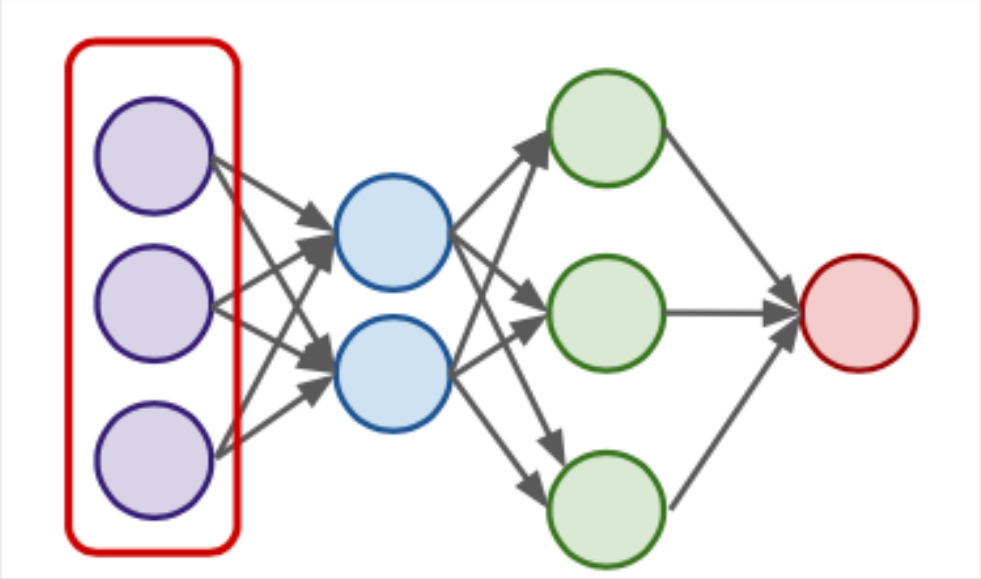


Generamos verticalmente redes de perceptrones singulares y conectamos sus salidas como entradas a la siguiente capa, así queda como la salida de una capa es la entrada de la siguiente.

Por ahora nos enfocaremos en el modelo “Feed forward” que significa que todo el output del primer layer va hacia adelante en el modelo hasta llegar a la salida, esto permite a la red comportarse como un “todo” y aprender relaciones e interacciones entre las variables de entrada /features.

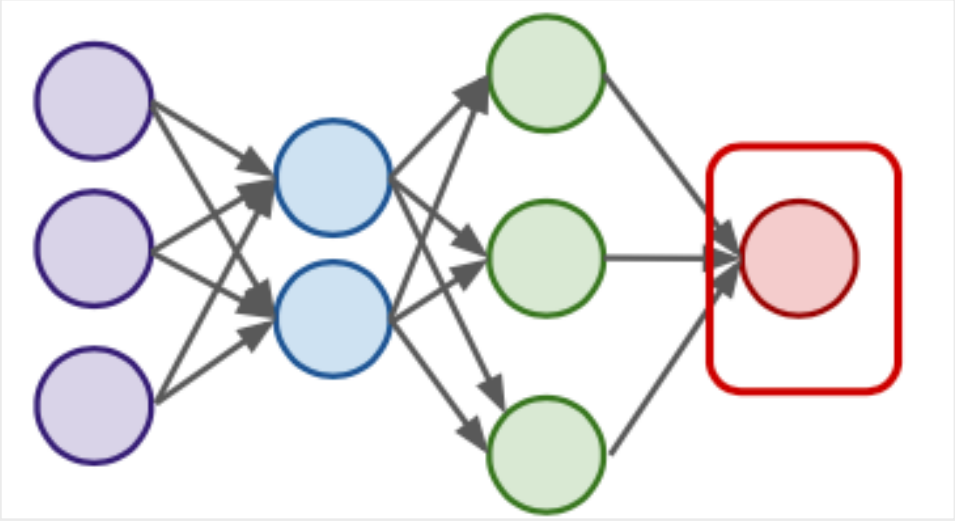
Tipos de capas

Capa de entrada



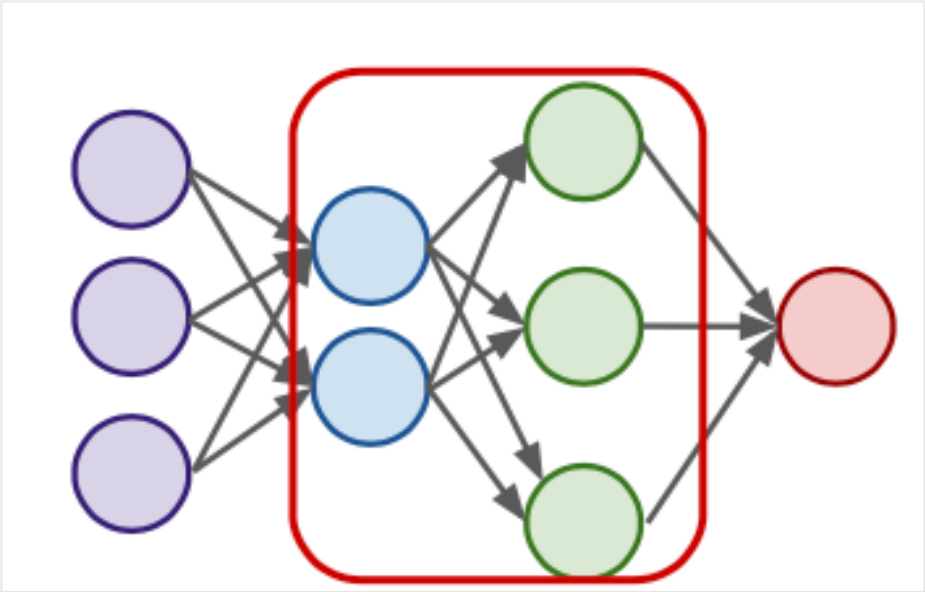
Capa de salida

nota, esta capa puede tener más de una neurona.



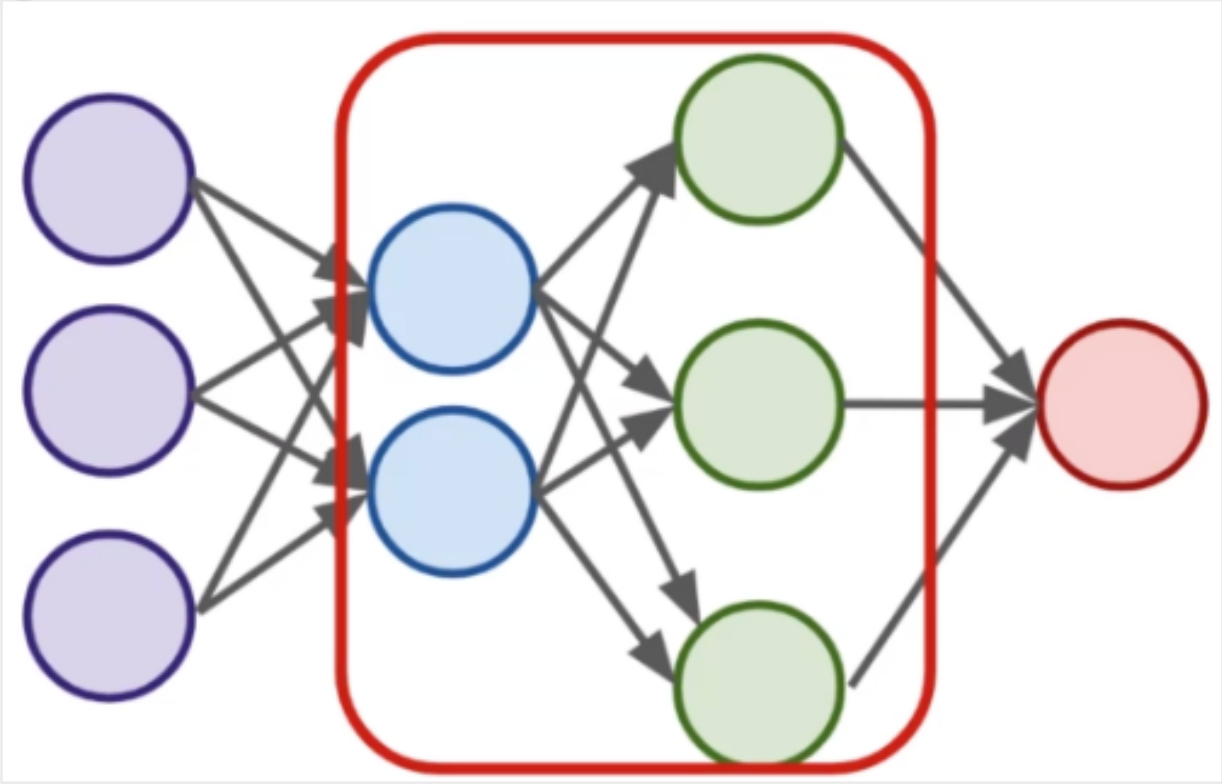
Capas ocultas

las capas entre la salida y la entrada son las capas ocultas.

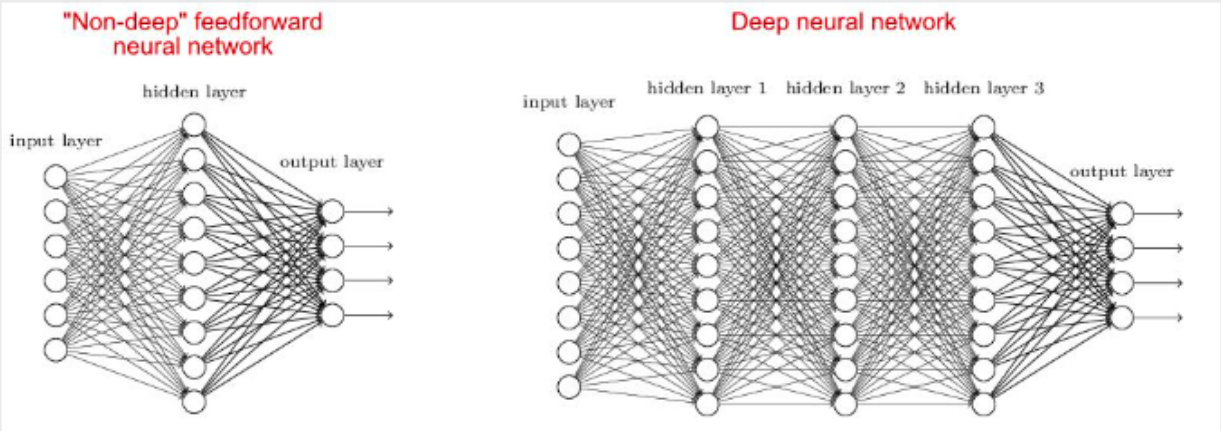


¿Cuándo una red neuronal se convierte en una “red neuronal profunda”?

La terminología indica que cuando se tienen 2 o más capas ocultas, se denomina red neuronal profunda.



Ejemplos



Datos interesantes:
Una de las cosas que es increible acerca del marco de trabajo de redes neuronales, es que puede ser usado para aproximar funciones continuas.
Zhou Lu y más tarde Boris Hanin probaron matematicamente que las redes neuronales pueden aproximar cualquier función convexa continua.
Para más info de este punto, ver en wikipedia “Universal approximation theorem”.

OJO
En las presentes notas se habla de la función que contiene las redes neuronales como una suma, pero en realidad eso no será útil en gran parte de los casos, se querrá fijar restricciones a las salidas, especialmente en tareas de clasificación, para ello se detallará a continuación sobre las **funciones de activación.**

Funciones de activación

Podemos interpretar el bias como un offset value, haciendo que la entradas ponderadas por su peso deban alcanzar cierto treshold para poder tener efecto.
Por ejemplo, si tenemos que
b = -10
x*w + b
Entonces vemos que x*w no empezará a tener efecto hasta que sobrepase al bias teniendo un producto superior a 10.

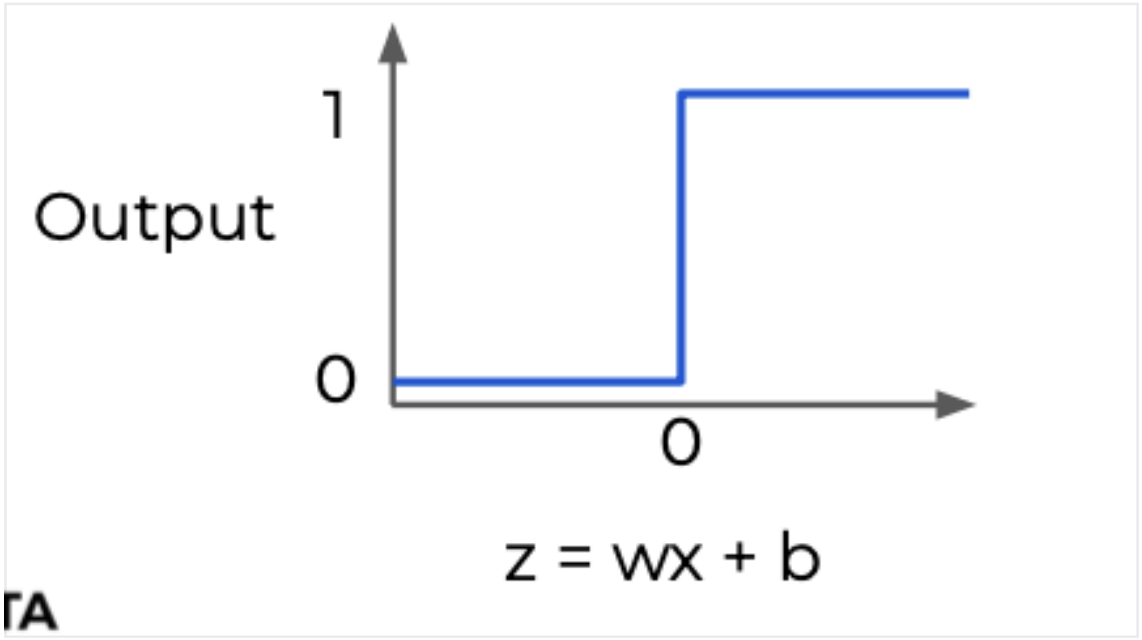
Luego, queremos ajustar un set de limites para la salida total de x*w + b, es decir, pasarlo por unas **funciones de activación** que limiten su valor.

Mucha investigación se ha hecho en las funciones de activación y su efectividad.

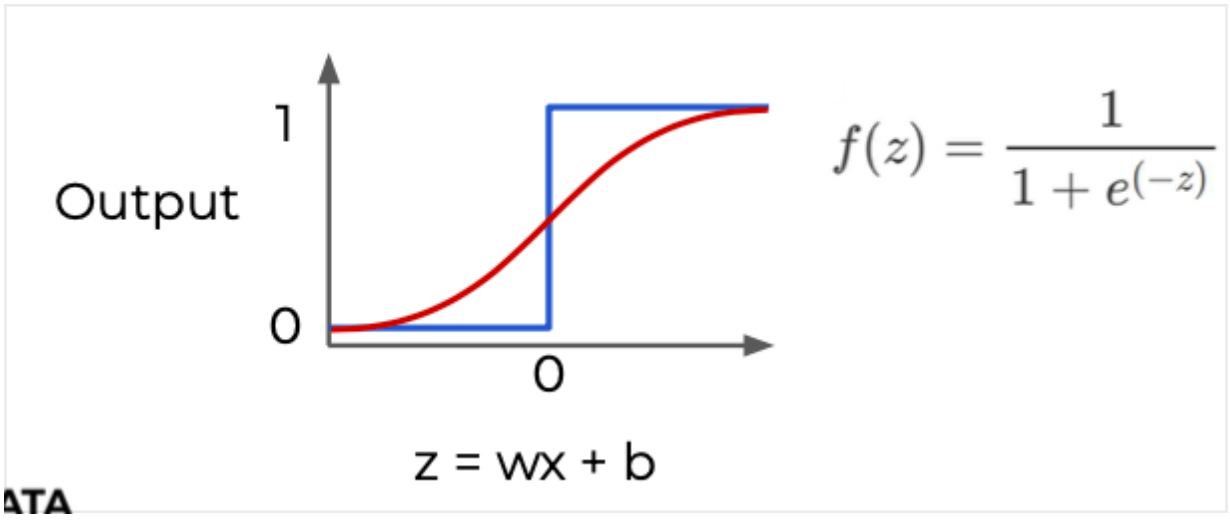
Exploremos algunas útiles:

- Si tuvieramos un problema de clasificación binaria, querríamos un output o 0 ó 1, por lo que podríamos ocupar una función de escalón que mapee ó 0 ó 1.

Independiente del valor, siempre va a dar como salida ó 0 ó 1.

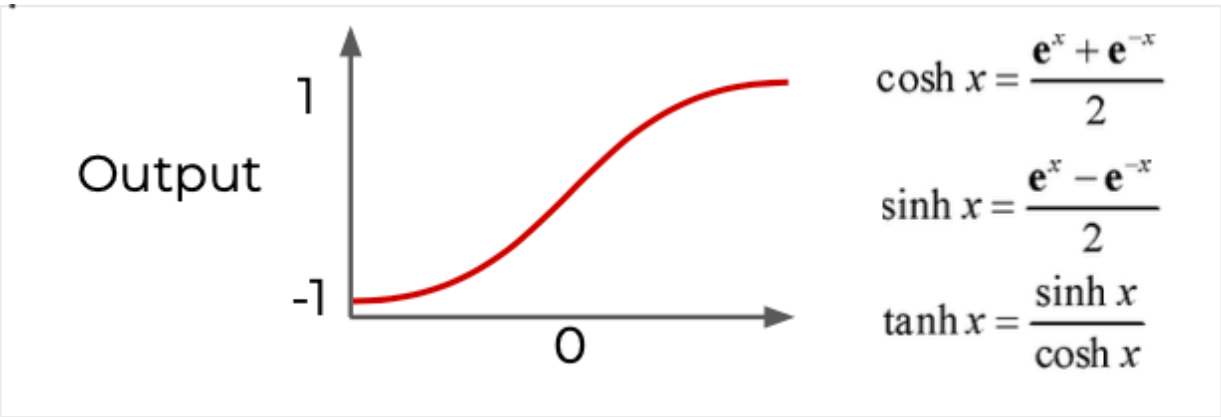


Sin embargo, si quisieramos una función más dinámica que mantenga una forma similar, podríamos utilizar la función sigmoide.

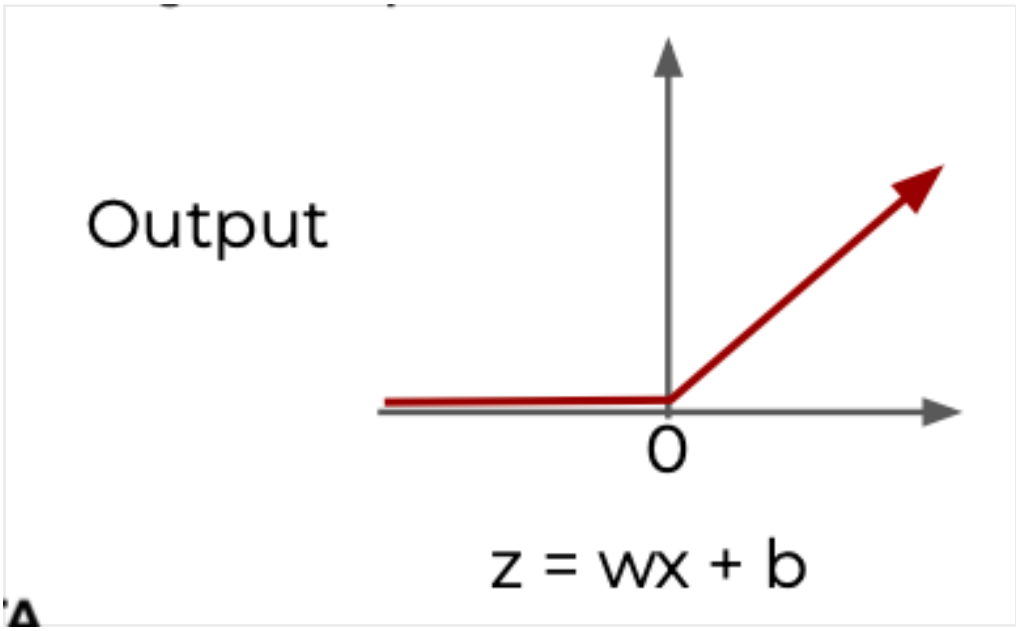


Cambiar la función de activación puede ser bueno dependiendo la tarea, la sigmoide por ejemplo todavía sirve para clasificación y es más sensitiva a pequeños cambios.

Otra función de activación que podrá encontrarse es la tangente hiperbólica.
Tanh(z)



Una de las más usadas y efectivas es la Rectified Linear Unit (ReLU), en realidad bastante simple
max(0,z)



ReLU ha tenido bastante buen performance, especialmente lidiando con el problema del “vanishing gradient”.
Se recomienda siempre ir por default a ReLU dado su buen performance general.

Funciones de activación Multi-Class

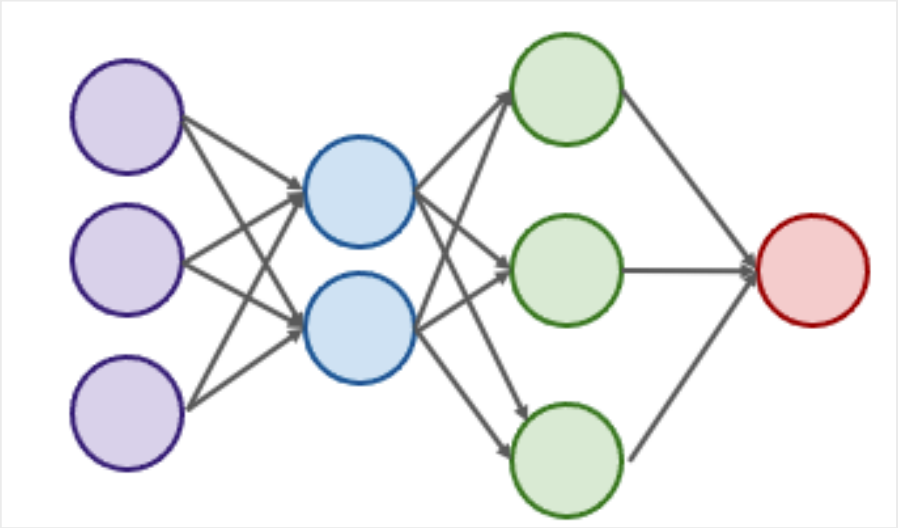
Las funciones de activación vistas previamente hacen sentido para un single output, pero qué pasa cuando tengamos una situación “multi clase”?

Generalizando, este problema se podría dar mayormente por las siguientes 2 situaciones:

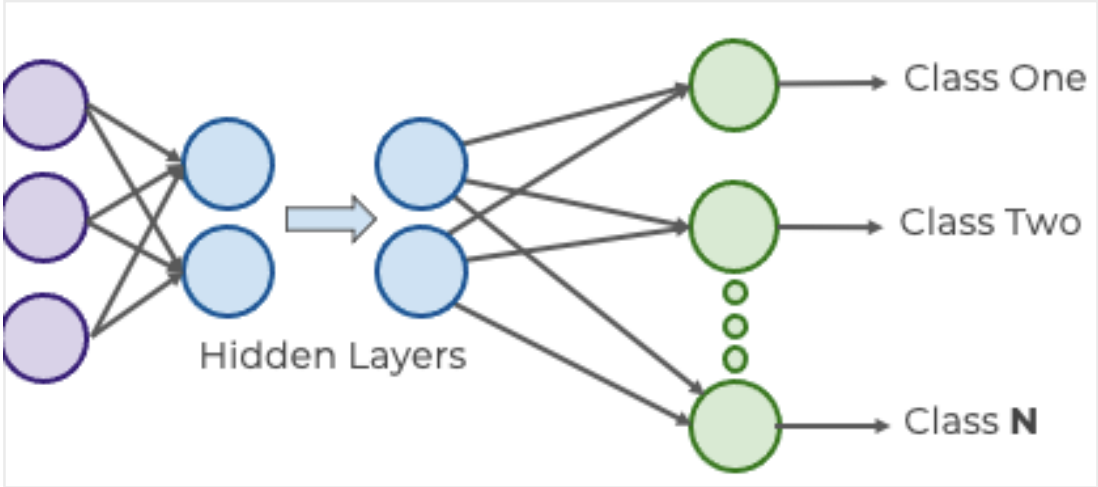
1. Clases no-exclusivas
    a. Un data point puede tener múltiples clases/categorias asignadas a él mismo (ej: una foto puede tener 1 o más tags)
2. Clases mutuamente exclusivas
    a. Sólo una clase por data point. (Ej: Color de una foto (blanco ó a color).

Lo primero es saber como plantear la arquitectura de la red si sabemos que tenemos un problema multiclase, la manera más fácil es tener sólo 1 nodo de salida por clase.

Previamente ilustramos la red neuronal de single output,



Podemos expandir esta idea para multi clase



Lo cual significa que tendremos que organizar las categorias para la capa de salida, no podemos tener strings de tipo “clase 1”, “clase 2”, etc. Sino que tendremos que utilizar técnicas como **one hot encoding** para transformar el dataset de manera entendible para la red.

Ejemplo para clases mutualmente exclusivas

Data Point 1	RED
Data Point 2	GREEN
Data Point 3	BLUE
...	...
Data Point N	RED

→

	RED	GREEN	BLUE
Data Point 1	1	0	0
Data Point 2	0	1	0
Data Point 3	0	0	1
...	...	...	...
Data Point N	1	0	0

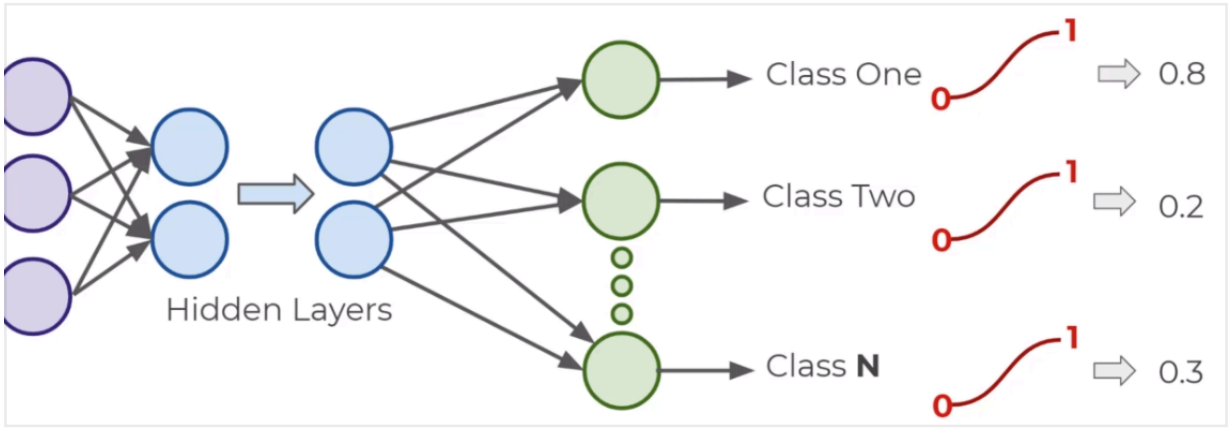
Ejemplo para clases no-exclusivas

Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

Ahora que tenemos nuestros data ordenada, debemos escoger correctamente la función de activación de clasificación que la última output layer tendrá.

Para non-exclusive, ocupamos la función sigmoide, cada salida representará la probabilidad entre 0 y 1 de asignar la X clase.



Recordar que para el caso no exclusivo, cada neurona de salida será independiente de la otra, permitiendo al data point pertener a más de una clase asignada.

Para casos mutualmente exclusivos, ocupamos la función de activación **softmax**.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

Calcula las distintas probabilidades en un espacio de K opciones. Y la suma de todas las opciones sumará a 1.
Escogemos la probabilidad más alta para asignarle como clase.

- [Red , Green , Blue]
- [0.1 , 0.6 , 0.3]

Funciones de costo y descenso del gradiente

Lo que entendemos hasta ahora:

- Las redes neuronales tienen inputs, los cuales luego los multiplican por pesos y agregan biases/sesgos.
- Los resultados son pasados por funciones de activación las cuales al final de las capas, entregan cierto output.
- Este output es la estimación del modelo con respecto a lo que representa el data point,
- Así que, después de que la red crea la predicción, ¿cómo la evaluamos?
- Y después de la evaluación, cómo actualizamos los pesos y biases/ sesgos de la red?

Respondiendo a lo anterior, tenemos que tomar las predicciones de la red y compararlas con los valores reales (esto es tomando los datos del set de entrenamiento durante la etapa de ajuste/entreno del modelo).

La función de costo (también referida como la función de pérdida / loss function) escencialmente es una medición de qué tan lejos estoy del valor real y debe ser un average.

Definición de variables para continuar

y: valor real
a: predicción de la red

pesos y bias /sesgo
w*x + b = z
pasar z a la función de activación o(Z) = a

Función de costo cuadrático

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Calculamos la diferencia entre los valores reales contra su predicción.

(la notación a^L hace referencia a que L es la capa de salida, es decir, si se quisiera hacer referencia a capas anteriores tendría que notarse a^L-n)
(nota: la notación hace referencia a vectores de entrada y salida, dado que estaremos lidiando con "batchs" de datos de entrenamiento y predicción)

Al elevar al cudrado logramos 2 objetivos útiles:

1. Mantener todo en un dominio positivo,
2. Castigar fuerte los errores grandes.

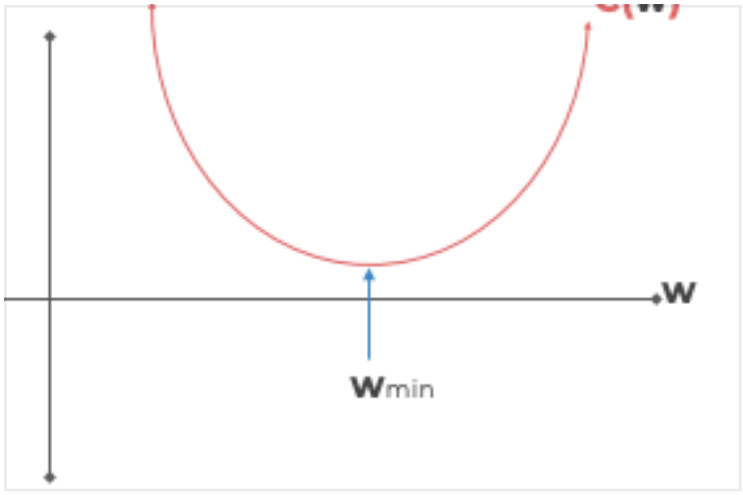
En general, podemos pensar en funciones de costo como funciones de 4 grandes parámetros

$$C(W, B, S^r, E^r)$$

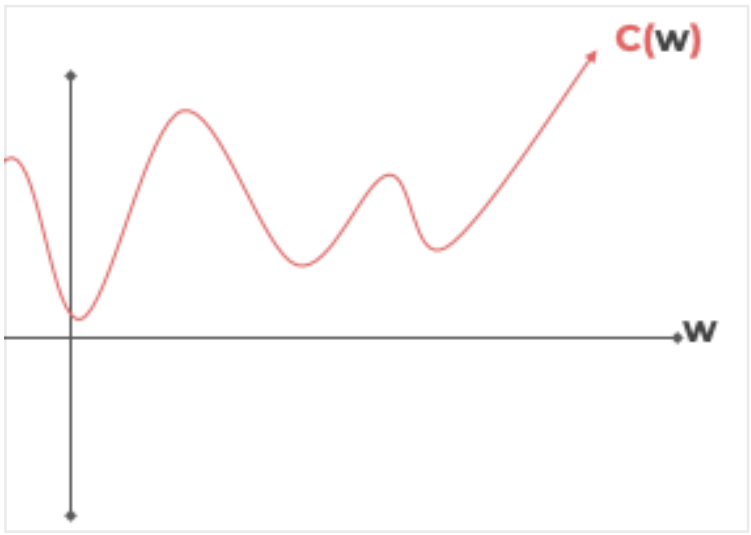
W: Pesos de la red neuronal
B: Biases /sesgos de la red neuronal
S^r: Input de un single training sample
E^r: Output deseado de ese training sample

Queremos **minimizar** nuestra función de costo (overall error). Lo que significa que tenemos que obtener cuales son los valores W resultan en el mínimo C(w).

Rapidamente podríamos pensar en derivar y resolver para 0.



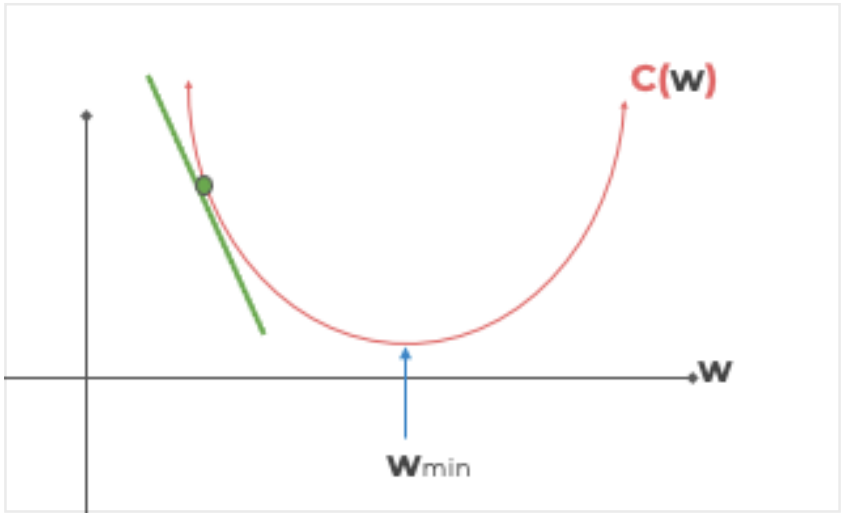
Pero qué pasa con las funciones muy complejas?, además n-dimensionales!



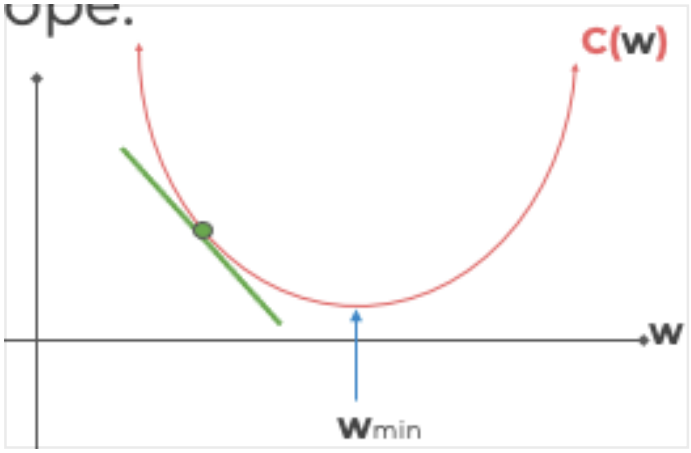
Para ello, podemos ocupar el **descenso del gradiente** para resolver este problema.

Para ejemplificarlo en 2 dimensiones:

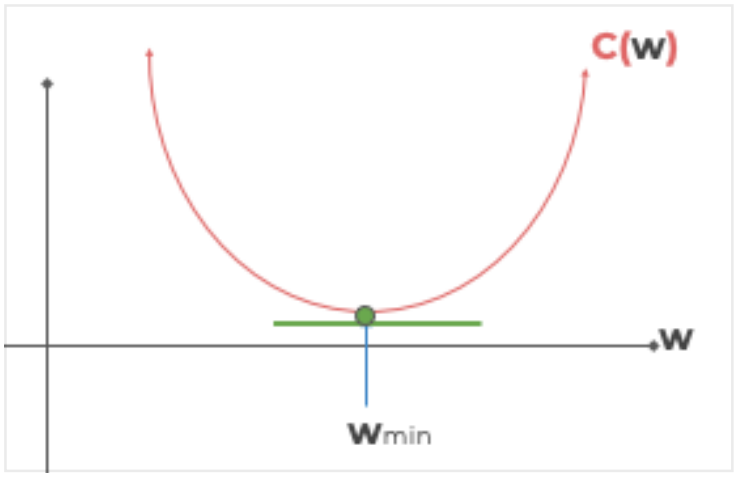
Calculamos la pendiente de punto en la función



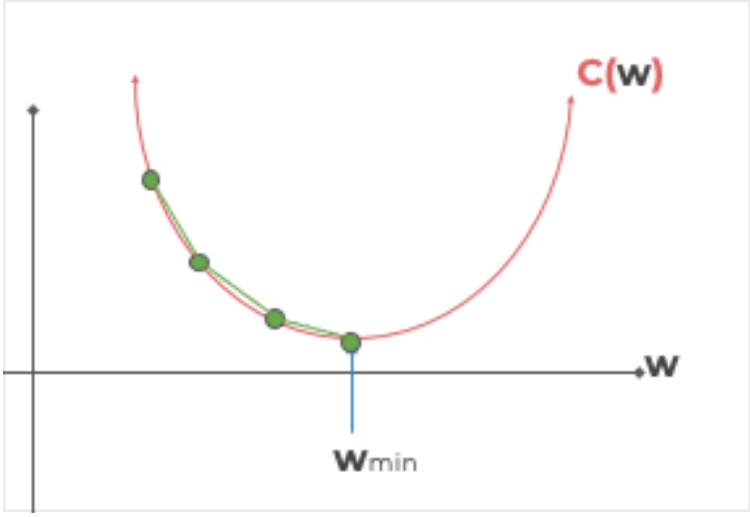
y nos movemos hacia abajo en dirección de la pendiente



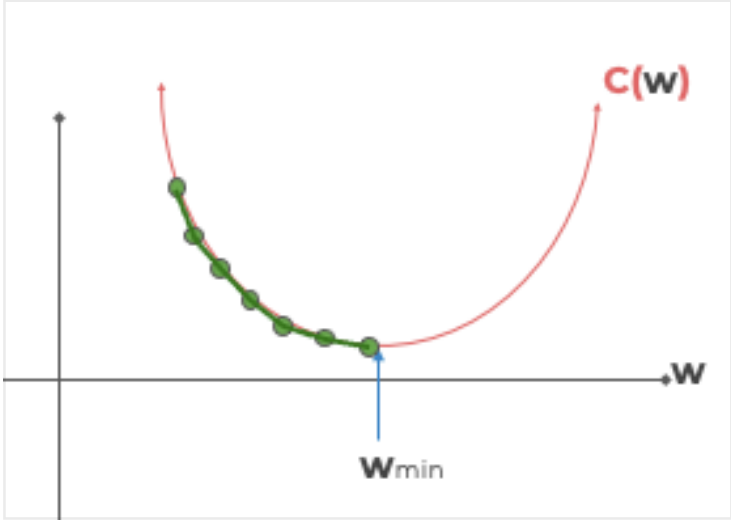
hasta converger a 0, indicando el mínimo.



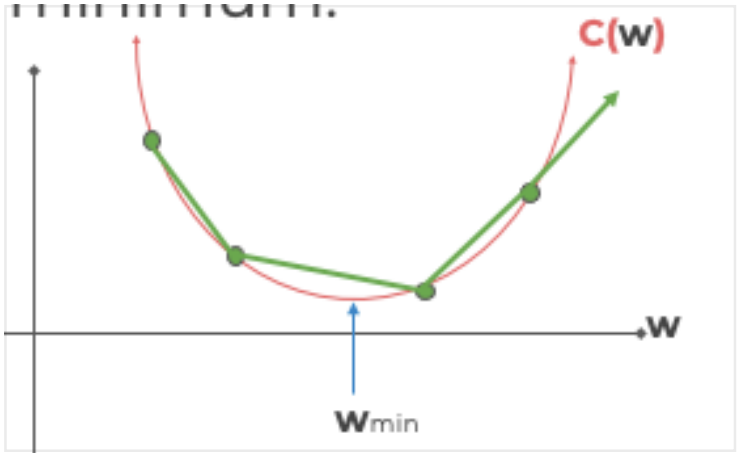
El largo de los pasos seguidos hacia la pendiente se le denomina “learning rate”



A menor learning rate mayor será el tiempo que tomará para encontrar el mínimo



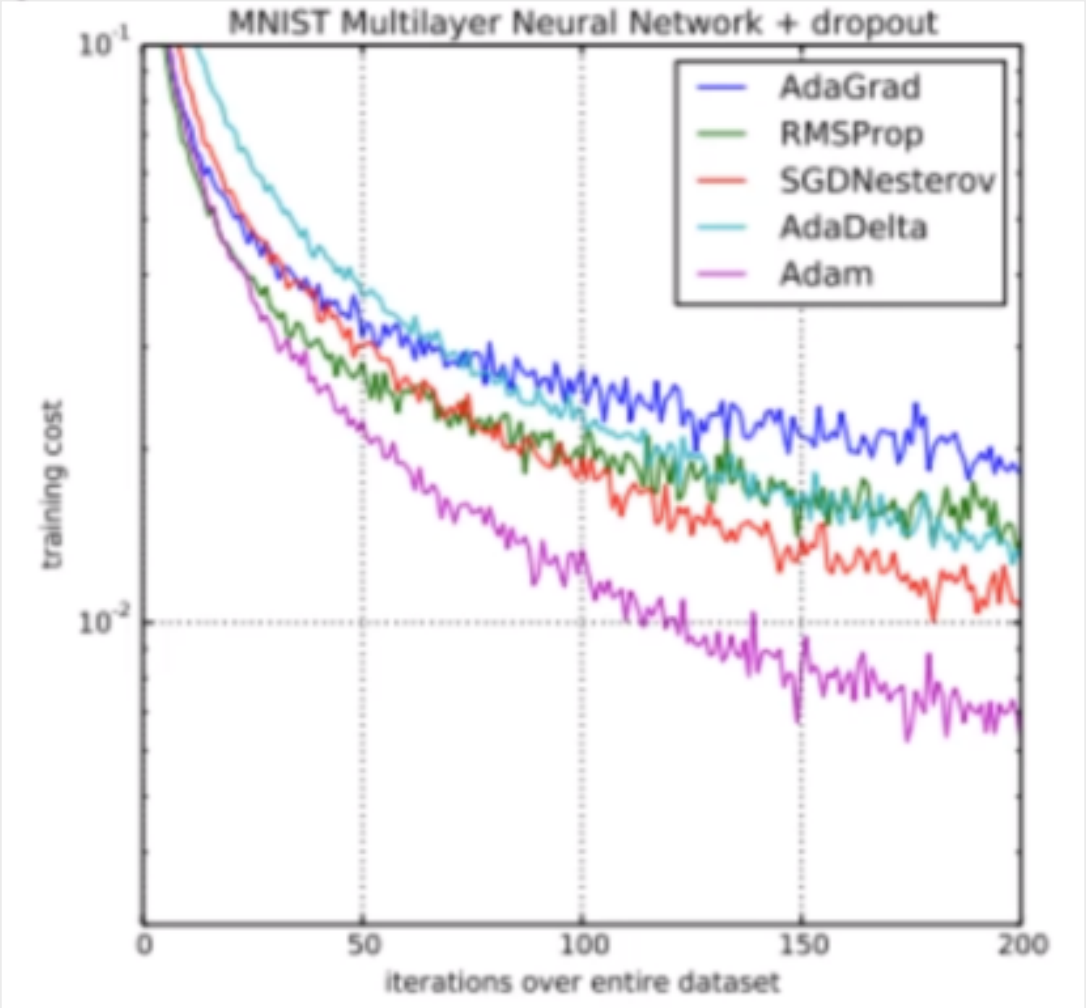
Y a mayor el learning rate podría tardar menos pero corremos el riesgo de overkillear el problema.



El learning rate mostrado en los ejemplos es una constante, pero podemos ser inteligentes y ajustar el learning rate mediante vayamos descendiendo, como empezar con learning rates altos e ir disminuyendo mediante vaya convergiendo a 0. Esta técnica es conocida como **descenso del gradiente adaptativo.**

Un algoritmo optimizado del descenso del gradiente es “Adam”, publicado por Kinga y Ba en 2015, usaremos este algoritmo en el código.

Adam versus otros algoritmos:



Comentarios sobre lidear con N-dimensiones
Cuando se habla de vectores n-dimensionales (tensores), la notación cambia de derivadas a gradiente,
esto significa que calculamos:

$$\nabla C(w1,w2,...wn)$$

Nota:
Para problemas de clasificación usualmente ocuparemos la función de costo “cross entropy”. El supuesto es que tu modelo predice una distribuciópn de probabilidad p(y=i) para clada clase i=1, 2,...,n.

Clasificación binaria:

$$-(y \log(p) + (1-y) \log(1-p))$$

Clasificación M(clases) > 2

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Hasta ahora entendemos cómo las redes pueden recibir una entrada, y afectarla con pesos, sesgos y funciones de activación para producir una salida estimada. Luego aprendimos a evaluar ese resultado.

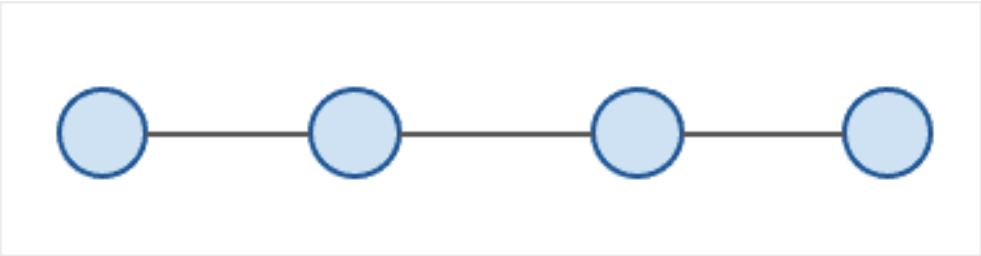
Lo último que tenemos que aprender sobre la teoría es:
Una vez que obtenemos nuestro valor de coste/pérdida, ¿cómo volvemos a ajustar nuestros pesos y sesgos?
Esto es **backpropagation.**

Backpropagation

Empezaremos por construir una intuición detrás de la backpropagation, y luego nos sumergiremos en el cálculo y la notación.

Fundamentalmente, queremos saber cómo cambia el resultado de la función de coste con respecto a los pesos de la red, para poder actualizar los pesos para minimizar la función de costo.

Ejemplo: Red de 1 neurona por cada capa

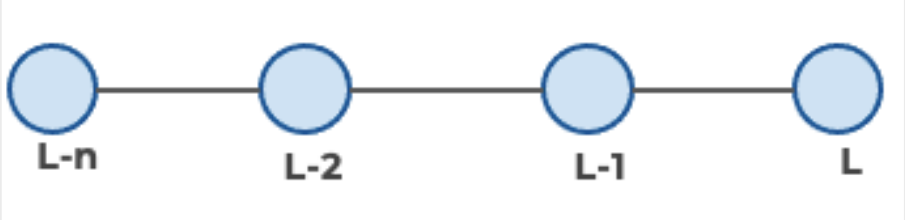


Cada input recibe un peso y un bias /sesgo

Es decir tenemos:

$$C(w1,b1,w2,b2,w3,b3)$$

Con L capas:



Concentrándonos en las últimas 2 capas, definamos:

$$z=wx+b$$

Y aplicamos la función de activación

$$a = \sigma(z)$$

Esto significa que tenemos:

- $z^L = w^L a^{L-1} + b^L$
- $a^L = \sigma(z^L)$
- $C_0(\ldots) = (a^L - y)^2$

Y escencialmente queremos entender qué tan sensible es la función de costo con cambios en w:

$$\frac{\partial C_0}{\partial w^L}$$

La idea principal aquí es que podemos utilizar el gradiente para volver a través de la red y ajustar nuestros pesos y sesgos para minimizar la salida del vector de error en la última capa de salida.