Single-layer Perceptron

The perceptron machine learning algorithm was devised in 1957 by Frank Rosenblatt at Cornell University. It is executed here as a **supervised learning algorithm**, meaning a desired, or known, output exists. We train the perceptron to do our bidding based on how closely its guesses at each iteration correspond to the known output.

We can think of the perceptron as a group of n input neurons that communicate at n synapses with a single output neuron. Each input neuron receives an input x_i, and the affect of this stimulation on the output neuron depends on the strength w_i of the synaptic connection between them. Training the perceptron involves changing these synaptic weights over many iterations to arrive at the set of weights $w_1 ... w_n$ producing an output o that matches our desired output, y.

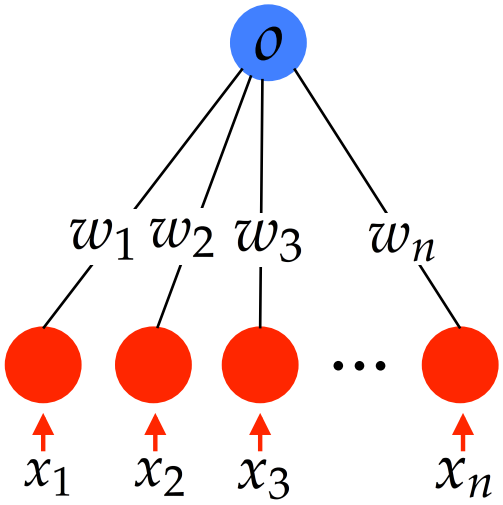


Figure 1: Synaptic weights $w_1 ... w_n$ updated upon every iteration determine how much each input ($x_1 ... x_n$) contributes to the output (o) of the perceptron.

Our training routine will consider binary inputs and outputs. Specifically, each input neuron is either stimulated ($x_i = 1$) or not ($x_i = 0$), and the output (a weighted sum of binary inputs) is either 1 or 0.

You may be wondering how an output that is a weighted sum of 1's and 0's is restricted to being either 1 or 0. We appeal to basic principles of neuroscience and realize that neural firing is an all-or-none event. In other words, a neuron fires ($o = 1$) if the weighted sum of inputs exceeds a given threshold, and doesn't fire ($o = 0$) if it does not reach threshold. To simplify matters, we set this threshold to 0 and pass our weighted sum θ to a threshold function $f(\theta)$ which returns a 1 if $\theta > 0$ and a 0 if $\theta \leqslant 0$.

To train a perceptron, you are coding a learning algorithm that governs the evolution of synaptic weights over time. How do these weights change?

The Perceptron Learning Algorithm

At each iteration j, the perceptron calculates the output based on the current input pattern (a vector of $x_1 ... x_n$ values) and weights (a vector of $w_1 ... w_n$ values), then updates the weights based on how much its output o differs from the desired output y. The algorithm involves three basic steps at each j:

The current output is given by

$$o^j = f(\sum_{i=1}^{n} w_i x_i) \text{ where } f(\theta) = \begin{cases} 1, & \text{if } \theta > 0. \\ 0, & \text{if } \theta \leqslant 0. \end{cases} \tag{1}$$

The difference between o^j and desired output y can be calculated as

$$d^j = y - o^j \tag{2}$$

Each weight is then updated by ΔW_i where

$$\Delta W_i = \ell d^j x_i^j \text{ for } i = 1...n, w_i = w_i + \Delta w_i \tag{3}$$

with ℓ as the learning rate that scales changes in weight.

Why do we need a learning rate? Imagine a scenario where the input neurons received an abnormal stimulation pattern; we wouldn't want these inputs to affect the weights to such an extent that it would subsequently take many normal inputs to get back on track. Setting $\ell = 0.01$ and $j_{max} = 1000$ should keep things in check.

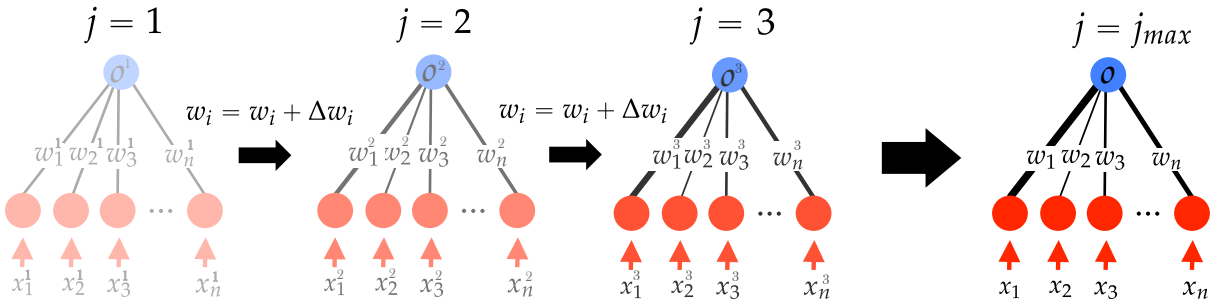


Figure 2: We see in this example that weights `[w1...wn]` change `for j = 1:jmax` as the perceptron arrives at its final, well-trained form. Input neuron 1 becomes most influential and input neuron 2 becomes least influential (weight w_1 is greatest, w_2 is smallest). The strengths of synapses 3 and 4, w_3 and w_4, also evolve over time.

We will implement the perceptron learning algorithm to solve three classification problems: an AND problem, an OR problem, and an XOR problem.

Consider a perceptron where $n = 2$, i.e., where there are only two input neurons. In each of Tables 1 through 3 below, we consider the desired output y for 4 combinations of inputs. Training a perceptron to arrive at $o = y = 1$ would involve, `for j=1:jmax`, handing each of the four combinations of inputs `[x1 x2]` to the perceptron learning algorithm.

The AND Problem

We seek a vector of weights `[w1...wn]` such that the output neuron fires ($o = 1$) when all input neurons are stimulated (x_1 *and* $x_2 = 1$).

x_1	x_2	y
1	1	1
1	0	0
0	1	0
0	0	0

Table 1: AND with two inputs

The OR Problem

We seek a vector of weights `[w1...wn]` such that the output neuron fires ($o = 1$) when at least one of the input neurons is stimulated ($x_1 = 1$ *or* $x_2 = 1$, or both).

x_1	x_2	y
1	1	1
1	0	1
0	1	1
0	0	0

Table 2: OR with two inputs

The XOR Problem

We seek a vector of weights `[w1...wn]` such that the output neuron fires ($o = 1$) when exactly one of the input neurons is stimulated (either $x_1 = 1$ or $x_2 = 1$, but not both).

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

Table 3: XOR with two inputs

Project: Perceptron Classification

Your project consists of writing two functions, a driver `pdrive` which calls a function `perceptron` in the form `[w, w0] = perceptron(x,y)`, where `x` is a set of `length(x)` different inputs to `n` input neurons, and `y` is the vector of *desired* outputs corresponding to each of these input patterns. In `perceptron` you will code the perceptron learning algorithm such that, when the perceptron is handed this set of inputs, its output `o` matches the desired ouput `y`. As proof of its success, the function `perceptron` will return to you the set of optimal weights `w = [w1...wn]` that produce an output corresponding to `y` for each input pattern `x`.

`pdrive` will run `perceptron` on the AND, OR, and XOR problems considered above. Tables 1, 2, and 3 hint at your input structure. For example, to run the perceptron on the AND problem, you would use:

```
x =  1 1     y = 1
     1 0         0          [w, w0] = perceptron(x,y)
     0 1         0
     0 0         0
```

You will notice that `x` is the same for each classification problem, whereas `y` changes based on whether we want to perform an AND, OR, or XOR operation on the inputs. Furthermore, each row of `x` presents a set of inputs `x1...xn` with desired output in the corresponding row of `y`. For the perceptron to arrive at weights that perform the correct operation on any of the inputs, your training routine will need to update weights `w1...wn` across successive presentations of the different input vectors, of which there are `length(x)` in number (in our example, `length(x) = 4`). Your learning algorithm should accomodate an `x` of arbitrary size, and pass each row of x through the learning algorithm `jmax = 1000` times. You may set all weights initially to 1.

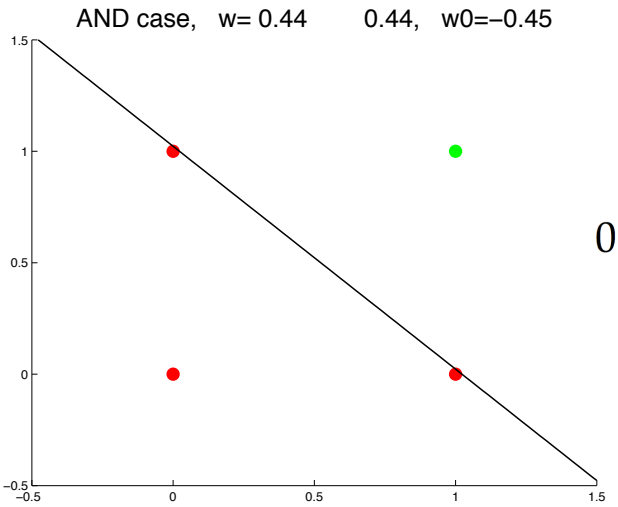
pdrive

Finally, we seek a way to visualize these solutions, and ask ourselves: what would happen if we handed our trained perceptron an input vector that was not contained in the set of training inputs `x`?

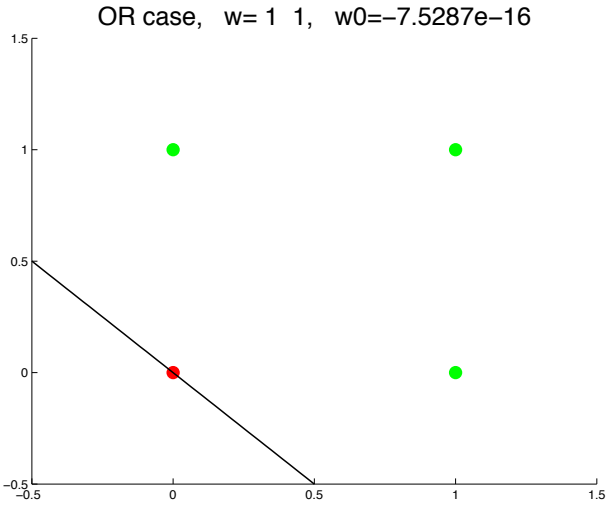
In fact, we can answer this question using only the set of optimal weights returned by our perceptron. We can also use these weights to display the perceptron's output for all potential inputs `[x1 x2]`. To do this, recall that the perceptron's output is returned from a thresholding function, which compares the weighted sum of inputs, $w_1 x_1 + w_2 x_2 = \theta$, to the threshold 0, and returns a 1 if $\theta > 0$ and a 0 if $\theta \leqslant 0$. We consider the line

$$w_1 x_1 + w_2 x_2 - \theta = 0 \, ,$$

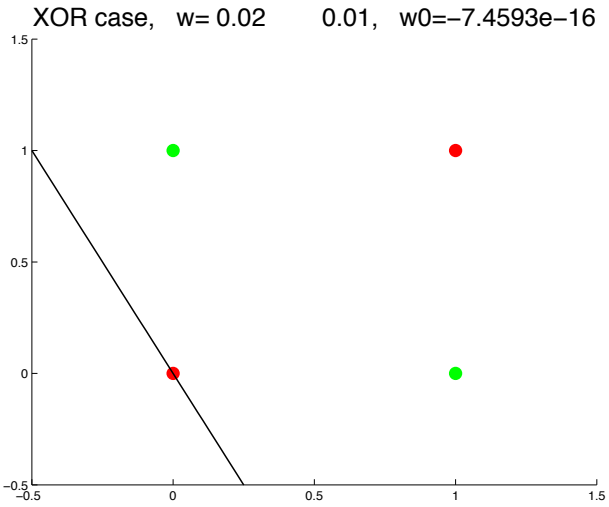
and realize that the perceptron will return a 1 for all point pairs (x_1, x_2) above this line, and a 0 for all pairs of inputs below this line.



$$0.44x_1 + 0.44x_2 - 0.45 = 0$$



$$1x_1 + 1x_2 - 0 = 0$$



$$0.02x_1 + 0.01x_2 - 0 = 0$$

The above plots are those that you will create in `pdrive` after calling `perceptron` on the AND, OR, and XOR problems. `Scatter` with appropiate color choices (and a markersize of at least 800) paints the point with coordinates `(x1,x2)` red if the perceptron will return a 0 for that input pattern, and green if the perceptron will return a 1. The four points plotted on each set of axes are those defined by `x`, with colors specified by `y`. For example, to plot only the point `(1,1)` with a large marker in green, one could execute:

```
scatter(1,1,800,'g.')
```

You will notice in the AND and OR cases that we are able to draw a line dividing input pairs into two output classes, $\{1, 0\}$. Yet in the XOR case, we examine the scattering of four colored inputs and recognize that there is no way to draw a line dividing the plane into the appropriate sections, so the perceptron cannot solve the XOR problem. Formally we refer to the classes in the XOR case as being **not linearly separable**.

Thus we recognize that the perceptron can only solve linearly separable problems.

Our work here is almost done. However, we have yet to establish how θ is chosen to draw the line $w_1x_1 + w_2x_2 - \theta = 0$. Your perceptron will find θ automatically if you create another weight, w_0, assigned to an input that is always 1. We call this w_0 the **bias bit**, which is the second output of the `perceptron` function.

In `perceptron`, you will need to add this bias bit to the vector of weights before running the perceptron algorithm, and afterwards strip off the bias weight from the computed `[w1...wn+1]` to return `[w w0]`. How do you handle the extra input that is always one? Update `x` within `perceptron` by adding an extra column of ones on the right.

Your work will be graded as follows:

```
 5 pts for header containing detailed usage of each function
 5 pts for further comments in code
 3 pts for indentation
15 pts for correct perceptron function
10 pts for correct pdrive
12 pts for reproducing the titled figures on the previous page corresponding
to AND, OR, and XOR problems
```