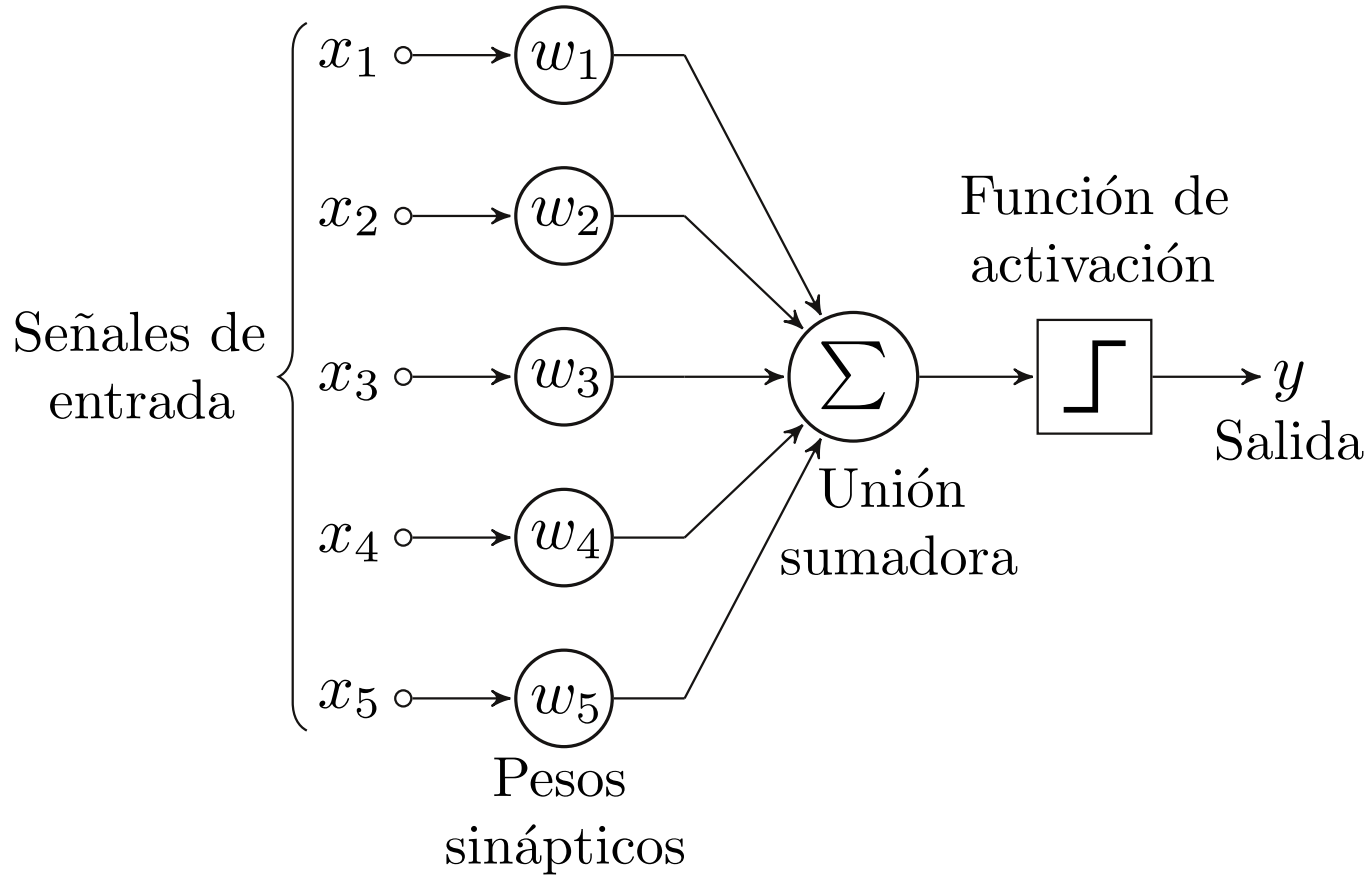
Programando un Perceptron en Python

Aprenderas a entender los múltiples algoritmos de machine learning. En este notebooks revisaremos cómo funciona un perceptron e implementaremos un ejemplo en python usando el conocido dataset Iris.

Autor:Mg. Rubén Quispe

EL PERCEPTRON

El Perceptron simple, también conocido una red neuronal de una sola capa (Single-Layer Neural Network), es un algoritmo de clasificación binaria creado por Frank Rosenblatt a partir del modelo neuronal de Warren McCulloch y Walter Pitts desarrollado en 1943.



La neurona recibe impulsos externos (x) que son considerados con distinta importancia o peso (w) en una función de activación (z). Si el estímulo agregado sobrepasa cierto umbral (θ), la neurona se activa.

Matemáticamente, definimos x como el vector de estímulos y w como el vector de pesos, ambos m dimensiones, y z como la función de activación.
El perceptron φ(z) se considera activo cuando su valor es mayor o igual al umbral θ o inactivo en cualquier otro caso. Formalmente esta es una función escalón puede ser escrita de la siguiente forma:
Si incorporamos θ a la expresión, definiendo w0=−θyx0=1 podemos escribir z=w0x0+w1x1+…+wmxm.

Donde:

η es la tasa de aprendizaje que es un valor entre 0 y 1.0
y(i) es el valor real
y^(i) es el valor de salida calculado (notar el sombrero en la y)
x(i)j es el valor de la muestra asociado

Para este ejemplo usaremos un conjunto de datos llamado Iris que es quizás el dataset más conocido en el mundo del machine learning. Este es un dataset multivariado que contiene 50 muestras de 3 especies de la flor Iris (iris setosa, Iris virgínica e Iris versicolor). En cada caso, se midió en centímetros largo y ancho del sépalo y del pétalo. Pueden descargar los datos desde el Machine Learning Repository de la Universidad de California

Los atributos disponibles en el dataset son los siguientes:

Largo del sépalo en cm
Ancho del sépalo en cm
Largo del pétalo en cm
Ancho del pétalo en cm
Clase (Iris setosa, Iris versicolor, Iris virgínica)
Una de las características de este dataset, es que algunos de sus atributos son linealmente separables, que es un requisito para el buen funcionamiento del perceptron. Que un set sea linealmente separable significa que una puede trazarse una recta (2D) o un plano (3D) y aislar cada clase de datos correctamente. Esto se puede observar claramente en el siguiente ejemplo.

Implementando la regla del perceptron en python

Puedes descargar el cuaderno de Jupyter https://jupyter.org/ y así ir siguiendo esta implementación paso a paso.

Primero partiremos implementando un clase en python. Esta clase define los siguientes métodos:

1. **__init__**: Define la tasa de aprendizaje del algoritmo y el numero de pasadas a hacer por el set de datos.
2. fit: Implementa la regla de aprendizaje, definiendo inicialmente los pesos en 0 y luego ajustándolos a medida que calcula/predice el valor para cada fila del dataset.
3. predict: Es la función escalón φ(z). Si el valor de z es mayor igual a 0, tiene por valor 1. En cualquier otro caso su valor es -1.
4. net_input: Es la implementación de la función de activación z. Si se fijan en el código, hace producto punto en los vectores x y w.

In [3]:

```
import numpy as np

class Perceptron:
    """Clasificador Perceptron basado en la descripción del libro
    "Python Machine Learning" de Sebastian Raschka.

    Parametros
    ----------

    eta: float
        Tasa de aprendizaje.
    n_iter: int
        Pasadas sobre el dataset.

    Atributos
    ---------
    w_: array-1d
        Pesos actualizados después del ajuste
    errors_: list
        Cantidad de errores de clasificación en cada pasada

    """
    def __init__(self, eta=0.1, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Ajustar datos de entrenamiento

        Parámetros
        ----------
        X:  array like, forma = [n_samples, n_features]
            Vectores de entrenamiento donde n_samples es el número de muestras y
            n_features es el número de carácteristicas de cada muestra.
        y:  array-like, forma = [n_samples].
            Valores de destino

        Returns
        -------
        self:   object
        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def predict(self, X):
        """Devolver clase usando función escalón de Heaviside.
        phi(z) = 1 si z >= theta; -1 en otro caso
        """
        phi = np.where(self.net_input(X) >= 0.0, 1, -1)
```

```
        return phi

    def net_input(self, X):
        """Calcular el valor z (net input)"""
        # z = w · x + theta
        z = np.dot(X, self.w_[1:]) + self.w_[0]
        return z
```

Ahora que tenemos la clase definida, debemos primero cargar los datos y luego entrenar el perceptron.
En este caso usaremos Iris, que contiene información de flores (largo de pétalo, sépalo, especie, etc.). Pueden leer más en wikipedia https://es.wikipedia.org/wiki/Conjunto_de_datos_flor_iris y descargarlo desde este link: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data. Yo ya tengo descargado el dataset en la carpeta datasets de este mismo repositorio (también pueden verlo ahí)

In [12]:

```
from sklearn.datasets import load_iris
import numpy as np
iris= load_iris()
print (iris)
```

```
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]), 'target_nam

es': array(['setosa', 'versicolor', 'virginica'], dtype='<U10'), 'DESCR':
'.. _iris_dataset:\n\nIris plants dataset\n--------------------\n\n**Data
Set Characteristics:**\n\n    :Number of Instances: 150 (50 in each of thr
ee classes)\n    :Number of Attributes: 4 numeric, predictive attributes a
nd the class\n    :Attribute Information:\n        - sepal length in cm\n
- sepal width in cm\n        - petal length in cm\n        - petal width i
n cm\n        - class:\n                - Iris-Setosa\n                - I
ris-Versicolour\n                - Iris-Virginica\n                \n    :
Summary Statistics:\n\n    ============== ==== ==== ======= ===== ========
============\n                    Min  Max   Mean    SD   Class Correlatio
n\n    ============== ==== ==== ======= ===== ====================\n    se
pal length:   4.3  7.9   5.84   0.83    0.7826\n    sepal width:    2.0
4.4   3.05   0.43   -0.4194\n    petal length:   1.0  6.9   3.76   1.76
0.9490  (high!)\n    petal width:    0.1  2.5   1.20   0.76    0.9565  (hi
gh!)\n    ============== ==== ==== ======= ===== ====================\n\n
:Missing Attribute Values: None\n    :Class Distribution: 33.3% for each o
```

In [16]:

```
iris.feature_names
```

Out[16]:

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

In [17]:

```
iris.DESCR
```

Out[17]:

```
'.. _iris_dataset:\n\nIris plants dataset\n--------------------\n\n**Data Se
t Characteristics:**\n\n    :Number of Instances: 150 (50 in each of three c
lasses)\n    :Number of Attributes: 4 numeric, predictive attributes and the
class\n    :Attribute Information:\n        - sepal length in cm\n        -
sepal width in cm\n        - petal length in cm\n        - petal width in cm
\n        - class:\n                - Iris-Setosa\n                - Iris-Ve
rsicolour\n                - Iris-Virginica\n                \n    :Summary
Statistics:\n\n    ============== ==== ==== ======= ===== ====================
==\n                    Min  Max   Mean    SD   Class Correlation\n    =====
========= ==== ==== ======= ===== ====================\n    sepal length:
4.3  7.9   5.84   0.83    0.7826\n    sepal width:    2.0  4.4   3.05   0.43
-0.4194\n    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)\n
petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)\n    =============
= ==== ==== ======= ===== ====================\n\n    :Missing Attribute Val
ues: None\n    :Class Distribution: 33.3% for each of 3 classes.\n    :Creat
or: R.A. Fisher\n    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
\n    :Date: July, 1988\n\nThe famous Iris database, first used by Sir R.A.
Fisher. The dataset is taken\nfrom Fisher\'s paper. Note that it\'s the same
as in R, but not as in the UCI\nMachine Learning Repository, which has two w
rong data points.\n\nThis is perhaps the best known database to be found in
the\npattern recognition literature.  Fisher\'s paper is a classic in the fi
eld and\nis referenced frequently to this day.  (See Duda & Hart, for exampl
e.)  The\ndata set contains 3 classes of 50 instances each, where each class
refers to a\ntype of iris plant.  One class is linearly separable from the o
ther 2; the\nlatter are NOT linearly separable from each other.\n\n.. topi
c:: References\n\n   - Fisher, R.A. "The use of multiple measurements in tax
onomic problems"\n     Annual Eugenics, 7, Part II, 179-188 (1936); also in
"Contributions to\n     Mathematical Statistics" (John Wiley, NY, 1950).\n
- Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysi
s.\n     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.\n
- Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System\n
Structure and Classification Rule for Recognition in Partially Exposed\n
Environments".  IEEE Transactions on Pattern Analysis and Machine\n     Inte
lligence, Vol. PAMI-2, No. 1, 67-71.\n   - Gates, G.W. (1972) "The Reduced N
earest Neighbor Rule".  IEEE Transactions\n     on Information Theory, May 1
972, 431-433.\n   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et a
l"s AUTOCLASS II\n     conceptual clustering system finds 3 classes in the d
ata.\n   - Many, many more ...'
```

In [18]:

```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica

    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is take
n
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field an
d
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to
a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

.. topic:: References

   - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analys
```

```
is.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transacti
ons
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

In [25]:

```
from sklearn.utils import shuffle
```

In [26]:

```
x= iris.data
```

In [27]:

```
y= iris.target
```

In [28]:

```
x,y = shuffle(x,y, random_state=0)
```

In [29]:

```
print(x)
```

```
[4.7 3.2 1.6 0.2]
[7.4 2.8 6.1 1.9]
[5.  3.3 1.4 0.2]
[6.3 3.4 5.6 2.4]
[5.7 2.8 4.1 1.3]
[5.8 2.7 3.9 1.2]
[5.7 2.6 3.5 1. ]
[6.4 3.2 5.3 2.3]
[6.7 3.  5.2 2.3]
[6.3 2.5 4.9 1.5]
[6.7 3.  5.  1.7]
[5.  3.  1.6 0.2]
[5.5 2.4 3.7 1. ]
[6.7 3.1 5.6 2.4]
[5.8 2.7 5.1 1.9]
[5.1 3.4 1.5 0.2]
[6.6 2.9 4.6 1.3]
[5.6 3.  4.1 1.3]
[5.9 3.2 4.8 1.8]
[6.3 2.3 4.4 1.3]
[5.1 3.5 1.4 0.2]
```

In [33]:

```
from sklearn.model_selection import train_test_split
```

In [34]:

```
x_train, x_test, y_train, y_test=train_test_split(x,y, test_size=0.3, random_state=42)
```

In [35]:

```
x_train.shape
```

Out[35]:

```
(105, 4)
```

In [36]:

```
y_train.shape
```

Out[36]:

```
(105,)
```

In [42]:

```
import pandas as pd
```

Código de perceptrón de una sola capa
Ahora que entendemos bien cómo funcionan los perceptrones, demos un paso más y solidifiquemos las matemáticas en código. Usaremos principios orientados a objetos y crearemos una clase. Para construir nuestro perceptrón, necesitamos saber cuántas entradas hay para crear nuestro vector de peso. La razón por la que agregamos uno al tamaño de entrada es para incluir el sesgo en el vector de peso.

In [47]:

```
import numpy as np

class Perceptron(object):
    """Implements a perceptron network"""
    def __init__(self, input_size):
        self.W = np.zeros(input_size+1)
```

También necesitaremos implementar nuestra función de activación. Simplemente podemos devolver 1 si la entrada es mayor o igual a 0 y 0 de lo contrario.

In [48]:

```
def activation_fn(self, x):
    return 1 if x >= 0 else 0
```

Finalmente, necesitamos una función para ejecutar una entrada a través del perceptrón y devolver una salida. Convencionalmente, esto se llama predicción. Agregamos el sesgo en el vector de entrada. Entonces podemos simplemente calcular el producto interno y aplicar la función de activación.

In [49]:

```
def predict(self, x):
    x = np.insert(x, 0, 1)
    z = self.W.T.dot(x)
    a = self.activation_fn(z)
    return a
```

Código de algoritmo de aprendizaje de perceptrón
Con la regla de actualización en mente, podemos crear una función para seguir aplicando esta regla de actualización hasta que nuestro perceptrón pueda clasificar correctamente todas nuestras entradas. Necesitamos seguir iterando a través de nuestros datos de entrenamiento hasta que esto suceda; Una época es cuando nuestro perceptrón ha visto todos los datos de entrenamiento una vez. Por lo general, ejecutamos nuestro algoritmo de aprendizaje para varias épocas.
Antes de codificar el algoritmo de aprendizaje, necesitamos hacer algunos cambios en nuestra función init para agregar la tasa de aprendizaje y el número de épocas como entradas.

In [50]:

```
def __init__(self, input_size, lr=1, epochs=10):
    self.W = np.zeros(input_size+1)
    # add one for bias
    self.epochs = epochs
    self.lr = lr
```

Ahora podemos crear una función, entradas dadas y salidas deseadas, ejecutar nuestro algoritmo de aprendizaje perceptrón. Seguimos actualizando los pesos durante varias épocas e iteramos a través de todo el conjunto de entrenamiento. Insertamos el sesgo en la entrada al realizar la actualización de peso. Entonces podemos crear nuestra predicción, calcular nuestro error y realizar nuestra regla de actualización.

In [51]:

```
def fit(self, X, d):
    for _ in range(self.epochs):
        for i in range(d.shape[0]):
            y = self.predict(X[i])
            e = d[i] - y
            self.W = self.W + self.lr * e * np.insert(X[i], 0, 1)
```

El código completo de nuestro perceptrón se muestra a continuación.

In [52]:

```
class Perceptron(object):
    """Implements a perceptron network"""
    def __init__(self, input_size, lr=1, epochs=100):
        self.W = np.zeros(input_size+1)
        # add one for bias
        self.epochs = epochs
        self.lr = lr

    def activation_fn(self, x):
        #return (x >= 0).astype(np.float32)
        return 1 if x >= 0 else 0

    def predict(self, x):
        z = self.W.T.dot(x)
        a = self.activation_fn(z)
        return a

    def fit(self, X, d):
        for _ in range(self.epochs):
            for i in range(d.shape[0]):
                x = np.insert(X[i], 0, 1)
                y = self.predict(x)
                e = d[i] - y
                self.W = self.W + self.lr * e * x
```

¡Ahora que tenemos nuestro perceptrón codificado, podemos intentar darle algunos datos de entrenamiento y ver si funciona! Un conjunto fácil de datos para dar es la puerta AND. Aquí hay un conjunto de entradas y salidas.

In [54]:

```
if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    d = np.array([0, 0, 0, 1])

    perceptron = Perceptron(input_size=2)
    perceptron.fit(X, d)
    print(perceptron.W)
```

```
[-3.  2.  1.]
```

¡En solo unas pocas líneas, podemos comenzar a usar nuestro perceptrón! Al final, imprimimos el vector de peso. Usando los datos de la compuerta AND, deberíamos obtener un vector de peso de [-3, 2, 1]. Esto significa que el sesgo es -3 y los pesos son 2 y 1 parax_1 y x_2, respectivamente.

In [55]:

```
if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    d = np.array([0, 0, 0, 1])

    perceptron = Perceptron(input_size=2)
    perceptron.fit(X, d)
    print(perceptron.W)
```

```
[-3.  2.  1.]
```

Para verificar que este vector de peso sea correcto, podemos intentar pasar por algunos ejemplos. Si ambas entradas son 0, la preactivación será -3 + 0 * 2 + 0 * 1 = -3. ¡Al aplicar nuestra función de activación, obtenemos 0, que es exactamente 0 Y 0! También podemos probar esto para otras puertas. Tenga en cuenta que este no es el único vector de peso correcto. Técnicamente, si existe un solo vector de peso que puede separar las clases, existe un número infinito de vectores de peso. El vector de peso que obtengamos depende de cómo inicializamos el vector de peso.

Para resumir, los perceptrones son el tipo más simple de red neuronal: toman una entrada, ponderan cada entrada, toman la suma de las entradas ponderadas y aplican una función de activación. Como fueron modelados a partir de neuronas biológicas por Frank Rosenblatt, toman y producen solo valores binarios. En otras palabras, podemos realizar una clasificación binaria usando perceptrones. Una limitación de los perceptrones es que solo pueden resolver problemas linealmente separables. En el mundo real, sin embargo, muchos problemas son en realidad linealmente separables. Por ejemplo, podemos usar un perceptrón para imitar una compuerta AND u OR. Sin embargo, dado que XOR no es linealmente separable, no podemos usar perceptrones de una capa para crear una puerta XOR. El algoritmo de aprendizaje de perceptrón se ajusta a la intuición de Rosenblatt: inhibir si una neurona se dispara cuando no debería, y excita si una neurona no se dispara cuando debería. Podemos tomar ese principio simple y crear una regla de actualización para nuestros pesos para dar a nuestro perceptrón la capacidad de aprender.

Los perceptrones son la base de las redes neuronales, por lo que tener una buena comprensión de ellos ahora será beneficioso al aprender sobre redes neuronales profundas.