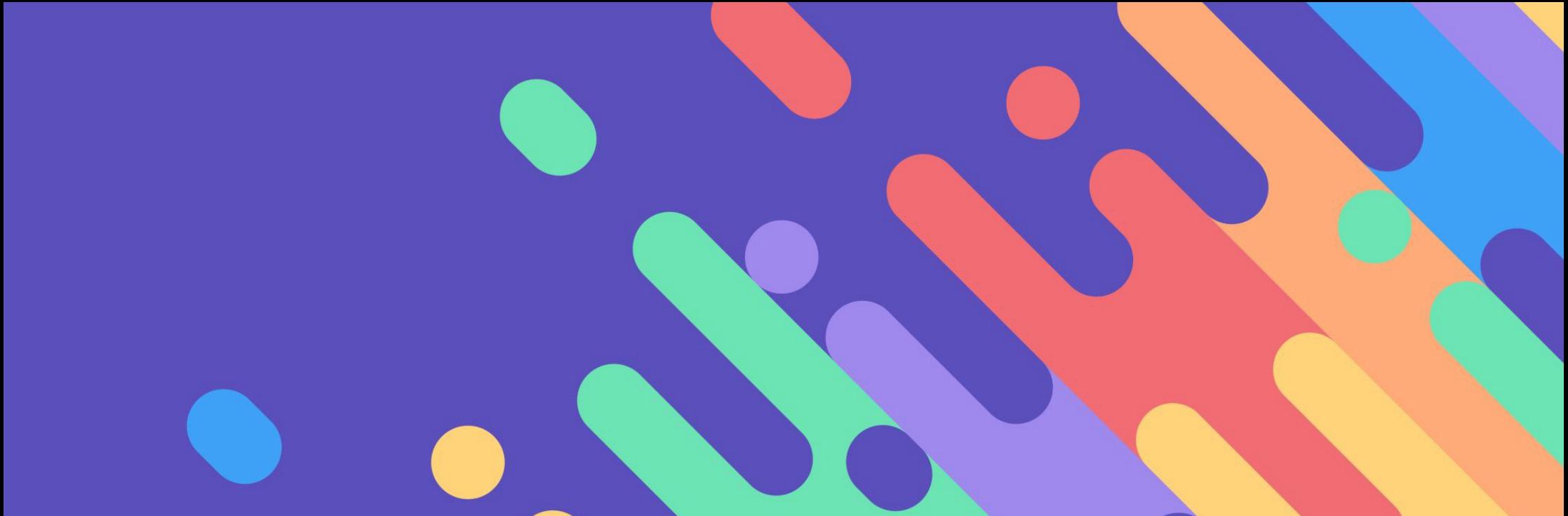


# REDES NEURONALES



Aprendizaje  
Automático  
CEIoT - FIUBA

Dr. Ing. Facundo Adrián  
Lucianna



---

LO QUE VIMOS LA CLASE ANTERIOR...

---

# ÁRBOLES DE DECISIÓN

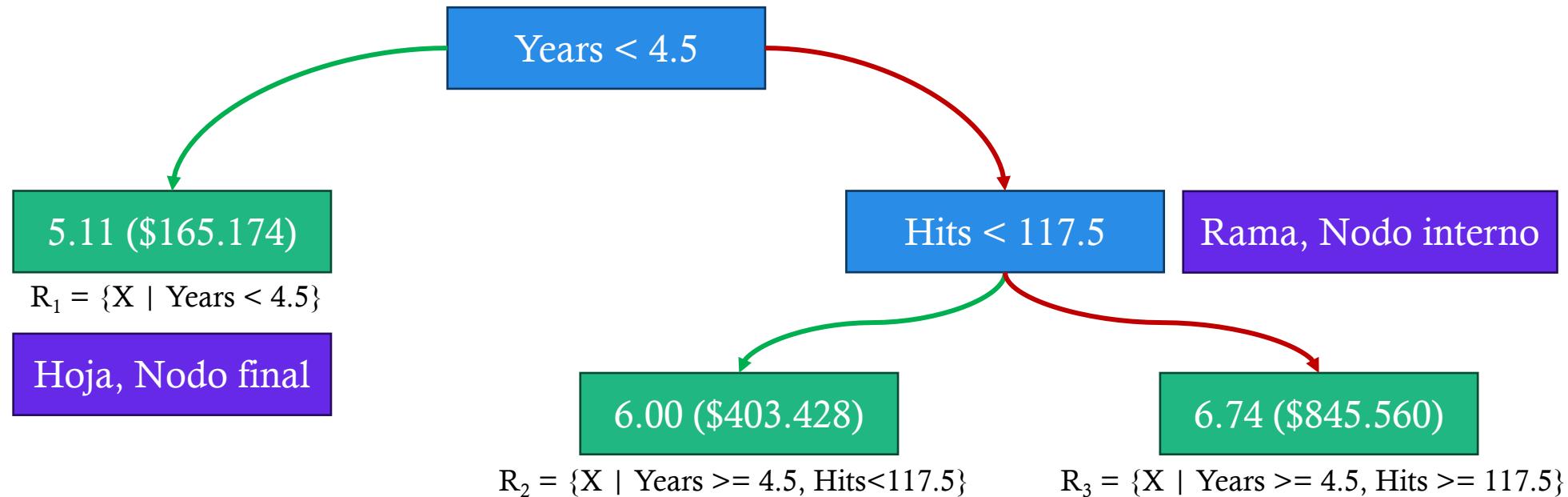
Los árboles de clasificación y regresión, conocidos como CART (Classification and Regression Trees), son una poderosa técnica de aprendizaje automático que se utiliza ampliamente para resolver problemas tanto de clasificación como de regresión.

Los árboles CART son modelos de decisión que utilizan una estructura de árbol para realizar predicciones basadas en reglas **lógicas sencillas y fáciles de interpretar**.

Arboles de clasificación

Arboles de regresión

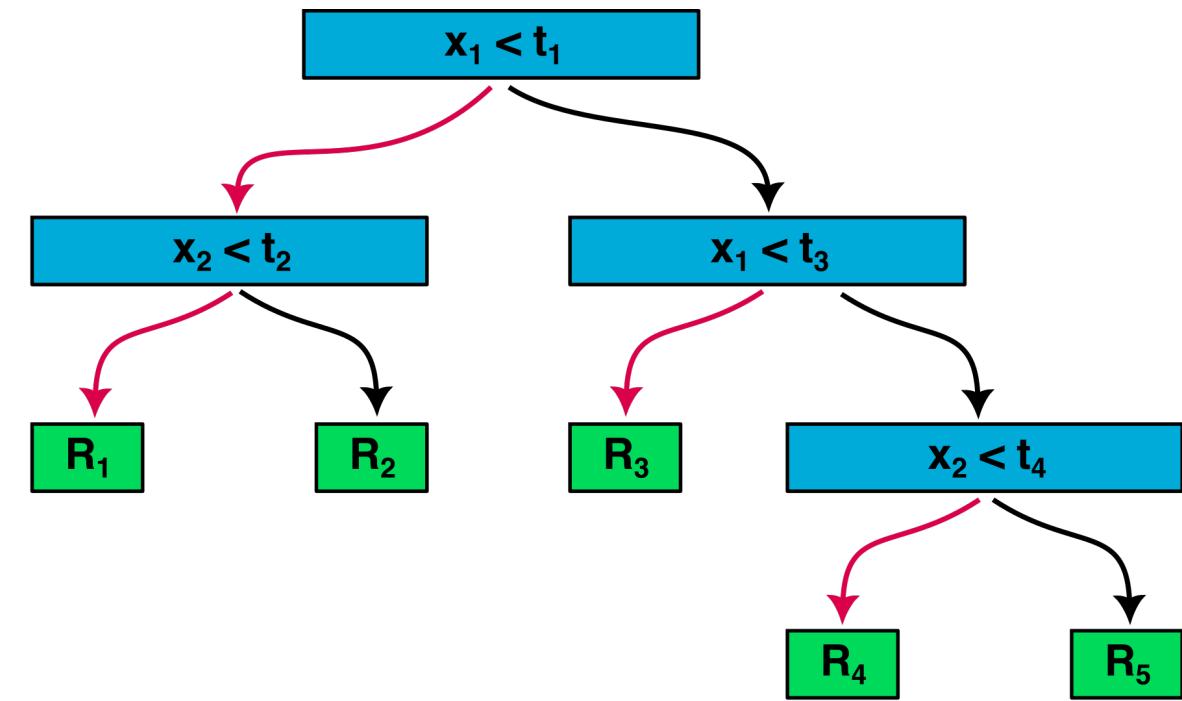
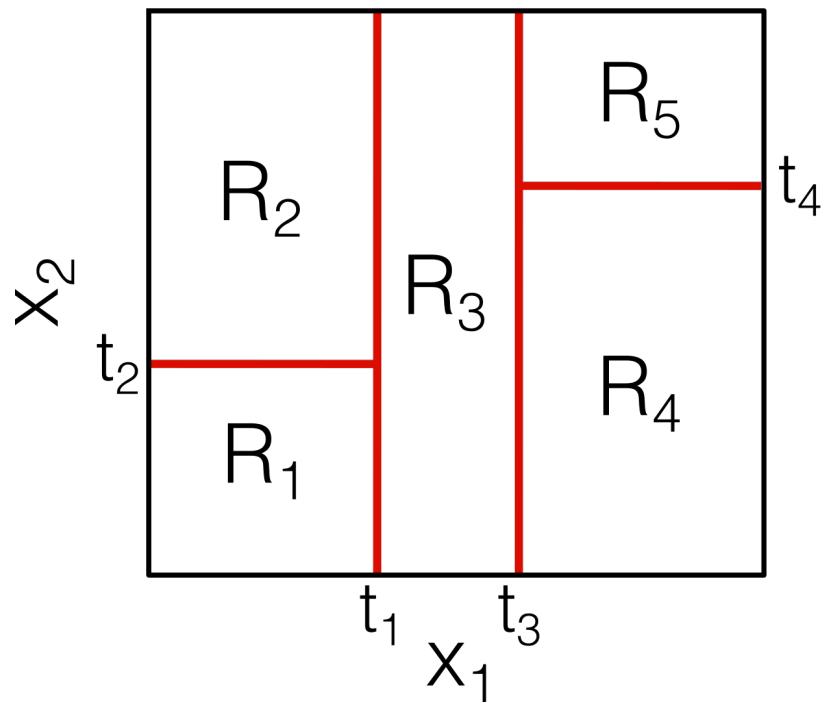
# ÁRBOLES DE REGRESIÓN



# ÁRBOLES DE REGRESIÓN

Recursive binary splitting

$$RSS = \sum_{j=1}^J \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$



---

# ÁRBOL DE CLASIFICACIÓN

Un árbol de clasificación es muy similar a uno de regresión, pero ahora se usa para predecir una **variable cualitativa**.

En el caso de regresión, al llegar la hoja, obteníamos el valor con el promedio de los valores en la hoja. Ahora, obtenemos la clase en base a la clase que más ocurre en las muestras que están en la hoja.

Al interpretar los resultados de un **árbol de clasificación**, a menudo estamos interesados no sólo en la predicción de clase correspondiente a una región de nodo terminal particular, sino también en las **proporciones de clase entre las observaciones de entrenamiento** que caen en esa región.

---

---

# BOSQUES ALEATORIOS

Es común que los árboles sobre-ajusten, dado que tan exacto tienden a adaptarse a los datos de entrenamiento.

Una forma de evitar esto es mediante **bosques aleatorios**, en el cual se construyen múltiples arboles de decisión y combinar sus salidas.

Si son de **clasificación**, estos árboles votan la clase.

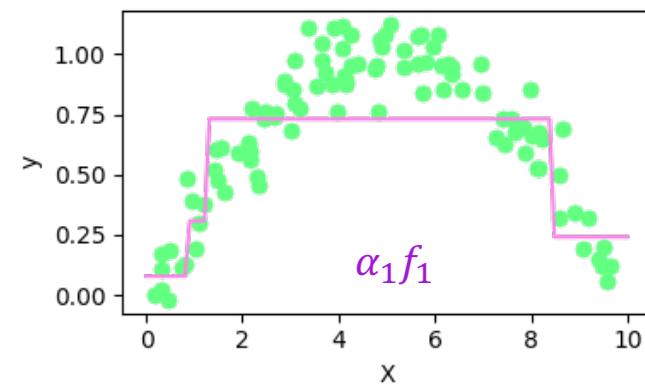
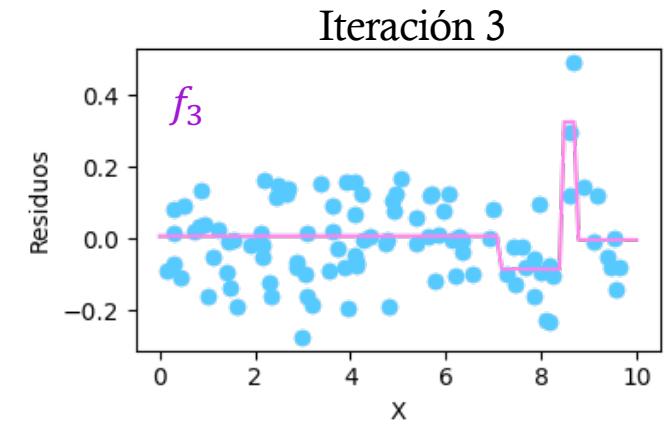
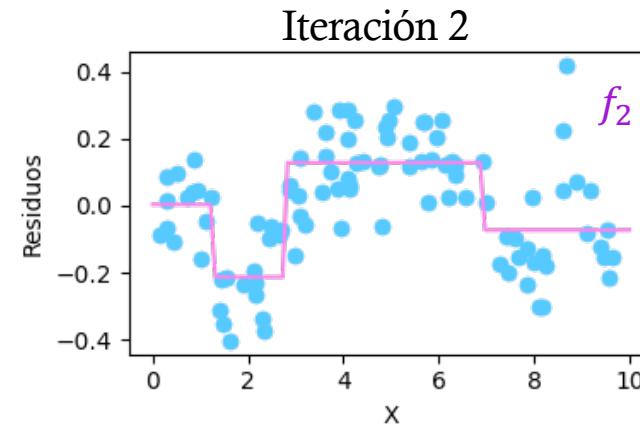
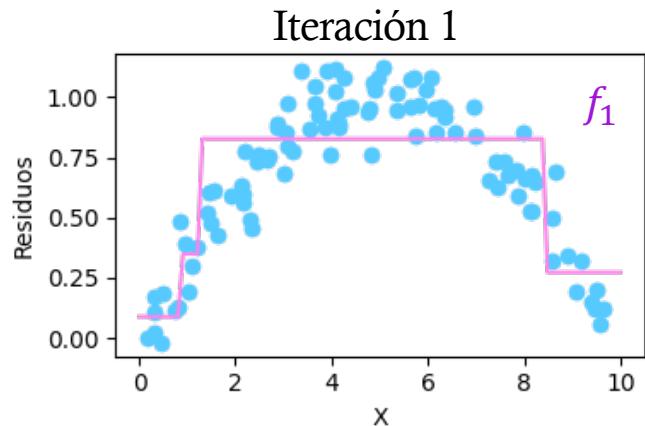
Si son de **regresión**, se promedia las predicciones.

---

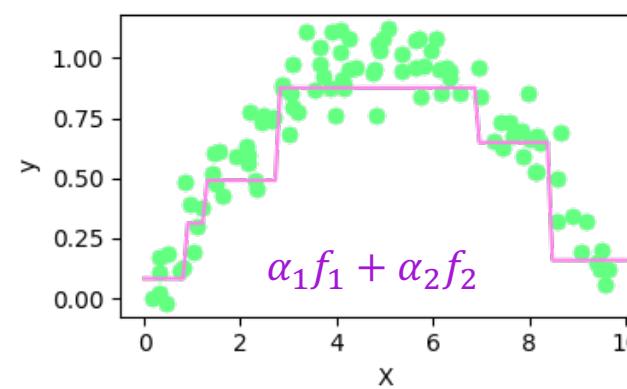
# BOSQUES ALEATORIOS

- **Bagging:** En el caso de los árboles, en vez de entrenar a los árboles con todas las observaciones del set de entrenamiento, se arma un nuevo set para cada árbol, usando **bootstrapping**.
- **Bosques aleatorios:** Los bosques aleatorios son una mejora de los obtenidos mediante **bagging**. Los bosques aleatorios hacen lo mismo que **bagging**, pero además se usa una cantidad aleatoria de atributos. El valor de cuantos atributos a usar se elige aproximadamente la raíz cuadrada de la cantidad de atributos totales.
- **Boosting:** La idea es similar a la de **Bagging**, pero en vez de construir arboles aleatoriamente dado por el proceso de **bootstrapping**, los nuevos árboles que se construyen usando la información de árboles anteriores

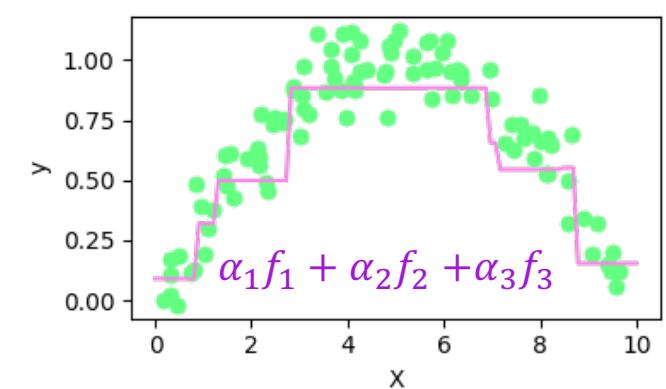
# BOSQUES ALEATORIOS



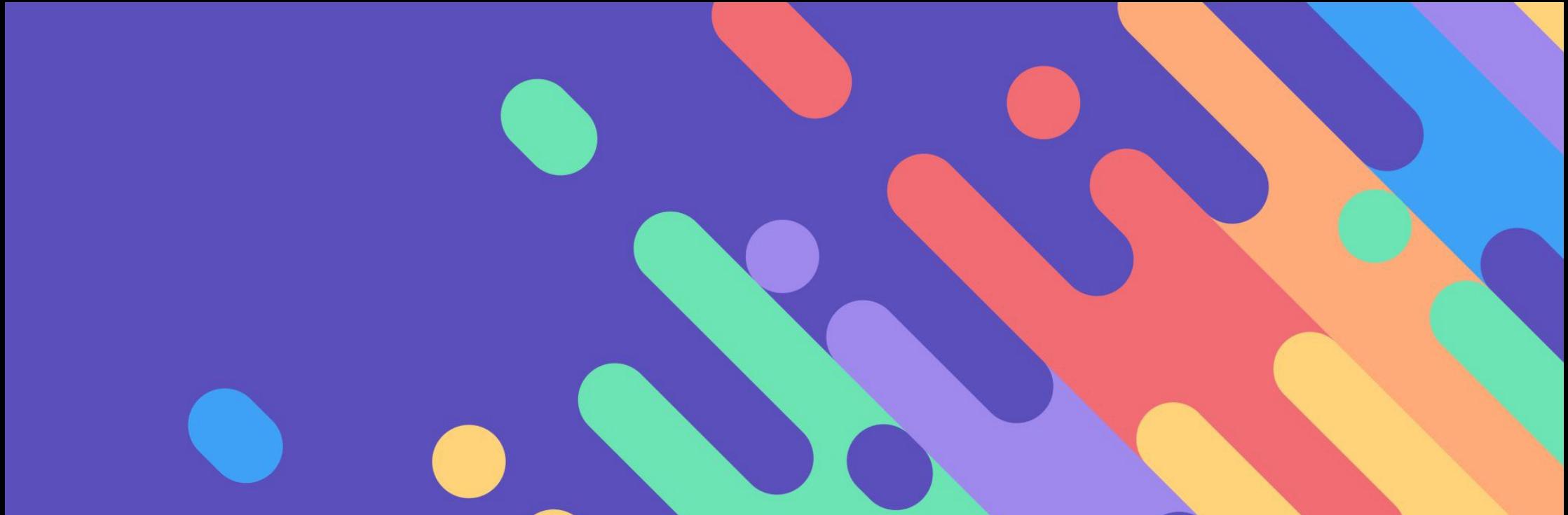
Salida de ensamble:  $F = \alpha_1 f_1$



Salida de ensamble:  $F = \alpha_1 f_1 + \alpha_2 f_2$



Salida de ensamble:  $F = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3$



---

# REDES NEURONALES

---

# REDES NEURONALES

Las redes neuronales originalmente se plantean como algoritmos matemáticos que tratan de imitar los cálculos complejos que tienen lugar en el cerebro.

Se busca imitar no solo la gran cantidad de unidades de procesamiento (neuronas) sino la interconexión entre ellas (sinapsis).

Esto genera dos grandes campos de aplicación, uno el de la neurociencia computacional y por otro el de **Deep Learning**. Siendo este último el gran popular de hoy en día con avances que vemos constantemente en el público general.

---

# REDES NEURONALES

Un poquito de historia...

Las redes neuronales podemos dar comienzo con la neurona de McCulloch-Pitts en 1943. Este modelo es la primera formulación del proceso de cálculo que lleva a cabo una neurona, basado en una formulación algebraica. La unidad actual usada de Deep Learning no es muy diferente a este modelo.

En 1952, Hodgkin y Huxley crean el modelo basado en conductancia de la neurona, que modela como los potenciales de acción de las neuronas se generan y se propagan. Es un modelo de ecuaciones diferenciales. Los resultados fueron tan importantes que se llevaron el premio Nobel de medicina en 1963.

Este modelo que describen como funciona la neurona siguieron camino al desarrollo de la **neurociencia computacional**.

---

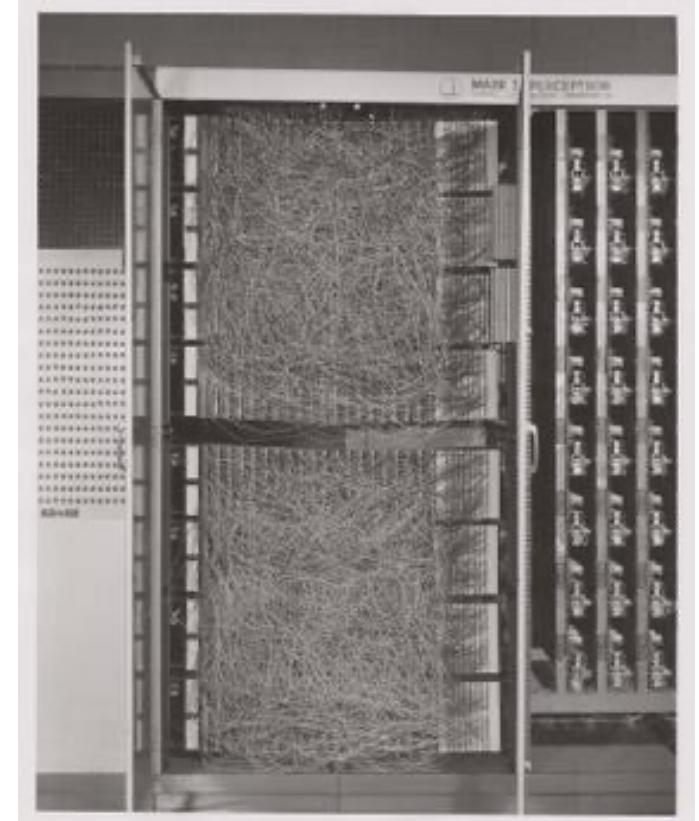
# REDES NEURONALES

Lo importante es que, en este estado embrionario del área, es que las investigaciones estaban en su cúspide.

En 1958, Rosenblatt realizó la primera implementación del perceptrón (basado en la neurona de McCulloch-Pitts).

Rosenblatt es el padre del **Deep Learning**.

Es quien perfeccionó el perceptrón moderno, y las redes de dos o tres capas.



---

# REDES NEURONALES

Pero en 1969 llegó el primer invierno en redes neuronales. En 1969 se publicó un libro llamado Perceptrons de Minsky y Papert que enfatizaba los límites de lo que los perceptrones podían hacer.

Este libro y su popularidad mató toda financiación en investigaciones hasta 1980. Entre muchas críticas, se planteó que el perceptrón como modelo neuronal no podía resolver la función lógica XOR, algo que una neurona biológica si podría...

---

# REDES NEURONALES

Pero en 1969 llegó el primer invierno en redes neuronales. En 1969 se publicó un libro llamado Perceptrons de Minsky y Papert que enfatizaba los límites de lo que los perceptrones podían hacer.

Este libro y su popularidad mató toda financiación en investigaciones hasta 1980.

Entre muchas críticas, se planteó que el perceptrón como modelo neuronal no podía resolver la función lógica XOR, algo que una neurona biológica si podría...

*Luego en experimentos fisiológicos se probó que la neurona biológica tampoco puede resolver la función lógica XOR, sino que necesita de una red.*

---

# REDES NEURONALES

Pero en 1969 llegó el primer invierno en redes neuronales. En 1969 se publicó un libro llamado Perceptrons de Minsky y Papert que enfatizaba los límites de lo que los perceptrones podían hacer.

Este libro y su popularidad mató toda financiación en investigaciones hasta 1980.

Entre muchas críticas, se planteó que el perceptrón como modelo neuronal no podía resolver la función lógica XOR, algo que una neurona biológica si podría...

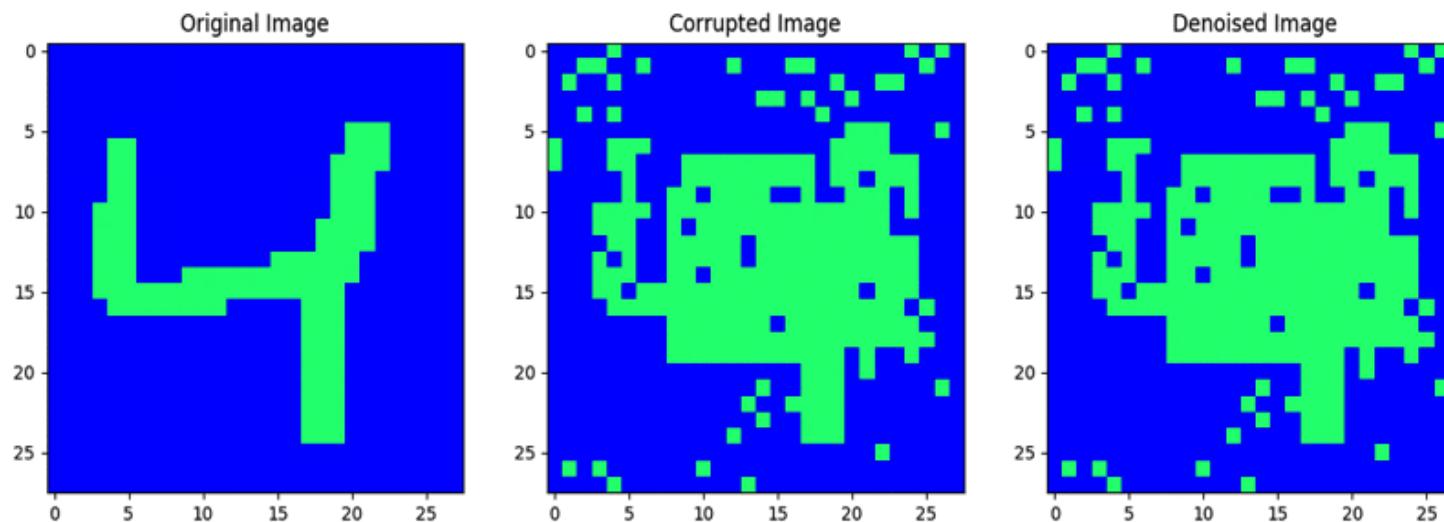
*Luego en experimentos fisiológicos se probó que la neurona biológica tampoco puede resolver la función lógica XOR, sino que necesita de una red.*

*The plot thickens (la trama se complica)... En 2020 se encontró que neuronas del cerebro humano si pueden.*

---

# REDES NEURONALES

El invierno termina en 1980-90 con los desarrollos de Rumelhart, Hopfield, entre otros. Aquí se desarrollaron el algoritmo de **Back-propagation**, redes de memoria y el concepto de propiedades emergentes.



Obtenido de <https://github.com/TarinZ/hopfield-nets>

---

# REDES NEURONALES

El invierno termina en 1980-90 con los desarrollos de Rumelhart, Hopfield, entre otros. Aquí se desarrollaron el algoritmo de **Back-propagation**, redes de memoria y el concepto de propiedades emergentes.

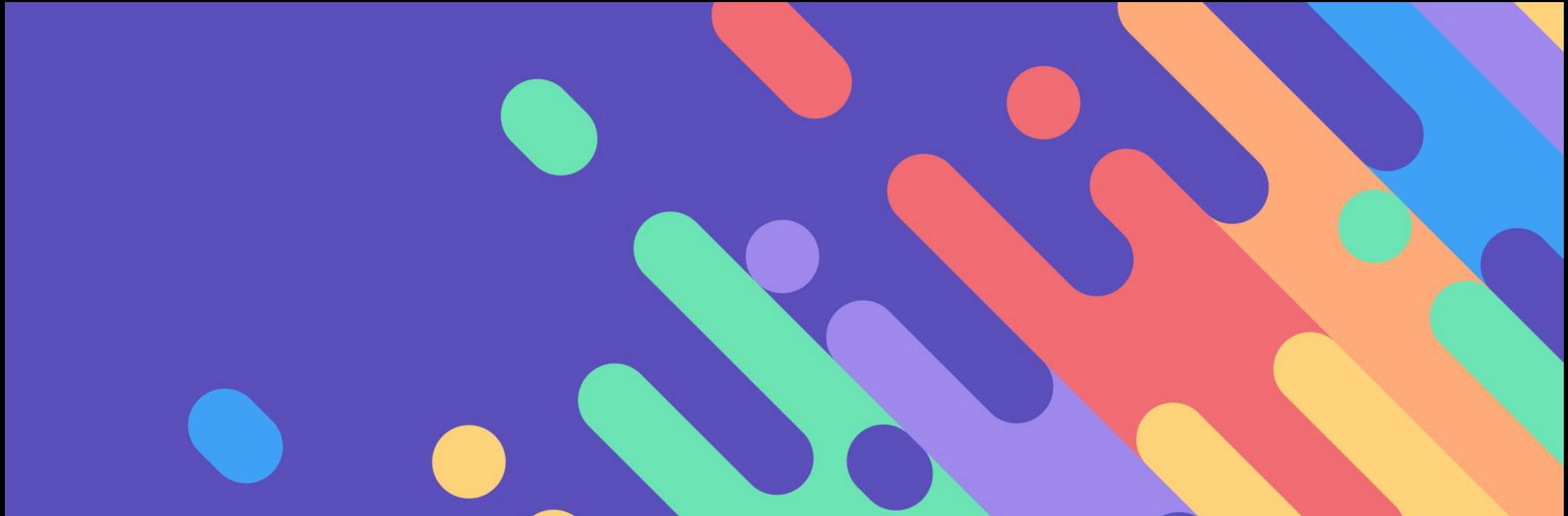
Luego llegó **un nuevo invierno** de redes neuronales con la llegada de algoritmos mucho más sencillos de usar y sin tanta dificultad de retoques de hiperparámetros, tales como SVM y los bosques aleatorios. Estos algoritmos eran más fáciles de usar y rendían mucho mejor que las redes neuronales.

---

# REDES NEURONALES

En 2010, volvió a la vida, y ahora, oficialmente se le empezó a llamar **Deep Learning**. Ahora se incorporaron nuevas arquitecturas de redes y características adicionales. Además, ahora estos algoritmos empezaron a superar a los demás en tareas de clasificación de imágenes y videos, y el modelado de voz y texto.

Parte del éxito es la posibilidad de contar con datasets más grandes (Big data) y mejores procesamientos de cálculo, además de la llegada de Google, Meta, Microsoft.

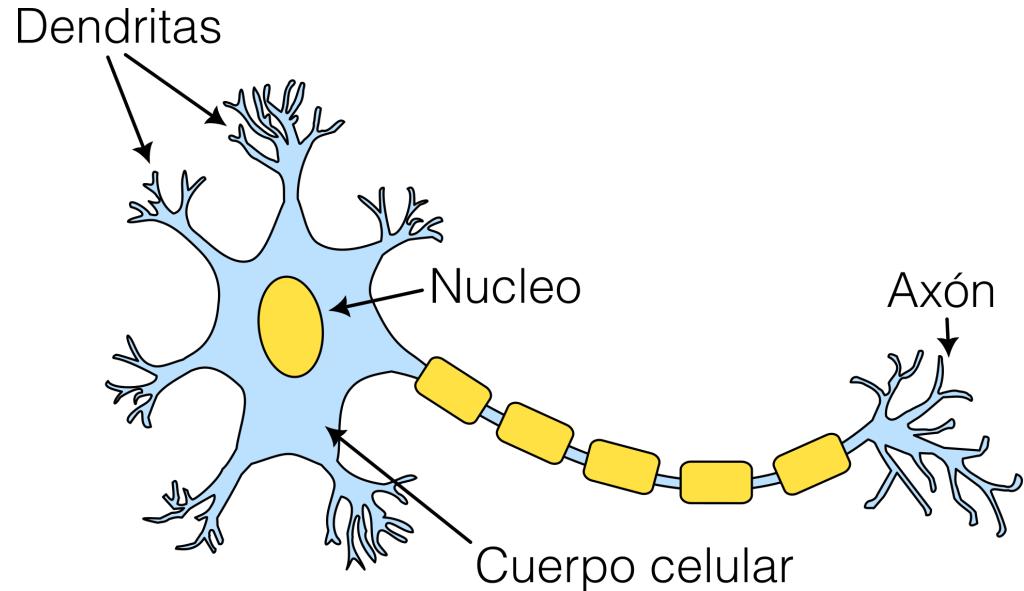


---

# PERCEPTRÓN Y NEURONAS SIGMOIDEAS

# PERCEPTRÓN

Empecemos con una neurona biológica:

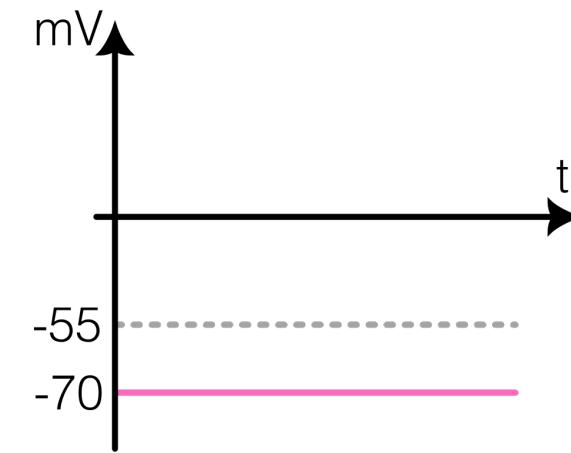
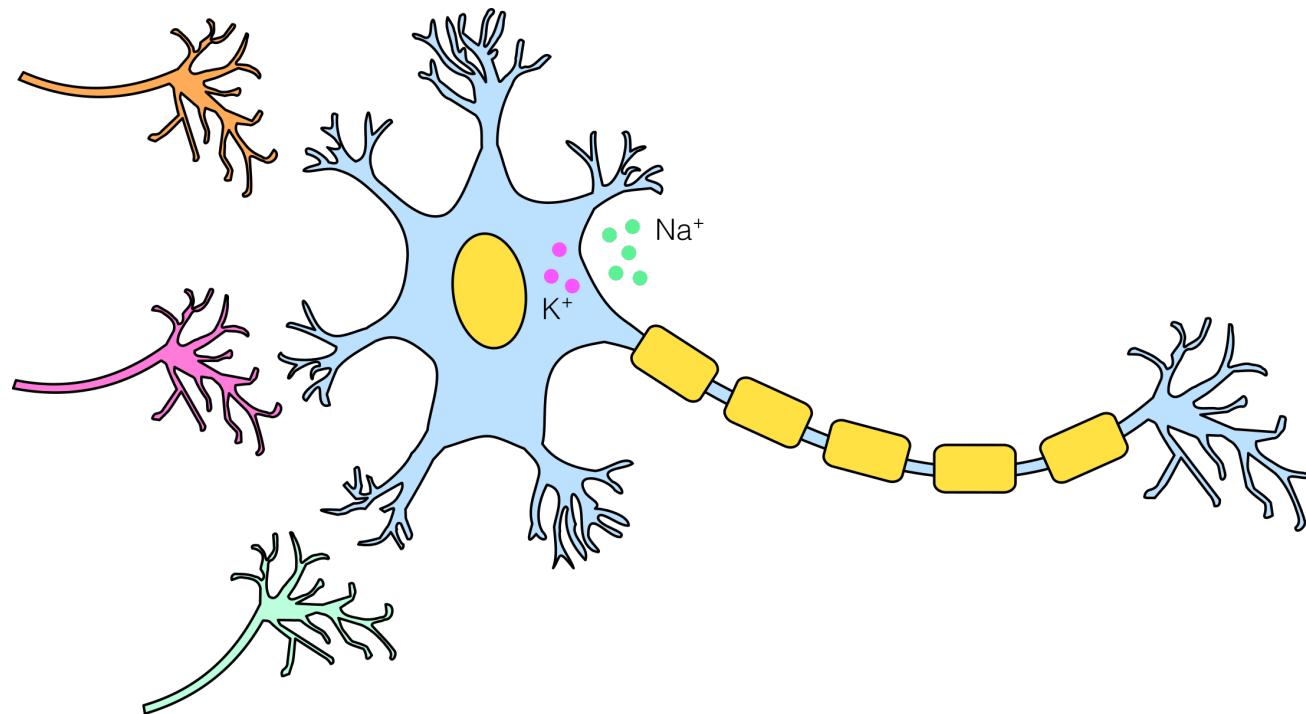


Tenemos:

- **Cuerpo celular o soma:** Donde está el núcleo y las organelas de la célula.
- **Dendritas:** Ramificaciones del soma, es donde la neurona recibe las sinapsis de otras neuronas.
- **Axón:** Es la prolongación encargada de transmitir el impulso nervioso. Es como se comunica la neurona.

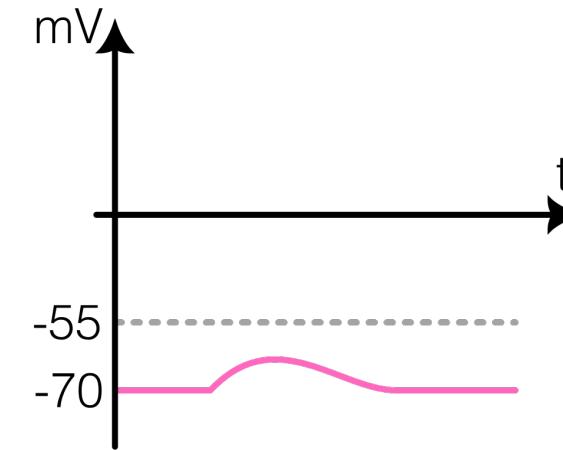
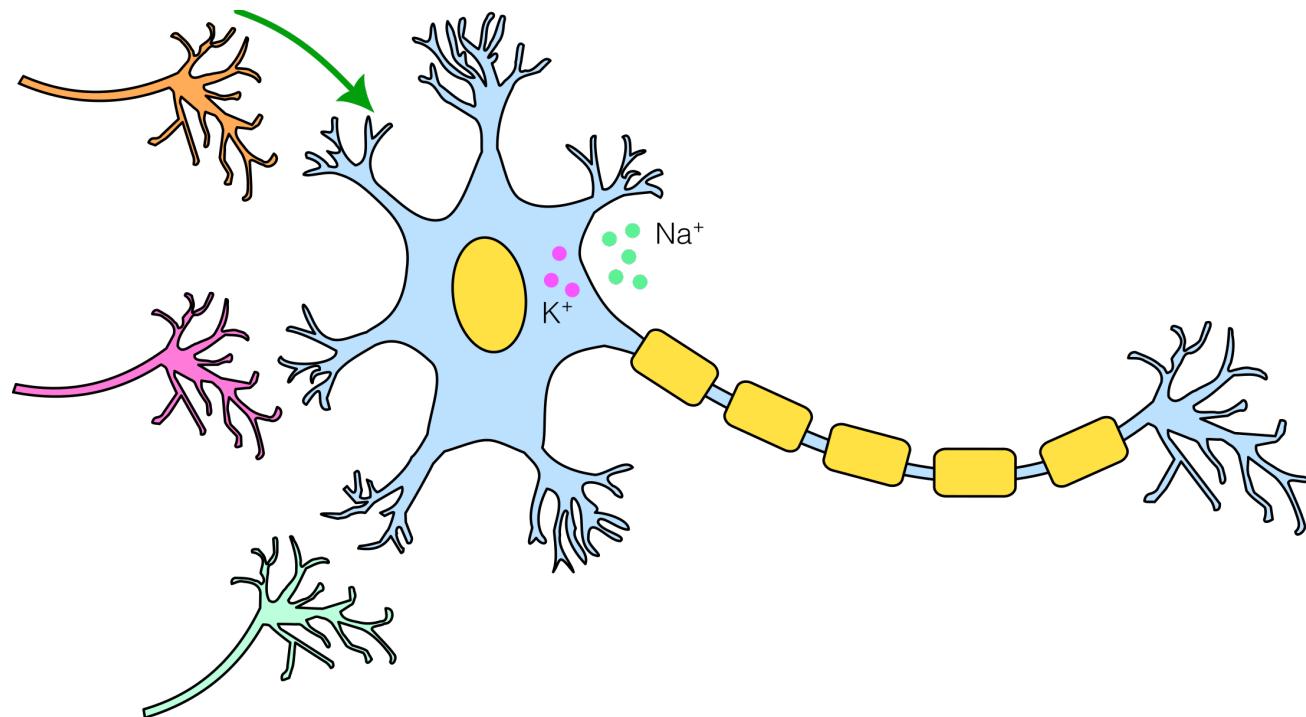
# PERCEPTRÓN

En estado de reposo, tenemos...



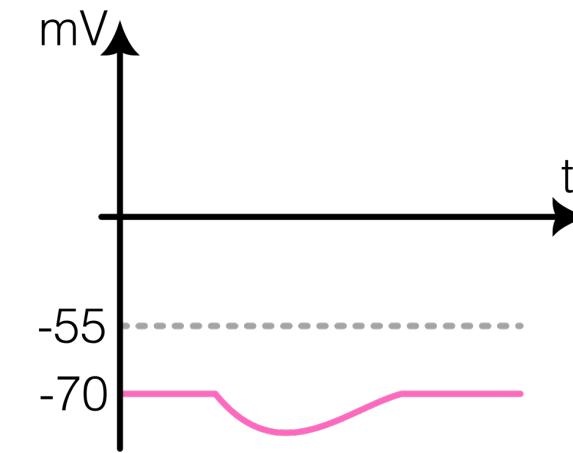
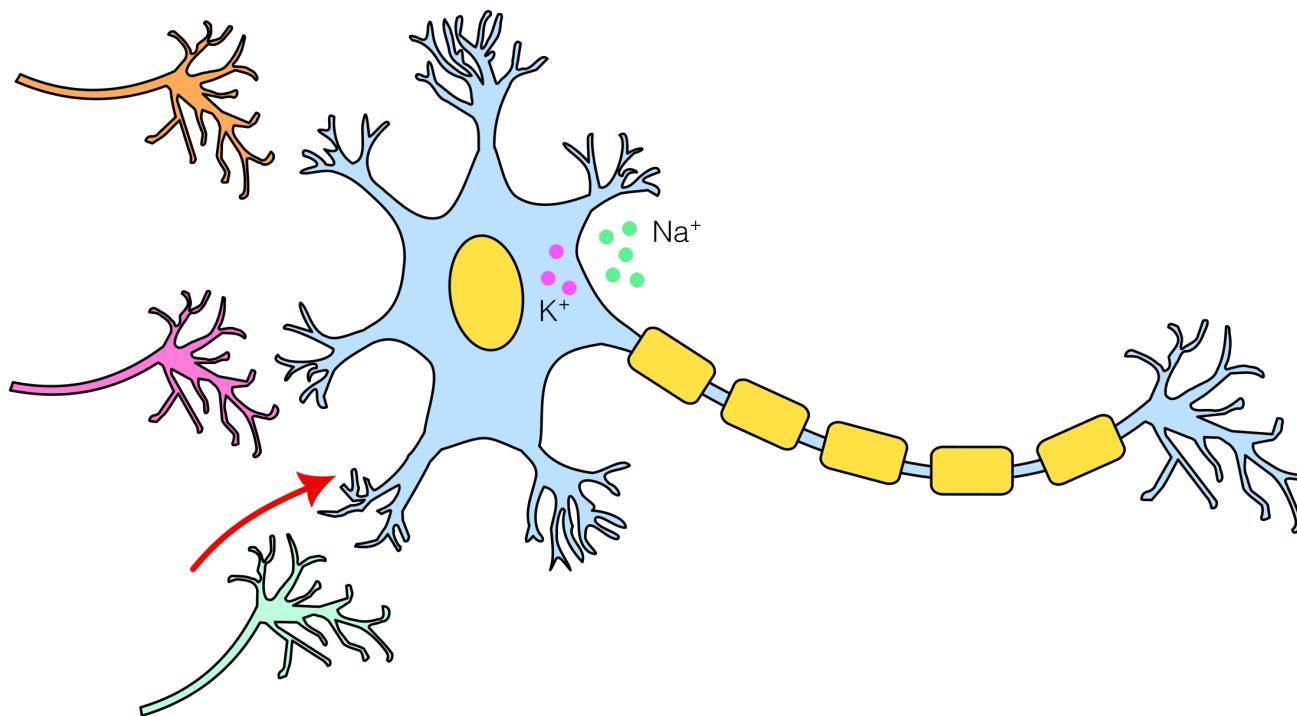
# PERCEPTRÓN

Si otra neurona excita, pero poco, aumenta le voltaje, pero rápidamente vuelve a su estado.



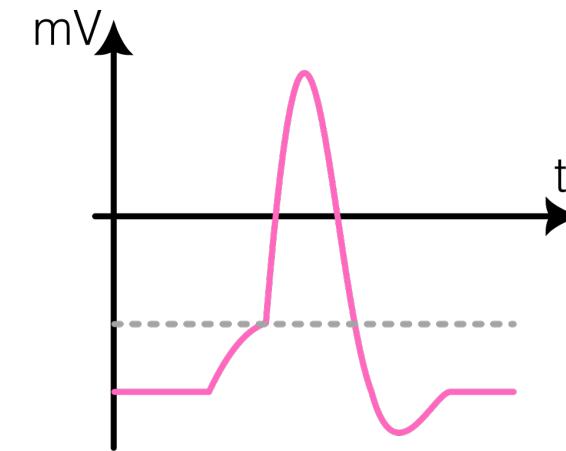
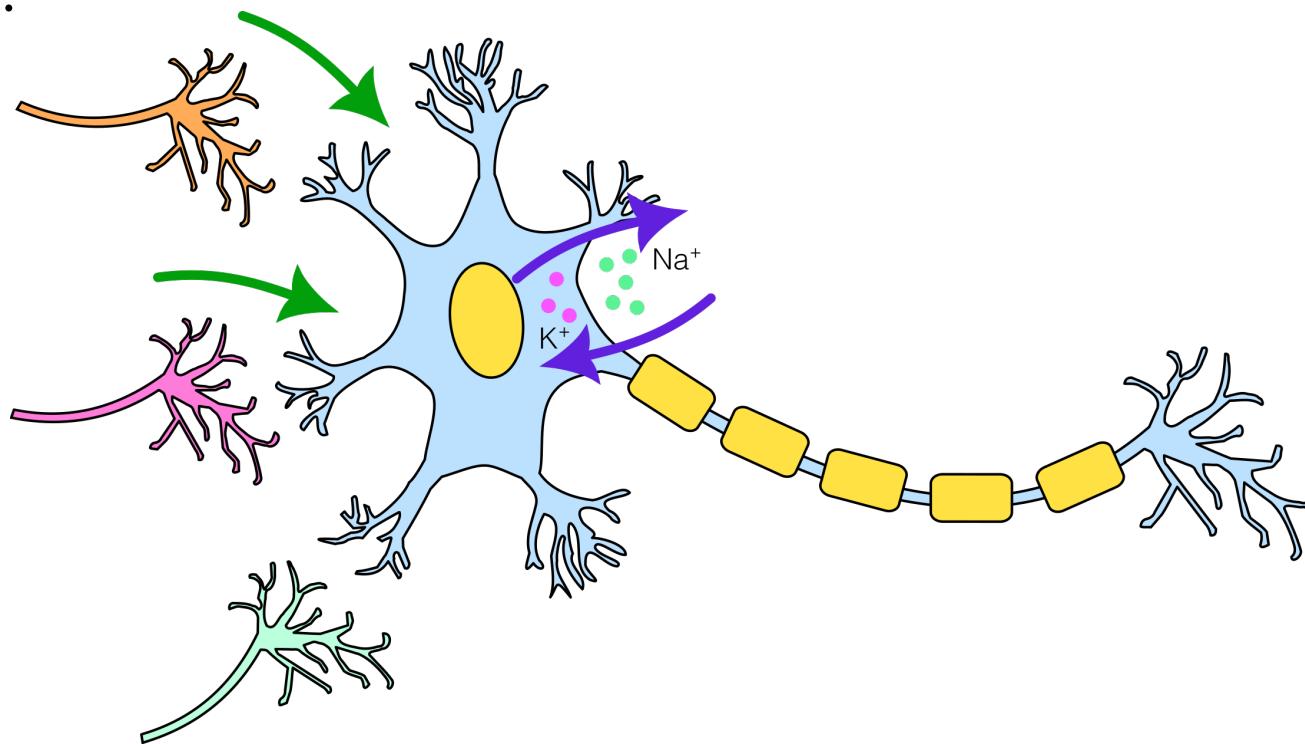
# PERCEPTRÓN

De similar forma, una sinapsis puede ser inhibitoria



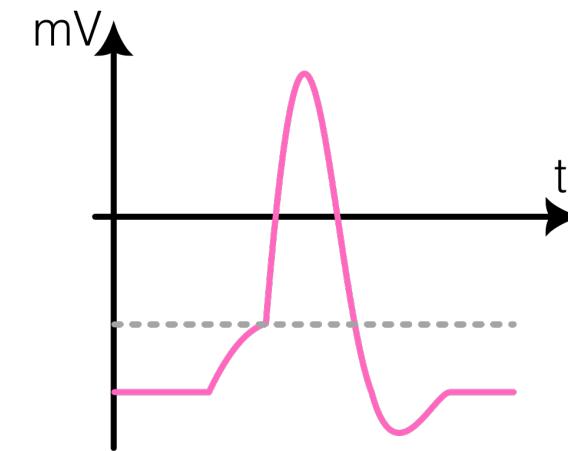
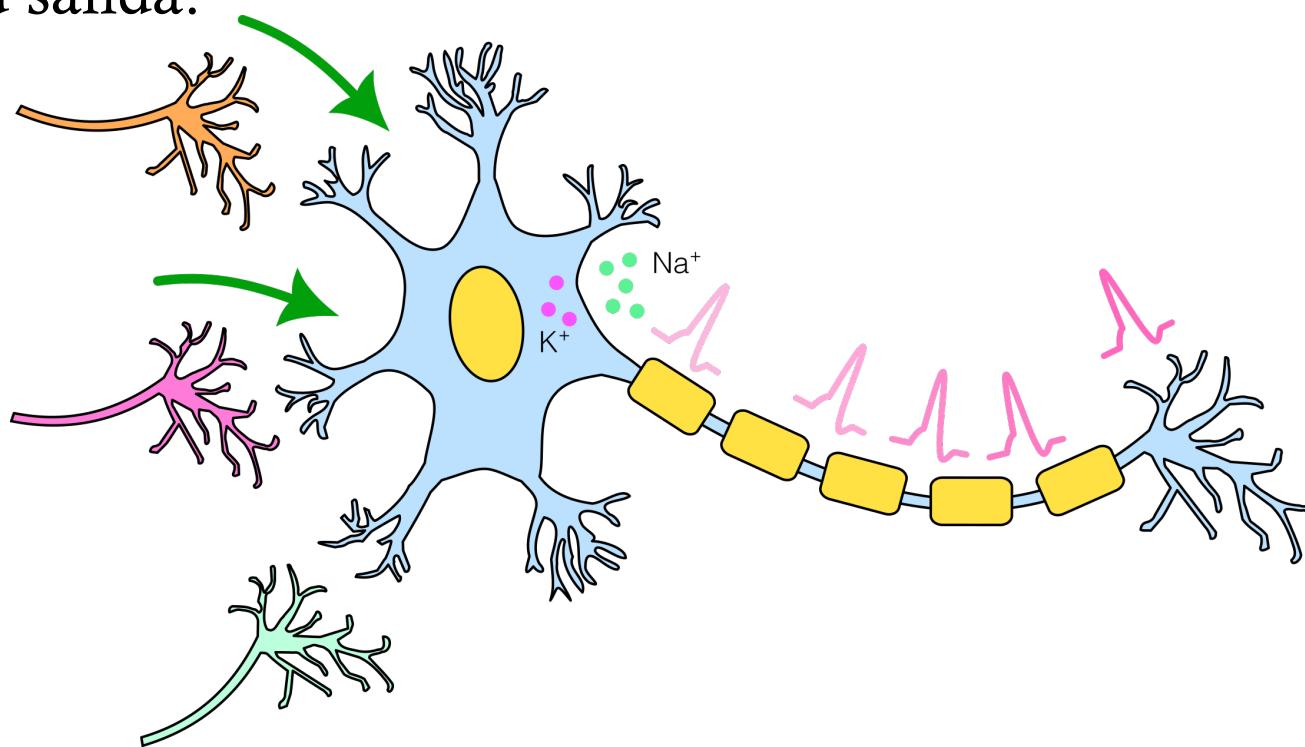
# PERCEPTRÓN

Ahora, si la excitación supera el umbral, se genera un impulso dado un efecto todo o nada.



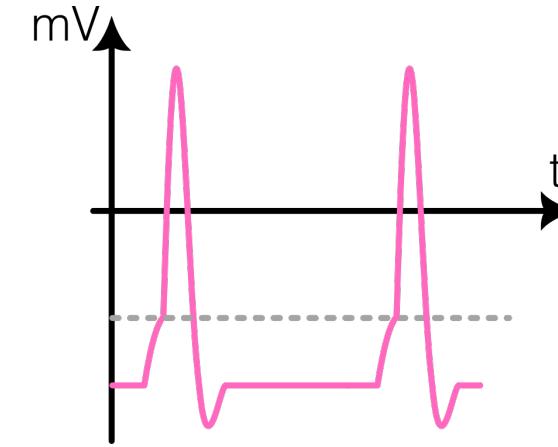
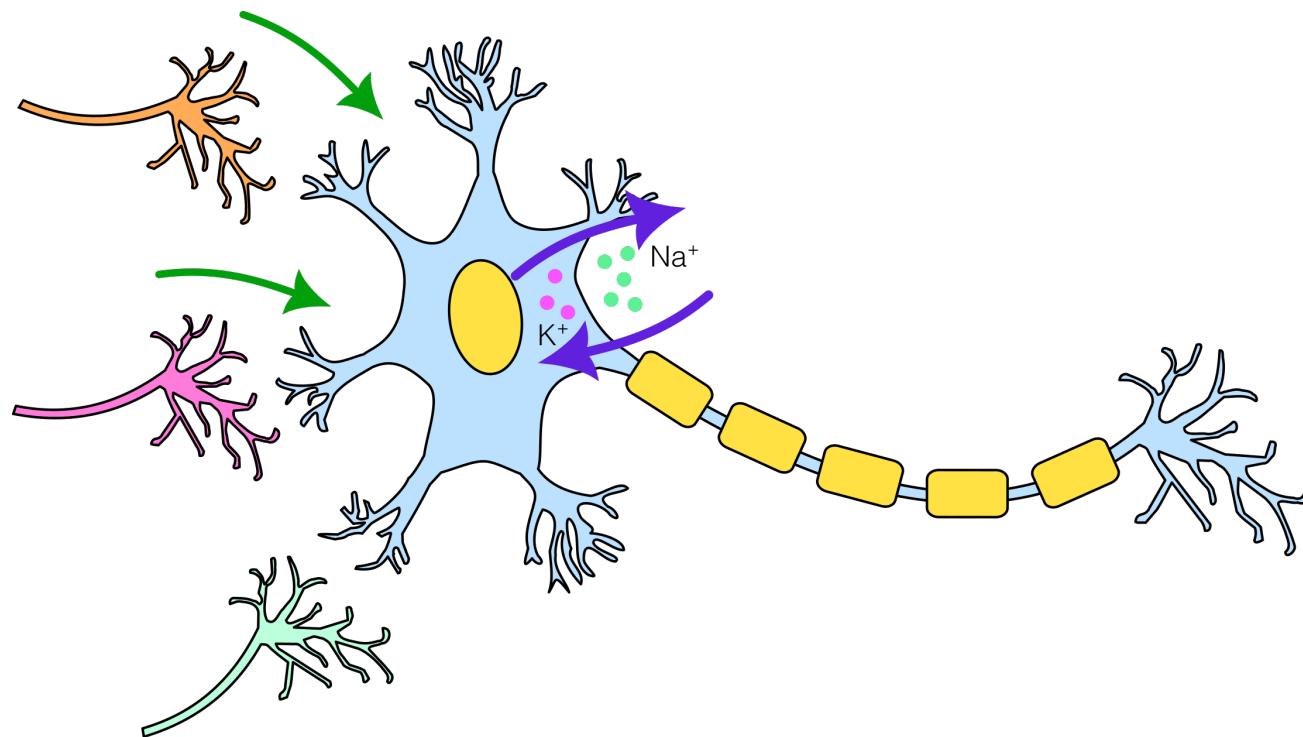
# PERCEPTRÓN

Y este impulso se propaga del cuerpo hasta el terminal axónico, el cual la neurona da su salida.



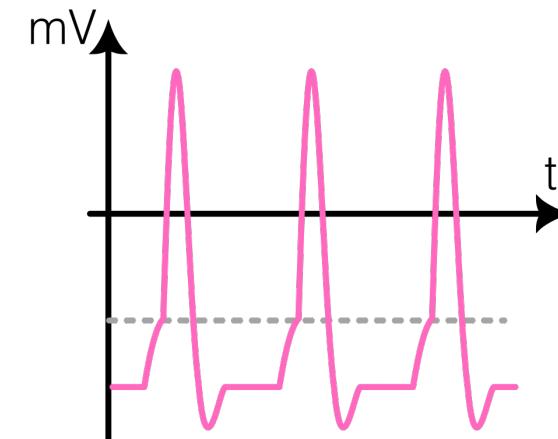
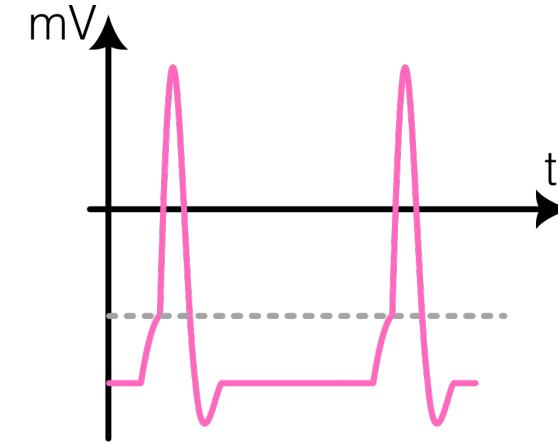
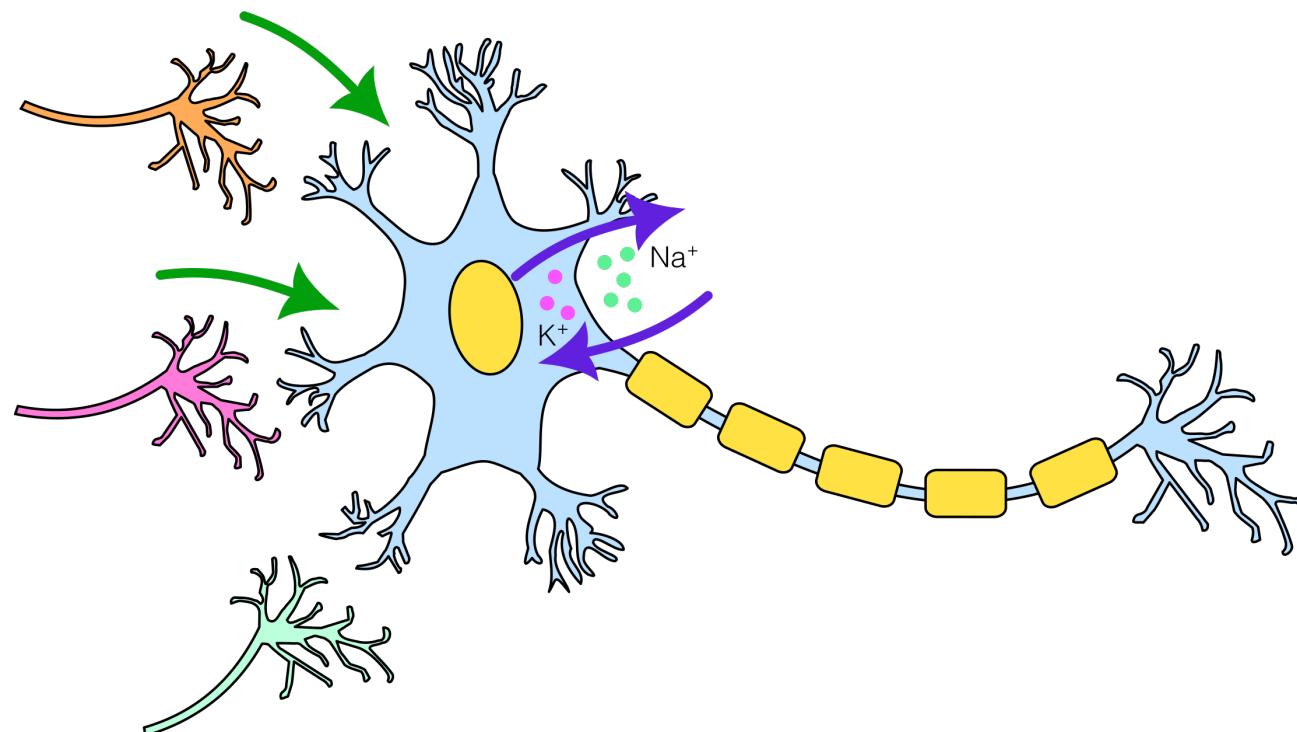
# PERCEPTRÓN

Niveles de excitación nos dan diferentes tasas de disparo...



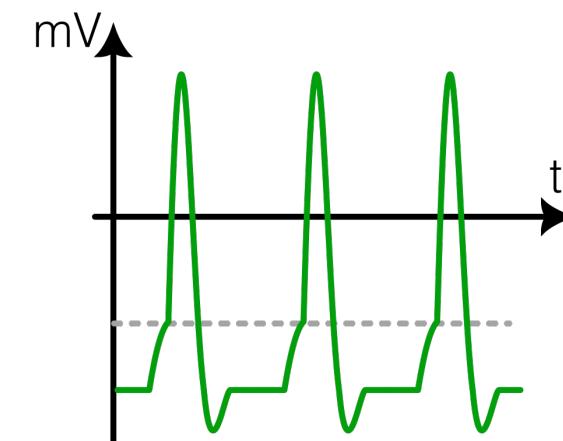
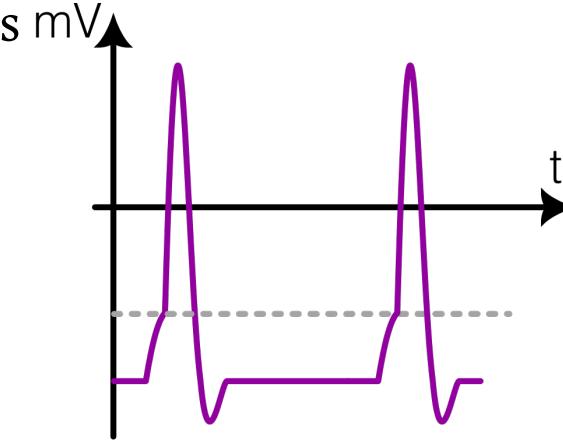
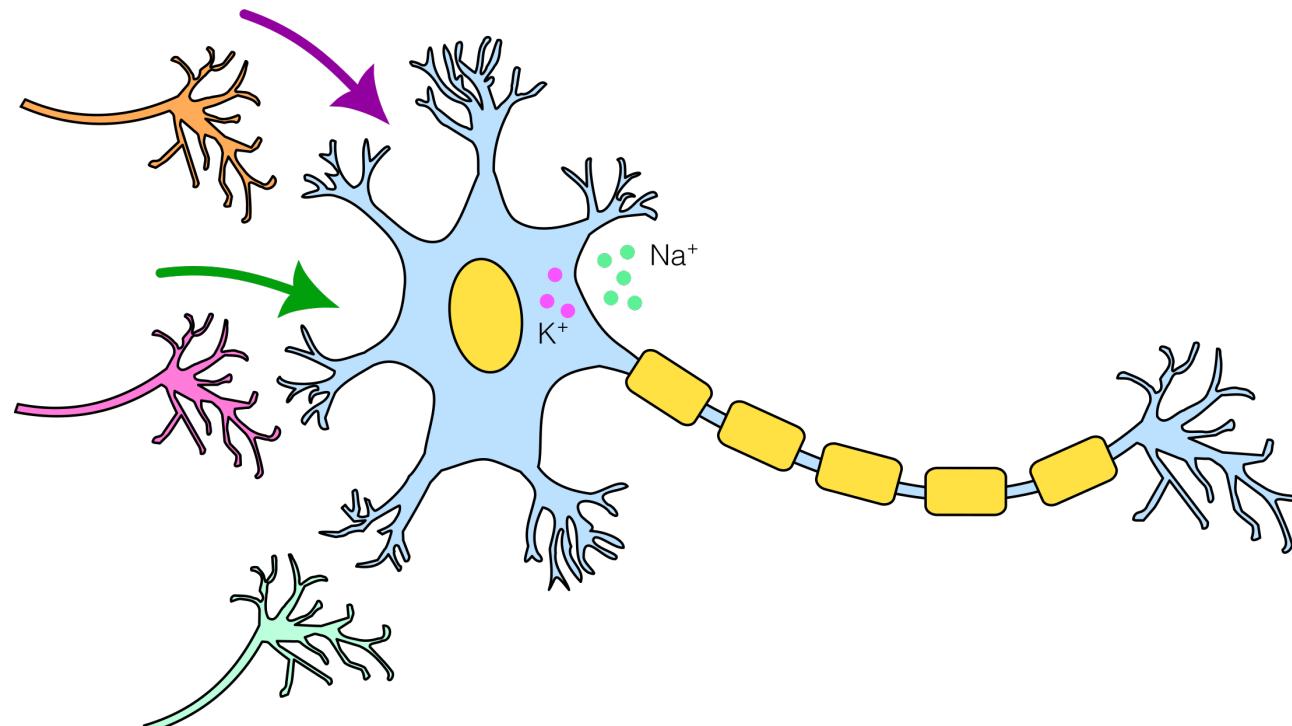
# PERCEPTRÓN

Niveles de excitación nos dan diferentes tasas de disparo...



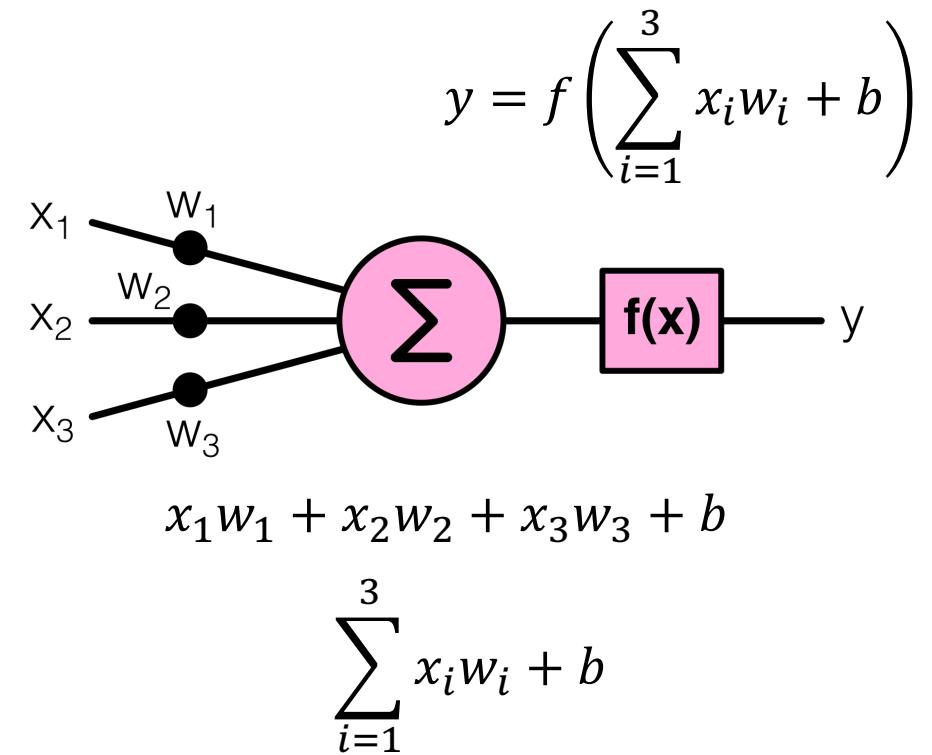
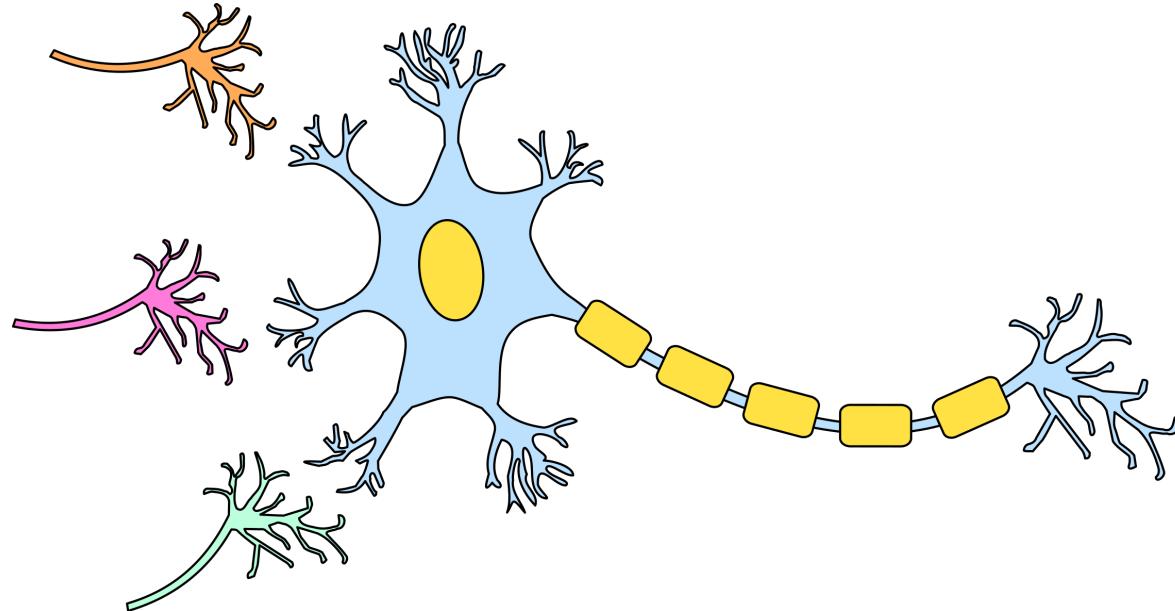
# PERCEPTRÓN

Y ante una misma excitación, hay sinapsis más sensibles que otras mV



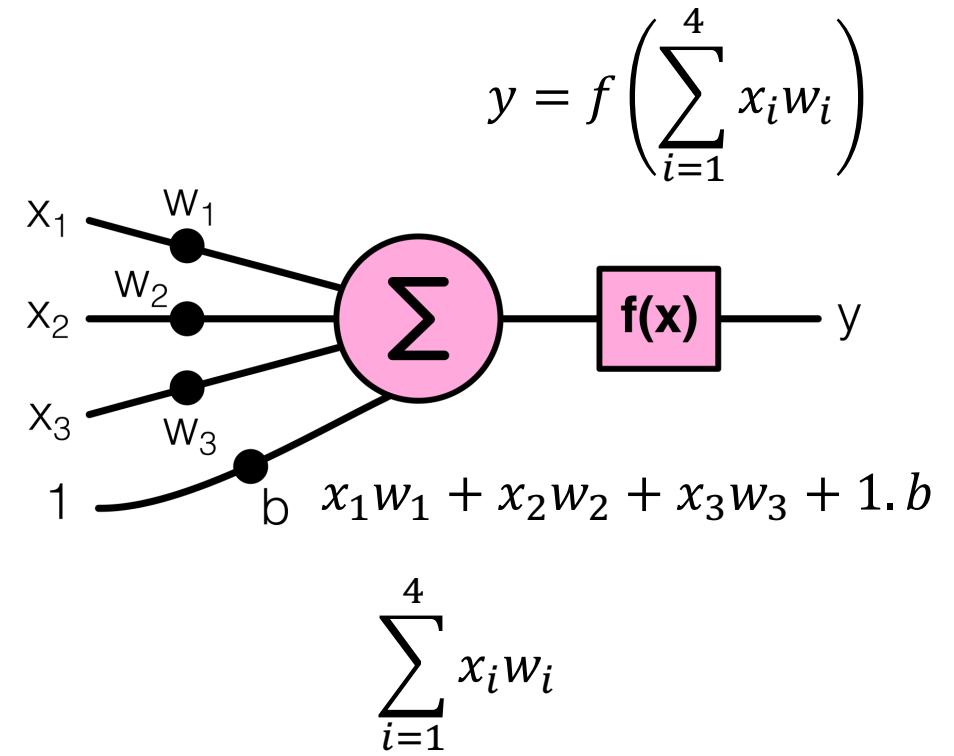
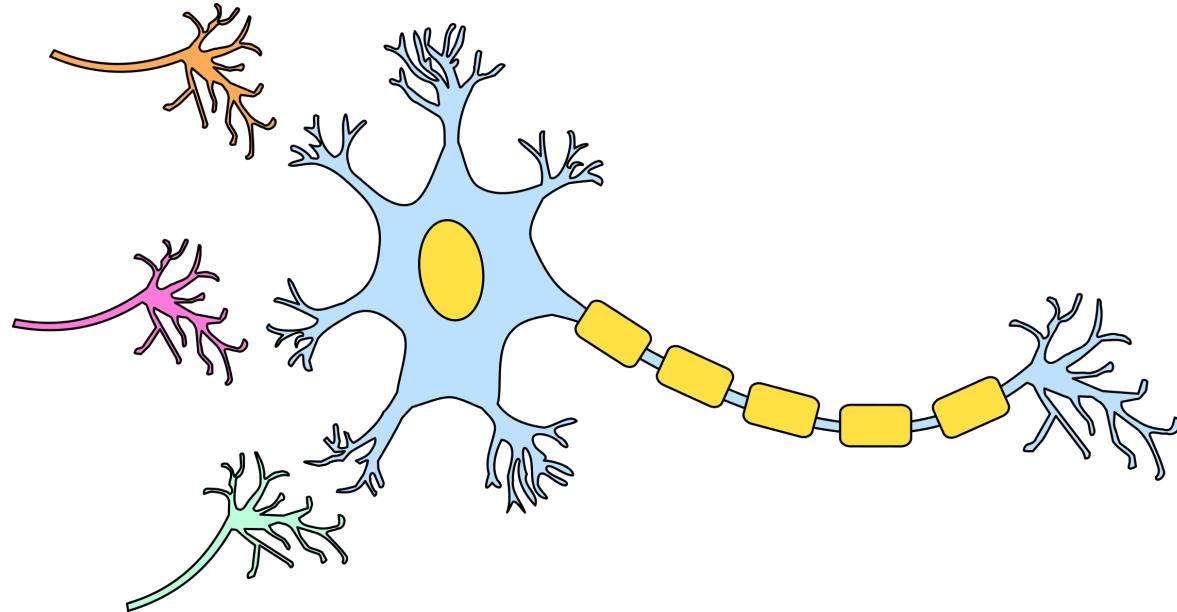
# PERCEPTRÓN

Con esto en mente, armemos el modelo de neurona



# PERCEPTRÓN

Con esto en mente, armemos el modelo de neurona

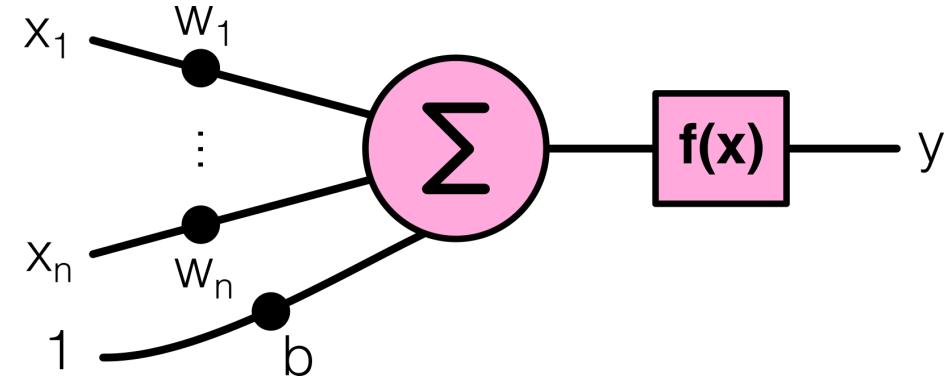


# PERCEPTRÓN

Si tenemos una observación de **n** atributos, la neurona tendrá **n** entradas con **n+1** pesos sinápticos.

Por lo que la parte lineal es:

$$\sum_{i=1}^n x_i w_i + b$$



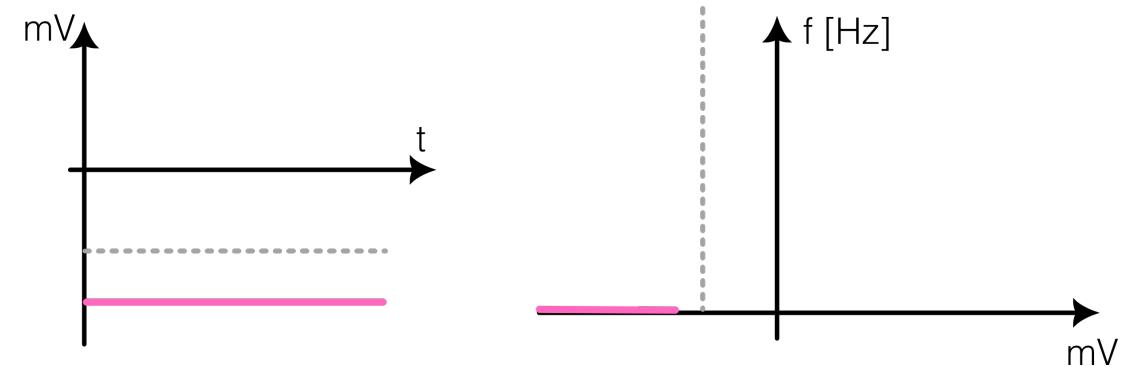
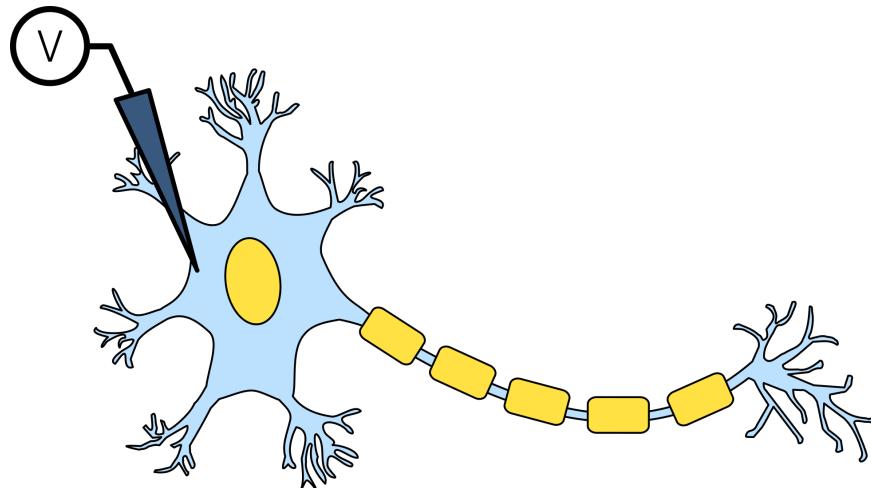
Donde  $w_i$ ,  $b$  son los pesos sinápticos, que representan si la conexión es excitatoria ( $w_i > 0$ ) o inhibitoria ( $w_i < 0$ ), o que no haya conexión sináptica ( $w_i = 0$ )

# PERCEPTRÓN

Veamos la función de activación. ¿Qué estamos modelando con ella?

En nuestro modelo, no nos interesa modelar el impulso nervioso, ni siquiera cuando ocurre, sino modelar la tasa de impulsos.

Hagamos un experimento: Pinchamos una neurona e inyectemos voltaje, y midamos la tasa de impulsos nerviosos versus el voltaje. Al principio, antes que llegue al valor umbral de disparo, la neurona no dispara y por lo tanto la tasa es 0 Hz.

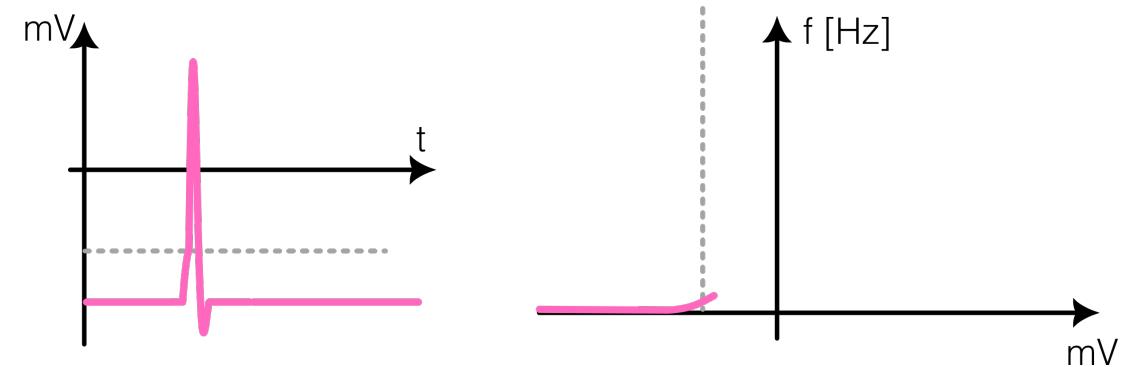
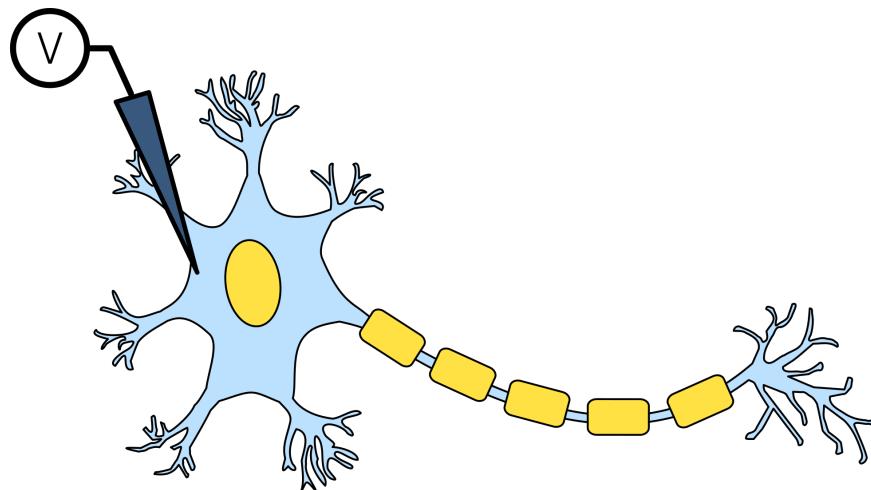


# PERCEPTRÓN

Veamos la función de activación. ¿Qué estamos modelando con ella?

En nuestro modelo, no nos interesa modelar el impulso nervioso, ni siquiera cuando ocurre, sino modelar la tasa de impulsos.

Hagamos un experimento: Pinchamos una neurona e inyectemos voltaje, y midamos la tasa de impulsos nerviosos versus el voltaje. Cuando aumentamos y pasamos el umbral, la neurona empieza a disparar.

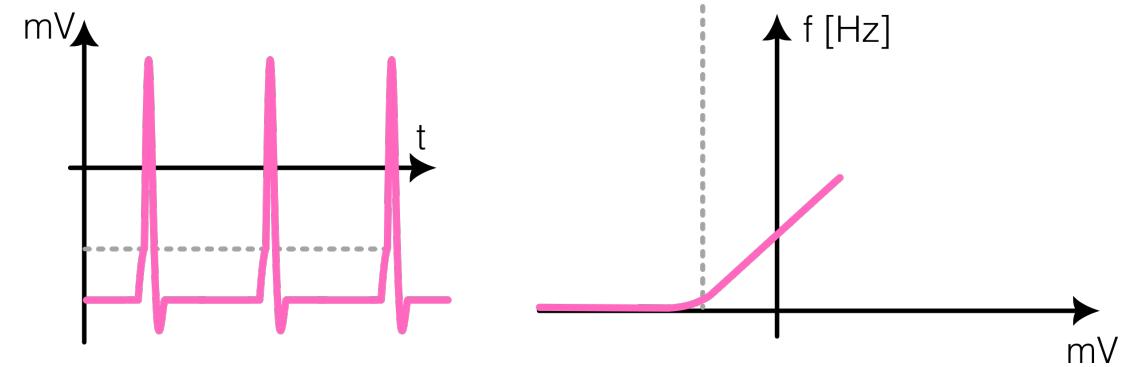
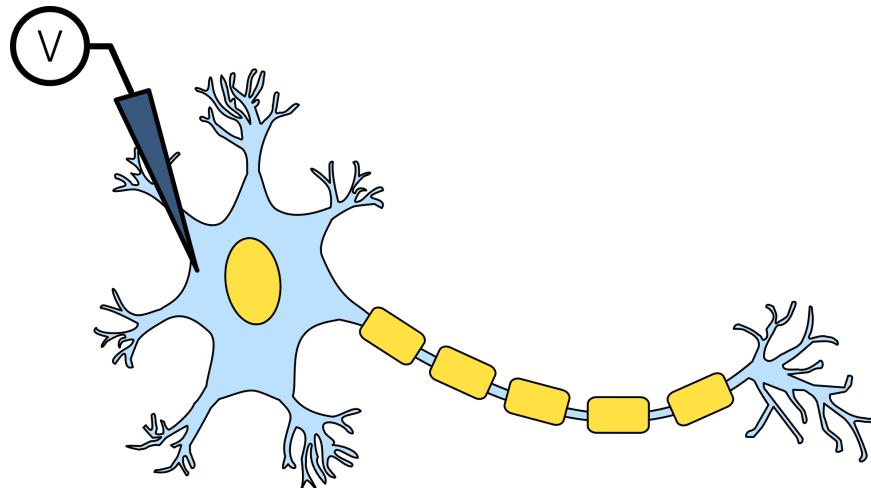


# PERCEPTRÓN

Veamos la función de activación. ¿Qué estamos modelando con ella?

En nuestro modelo, no nos interesa modela el impulso nervioso, ni siquiera cuando ocurre, sino modelar la tasa de impulsos.

Hagamos un experimento: Pinchamos una neurona e inyectemos voltaje, y midamos la tasa de impulsos nervioso versus el voltaje. Durante la etapa lineal, si duplicamos el voltaje, la neurona dispara al doble de la frecuencia.

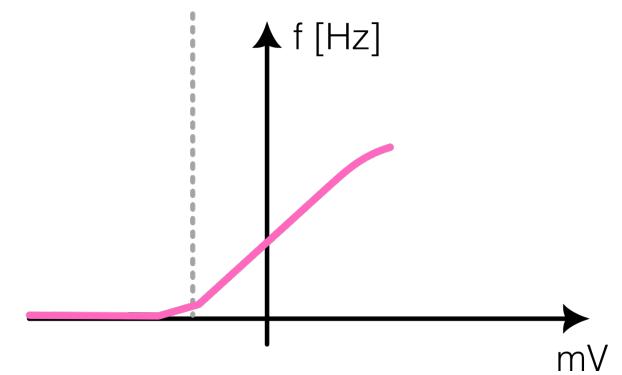
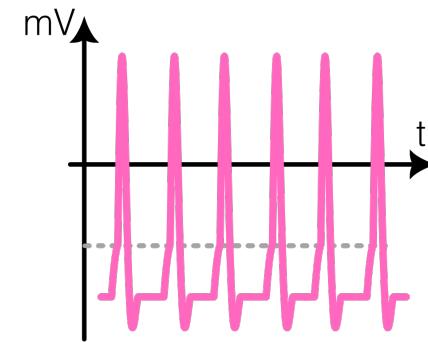
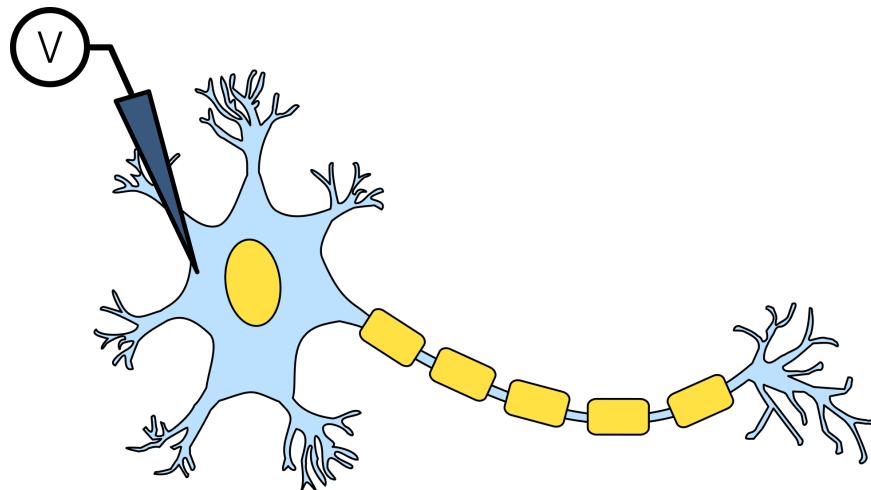


# PERCEPTRÓN

Veamos la función de activación. ¿Qué estamos modelando con ella?

En nuestro modelo, no nos interesa modela el impulso nervioso, ni siquiera cuando ocurre, sino modelar la tasa de impulsos.

Hagamos un experimento: Pinchamos una neurona e inyectemos voltaje, y midamos la tasa de impulsos nervioso versus el voltaje. Pero llega una etapa que el periodo refractario deja que aumente la tasa de disparo y se empiece a saturar.

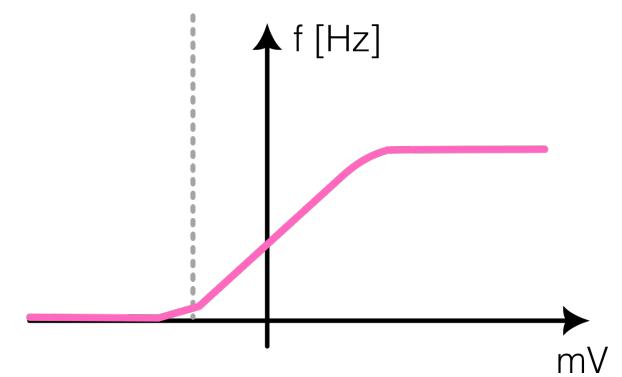
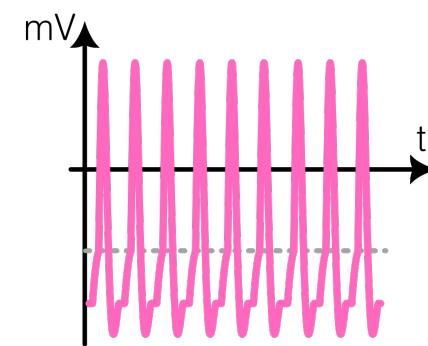
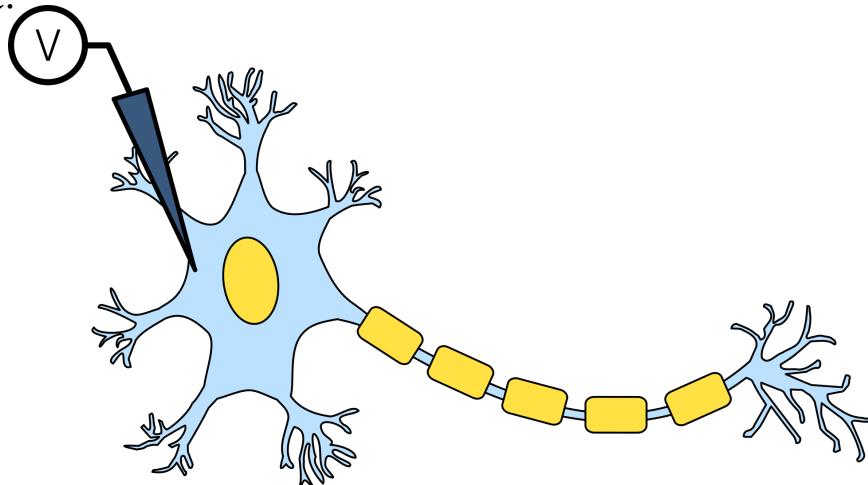


# PERCEPTRÓN

Veamos la función de activación. ¿Qué estamos modelando con ella?

En nuestro modelo, no nos interesa modela el impulso nervioso, ni siquiera cuando ocurre, sino modelar la tasa de impulsos.

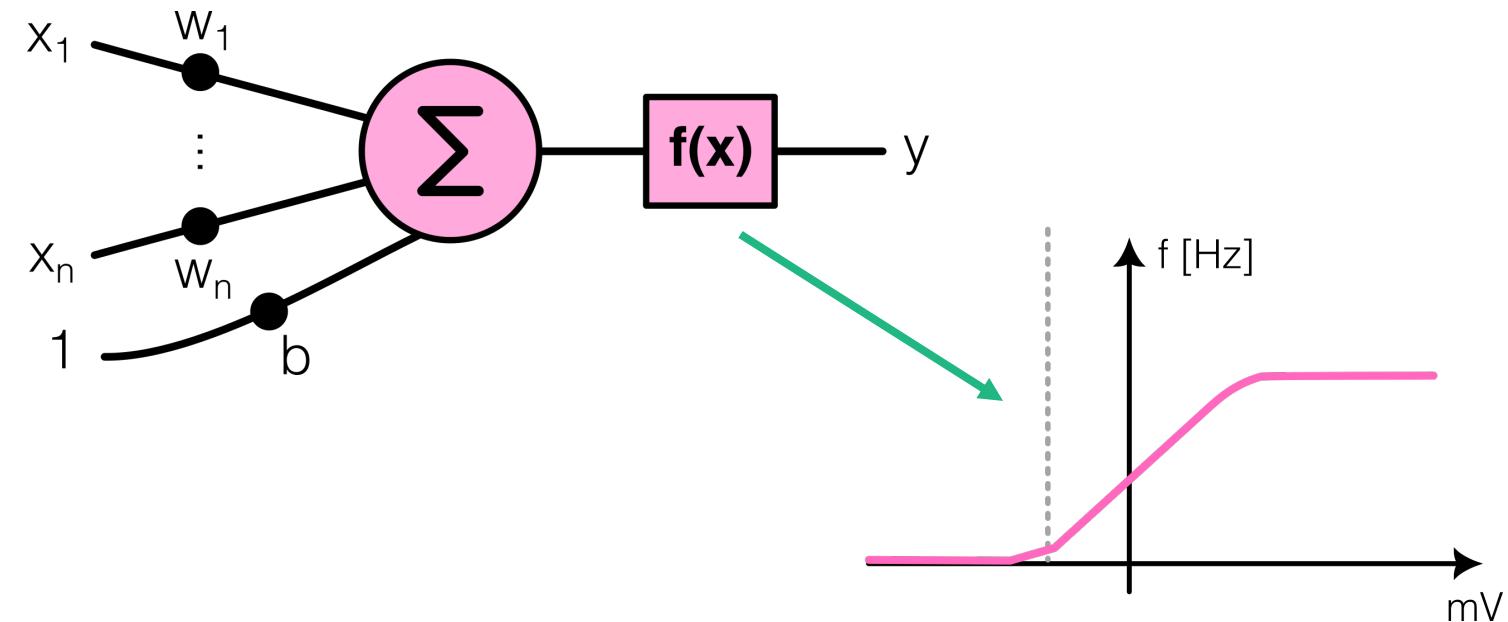
Hagamos un experimento: Pinchamos una neurona e inyectemos voltaje, y midamos la tasa de impulsos nerviosos versus el voltaje. Cuando llegamos a un punto, no importa cuánto voltaje introduzcamos, a la neurona dispara a la máxima frecuencia.



# PERCEPTRÓN

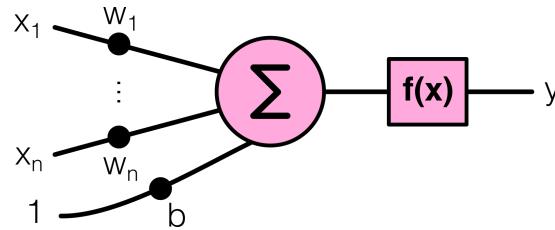
Por lo que la función de activación es una función no lineal que modela la variación de la tasa de disparo de la neurona.

Y con esto último, tenemos la expresión mínima de una neurona, el cual es una **unidad de cálculo no lineal**.



# PERCEPTRÓN

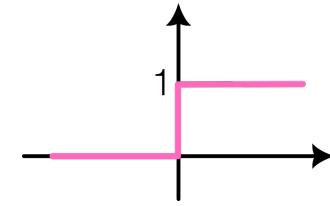
La función de activación que se usa en redes neuronales es una decisión de diseño:



$$y = f\left(\sum_{i=1}^n x_i w_i + b\right)$$

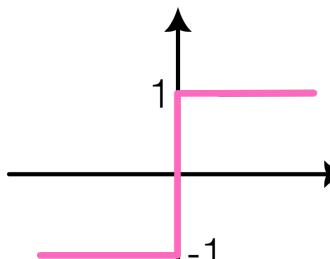
Función escalón

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



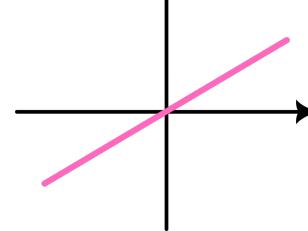
Función signo

$$f(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



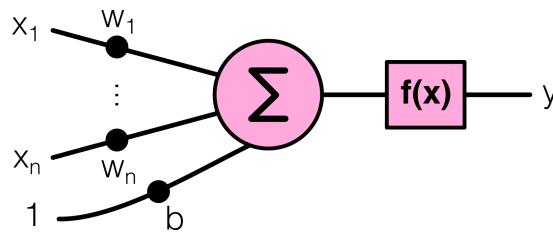
Función lineal

$$f(x) = x$$



# PERCEPTRÓN

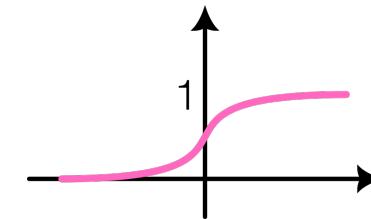
La función de activación que se usa en redes neuronales es una decisión de diseño:



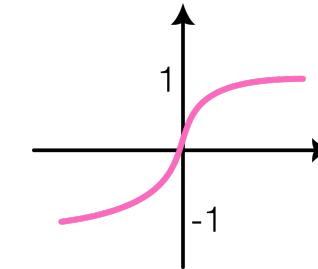
$$y = f\left(\sum_{i=1}^n x_i w_i + b\right)$$

Funciones sigmoideas

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

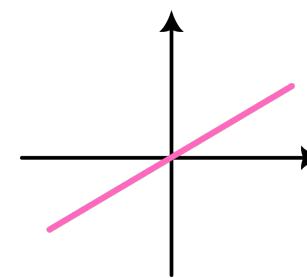


$$f(x) = \tanh(x)$$



Función ReLu

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



---

# PERCEPTRÓN

Con un perceptrón o neurona o red de una sola capa, se toma un vector de variables:

$$X = (x_1, x_2, \dots, x_n)$$

Donde, una neurona entrenada, puede predecir un **label** o variable a predecir,

$$y = f\left(\sum_{i=1}^n x_i w_i + b\right)$$

Si usamos la **función de activación lineal**... podemos ver una regresión lineal

Si usamos la **función sigmoidea**... podemos ver una regresión logística. Al usar funciones de activación con salida continua, tenemos una métrica no solo de que clase es, sino un valor probabilístico de que tan seguro es la predicción.

Como podemos ver una neurona puede hacer clasificaciones o regresiones. Una neurona en un espacio n-dimensional pone un hiperplano para separar clases o realizar una regresión.

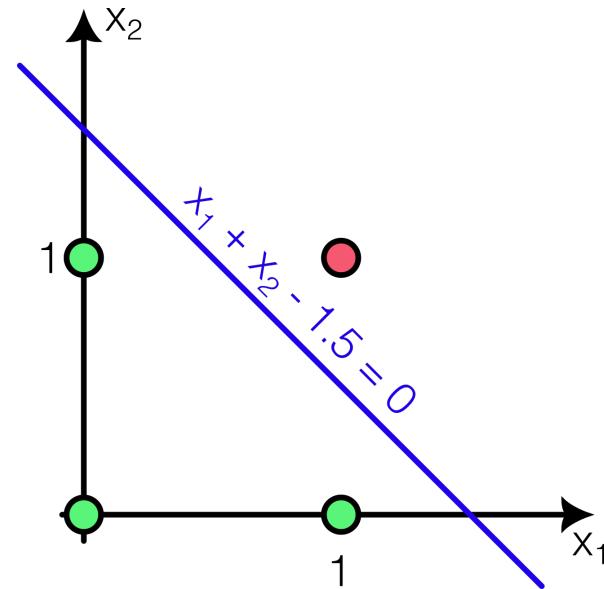
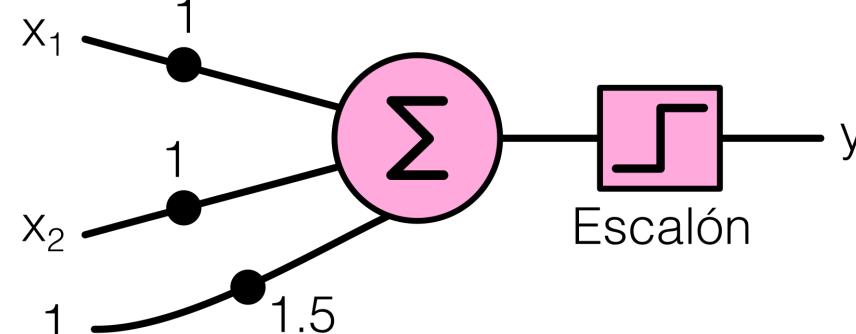
---

# PERCEPTRÓN

Veamos el ejemplo de funciones lógicas:

**AND** (2 entradas)

<b>x<sub>1</sub></b>	<b>x<sub>2</sub></b>	<b>y</b>
0	0	0
0	1	0
1	0	0
1	1	1

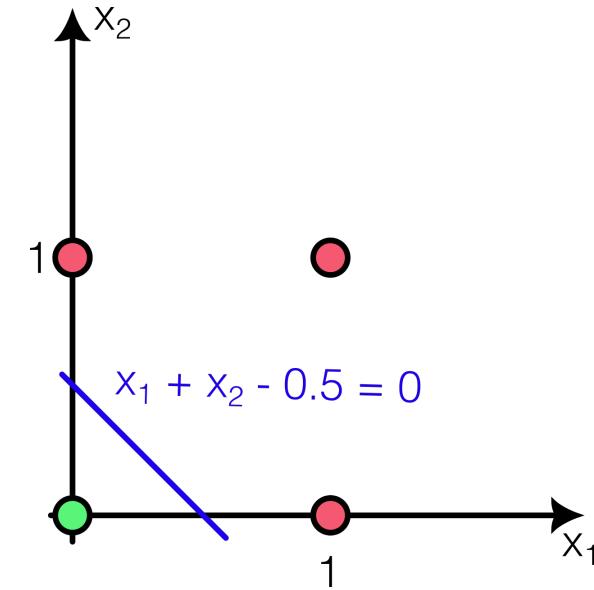
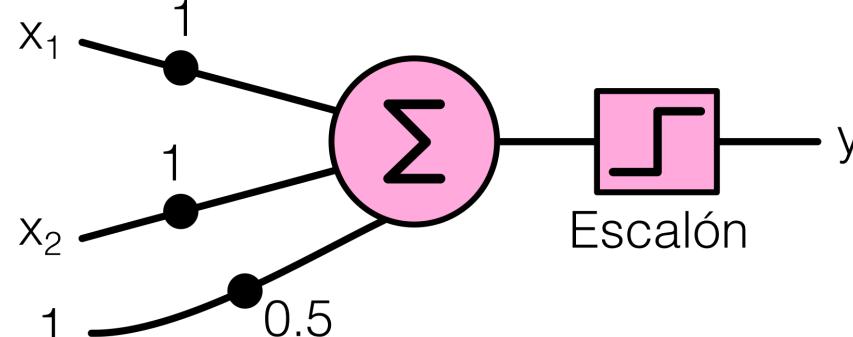


# PERCEPTRÓN

Veamos el ejemplo de funciones lógicas:

**OR (2 entradas)**

<b>x<sub>1</sub></b>	<b>x<sub>2</sub></b>	<b>y</b>
0	0	0
0	1	1
1	0	1
1	1	1

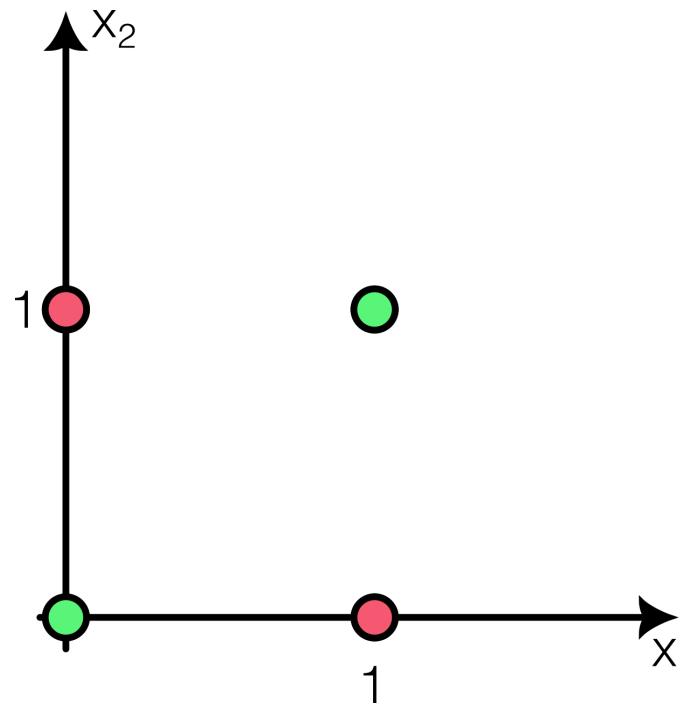


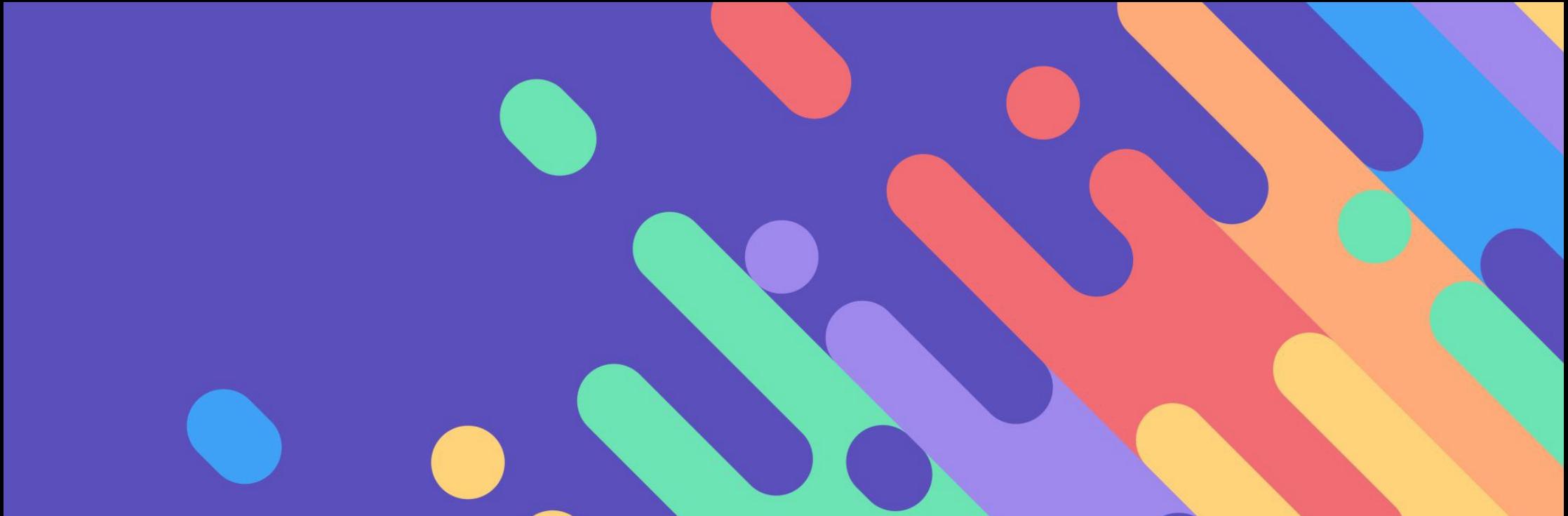
# PERCEPTRÓN

Veamos el ejemplo de funciones lógicas:

**XOR** (2 entradas)

<b>x<sub>1</sub></b>	<b>x<sub>2</sub></b>	<b>y</b>
0	0	0
0	1	1
1	0	1
1	1	0





---

# REDES FEED-FORWARD

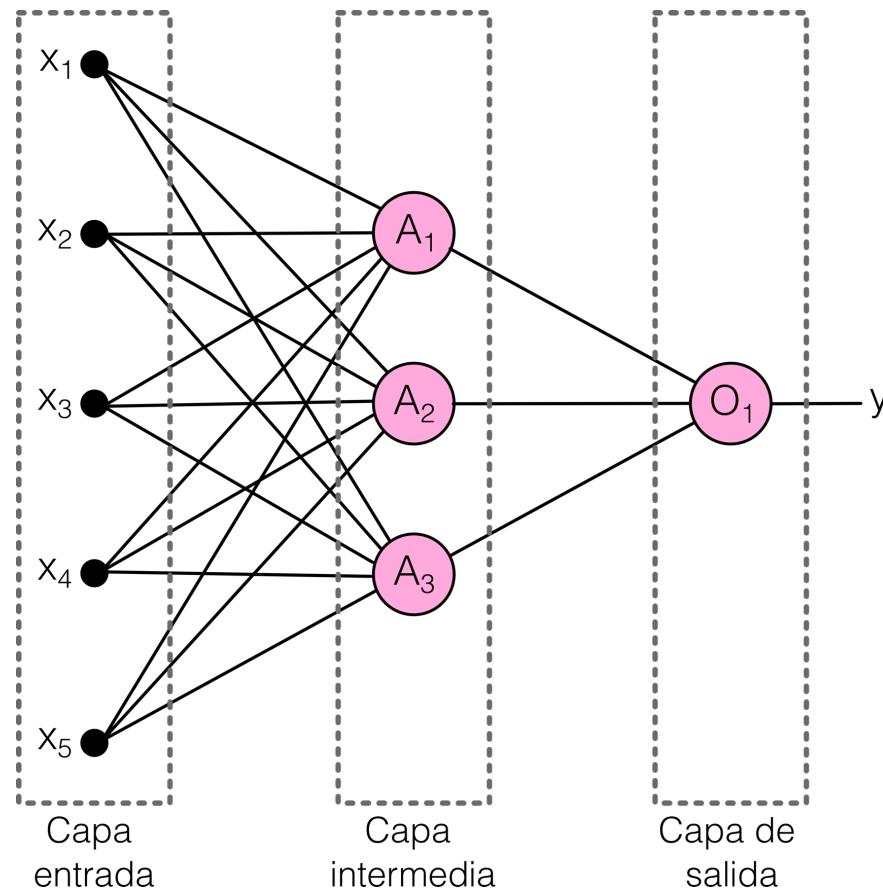
---

# REDES FEED-FORWARD

La forma de tener fronteras de decisión más complejas que la lineales o regresiones más complejas que la lineal, una forma de resolverlo es armando una red formada por neuronas en capas, en donde cada capa se conecta con la siguiente pero nunca hay retroalimentación.

Además, este tipo de redes, también llamadas redes densas, una neurona de una capa se conecta con todas las neuronas de la siguiente capa.

# REDES FEED-FORWARD



---

# REDES FEED-FORWARD

Este tipo de redes, la capa intermedia transforma el espacio de entrada en diferentes espacios generalmente no lineales.

Si todas las neuronas tienen función de activación lineal, la red colapsa a una sola neurona, por lo que una red feed-forward se justifica en casos no lineales.

---

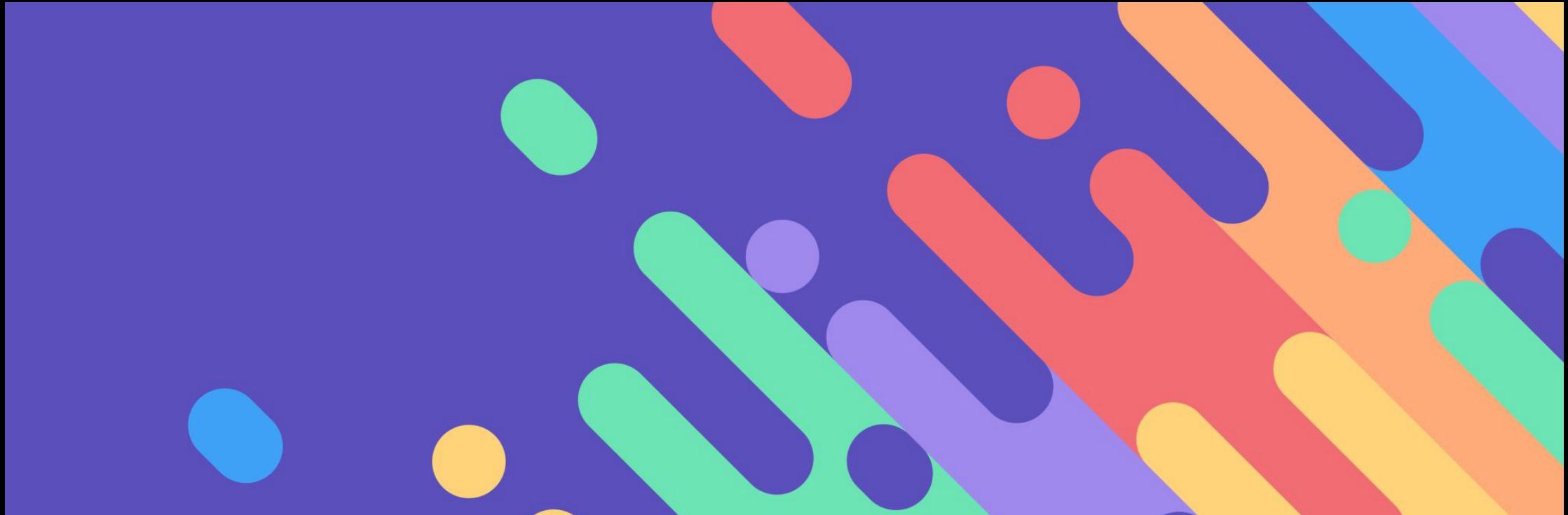
# REDES FEED-FORWARD

Cada neurona de la capa intermedia tiene la misma función de activación y su salida es:

$$A_k = g \left( \sum_{i=1}^n x_i w_{ki} + b_k \right)$$

Esta transformación producida por la capa intermedia, luego ingresan como entrada a la capa final:

$$y = f \left( \sum_{k=1}^K A_k \omega_k + \beta \right)$$



---

# BREVE INTRODUCCIÓN DE ENTRENAMIENTO DE REDES

---

# ENTRENAMIENTO

El entrenamiento de redes es algo complejo, y lo verán en más detalles en otra asignatura. Entrenar una red neuronal, es ajustar los pesos sinápticos para que los resultados coincidan con los valores a predecir del set de entrenamiento. Es decir, las redes neuronales feed-forward es de **aprendizaje supervisado**.

Para entrenar nuestro modelo, como vimos otros, es minimizar una función de perdida que sea acotada, de tal forma que podemos llegar, mediante alguna técnica de optimización al valor mínimo:

$$\underset{w_k, b_k, \omega_k, \beta}{\text{minimizar}} R(f(X_i))$$

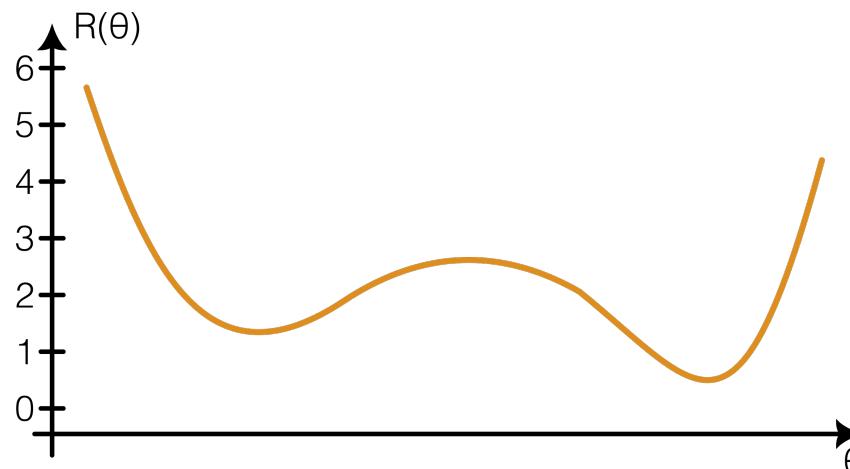
---

# ENTRENAMIENTO

minimizar  $R(f(X_i))$   
 $w_{kj}, b_k, \omega_k, \beta$

Donde:  $f(X_i) = \beta + \sum_{k=1}^K \omega_k g\left(\sum_{i=1}^n x_i w_{ki} + b_k\right)$

Esta función de perdida, que tiene tantas dimensiones como pesos sinápticos tiene, no es convexa y, por lo tanto, existen múltiples soluciones. Veamos a un ejemplo a que nos referimos a esto de un caso de optimización con un solo parámetro:



---

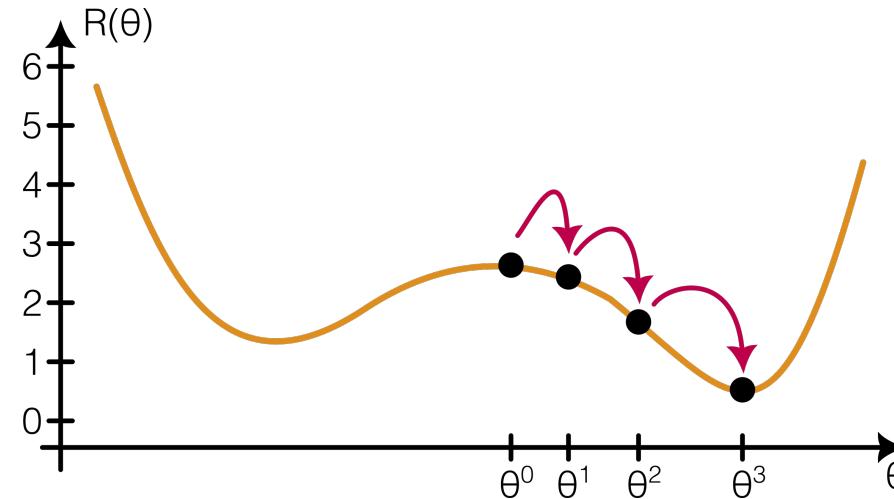
# ENTRENAMIENTO

Para lograr evitar caer en mínimos locales y también de sobreajustes lo entrenamos lentamente usando un algoritmo de **gradiente descendiente** y el proceso se corta cuando se detecta sobreajuste usando set de validación.

# ENTRENAMIENTO

La idea de gradiente descendiente es:

1. Comenzar con un valor de los pesos sinápticos al azar  $\theta_0$  en  $t=0$
2. Iterar hasta que  $R(\theta)$  deja de crecer o llega a un valor objetivo:
  1. Buscar un vector  $\delta$  que refleje un pequeño cambio en  $\theta$ , de tal forma que  $\theta^{(t+1)} = \theta^t + \delta$  reduzca a  $R(\theta)$ .
  2. Cambiar  $t$  a  $t+1$ , e iterar



---

# ENTRENAMIENTO

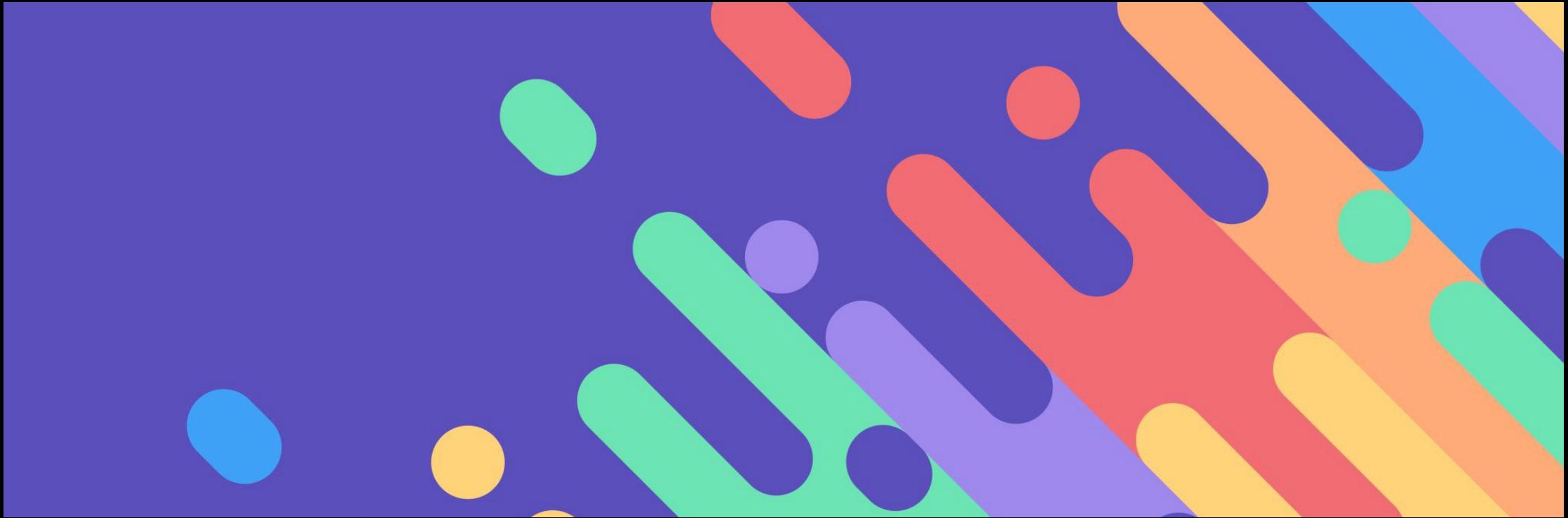
Este algoritmo tiene un problema el cual es muy lento para set de entrenamientos muy largos, ya que debe pasar por todos los valores y evaluar a R cada vez.

Una forma que podemos acelerar esto es submuestreando en batch (minibatch) de **m** observaciones,

Para cada minibatch, se calculan los gradientes de las **m** observaciones y los sumamos. Luego evaluamos R.

De esta forma se acelera el proceso porque no estamos calculando R cada paso, y esto acelera el proceso, pero aumentamos la probabilidad de no llegar a un resultado más optimo (pasamos a usar gradiente descendiente estocástico).

---



---

PYTORCH

---

# PYTORCH

**PyTorch** es una librería de Deep Learning. Es mantenida por Meta y es la principal competencia de otra famosa librería (tensorflow de Google).

Tiene un API para Python, como también para C++. PyTorch nos da toda una infraestructura de:

- Computación de tensores. Es similar a los arrays de Numpy pero con posibilidad de usarlos en GPU.
- Redes neuronales profundas.