Introducción a Redes Neuronales

Dr. Mauricio Toledo-Acosta

Diplomado Ciencia de Datos con Python

Índice

¿Dónde encontramos Deep Learning en la vida real?

- **Asistentes virtuales**:
  - LLMs: DeepSeek, Qwen, ChatGPT, Claude, etc.
  - Siri, Alexa (reconocimiento de voz y respuestas).
- **Redes sociales y fotos**:
  - Etiquetado automático de personas en fotos.
  - Filtros de cámara.
  - Sistemas de recomendación.
- **Seguridad y tecnología**:
  - Reconocimiento facial.
  - Detección de fraudes en tarjetas de crédito.
- **Transporte y mapas**:
  - Recomendación de rutas en Google Maps/Waze.
  - Coches autónomos.
- **Salud**:
  - Diagnóstico médico asistido.
  - Wearables.

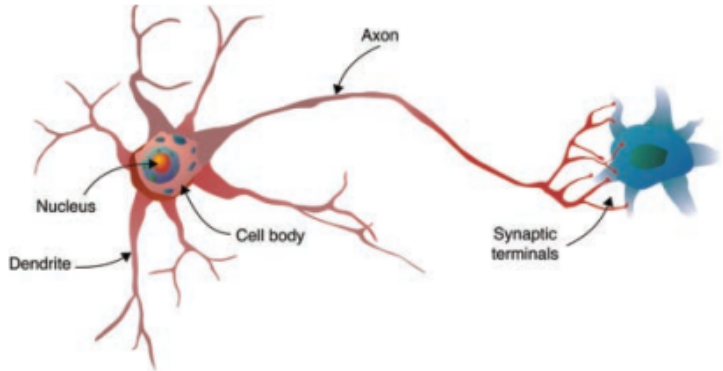
¿Cuándo usar Deep Learning?

- Datos grandes y/o complejos
- Patrones no lineales
- Datasets de imágenes, audio, texto
- Cuando los métodos tradicionales fallan

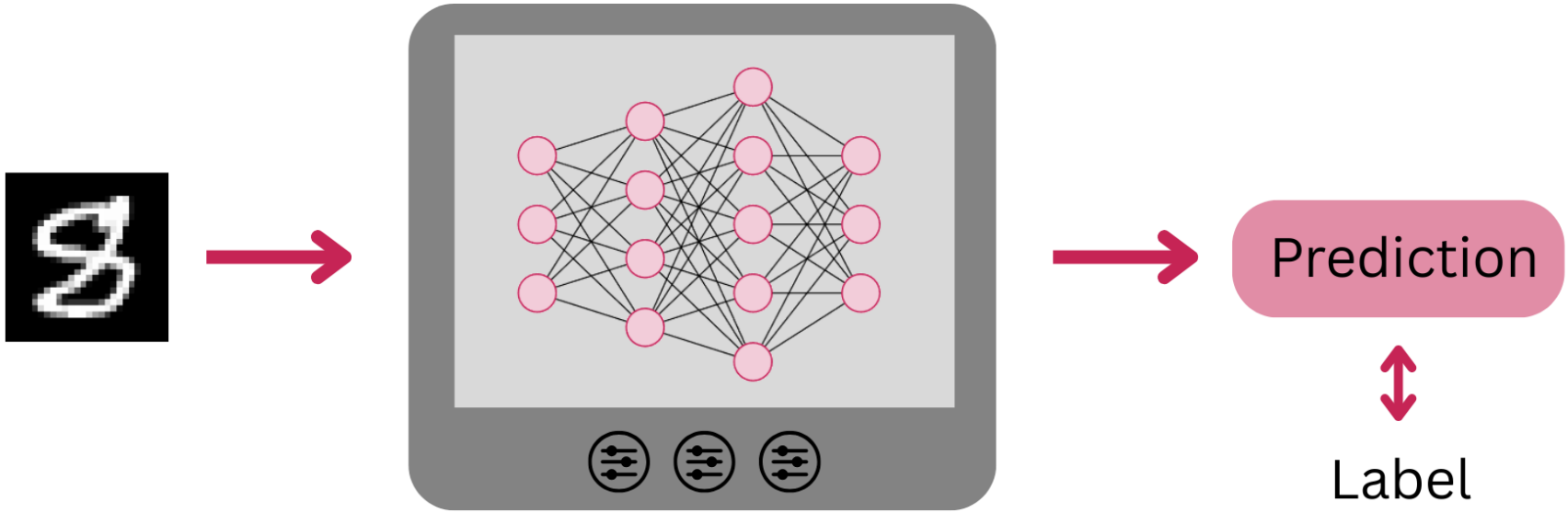
Introduction: Redes Neuronales

Redes Neuronales

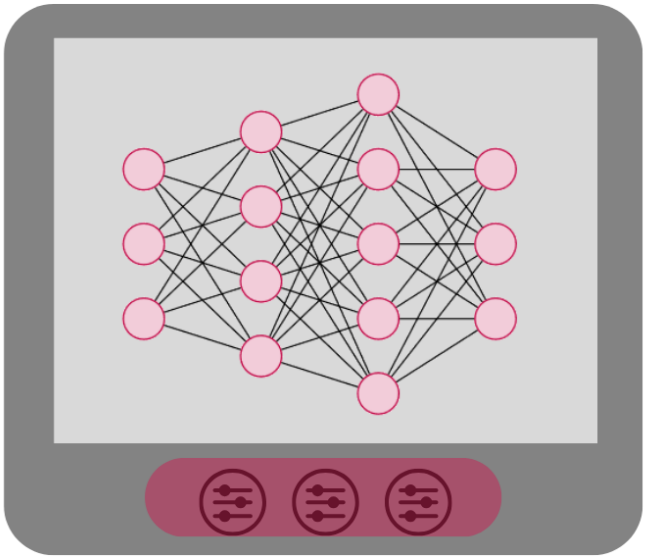
Las redes neuronales artificiales son técnicas de Machine Learning que simulan el mecanismo de aprendizaje de los organismos biológicos. El sistema nervioso humano contiene células (neuronas), conectadas entre sí por axones y dendritas. Las regiones de conexión entre axones y dendritas se denominan sinapsis. La fuerza de las conexiones sinápticas cambia en respuesta a estímulos externos. Así es como se produce el aprendizaje en los organismos vivos.



1. Feed Forward



2. Back Propagation



Prediction

Label

El perceptrón

- El algoritmo Perceptrón (McCulloch–Pitts-Rosenblatt, 1943-1961) desempeñó un papel importante en la historia del ML. Fue simulado por primera vez en una computadora IBM 704 en 1957. A principios de los 60, se diseñó un hardware específico (Mark I perceptron) para implementar el algoritmo.

El perceptrón

- El algoritmo Perceptrón (McCulloch–Pitts-Rosenblatt, 1943-1961) desempeñó un papel importante en la historia del ML. Fue simulado por primera vez en una computadora IBM 704 en 1957. A principios de los 60, se diseñó un hardware específico (Mark I perceptron) para implementar el algoritmo.

- Fue criticado por Marvin Minsky, quien demostró las limitaciones del algoritmo cuando se trataba de un conjunto no linealmente separable. Esto provocó un vacío en la investigación de la computación neuronal que duró hasta mediados de los 80s.

Deep Learning

Deep Learning

El aprendizaje profundo forma parte de una familia más amplia de métodos de ML, se basan en redes neuronales artificiales. El aprendizaje puede ser supervisado o no supervisado.

El adjetivo *deep* en deep learning hace referencia al uso de multiplies capas en una red, lo cual le da *profundidad*.

Deep Learning: Aproximadores Universales

Resultados teóricos demuestran que las redes neuronales profundas pueden aprender cualquier tipo de funciones complicadas que pueden representar abstracciones de alto nivel.

ELSEVIER

Neural Networks

Volume 2, Issue 5, 1989, Pages 359-366

Original contribution

Multilayer feedforward networks are universal approximators

Kurt Hornik, Maxwell Stinchcombe, Halbert White [1]

Deep Learning: Aproximadores Universales

Resultados teóricos demuestran que las redes neuronales profundas pueden aprender cualquier tipo de funciones complicadas que pueden representar abstracciones de alto nivel.

ELSEVIER

Neural Networks

Volume 2, Issue 5, 1989, Pages 359-366

Original contribution

Multilayer feedforward networks are universal approximators

Kurt Hornik, Maxwell Stinchcombe, Halbert White [1]

Las arquitecturas profundas se componen de múltiples niveles de operaciones no lineales, como en las redes neuronales con muchas capas ocultas.

Deep Learning

Las arquitecturas profundas son útiles para:

- Reducir la complejidad computacional en funciones complejas.

Deep Learning

Las arquitecturas profundas son útiles para:

- Reducir la complejidad computacional en funciones complejas.
- Generar representaciones compactas (embeddings) de los datos de entrada.

Deep Learning

Las arquitecturas profundas son útiles para:

- Reducir la complejidad computacional en funciones complejas.
- Generar representaciones compactas (embeddings) de los datos de entrada.
- Producir modelos que permitan generalizar funciones con entradas de alta variabilidad.

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).
- Transformadores: GPT, BERT, DeepSeek, ...

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).
- Transformadores: GPT, BERT, DeepSeek, ...
- Redes de Difusión (Diffusion Models)

Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).
- Transformadores: GPT, BERT, DeepSeek, ...
- Redes de Difusión (Diffusion Models)
- Redes Siamesas (Siamese Networks)

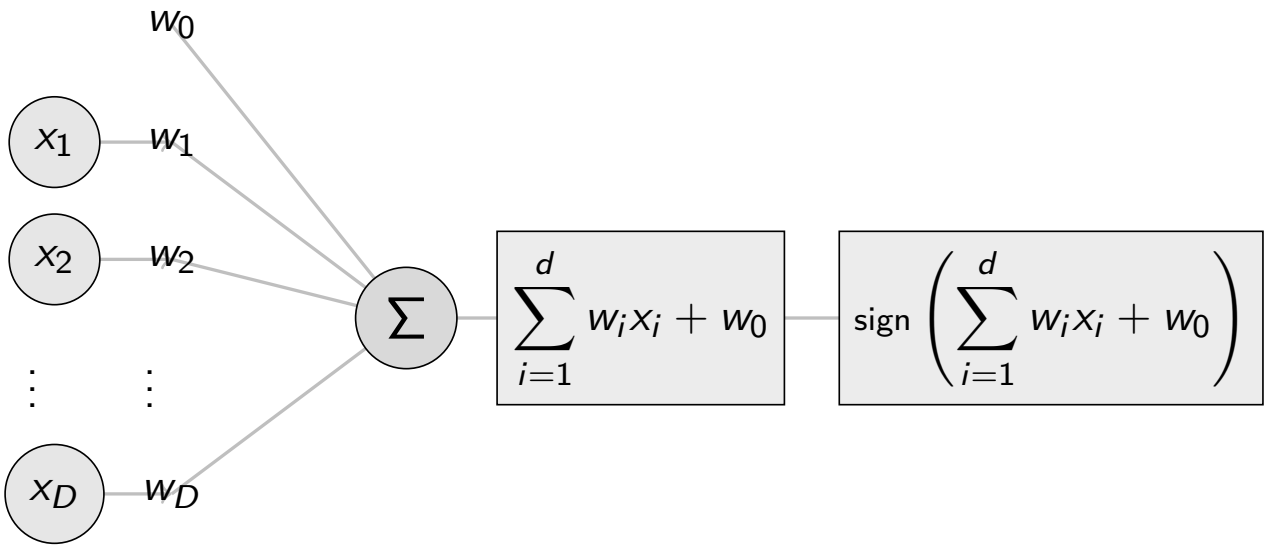
Algunas arquitecturas

Algunos tipos de Artificial neural networks:

- Perceptrones Multicapa (MLP) o Fully Connected Network.
- Redes Convolucionales (CNN).
- Autoencoders y Variational Autoenconders.
- Long short-term memory (LSTM) y Redes Recurrentes (RNN).
- Generative adversarial network (GAN).
- Self-Organizing Map (SOM).
- Transformadores: GPT, BERT, DeepSeek, ...
- Redes de Difusión (Diffusion Models)
- Redes Siamesas (Siamese Networks)
- Redes Neuronales de Grafos (GNN - Graph Neural Networks)

Índice

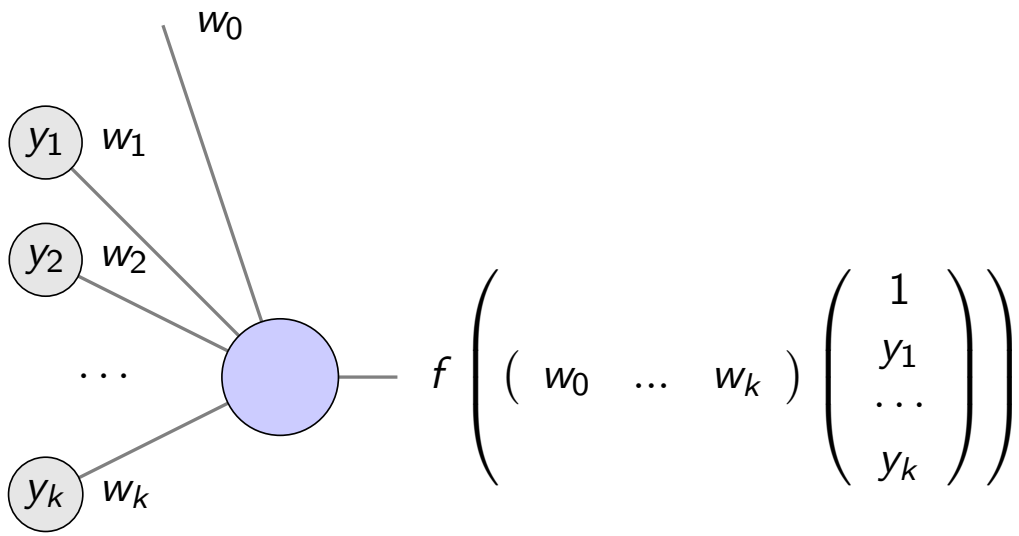
El perceptrón



- $\mathbf{w} = (w_0, w_1, ..., w_d)$ es el vector de pesos.
- Σ es una neurona.
- sign es la función de activación (no lineal).
- sign $\left(\sum_{i=1}^{d} w_i x_i + w_0 \right)$ es la salida de la neurona:

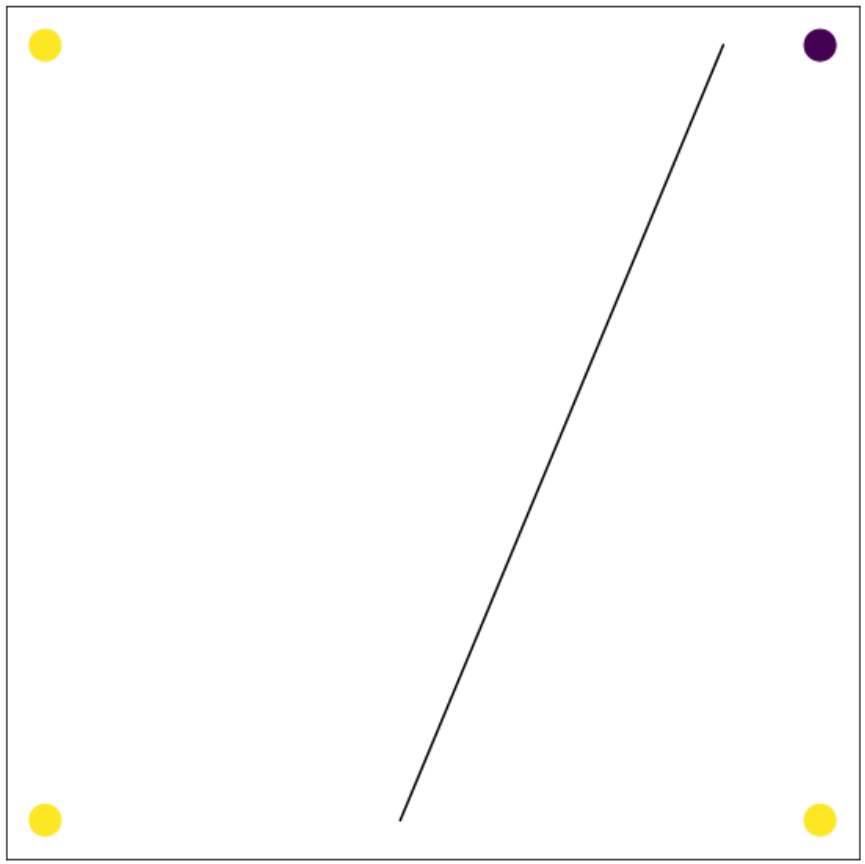
$$\text{output} = \text{activation}\,(\text{weights} \cdot \text{input})$$

El funcionamiento de una neurona

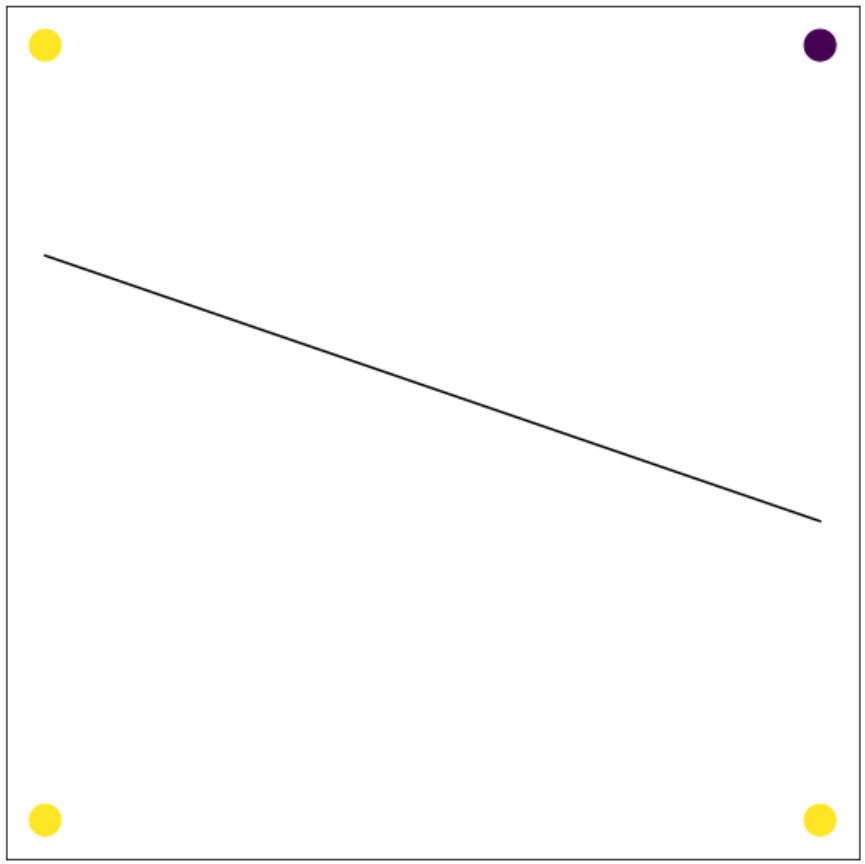


Tenemos k entradas (features) y una salida. La función de activación es f, los pesos w_i son valores a determinar.

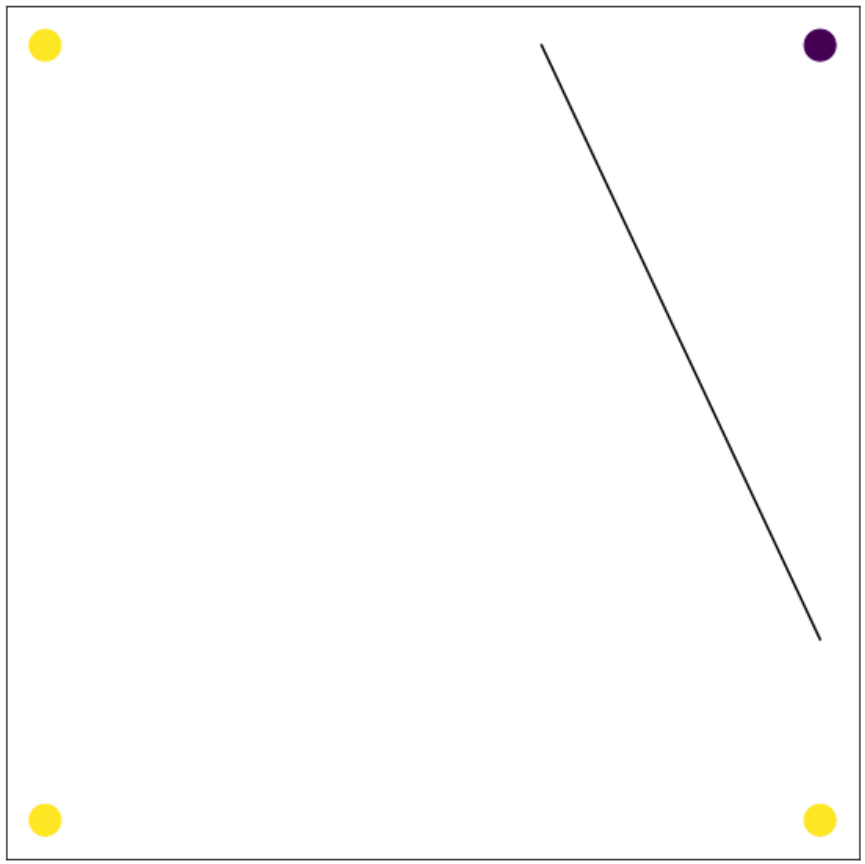
Ejemplo



Ejemplo

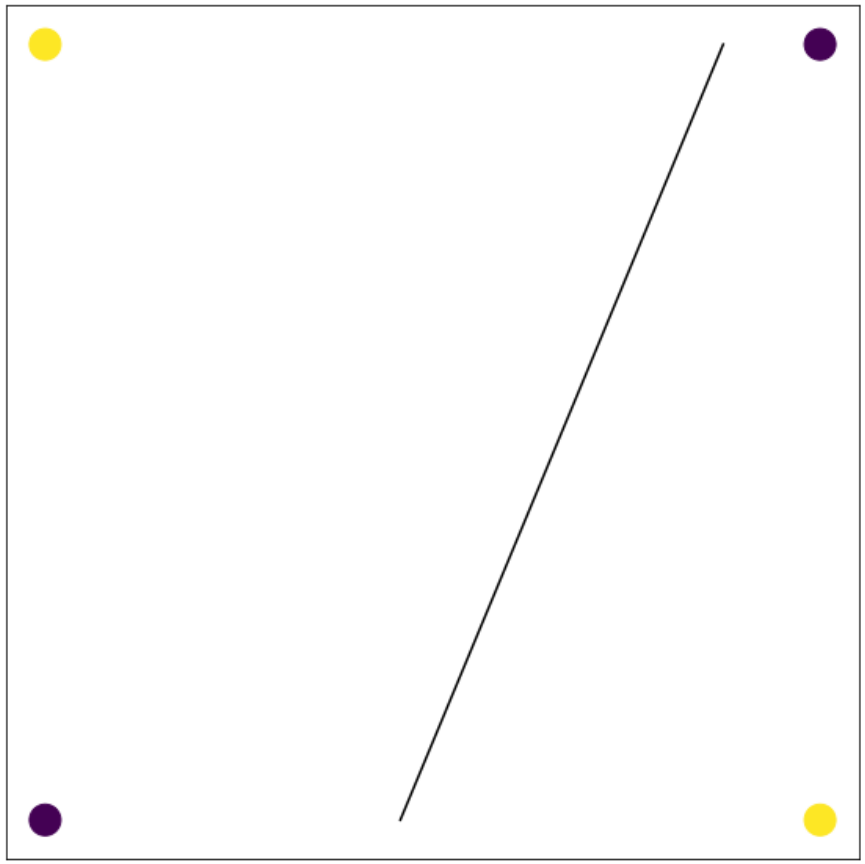


Ejemplo



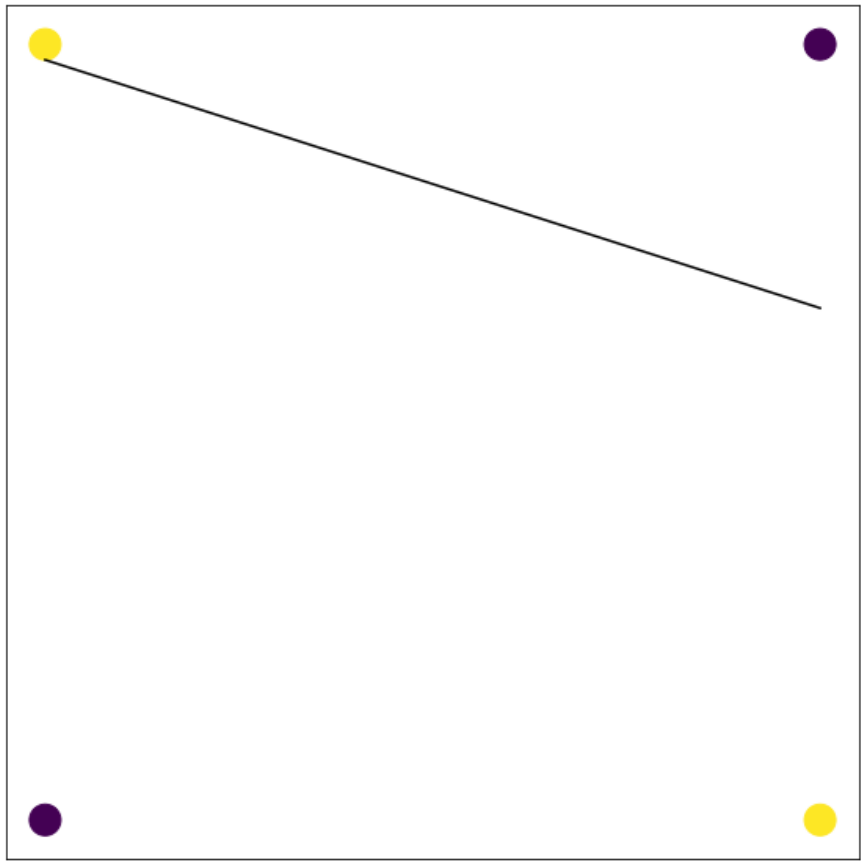
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



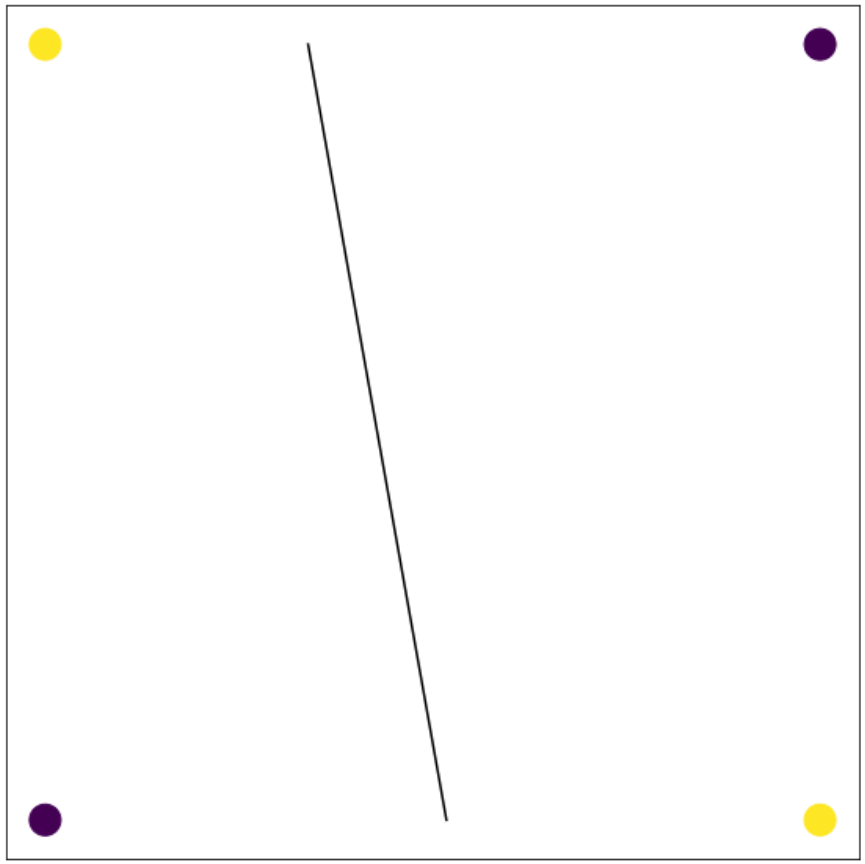
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



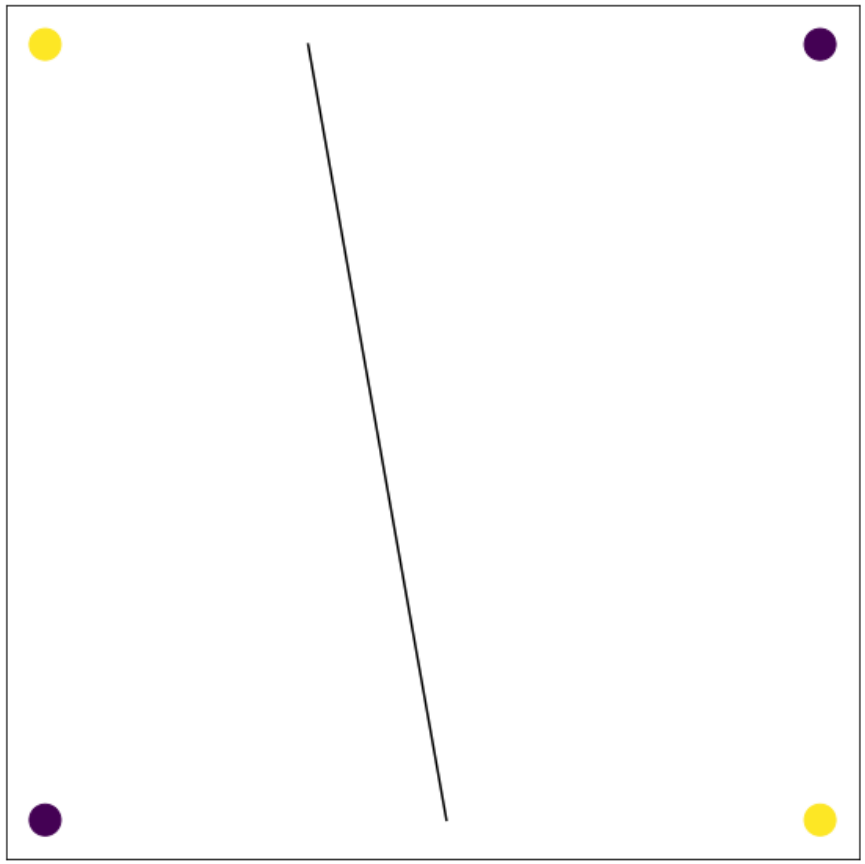
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



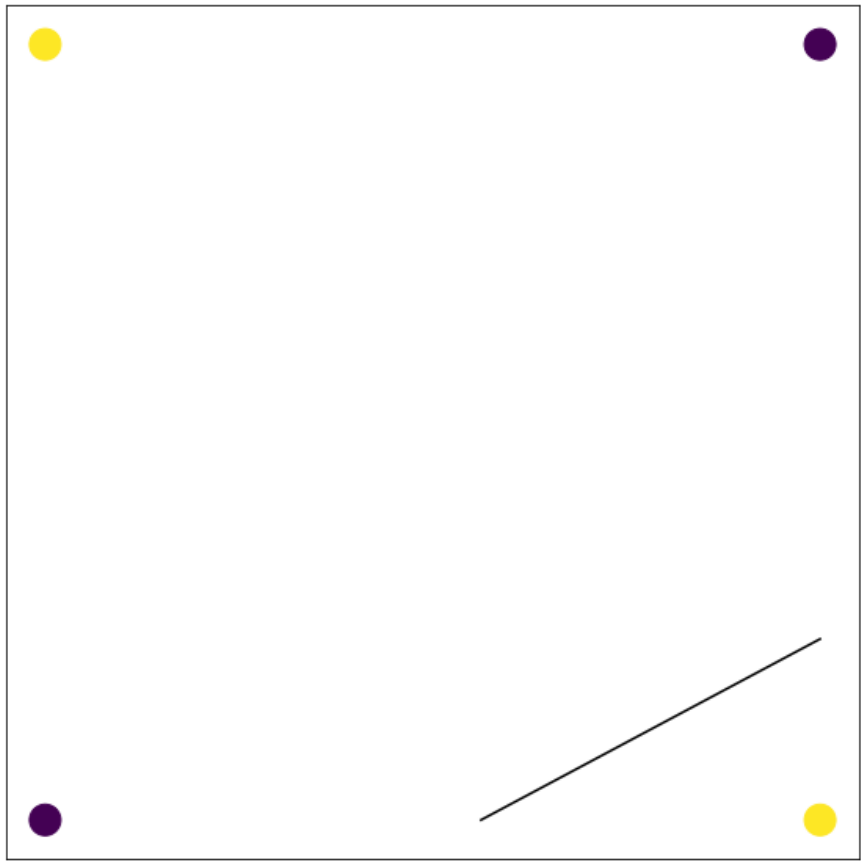
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



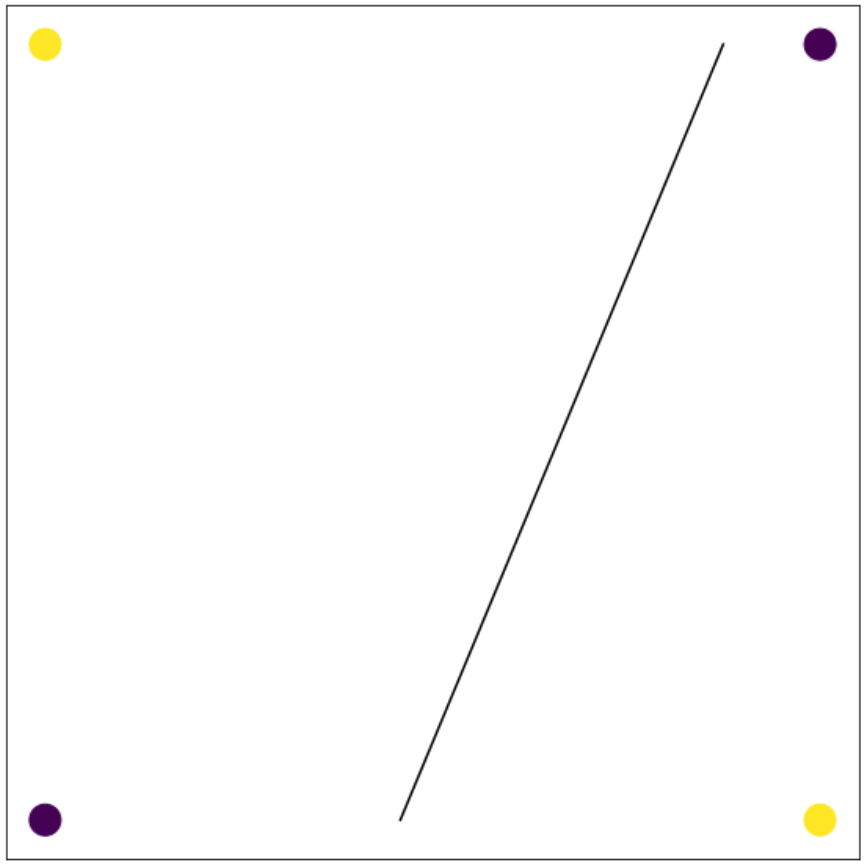
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



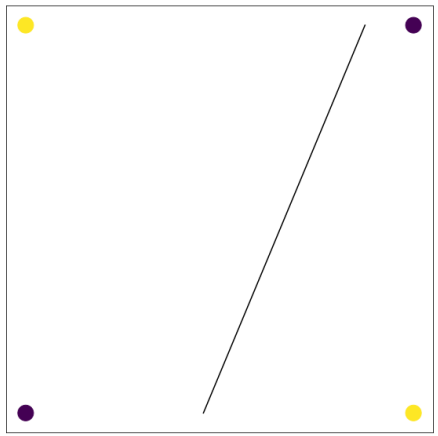
Limitaciones del Perceptrón

¿Qué pasa si los datos no son linealmente separables?



Limitaciones del Perceptrón

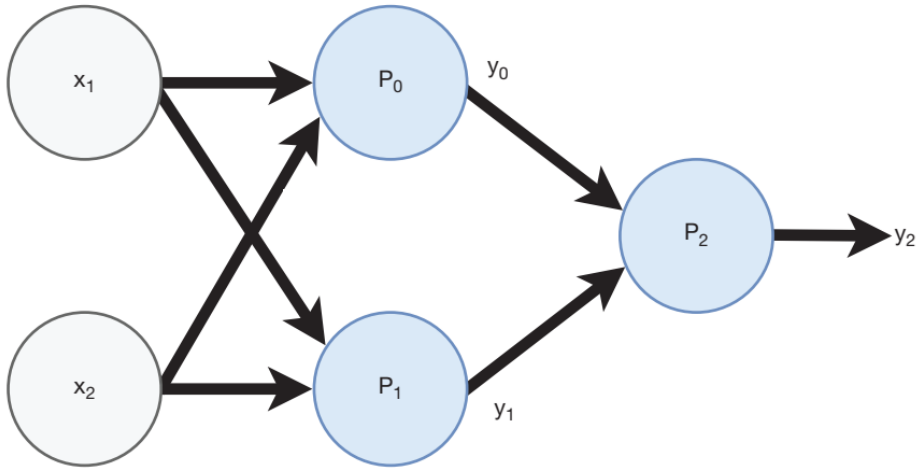
¿Qué pasa si los datos no son linealmente separables?



Alternativas:

- Cambiar el modelo de una neurona.
- Combinar multiples neuronas.

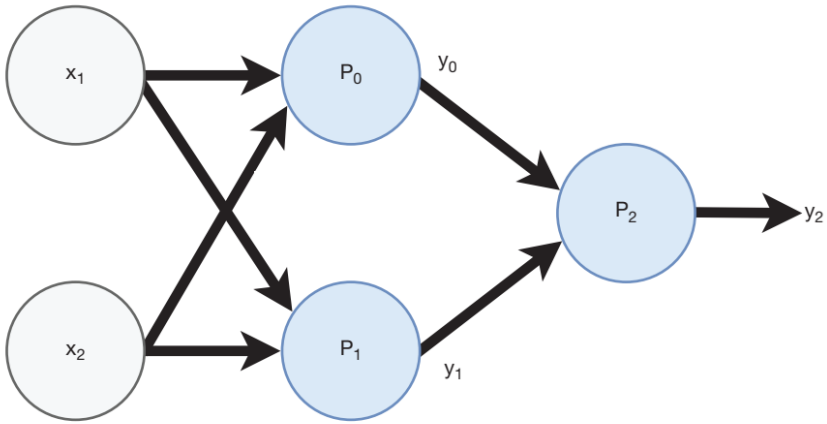
Combinando Perceptrones



Con esta arquitectura ya podemos resolver, de manera perfecta, este problema de clasificación

$$X = \begin{pmatrix} -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ 1 & 1 \end{pmatrix}, \; y = \begin{pmatrix} -1 \\ +1 \\ +1 \\ -1 \end{pmatrix}$$

Combinando Perceptrones

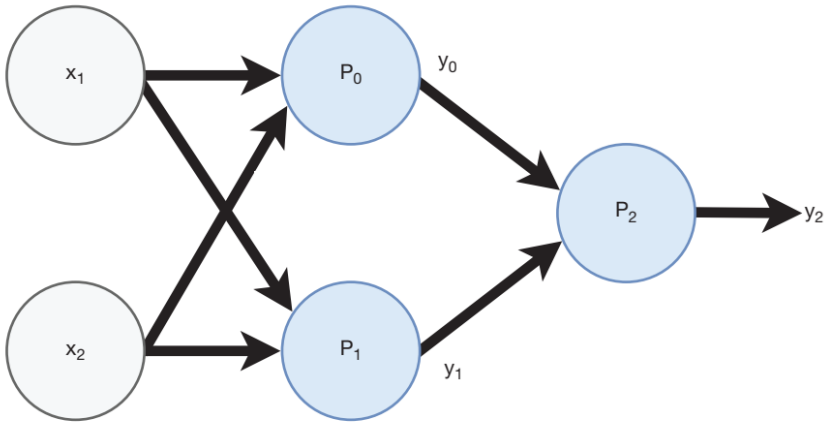


$$y_0 = f_0\left(\begin{pmatrix} w_0^{(0)} & w_1^{(0)} & w_2^{(0)} \end{pmatrix}\begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}\right)$$

$$y_1 = f_1\left(\begin{pmatrix} w_0^{(1)} & w_1^{(1)} & w_2^{(1)} \end{pmatrix}\begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}\right)$$

$$y_2 = f_2\left(\begin{pmatrix} w_0^{(2)} & w_1^{(2)} & w_2^{(2)} \end{pmatrix}\begin{pmatrix} 1 \\ y_0 \\ y_1 \end{pmatrix}\right)$$

Combinando Perceptrones

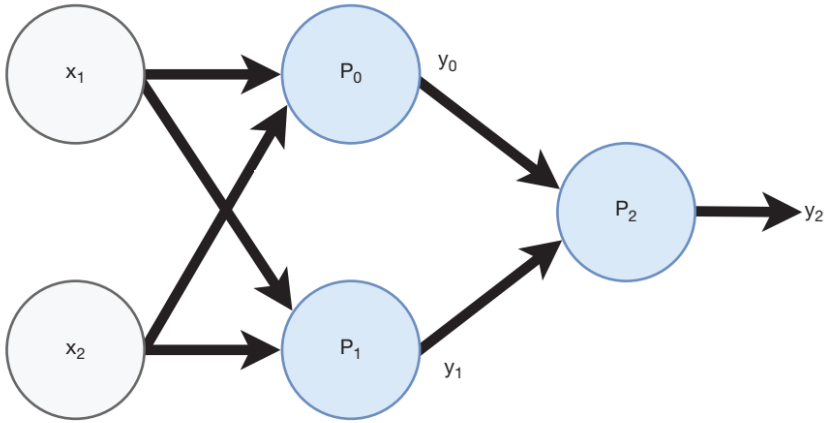


$$y_0 = \begin{pmatrix} w_0^{(0)} & w_1^{(0)} & w_2^{(0)} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$$

$$y_1 = \begin{pmatrix} w_0^{(1)} & w_1^{(1)} & w_2^{(1)} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}$$

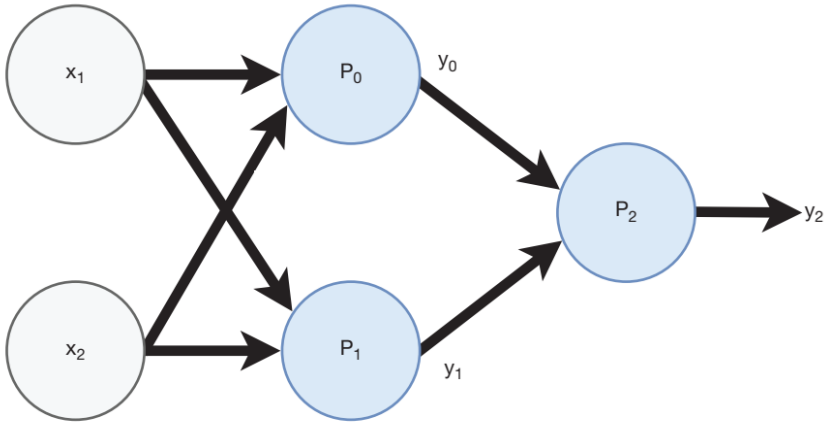
$$y_2 = \text{sign}\left(\begin{pmatrix} w_0^{(2)} & w_1^{(2)} & w_2^{(2)} \end{pmatrix} \begin{pmatrix} 1 \\ y_0 \\ y_1 \end{pmatrix} \right)$$

Combinando Perceptrones



Este arreglo de perceptrones (neuronas) es uno de los ejemplos más sencillos de red Feed-Forward Fully Connected. *Completamente conectada* significa que la salida de cada neurona de una capa está conectada a todas las neuronas de la capa siguiente. *Feed-forward* significa que no hay conexiones hacia atrás. Una red neuronal multinivel tiene una capa de entrada, una o más capas ocultas y una capa de salida. La capa de entrada no contiene neuronas, sino sólo las propias entradas (features).

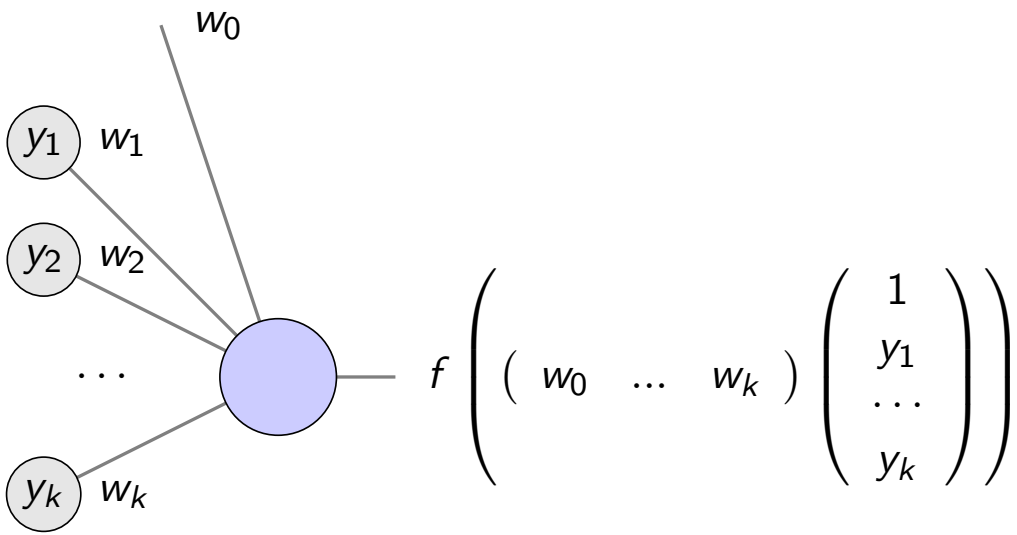
Combinando Perceptrones



- Tenemos dos features (blanco).
- Tenemos dos capas de neuronas (azul).
  - La capa oculta tiene dos neuronas, sin activación.
  - La capa de salida tiene una neurona con activación sign.

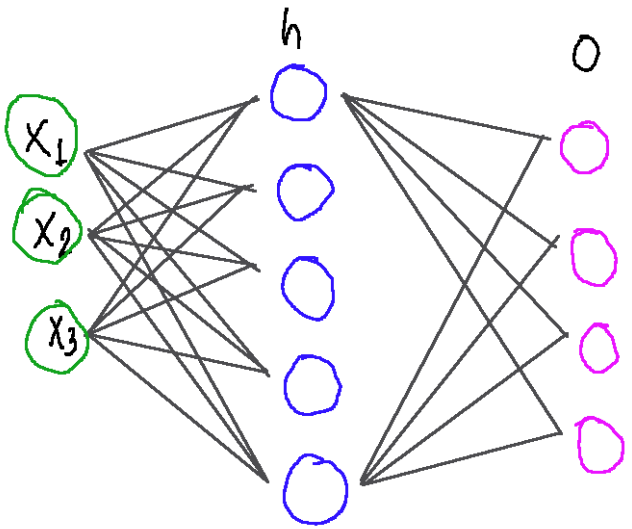
Entrenar esta red de perceptrones significa encontrar los mejores pesos (parámetros) $w_i^{(k)}$.

El funcionamiento de una neurona



La capa previa tiene k neuronas, la función de activación es f.

El output de una red neuronal



$$X = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ & \cdots & & \\ 1 & x_1^{(N)} & x_2^{(N)} & x_3^{(N)} \end{pmatrix}$$

$$h = \Phi_1\left(X \cdot W_1^T\right)$$

$$O = \Phi_2\left(\bar{h} \cdot W_2^T\right)$$

$W_1 \in M_{5\times 4}$ $\quad W_2 \in M_{4\times 6}$

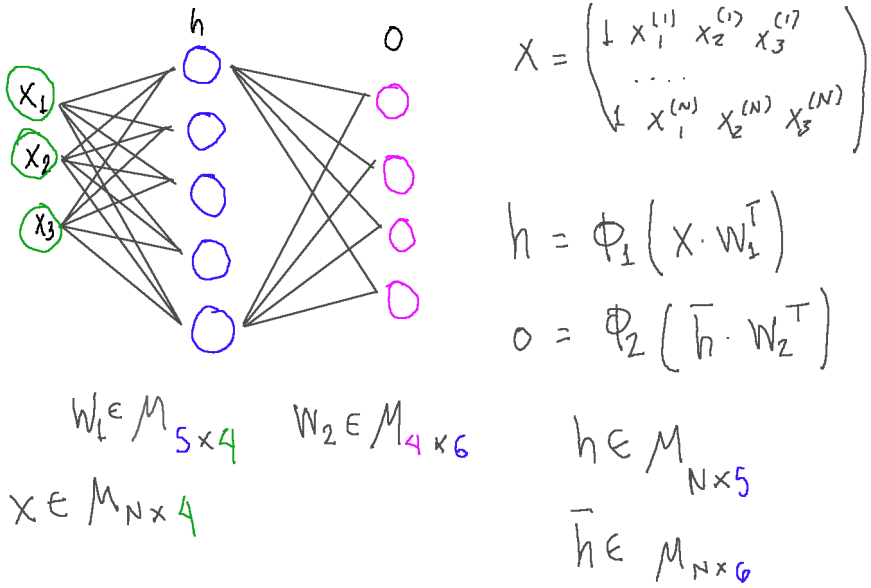
$X \in M_{N\times 4}$

$h \in M_{N\times 5}$

$\bar{h} \in M_{N\times 6}$

Los valores en las neuronas de salida dependen de la función de activación Φ_2.

El output de una red neuronal



Los valores de salida se comparan con las etiquetas del dataset y el error se mide con una función de perdida.

El aprendizaje de una red neuronal consiste en determinar los valores de las matrices W_i que minimicen el error total de salida.

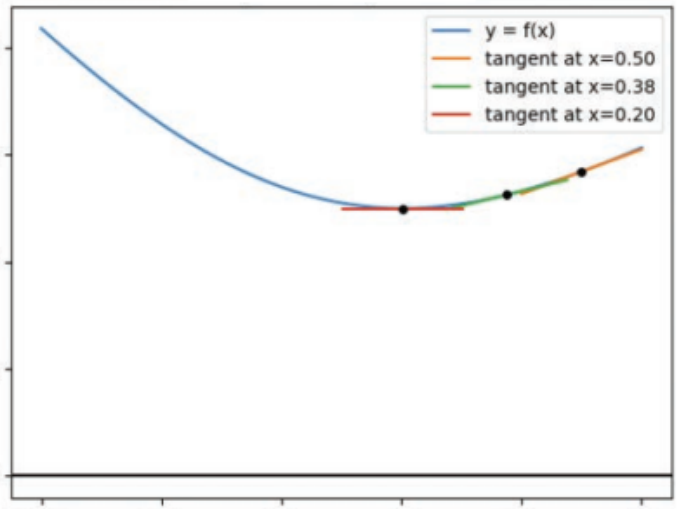
Índice

¿Cómo encontramos el mínimo de una función?

- Sea $f(x)$ una función, queremos encontrar el mínimo de f.

¿Cómo encontramos el mínimo de una función?

- Sea $f(x)$ una función, queremos encontrar el mínimo de f.
- La derivada en el punto x que minimiza el valor de f es 0.



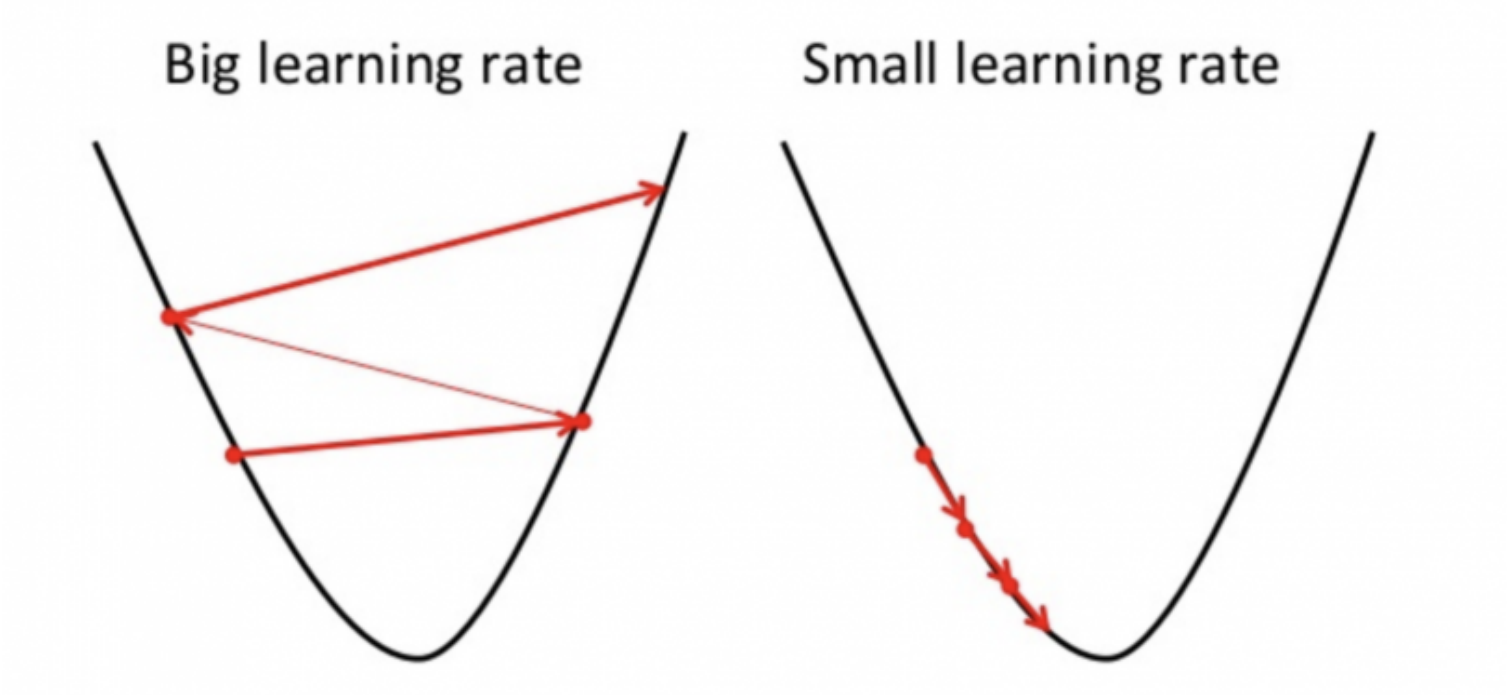
Descenso de gradiente

- A partir de un valor inicial x queremos encontrar un valor que minimiza f.
- El signo de $f'(x)$ nos indica la dirección en la cual *mover* x para reducir el valor de $f(x)$.
- La magnitud de $f'(x)$ nos indica que tanto ajustar x. Usamos un peso η, llamado learning rate,

$$x_{n+1} = x_n - \eta f'(x_n).$$

- Si η es muy grande, el método puede *pasarse* de la solución y no converger.
- El algoritmo no garantiza encontrar el mínimo global, puede quedarse atorado en un mínimo local.

Learning Rate

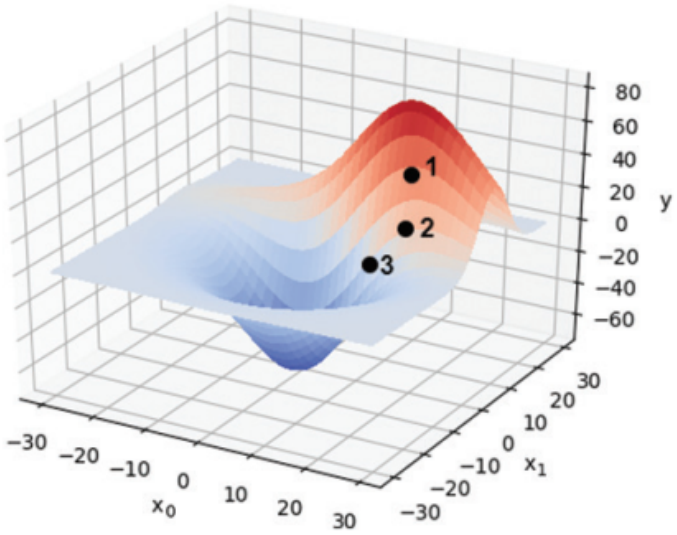


El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.

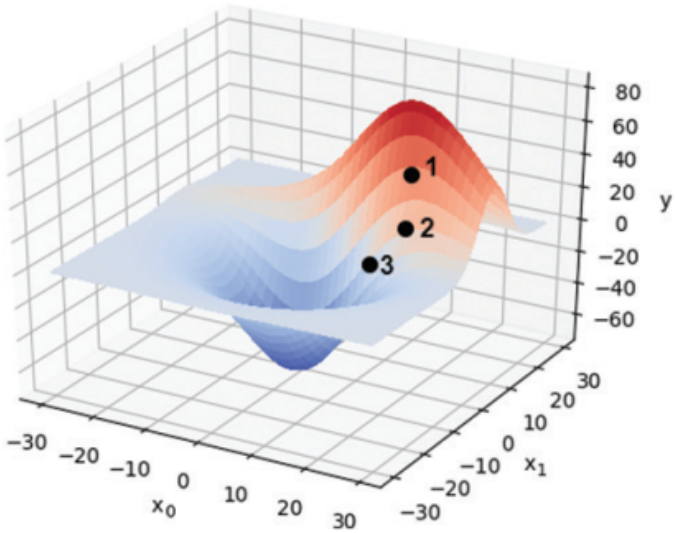
El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.
- El gradiente es un vector que consta de las derivadas e indica la dirección en el espacio de features en la que se produce el ascenso más pronunciado en el valor de f.



El caso de dimensión alta

- Consideremos la función $f(x_0, x_1)$.
- El gradiente es un vector que consta de las derivadas e indica la dirección en el espacio de features en la que se produce el ascenso más pronunciado en el valor de f.



- Si estamos en el punto $\mathbf{x_n} = \left(x_0^{(n)}, x_1^{(n)}\right)$ y queremos minimizar f, ajustamos el punto de la siguiente forma

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \eta \nabla f(\mathbf{x_n})$$

Gradient-based Learning

Al aplicar descenso de gradiente a nuestra red neuronal, consideramos las features **x** como constantes, el objetivo es encontrar los pesos **w** de tal forma que minimicemos el error.

- ¿Qué función queremos minimizar? Una función de perdida.

Gradient-based Learning

Al aplicar descenso de gradiente a nuestra red neuronal, consideramos las features **x** como constantes, el objetivo es encontrar los pesos **w** de tal forma que minimicemos el error.

- ¿Qué función queremos minimizar? Una función de perdida.
- En el contexto de optimización, la función que evalua una solución candidata se le llama función objectivo. En las redes neuronales, la función objetivo se le llama función costo o perdida (loss function).

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by

$$L^{(0/1)}(\mathbf{w}) = (y_i - \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle)^2$$

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by

$$L^{(0/1)}(\mathbf{w}) = \frac{1}{2}(y_i - \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle)^2 = 1 - y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle$$

- This function is not smooth, we use the smooth surrogate loss function

$$L(\mathbf{w}) = \max\{-y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle, 0\}.$$

Gradient-based Learning

- In the case of the Perceptron, the loss function is given by
- This function is not smooth, we use the smooth surrogate loss function

$$L(\mathbf{w}) = \max\{-y_i \text{sign}\langle \mathbf{w}, \mathbf{x}_i \rangle, 0\}.$$

- Applying gradient descent

$$\begin{aligned} \mathbf{w}_{n+1} &= \mathbf{w}_n - \eta \nabla L(\mathbf{w}) \\ &= \begin{cases} \mathbf{w}_n + \eta y_i \mathbf{x}_i, & \text{well classified} \\ \mathbf{w}_n, & \text{misclassified} \end{cases} \end{aligned}$$

Loss Functions

Other examples of loss functions

▸ Keras Losses

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.
- Good for regression tasks.

Loss Functions: MSE

Mean Square Error (MSE)

$$L(y, t) = (t - y)^2.$$

- L2 loss.
- Good for regression tasks.
- Trivial derivative for gradient descent.

Loss Functions: MAE

Mean Absolute Error (MAE)

$$L = |t - y|.$$

- L1 loss.
- More robust to outliers than mse.
- Good for regression tasks.
- Discontinuity in its derivative.

Loss Functions: Hinge

Hinge Loss

$$L = \max\{-y_i \hat{y}_i, 0\}.$$

- Used in SVMs and Perceptron.
- Penalizes errors, but also correct predictions of low confidence (probabilities).
- Good for binary classification tasks.

Loss Functions: Categorical cross entropy

Categorical cross entropy

$$L = \sum_{i}^{n} y_i \log\left(\hat{y}_i\right).$$

- Good for multi-class classification problems.
- Considers y to be a one-hot encoding vector in n classes.

Optimizers: Gradient Descent

Gradient descent is the most basic but most used optimization algorithm.

Optimizers: Gradient Descent

Gradient descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms.

Optimizers: Gradient Descent

Gradient descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.

Optimizers: Gradient Descent

Advantages:

- Easy computation.
- Easy to implement.
- Easy to understand.

Optimizers: Gradient Descent

Advantages:

- Easy computation.
- Easy to implement.
- Easy to understand.

Disadvantages:

- May trap at local minima.
- Weights are changed after calculating gradient on the whole dataset. May take a long time to converge.
- Requires large memory to calculate gradient on the whole dataset.

Optimizers

We can use other optimizing strategies:

- Gradient Descent
- Stochastic Gradient Descent
- Stochastic Gradient Descent with momentum
- Mini-Batch Gradient Descent
- Adagrad
- RMSProp
- AdaDelta
- Adam

▸ Keras Optimizers

Stochastic Gradient Descent

- Instead of taking the whole dataset for each iteration in each iteration, we randomly shuffle the data and take a batch.

Stochastic Gradient Descent

- Instead of taking the whole dataset for each iteration in each iteration, we randomly shuffle the data and take a batch.
- The path took by the algorithm is full of noise as compared to the gradient descent algorithm.

Stochastic Gradient Descent

- Instead of taking the whole dataset for each iteration in each iteration, we randomly shuffle the data and take a batch.
- The path took by the algorithm is full of noise as compared to the gradient descent algorithm.
- Due to an increase in the number of iterations, the overall computation time increases. Still, the computation cost is still less than that of the gradient descent optimizer.

Stochastic Gradient Descent

- Instead of taking the whole dataset for each iteration in each iteration, we randomly shuffle the data and take a batch.
- The path took by the algorithm is full of noise as compared to the gradient descent algorithm.
- Due to an increase in the number of iterations, the overall computation time increases. Still, the computation cost is still less than that of the gradient descent optimizer.
- If the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

Adagrad: Adaptive gradient descent

- The adaptive gradient descent algorithm uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training.

Adagrad: Adaptive gradient descent

- The adaptive gradient descent algorithm uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training.
- The more the weights change, the least the learning rate changes.

Adagrad: Adaptive gradient descent

- The adaptive gradient descent algorithm uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training.
- The more the weights change, the least the learning rate changes.
- The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithmss, as it reaches convergence at a higher speed.

Adagrad: Adaptive gradient descent

- The adaptive gradient descent algorithm uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training.
- The more the weights change, the least the learning rate changes.
- The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithmss, as it reaches convergence at a higher speed.
- One downside of AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small.

Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.

Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.
- Adam optimizer updates the learning rate for each network weight individually.

Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.
- Adam optimizer updates the learning rate for each network weight individually.
- The Adam optimizers inherit the features of both Adagrad and RMSProp algorithms.

Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.
- Adam optimizer updates the learning rate for each network weight individually.
- The Adam optimizers inherit the features of both Adagrad and RMSProp algorithms.
- Instead of adapting learning rates based upon the first moment (mean), it also uses the second moment of the gradients (variance).

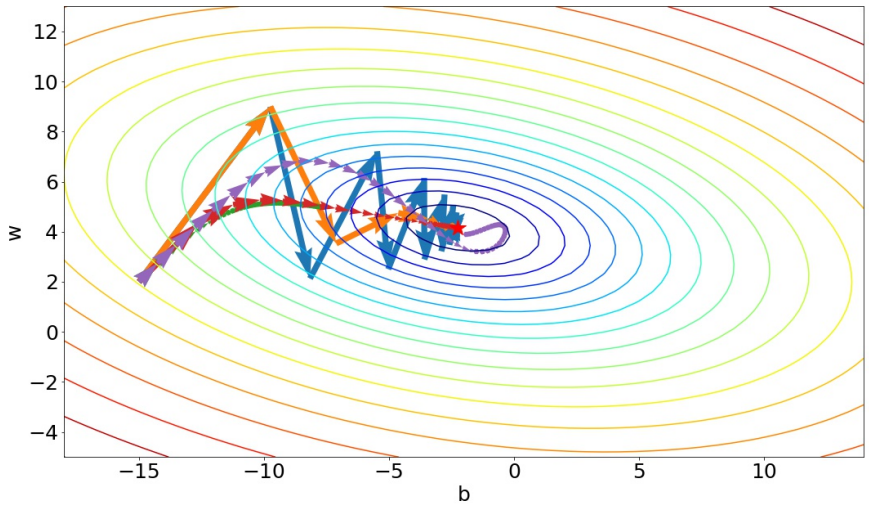
Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.
- Adam optimizer updates the learning rate for each network weight individually.
- The Adam optimizers inherit the features of both Adagrad and RMSProp algorithms.
- Instead of adapting learning rates based upon the first moment (mean), it also uses the second moment of the gradients (variance).
- It is often used as a default optimization algorithm. It has faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

Adam: ADAptive Moment estimation

- It is an extension of stochastic gradient descent.
- Adam optimizer updates the learning rate for each network weight individually.
- The Adam optimizers inherit the features of both Adagrad and RMSProp algorithms.
- Instead of adapting learning rates based upon the first moment (mean), it also uses the second moment of the gradients (variance).
- It is often used as a default optimization algorithm. It has faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.
- It tends to focus on faster computation time, it might not generalize the data well enough.

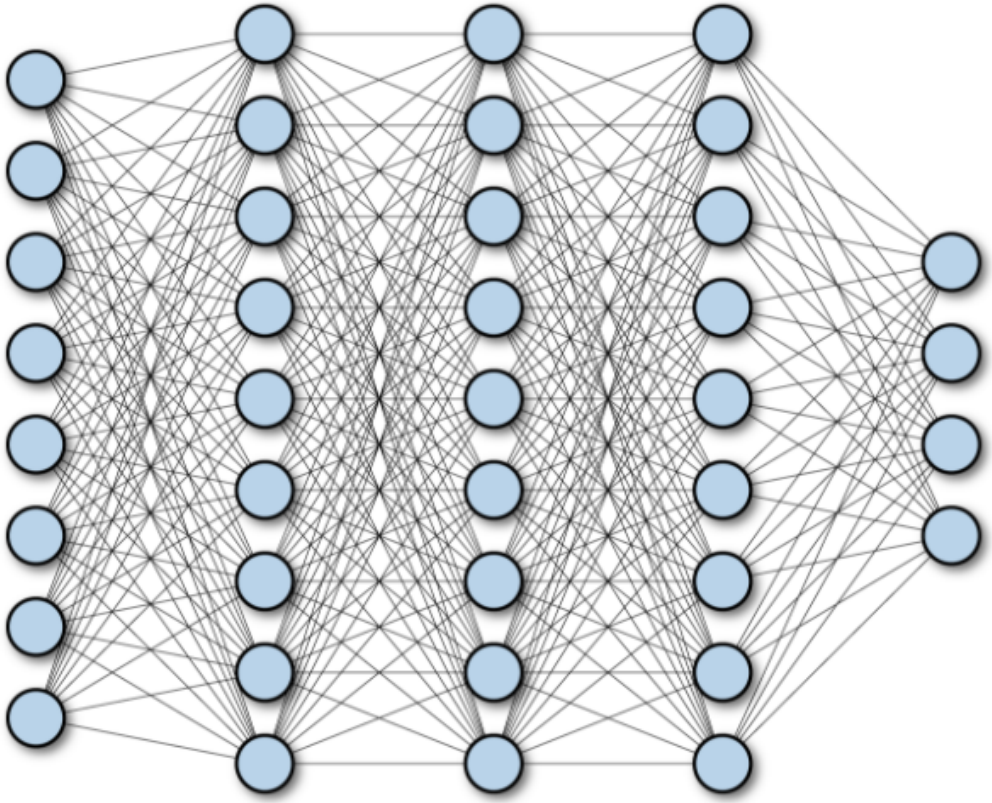
Different Optimizers



Visualizaciones, Several optimizers

Índice

Fully Connected Networks



A fully connected neural network consists of a series of fully connected layers that connect every neuron in one layer to every neuron in the other layer.

Advantages and Disadvantages

- The major advantage of fully connected networks is that they are no special assumptions needed to be made about the input.

Advantages and Disadvantages

- The major advantage of fully connected networks is that they are no special assumptions needed to be made about the input.
- While being structure agnostic makes fully connected networks very broadly applicable, such networks do tend to have weaker performance than special-purpose networks tuned to the structure of a problem space.

The Algorithm

The algorithm consists of three steps:

- First, present one or more training examples to the neural network: Feed-Forward.

Estos 3 pasos son una época.

The Algorithm

The algorithm consists of three steps:

- First, present one or more training examples to the neural network: Feed-Forward.
- Second, compare the output of the neural network to the desired value: Loss function.

Estos 3 pasos son una época.

The Algorithm

The algorithm consists of three steps:

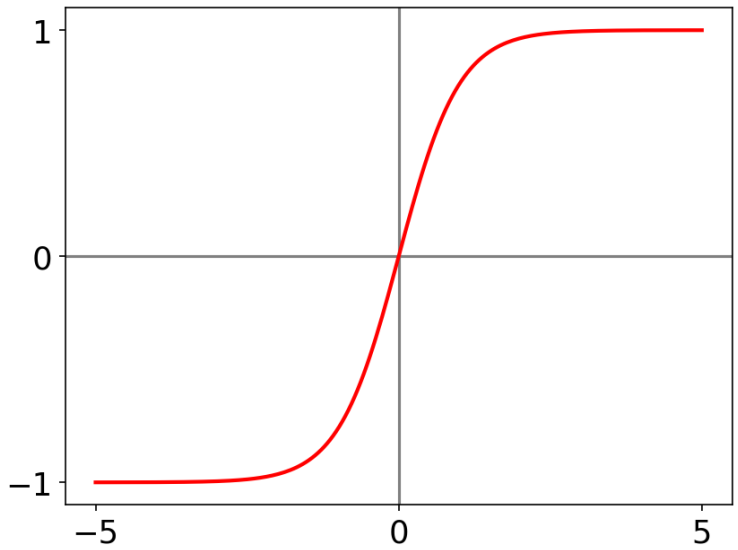
- First, present one or more training examples to the neural network: Feed-Forward.
- Second, compare the output of the neural network to the desired value: Loss function.
- Finally, adjust the weights to make the output get closer to the desired value using gradient descent: Back propagation.

Estos 3 pasos son una época.

Other Activation Functions: Hyperbolic Tangent

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \in (-1, 1)$$

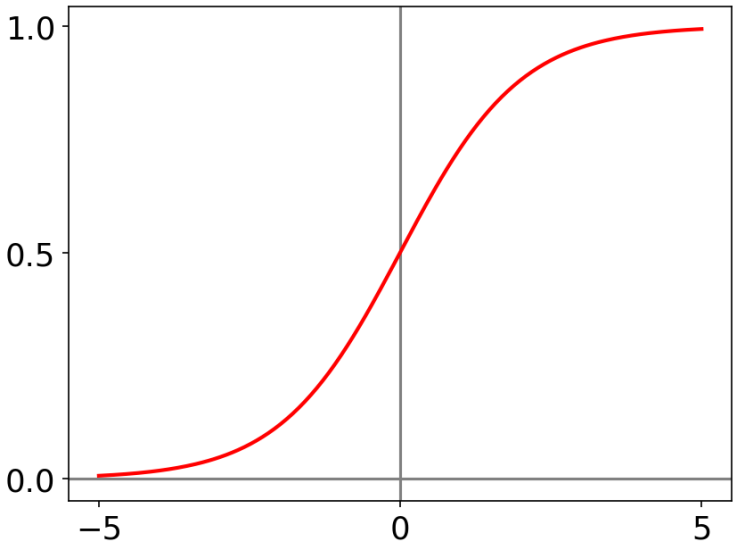
$$\tanh'(x) = 1 - \tanh^2(x)$$



It is often used in the hidden layer. It can also be used if the predicted value of a regression task has values in $(-1, 1)$.

Other Activation Functions: The Sigmoid

$$S(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$
$$S'(x) = S(x)(1 - S(x))$$



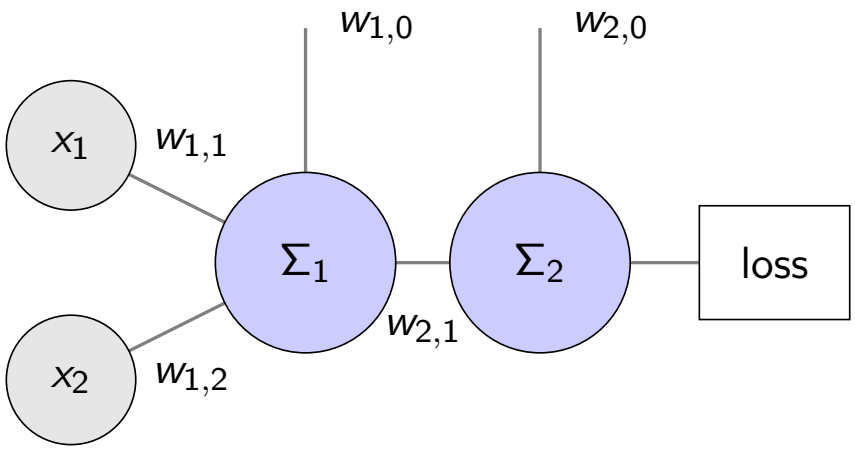
It is used when we need the output to be in $(0, 1)$. For example, in binary classification.

Choice of Activation Functions

There exist a large number of activation functions. Two popular choices are tanh and the logistic sigmoid function. When picking between the two, choose tanh for hidden layers and logistic sigmoid for the output layer.

▸ Keras Activation Functions

Example: Back-propagation



This neural network implements

$$f = S\left(w_{2,0} + w_{2,1}\tanh(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2)\right)$$

We use the MSE loss:

$$e(f) = \frac{1}{2}(y - f)^2.$$

Example: Back-propagation

- The error is given by

$$e(\mathbf{w}) = \frac{1}{2}\left(y - S\left(w_{2,0} + w_{2,1}\tanh(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2)\right)\right)^2$$

where $\mathbf{w} = (w_{1,0}, w_{1,1}, w_{1,2}, w_{2,0}, w_{2,1})$

- We write it as:

$$\begin{aligned} e(f) &= \frac{1}{2}(y - f)^2 \\ f(z_f) &= S(z_f) \\ z_f(w_{2,0}, w_{2,1}, g) &= w_{2,0} + w_{2,1}g \\ g(z_g) &= \tanh(z_g) \\ z_g(w_{1,0}, w_{1,1}, w_{1,2}) &= w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2 \end{aligned}$$

Example: Back-propagation

Compute the partial derivatives

$$\frac{\partial}{\partial w_{2,0}} e = -(y - f) S'(z_f)$$

$$\frac{\partial}{\partial w_{2,1}} e = -(y - f) S'(z_f) g$$

$$\frac{\partial}{\partial w_{1,0}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g)$$

$$\frac{\partial}{\partial w_{1,1}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g) x_1$$

$$\frac{\partial}{\partial w_{1,2}} e = -(y - f) S'(z_f) w_{2,1} \tanh'(z_g) x_2$$

Example: Back-propagation

Finally, we update the weights, via gradient descent

$$w_{2,0} \leftarrow w_{2,0} - \eta \frac{\partial e}{\partial w_{2,0}}$$

$$w_{2,1} \leftarrow w_{2,1} - \eta \frac{\partial e}{\partial w_{2,1}}$$

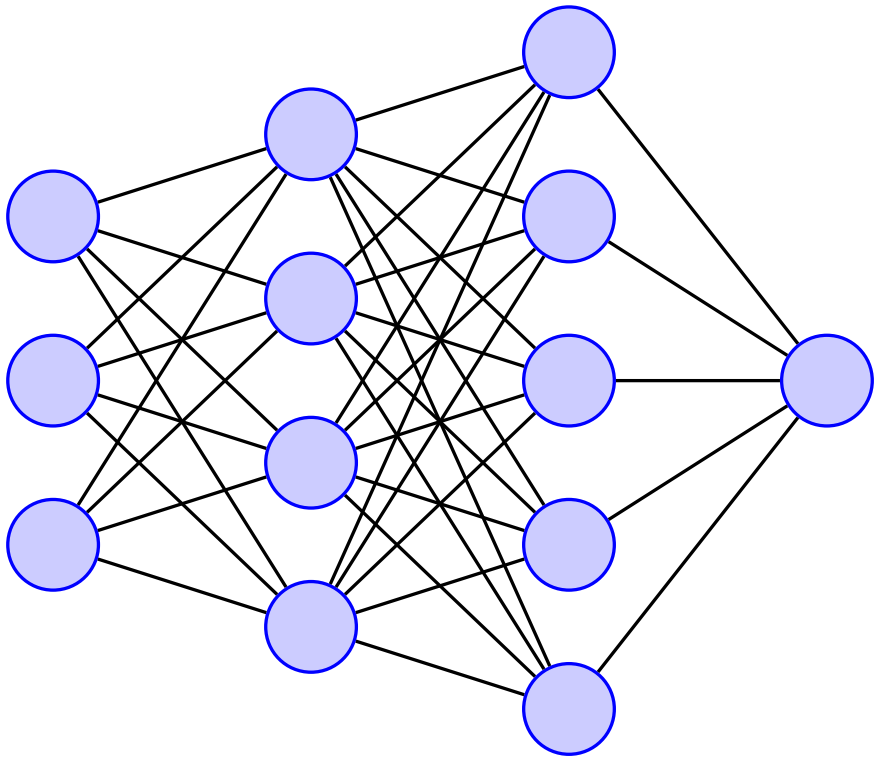
$$w_{1,0} \leftarrow w_{1,0} - \eta \frac{\partial e}{\partial w_{1,0}}$$

$$w_{1,1} \leftarrow w_{1,1} - \eta \frac{\partial e}{\partial w_{1,1}}$$

$$w_{1,2} \leftarrow w_{1,2} - \eta \frac{\partial e}{\partial w_{1,2}}$$

Regression

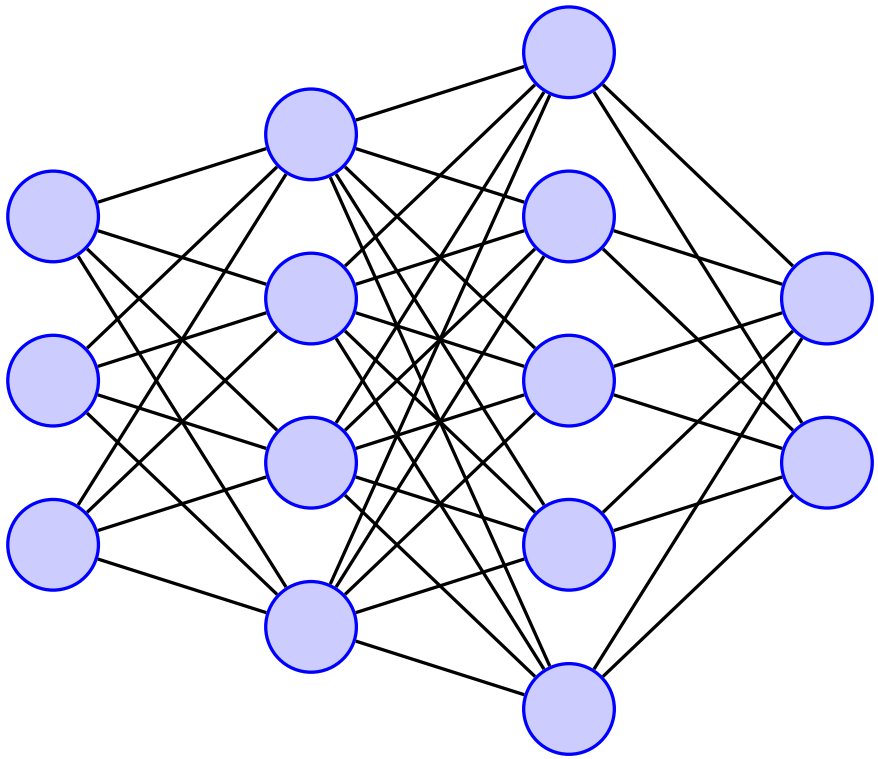
Si tenemos tres variables independientes y una variable dependiente



La función de activación en la capa de salida es *usualmente* `None` (la función identidad).

Multivariate Regression

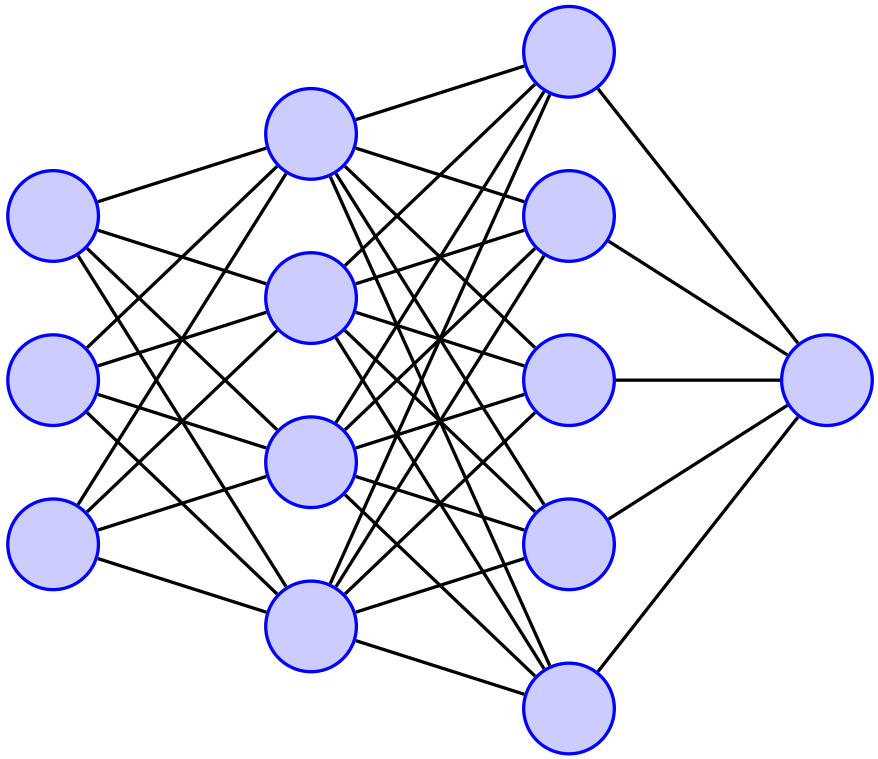
If we have three independent variables and two dependent variable



La función de activación en la capa de salida es *usualmente* `None` (la función identidad).

Binary Classification

If we have three independent variables and two classes

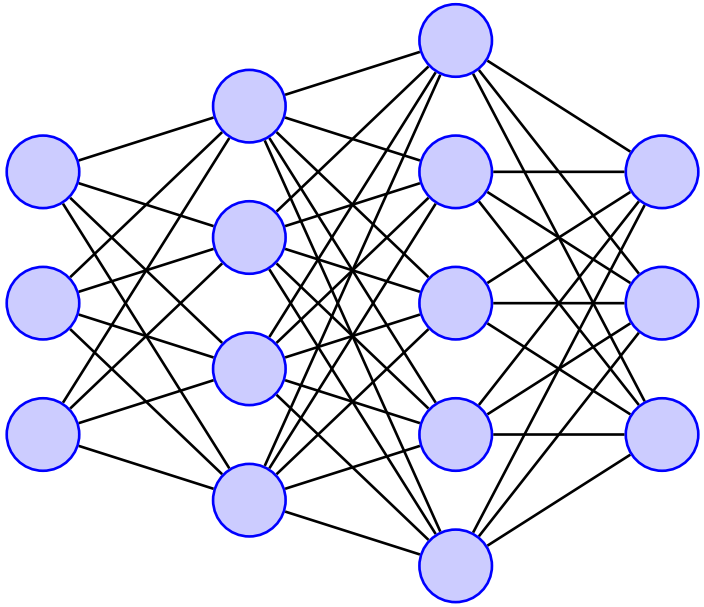


La función de activación en la capa de salida es la sigmoide

$$\sigma(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$

Multi-class Classification

Si tenemos tres, o más, clases



La función de activación en la capa de salida es softmax

$$\sigma\left(j;(x_1,...,x_n)\right)=\frac{e^{x_j}}{\sum_{i=1}^{n}e^{x_i}}\in(0,1)$$

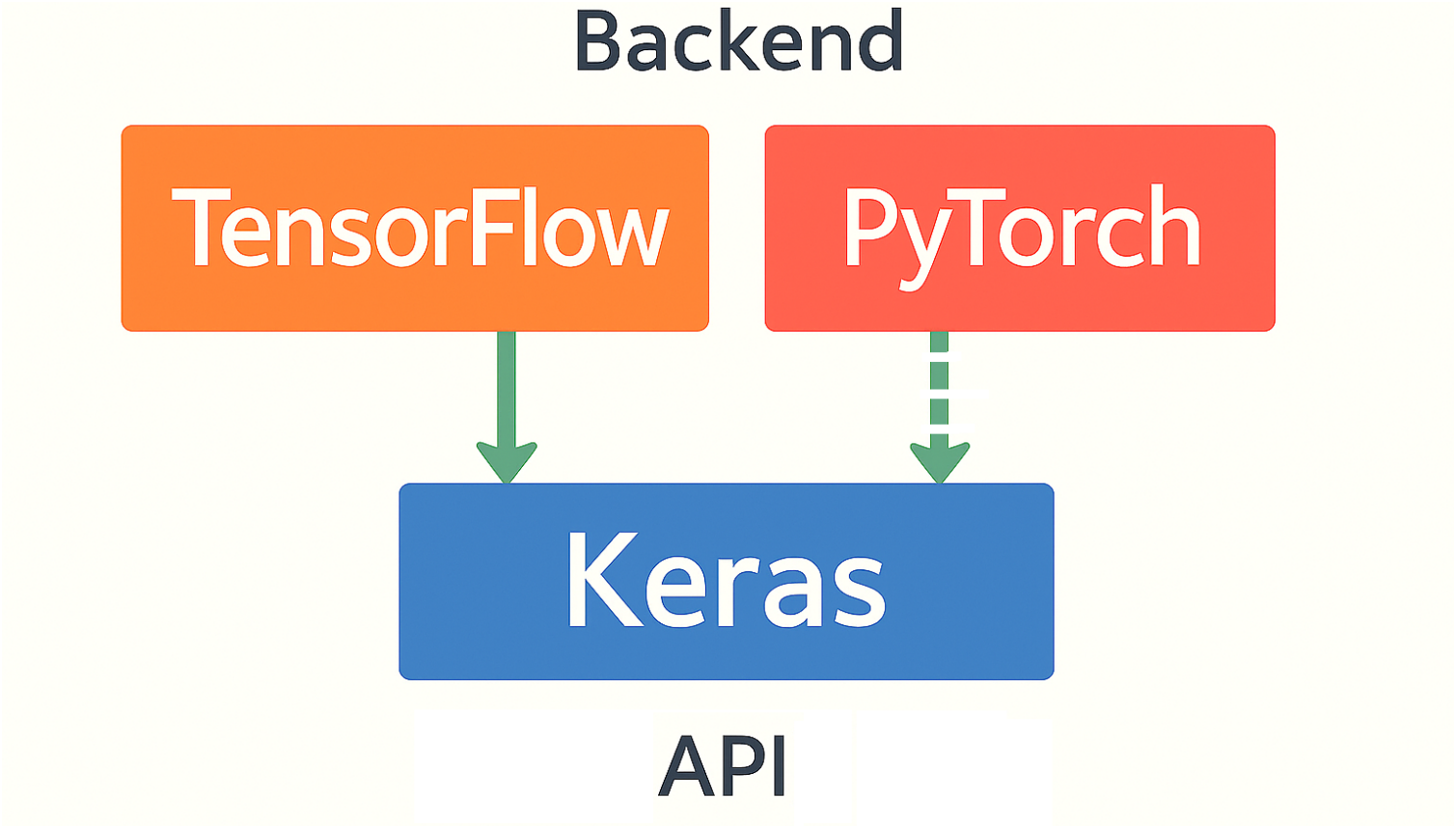
Índice

Deep Learning Libraries

Tensorflow

Pytorch

Keras, Tensorflow and Pytorch



Deep Learning Libraries

	Keras	**TensorFlow**	**PyTorch**
Nivel	Alto	Alto + Bajo	Alto + Bajo
Sintaxis	Simple	Híbrida	Imperativa
Datos	Backend	`tf.data`	`torch.data`
Debug	Fácil	Complejo	Muy fácil
Pre-trained models	Sí	Sí	Sí
Uso	Prototipado	Producción	Investigación

Comparación de los frameworks

```keras
from keras.models import Sequential
from keras.layers import Dense, Flatten

model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(8, activation='relu'),
    Dense(10, activation='softmax')
])
```

Keras

```tensorflow
import tensorflow as tf

# 1. Definir variables (pesos y biases)
W1 = tf.Variable(tf.random.normal([28*28, 8]))
b1 = tf.Variable(tf.zeros([8]))
W2 = tf.Variable(tf.random.normal([8, 10]))
b2 = tf.Variable(tf.zeros([10]))

# 2. Función forward
def model(x):
    x = tf.reshape(x, [-1, 28*28])  # Flatten
    x = tf.nn.relu(tf.matmul(x, W1) + b1)
    return tf.nn.softmax(tf.matmul(x, W2) + b2)
```

TensorFlow Puro

```pytorch
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.dense1 = nn.Linear(28*28, 8)
        self.dense2 = nn.Linear(8, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.dense1(x))
        return torch.softmax(self.dense2(x), dim=1)

model = MLP()
```

PyTorch

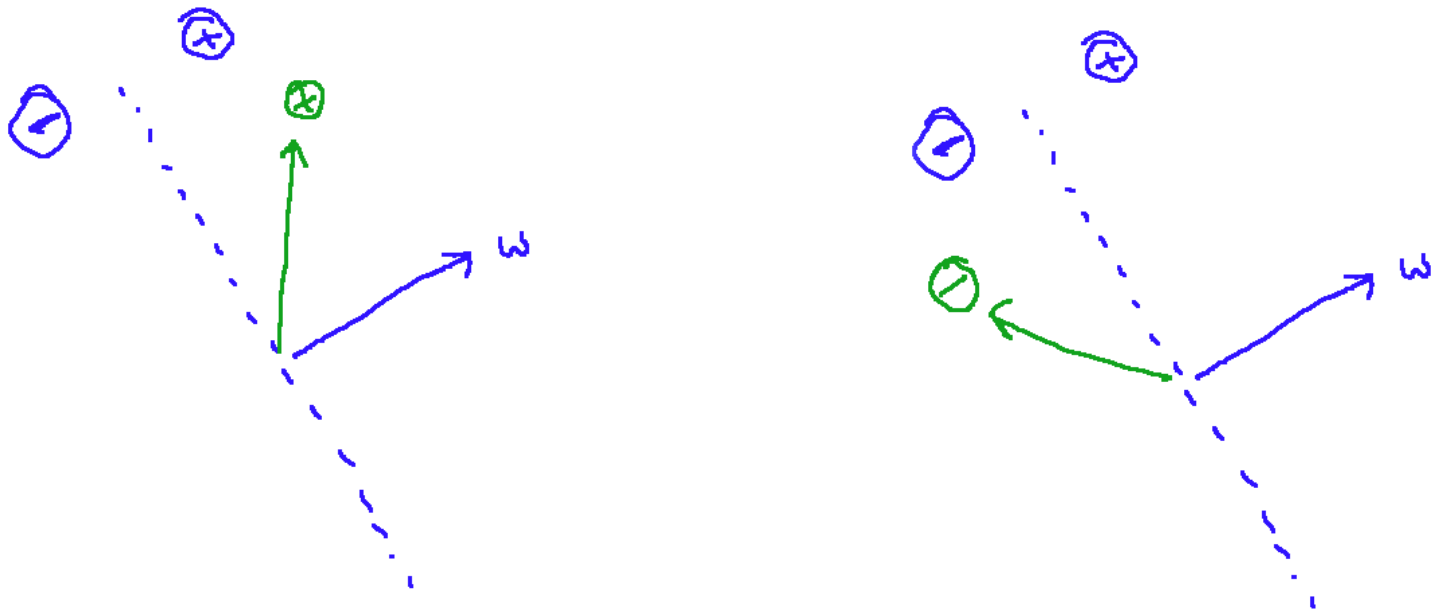
Índice

Casos del perceptron

We iterate over the training set

$$(x_1, y_1), (x_2, y_2) \cdots, (x_N, y_N).$$

a. If the point is well classified:

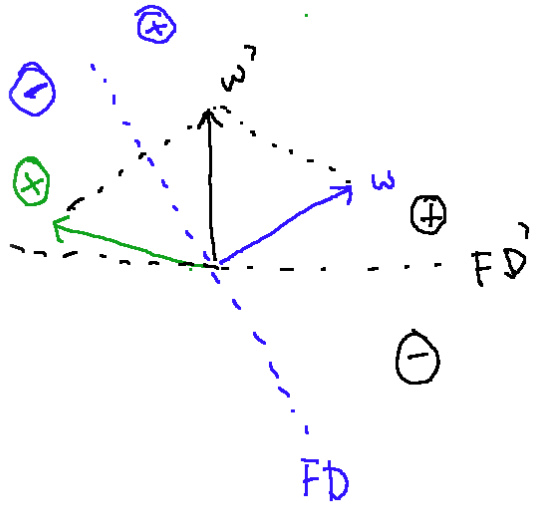


(a) If $y = +1$

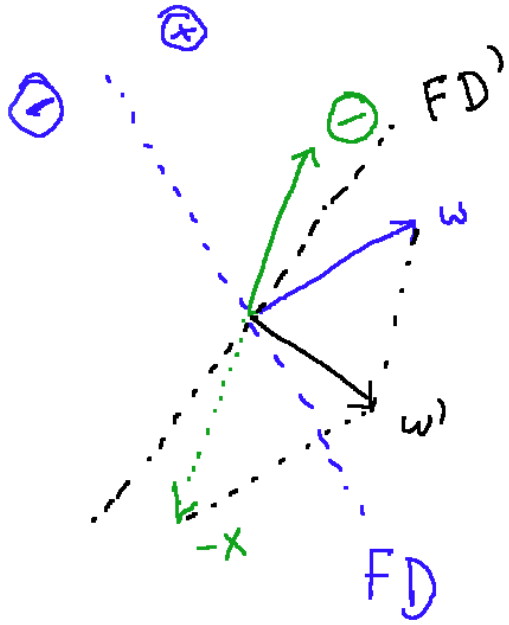
(b) If $y = -1$

Casos del perceptron

- b If the point is not well classified:



(a) If $y = +1$



(b) If $y = -1$

We update the vector w with $w + yx$

Regresar