Artificial Neural Networks

Course-3

Gavrilut Dragos

rev 1.1

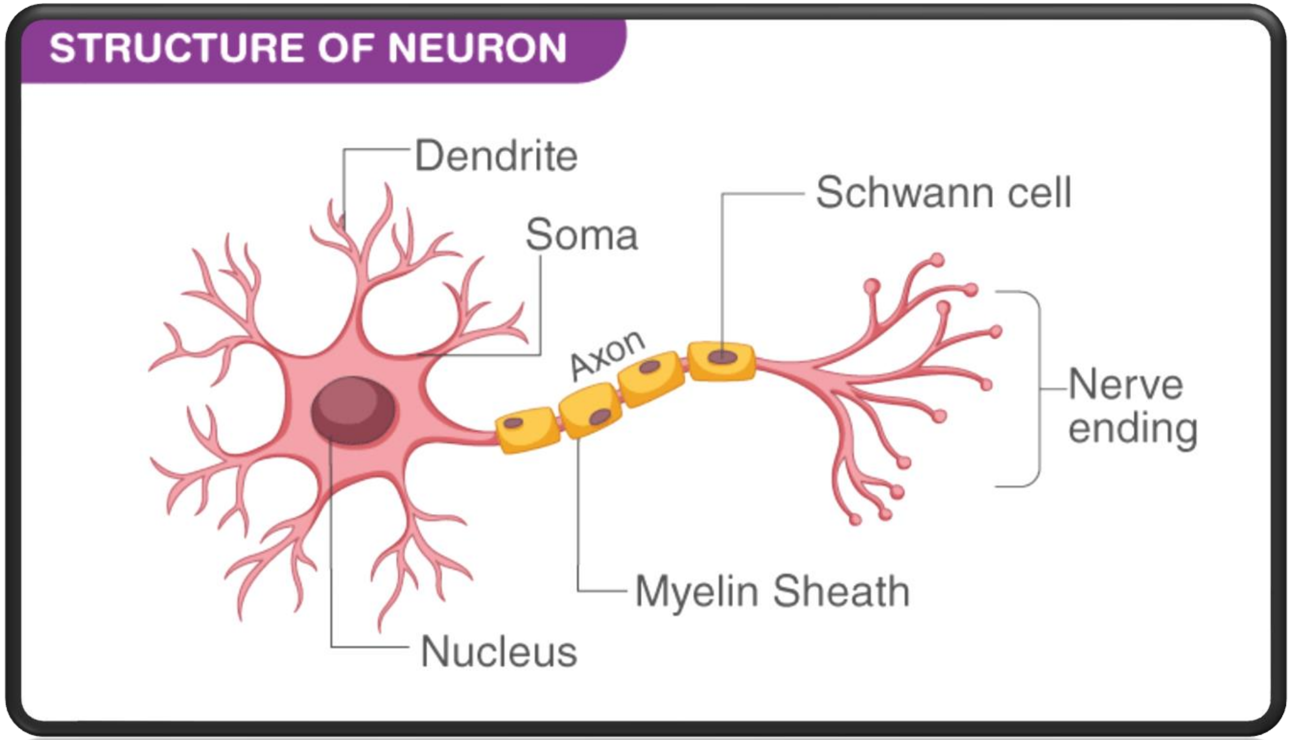
AGENDA FOR TODAY

- Perceptron Algorithm
  - History
  - Algorithm
  - Training
  - Demo
  - Batch Training
  - Adaline perceptron

Perceptron history

Perceptron history

- Proposed by Frank Rosenblatt in 1957
- Based on a model proposed by Warren McCulloch and Walter Pits in 1943
- It follows the way a simple neuron works

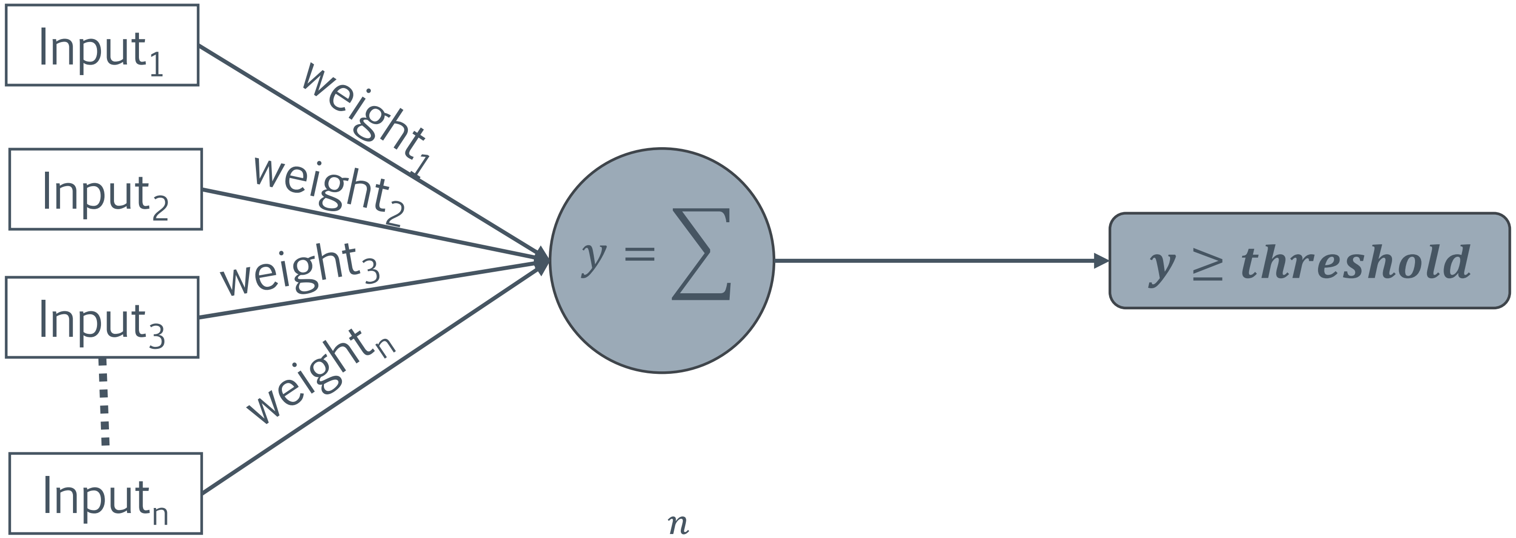


Picture from: https://cdn1.byjus.com/wp-content/uploads/2020/02/STRUCTURE-OF-NEURON.png

Algorithm

Algorithm

The perceptron algorithm can be described in several ways (the common one is to see it as the sum of products between the input data and various weights).



$$y = \sum_{i=1}^{n} Input_i \times weight_i \,, y \geq threshold$$

Algorithm

A more mathematical way to see this, is with some vectors and a dot product.

$$input = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}, weight = \begin{bmatrix} weight_1 \\ weight_2 \\ \vdots \\ weight_n \end{bmatrix}, y = input \cdot weight, y \geq threshold$$

Algorithm

A more mathematical way to see this, is with some vectors and a dot product.

$$input = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}, weight = \begin{bmatrix} weight_1 \\ weight_2 \\ \vdots \\ weight_n \end{bmatrix}, y = input \cdot weight, y \geq threshold$$

The previous equation can be written in a different way (to avoid using the threshold as a separate value).

$$input = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \\ 1 \end{bmatrix}, weight = \begin{bmatrix} weight_1 \\ weight_2 \\ \vdots \\ weight_n \\ -threshold \end{bmatrix}, y = input \cdot weight, y \geq 0$$

Algorithm

This form of the perceptron algorithm (that implies that a dot product must be bigger than 0) addresses the scenario with a binary label (a label can have two possible values – for example a label can indicate the presence or absence of a class).

Let’s write the perceptron equations for inputs with 2 elements:

$$input = \begin{bmatrix} input_1 \\ input_2 \\ 1 \end{bmatrix}, weight = \begin{bmatrix} weight_1 \\ weight_2 \\ -threshold \end{bmatrix}, result = input \cdot weight,$$

$$result = \ input_1 \times weight_1 + input_2 \times weight_2 - threshold$$

Algorithm

This form of the perceptron algorithm (that implies that a dot product must be bigger than 0) addresses the scenario with a binary label (a label can have two possible values – for example a label can indicate the presence or absence of a class).

Let's ... **ents:**

Assuming $input = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ and $weight = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ the resulting equation will be

ax+by+c = 0, the line equation in 2 dimensions.

$[\ 1\ ]$ ... $reshold]$

$$\boxed{result = input_1 \times weight_1 + input_2 \times weight_2 - threshold}$$

Algorithm

This means that the perceptron algorithm models a hyper-plane equation in "n" dimensions. For 2 dimensions this is one of the forms of a line equation.

So ... assuming we work on two dimension, the perceptron equation will be:

$$input = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, weight = \begin{bmatrix} a \\ b \\ c \end{bmatrix},$$

$$input \cdot weight > 0, or\ ax + by + c > 0$$

Algorithm

This means that the perceptron algorithm models a hyper-plane equation in "n" dimensions. For 2 dimensions this is one of the forms of a line equation.

So … assuming we work on two dimension, the perceptron equation will be:

If **ax+by+c>=0**, then if **a=b=c=0** this equation will always be ***true*** and as such we will not be able to classify anything.

So why use >0 in this inequality and not **>=0** ?

$$input \cdot weight \boxed{> 0}, or\ ax + by + c \boxed{> 0}$$

Algorithm

This means that the perceptron algorithm models a hyper-plane equation in “n” dimensions. For 2 dimensions this is one of the forms of a line equation.

So … assuming we work on two dimension, the perceptron equation will be:

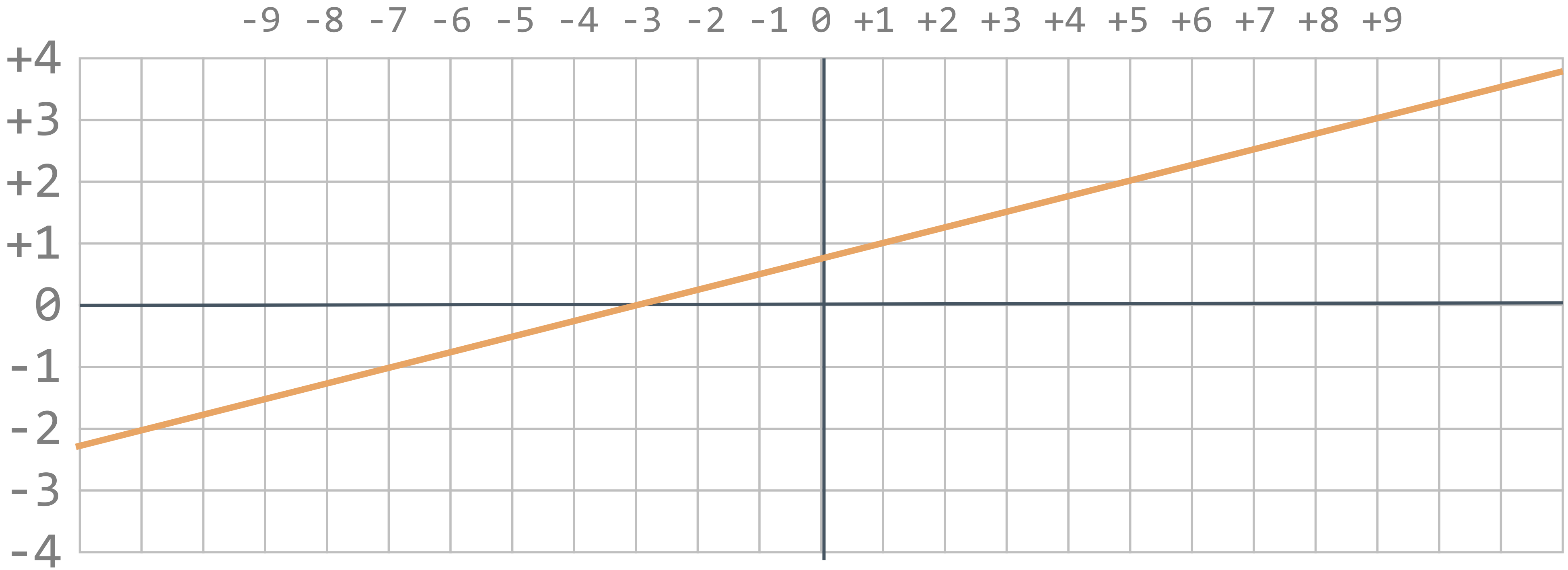
$input \quad [x \quad [a$

So, what is the purpose of that “>0” in this equation ?

$$input \cdot weight \boxed{> 0}, or\ ax + by + c \boxed{> 0}$$

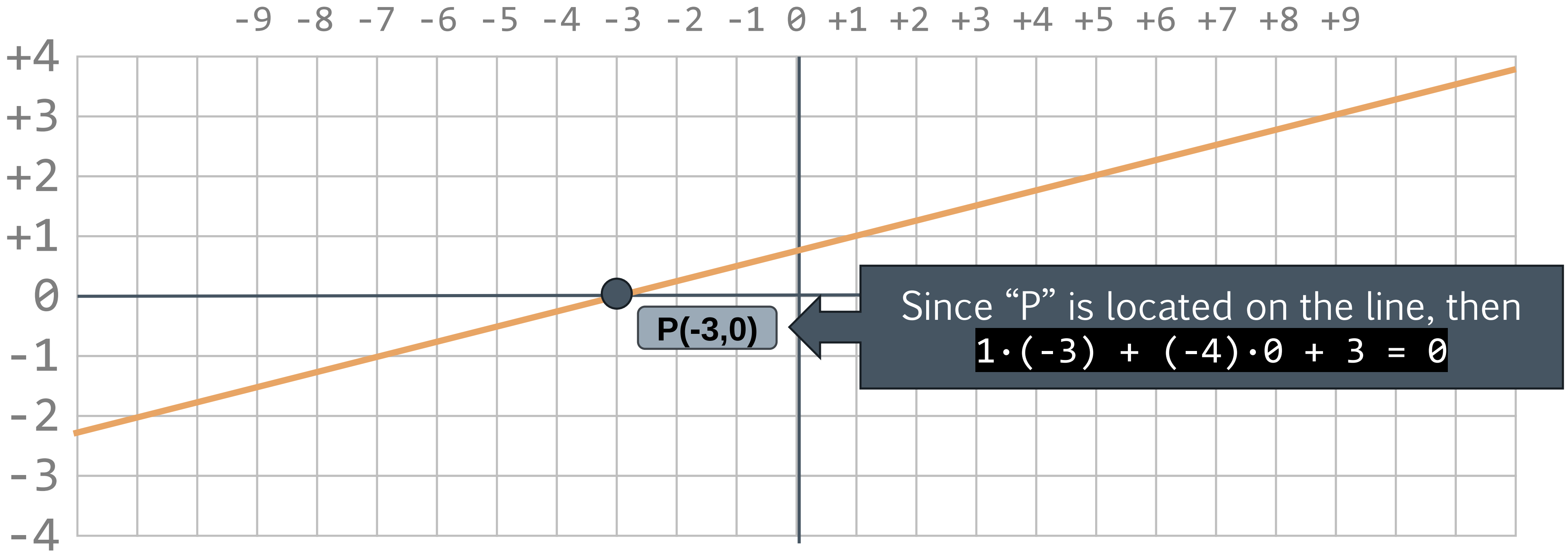
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



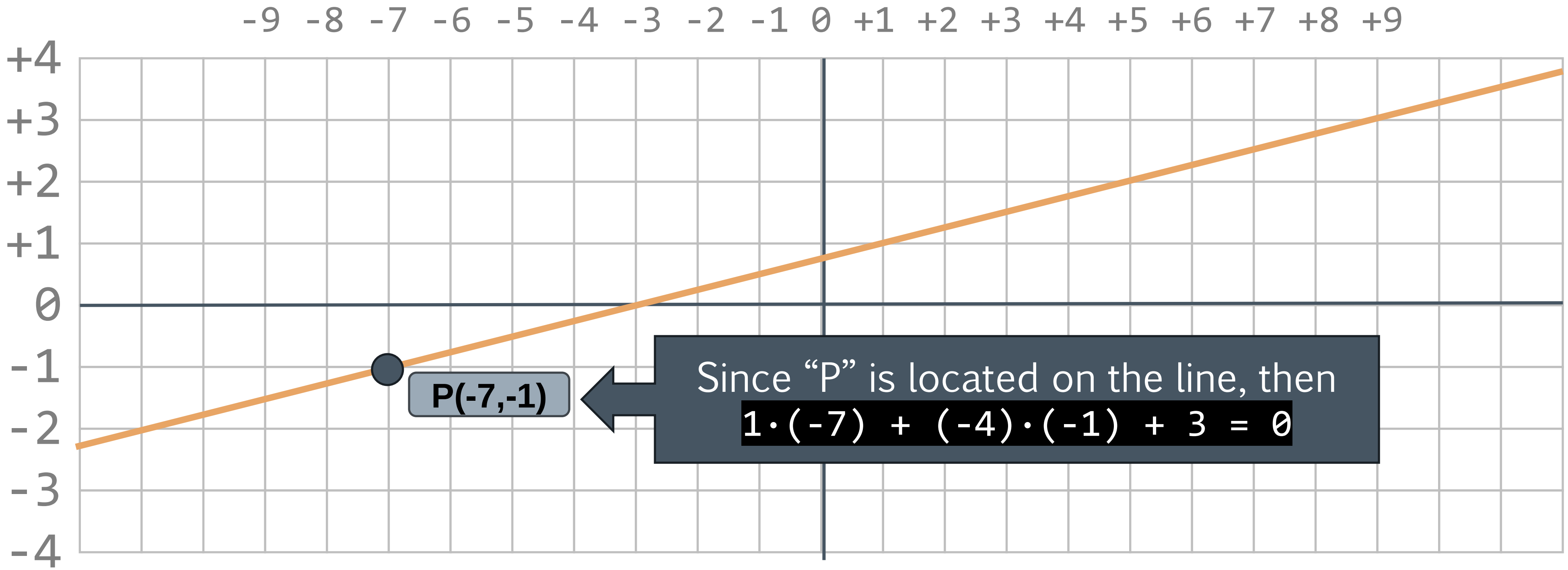
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



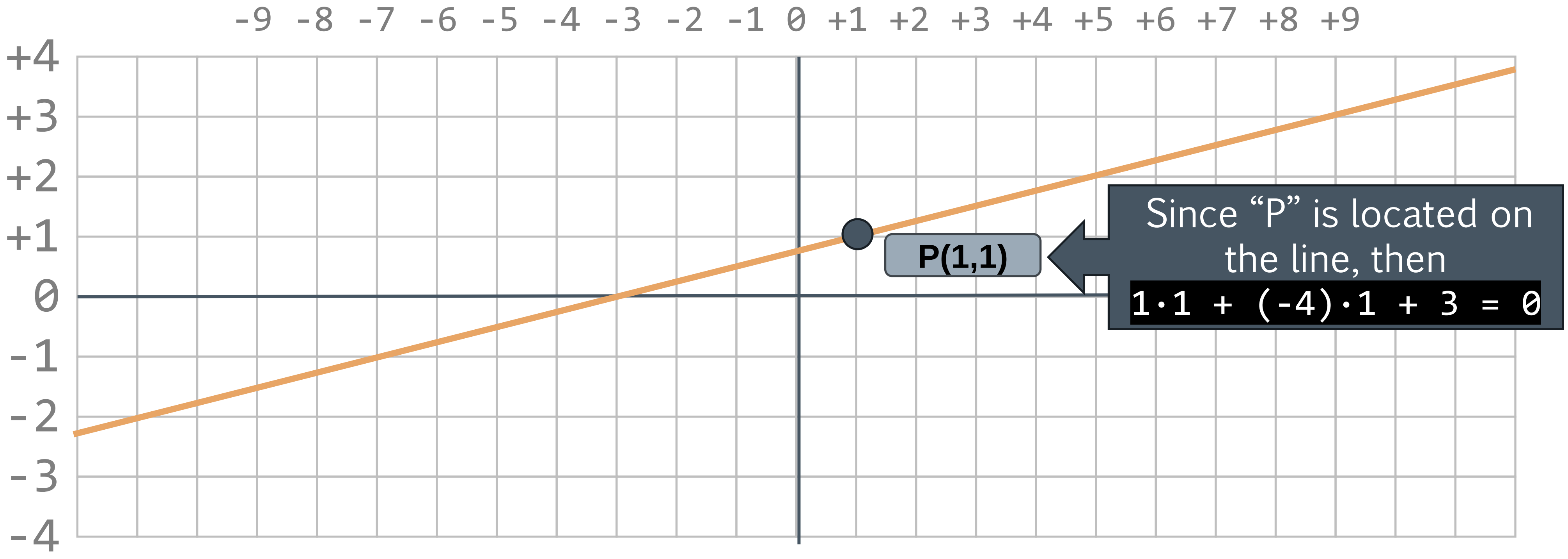
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



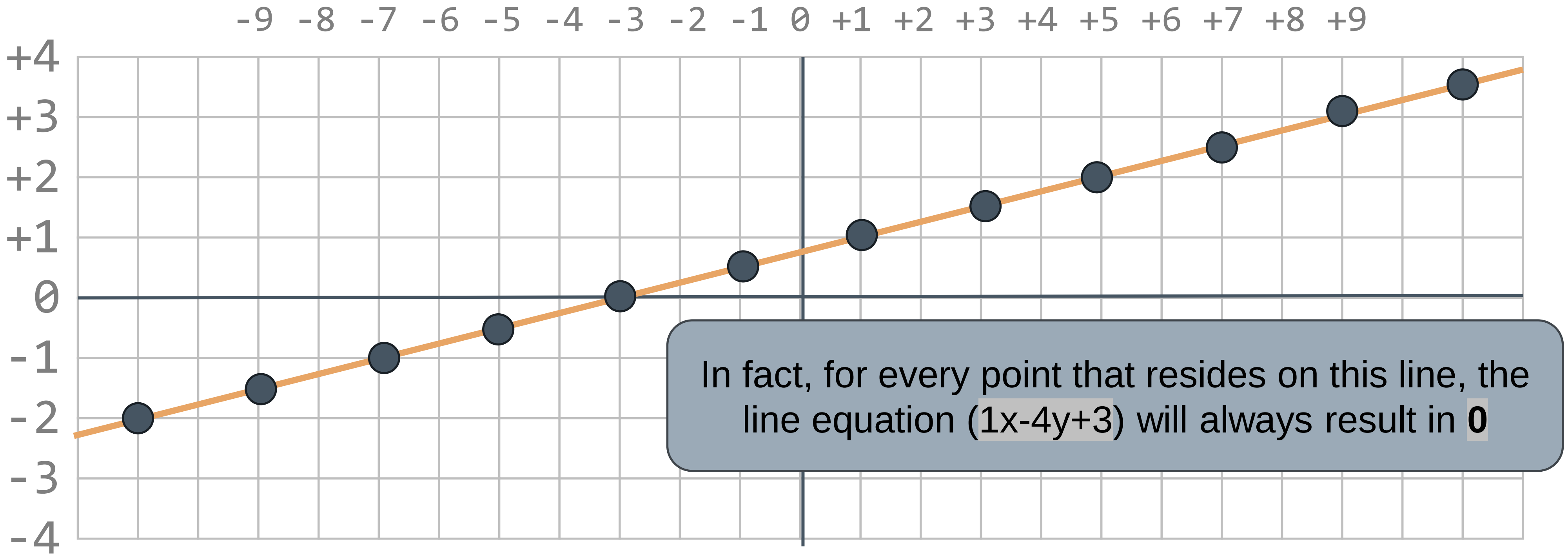
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot x + (-4) \cdot y + 3 = 0, a = 1, b = -4, c = 3}$ as described in the below picture.



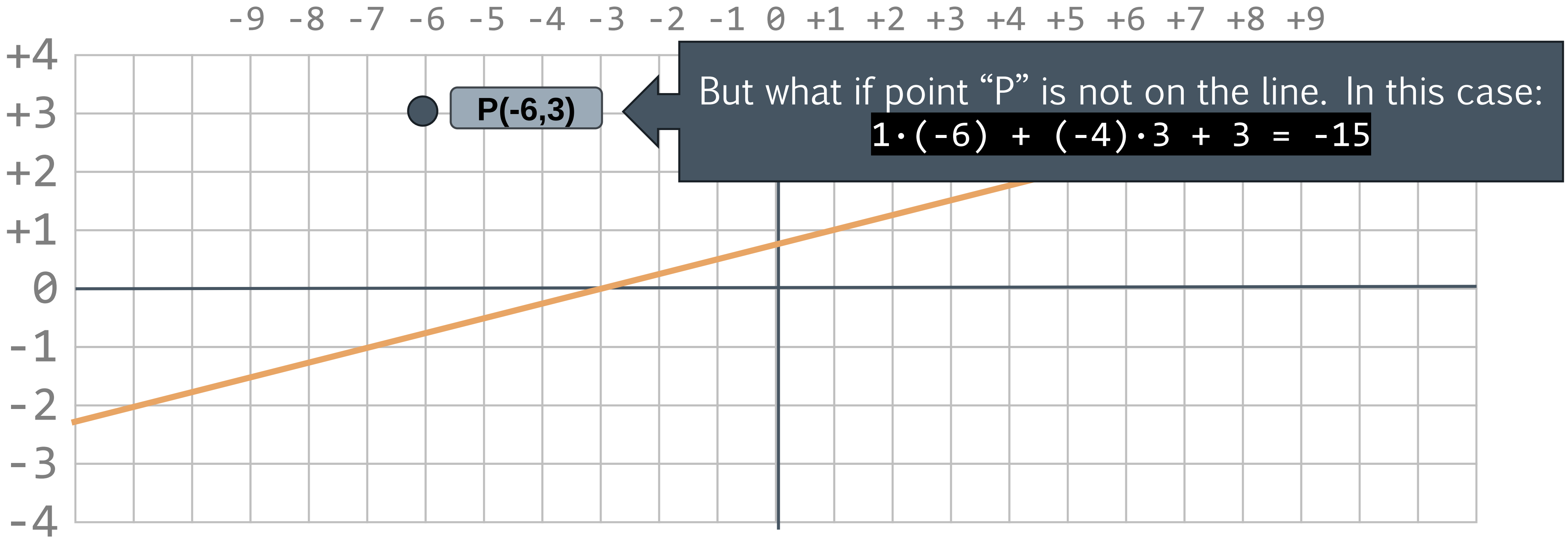
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



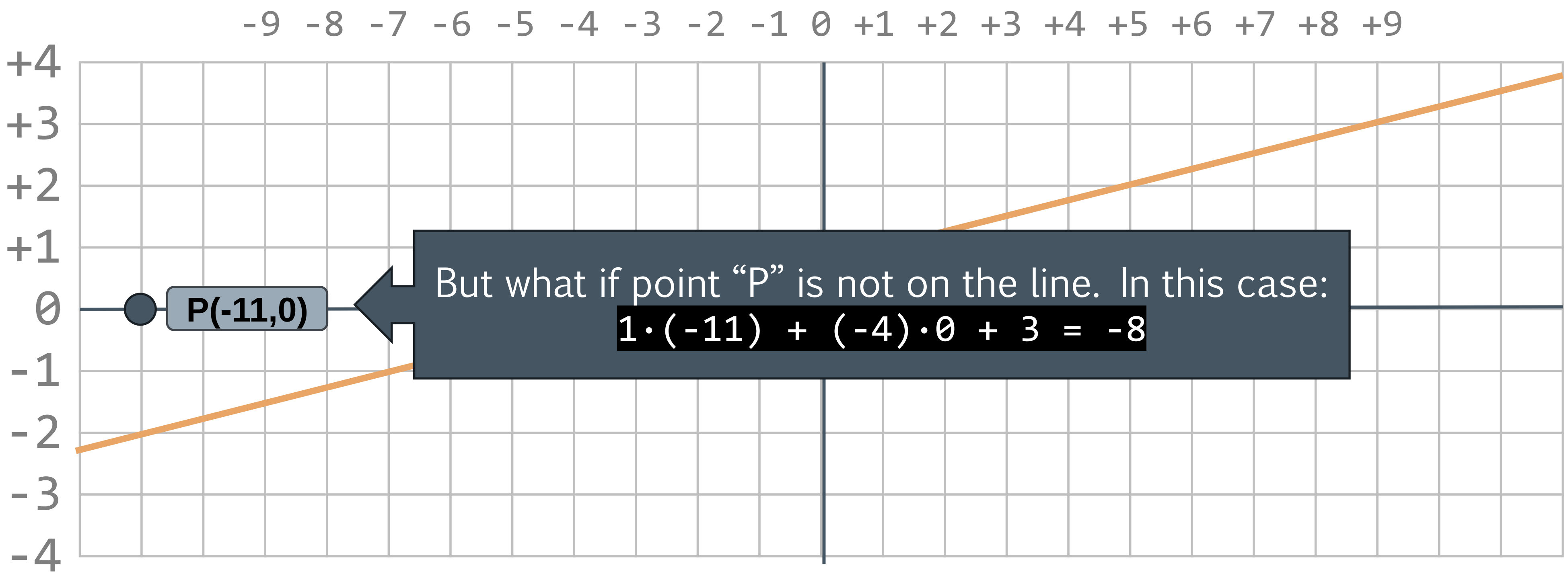
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



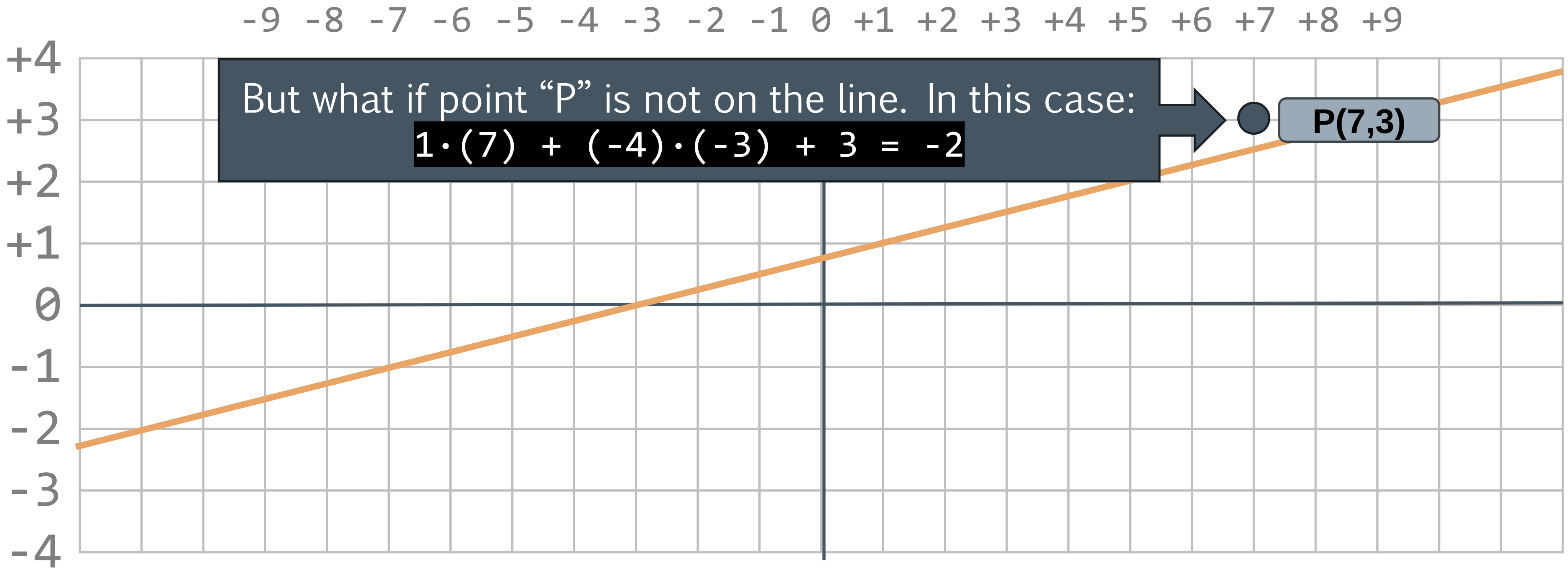
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot x + (-4) \cdot y + 3 = 0, a = 1, b = -4, c = 3}$ as described in the below picture.



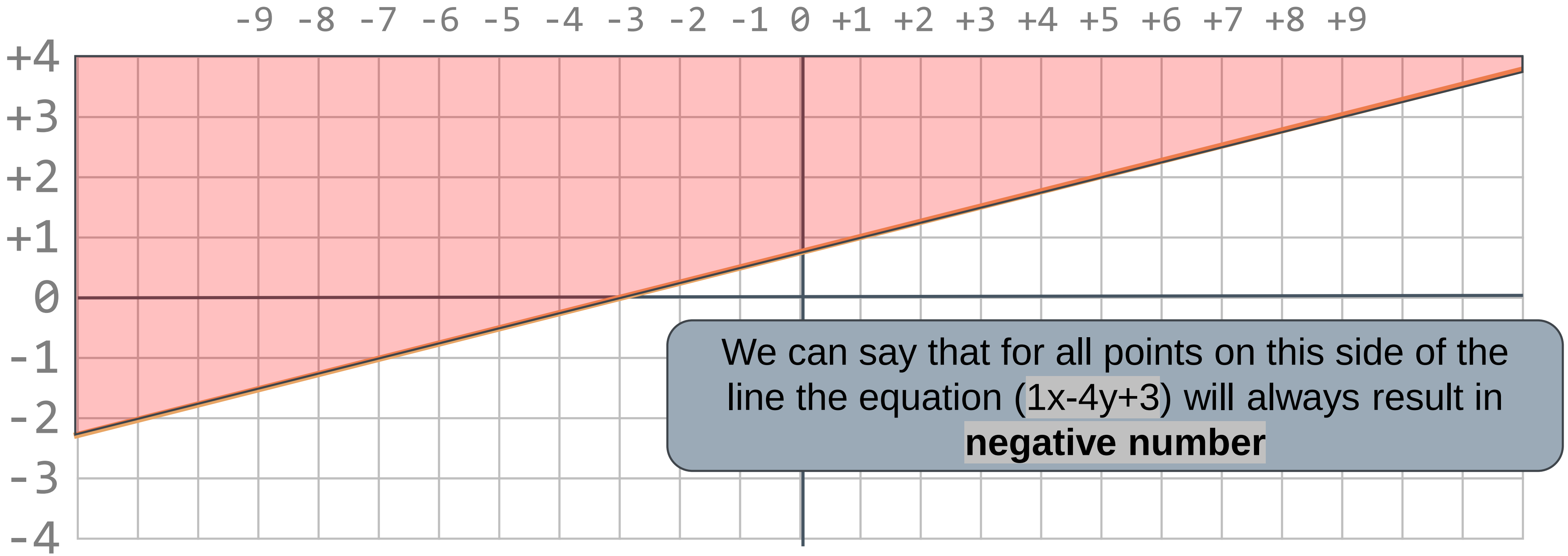
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



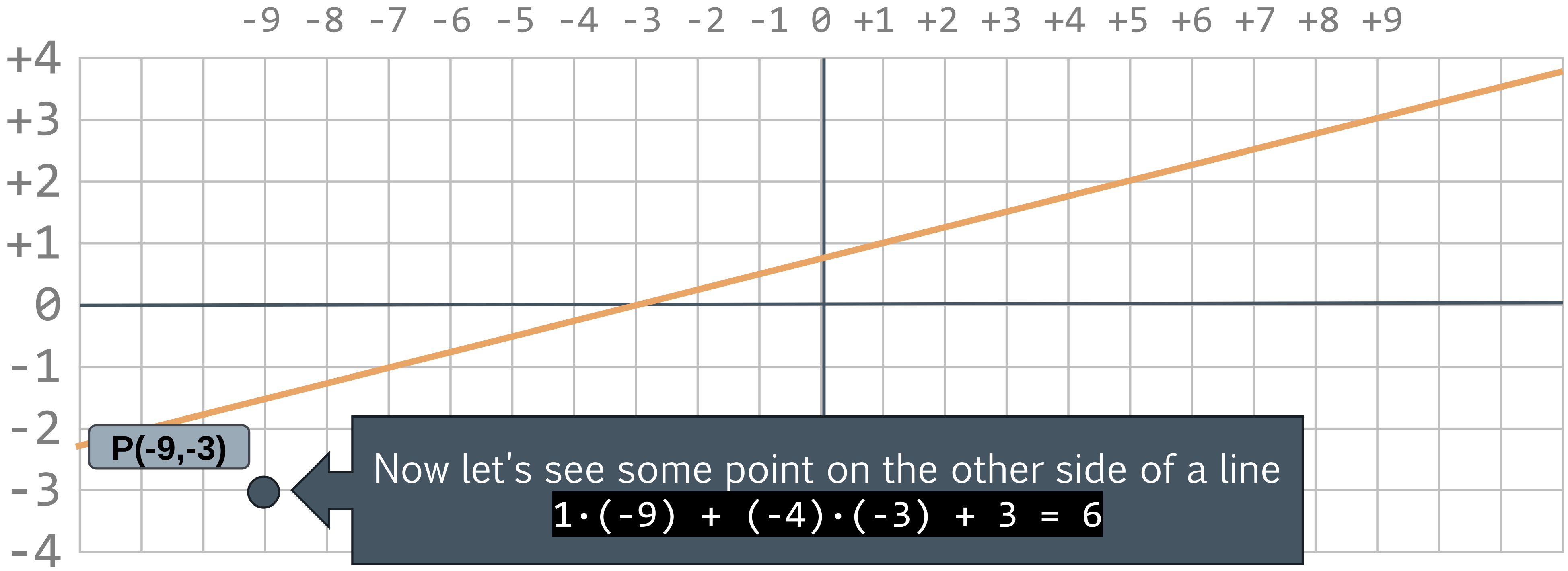
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (\mathbf{-4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = \mathbf{-4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



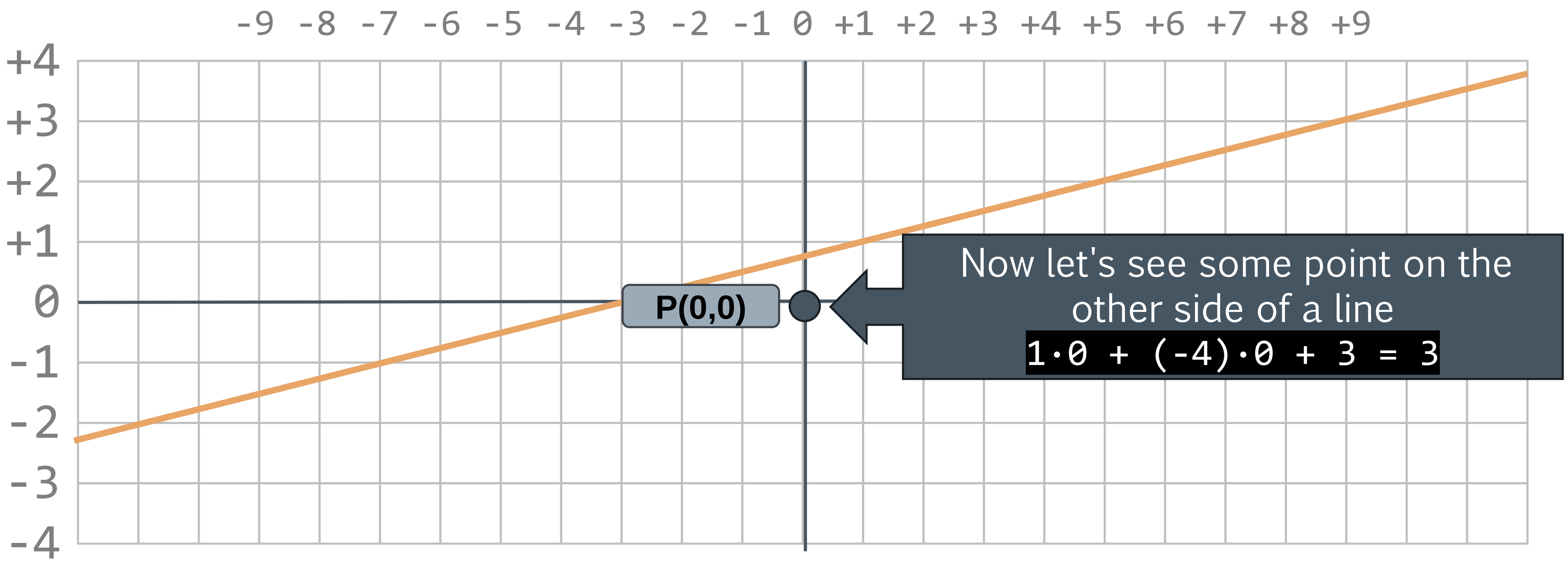
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



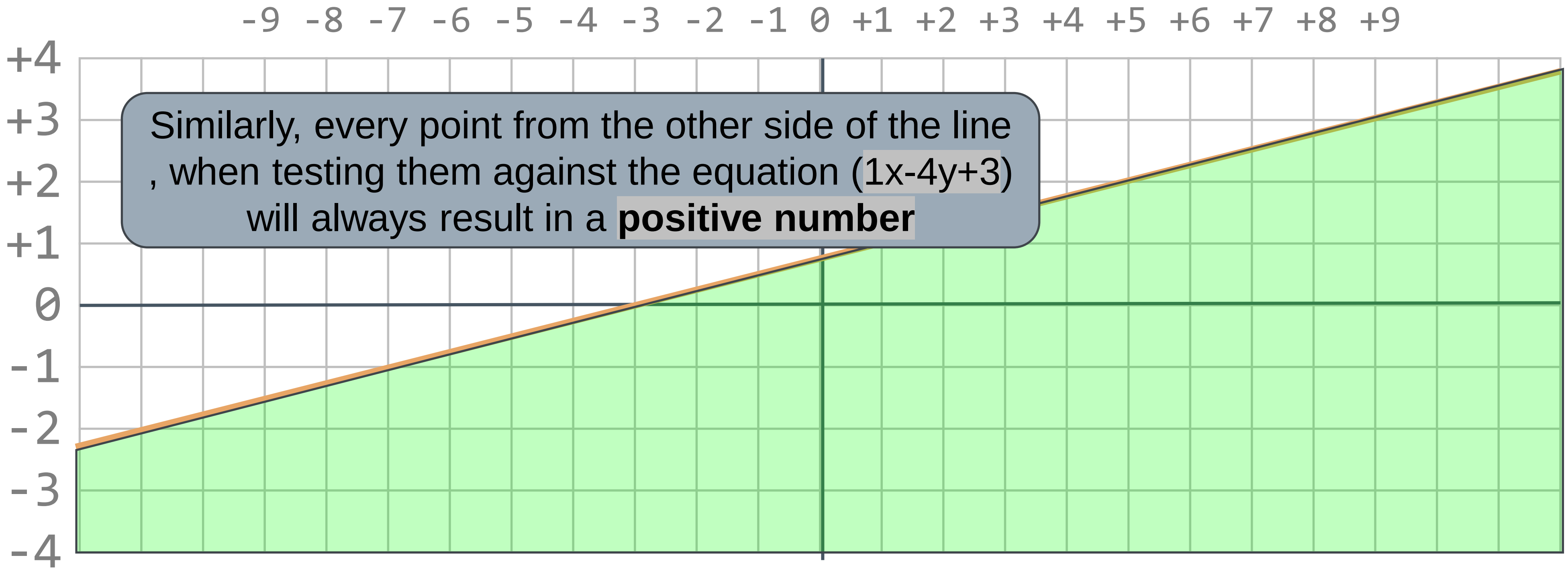
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot \boldsymbol{x} + (-\mathbf{4}) \cdot \boldsymbol{y} + \mathbf{3} = \mathbf{0}, \boldsymbol{a} = \mathbf{1}, \boldsymbol{b} = -\mathbf{4}, \boldsymbol{c} = \mathbf{3}}$ as described in the below picture.



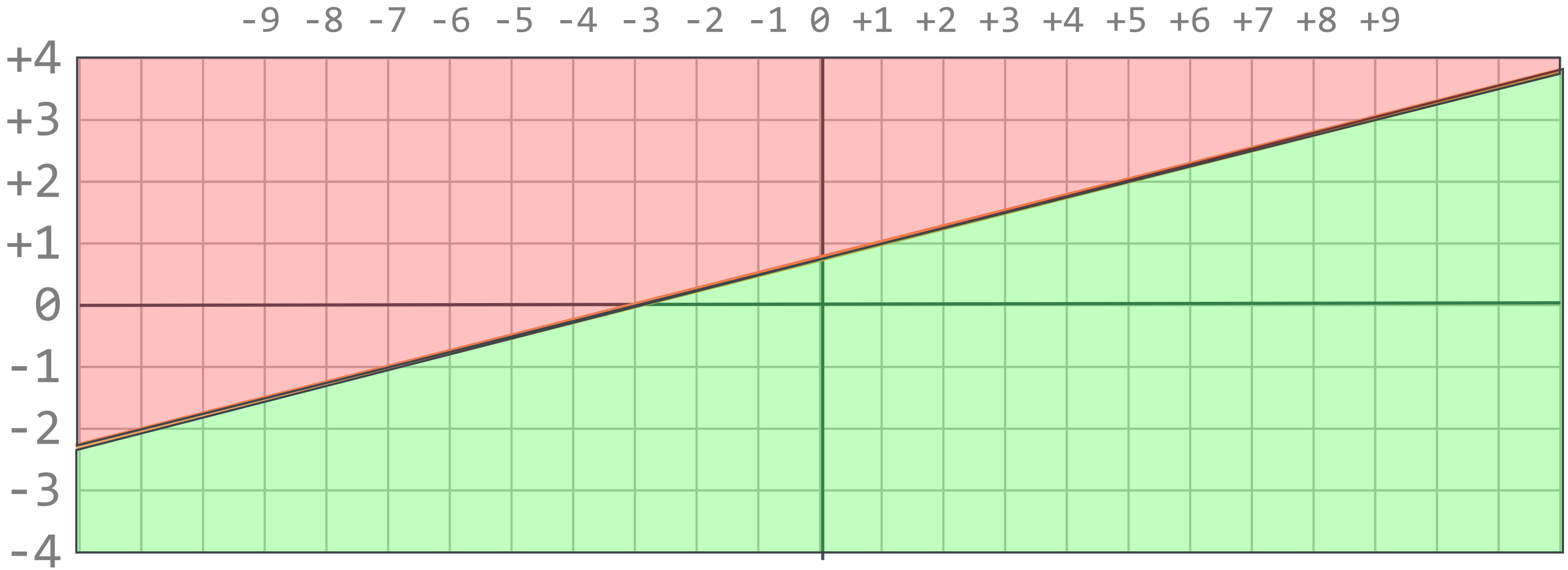
Algorithm

To understand the meaning of that inequality, lets consider the following line: $\boxed{1 \cdot x + (-4) \cdot y + 3 = 0, a = 1, b = -4, c = 3}$ as described in the below picture.



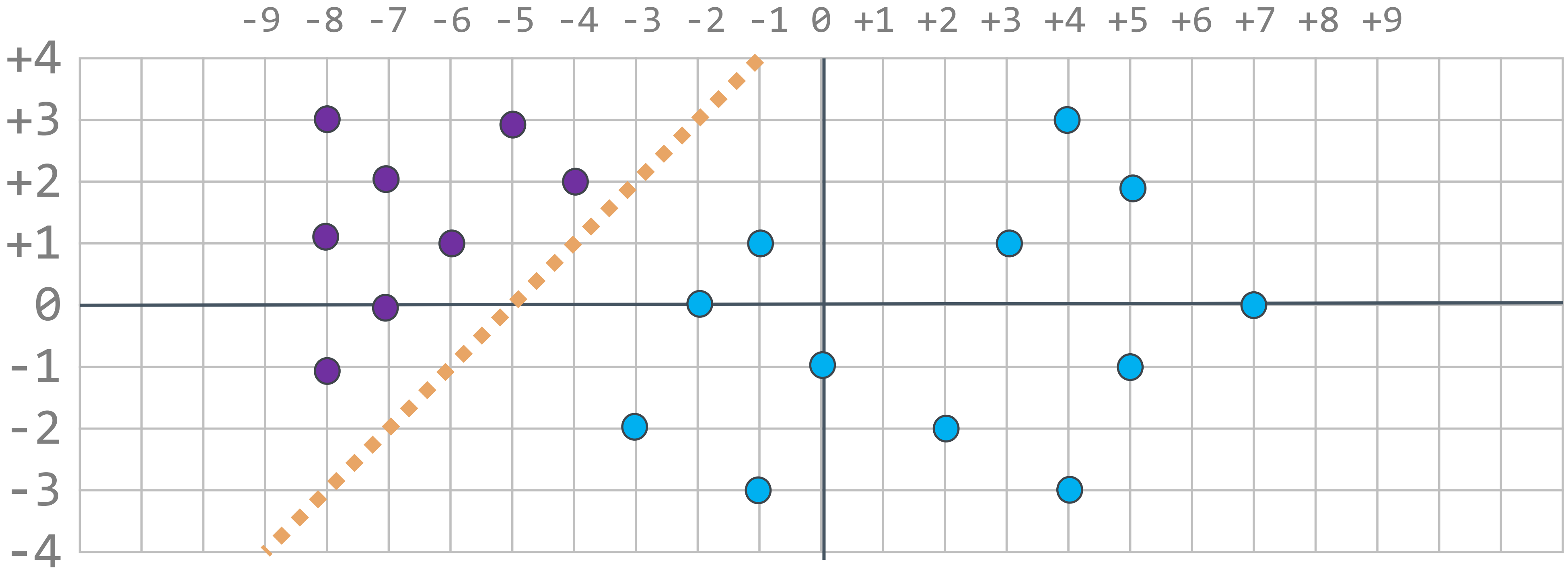
Algorithm

This means that the hyper-plane equation can used to separate points (some on one side that result in a positive value, and the other one on the other side that result in a negative value).



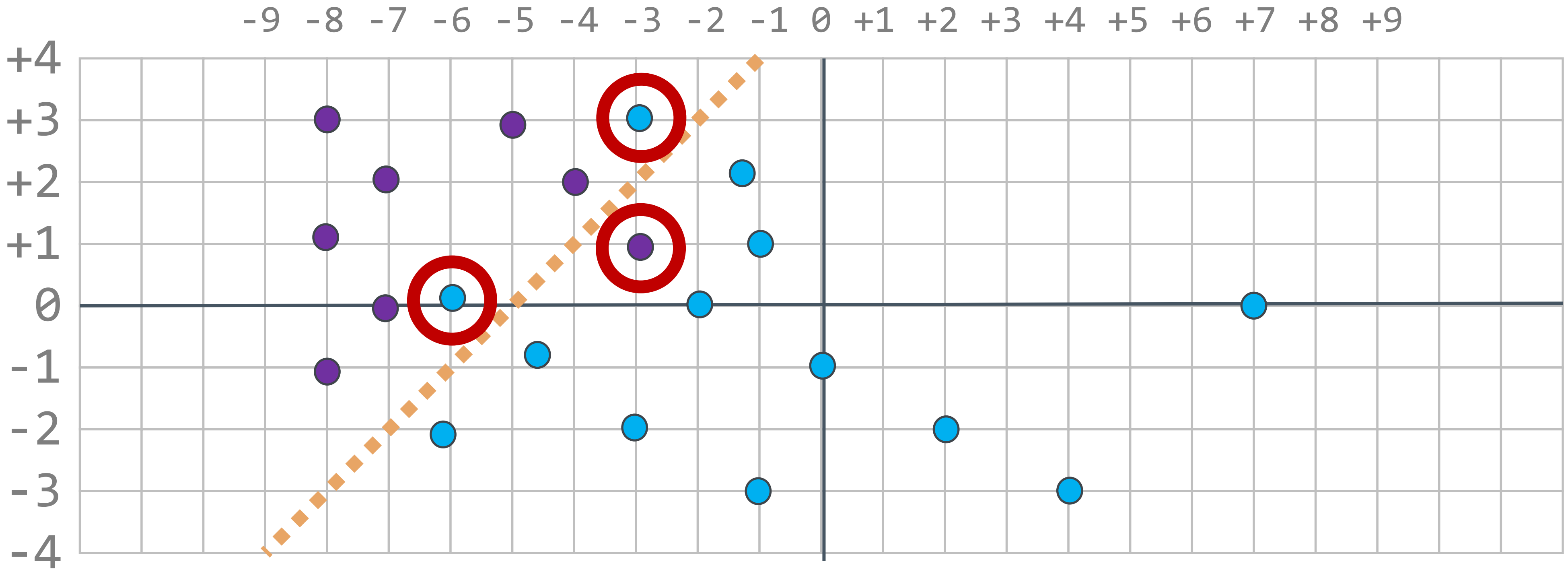
Algorithm

So, the perceptron algorithm can be described as follows: given dataset, single class, labeled, can we find a hyperplane that separates all example of one class from the other ones ? (in the case below – can we separate the **magenta** points from the **blue** ones ?)



Algorithm

In practice, data sets are not always ***linear-separable***. In this case, the purpose is to find a hyper-plane that minimizes or maximizes one of the metrics (FPR, ACC, etc).



Training

Training

At this point, we know that the perceptron algorithm is meant to search for a hyper-plane that can separate two data sets (as best of possible).

Let's assume that we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	YES
#2	$I_{(2,1)}$	$I_{(2,2)}$		$I_{(2,n)}$	NO
…					
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES

Training

At this point, we know that the perceptron algorithm is meant to search for a hyper-plane that can separate two data sets (as best of possible).

Let's assume that we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Labe
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	
#2	$I_{(2,}$	$I_{(2,}$		$I_{(2,}$	NO
			…		
#m				I	YES

These a scalar values that represent the data associated with a particular entry. We can say that these could be considered coordinates in a "n" dimensional space

Coordinate on dimension 1

Coordinate on dimension 2

Coordinate on dimension n

Training

At this point, we know that the perceptron algorithm is meant to search for a hyper-plane that can separate two data sets (as best of possible).

Let's assume that we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	YES
#2	$I_{(2,1)}$	$I_{(2,2)}$		$I_{(2,n)}$	NO
#m	$I_{(m,1)}$	I			

The label could be anything:

- **Yes**/**No** or **True**/**False** (to reflect that the sample belongs or not to the class)
- A string that reflects the name of the class and another one that reflects the rest (e.g. **CAT** and **OTHERS**) or (**CLASS-A** and **CLASS-B**)
- Some numerical values (**1**/**0**)

Training

At this point, we know that the perceptron algorithm is meant to search for a hyper-plane that can separate two data sets (as best of possible).

Let’s assume that we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	YES
#2	$I_{(2,1)}$	$I_{(2,2)}$		$I_{(2,n)}$	NO
			…		
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES

The target is to find a hyper-plane equation / or a vector “$w=[w_1,w_2…w_n]$” and a threshold that will classify most (if not all) of the samples from the dataset that are labeled with **YES** on one side of the hyperplane and most (if not all) of the samples from the dataset that are labeled with **NO** to the other side.

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO
…					
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES
…					
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES

$+ \; our \; model \; w = [w_1 \quad \cdots \quad w_n]$ and threshold

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO
…						
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES
…						
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES

$+ \; our \; model \; w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column $Input_{n+1}$)

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO
…					
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES
…					
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO
…						
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES
…						
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES

$+\ our\ model\ w = [w_1 \quad \cdots \quad w_n]$ and threshold

$$g(k) = \sum_{i=1}^{n} \left(w_i \times I_{(k,i)}\right) + threshold$$

$+\ our\ model\ w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column $Input_{n+1}$)

$$g(k) = \sum_{i=1}^{n+1} \left(w_i \times I_{(k,i)}\right) = w \cdot I_k$$

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO
		…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES
		…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO
			…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES
			…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES

$+ \, our \; model \; w = [w_1 \quad \cdots \quad w_n]$ and threshold

$$g(k) = \sum_{i=1}^{n} \left(w_i \times I_{(k,i)}\right) + threshold$$

$+ \, our \; model \; w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column $Input_{n+1}$)

$$g(k) = \sum_{i=1}^{n+1} \left(w_i \times I_{(k,i)}\right) = w \cdot I_k$$

$$Classified(k) = \begin{cases} true & g(k) > 0, Label_k = YES \\ true & g(k) \leq 0, Label_k = NO \\ false & g(k) > 0, Label_k = NO \\ false & g(k) \leq 0, Label_k = YES \end{cases}$$

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO
…					
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES
…					
#m	$I_{(m,1)}$	$I_{(m,2)}$			

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO
…						
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES
…						
						YES

$+ \; our \; model \; w =$

$$g(k) = \sum_{i=1}^{n} (w_i \times I_{(k,i)})$$

To simplify this equation, lets consider that all samples that are labeled **YES** are the ones that should be on the side of the hyper-plane where *f(x)* computes a positive value, and the ones labeled **NO** should be on the side of the hyperplane where *f(x)* computes a negative value.

We can change the labels as follows:
YES will be converted into 1, and **NO** will be converted into -1

Training

Assuming we have the following dataset (for training):

Entry	Input_1	Input_2	…	Input_n	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO **(-1)**
		…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES **(1)**
		…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES **(1)**

$+\ our\ model\ w = [w_1 \quad \cdots \quad w_n]$ and threshold

Entry	Input_1	Input_2	…	Input_n	Input_{n+1}	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO **(-1)**
			…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES **(1)**
			…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES **(1)**

$+\ our\ model\ w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column Input_{n+1})

We can change the labels as follows:
YES will be converted into 1, and **NO** will be converted into -1

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO **(-1)**
…					
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES **(1)**
…					
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES **(1)**

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO **(-1)**
…						
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES **(1)**
…						
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES **(1)**

$+\, our\ model\ w = [w_1 \quad \cdots \quad w_n]$ and threshold

$$f(k) = Label_k \times \left(\sum_{i=1}^{n} \left(w_i \times I_{(k,i)}\right) + threshold \right)$$

$+\, our\ model\ w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column $Input_{n+1}$)

$$f(k) = Label_k \times \left(\sum_{i=1}^{n+1} \left(w_i \times I_{(k,i)}\right) \right) = Label_k \times \left(w \cdot I_k\right)$$

Training

Assuming we have the following dataset (for training):

Entry	$Input_1$	$Input_2$	…	$Input_n$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	NO **(-1)**
		…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	YES **(1)**
		…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	YES **(1)**

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Input_{n+1}$	Label
#1	$I_{(1,1)}$	$I_{(1,2)}$		$I_{(1,n)}$	1	NO **(-1)**
			…			
#k	$I_{(k,1)}$	$I_{(k,2)}$		$I_{(k,n)}$	1	YES **(1)**
			…			
#m	$I_{(m,1)}$	$I_{(m,2)}$		$I_{(m,n)}$	1	YES **(1)**

$+\, our\ model\ w = [w_1 \quad \cdots \quad w_n]$ and threshold

$$f(k) = Label_k \times \left(\sum_{i=1}^{n} \left(w_i \times I_{(k,i)}\right) + threshold \right)$$

$+\, our\ model\ w = [w_1 \quad \cdots \quad w_{n+1}]$

No need for threshold (column $Input_{n+1}$)

$$f(k) = Label_k \times \left(\sum_{i=1}^{n+1} \left(w_i \times I_{(k,i)}\right) \right)$$
$$= Label_k \times \left(w \cdot I_k\right)$$

$$Classified(k) = \begin{cases} true, & f(k) > 0 \\ false, & f(k) \leq 0 \end{cases}$$

Training

Once we know that a sample is correctly classified or not, we can readjust the weight vector.

Let's consider that sample "k" is not correctly classified:

$$f(k) = Label_k \times (w \cdot I_k), f(k) \leq 0$$

In this cases, we need to compute an error and adjust the "w" vector with that error so that the next time we compute f(k) the result will be better than the previous one.

For this we introduce a learning rate termen ($\alpha > 0$) (values like 0.01, or 0.1 are often used). While in the next description, the learning rate will be a fixed value, there are techniques (**L**earning **R**ate (LR) Scheduling) that change the learning rate dynamically based on current iteration or epoch).

Training

Then, we can adjust "w" vector in the following way:

- First let's consider "$w_{(\text{iteration } t)}$" the weight vector at a specific iteration
- We can also consider "$f_t(k)$" the output of the perceptron equation for sample "k" at iteration "t"

$$f_t(k) = w_{(iteration\ t)} \cdot I_k, with\ Label_k \times f_t(k) \leq 0$$

- If $f_t(k)$ was not correctly classified, we need to change something in the equation to fix this. We can only modify the vector "w" (the rest of the parameters → I and Label are fixed as they represent input data). This means that we want to change "w" in such a way that:

$$Label_k \times f_t(k) > 0$$

Training

So ... our target is:

$$Label_k \times f_t(k) > 0 \rightarrow Label_k \times \sum_{j=1}^{n+1} (w_j \times I_{(k,j)}) > 0 \rightarrow \sum_{j=1}^{n+1} (w_j \times I_{(k,j)} \times Label_k) > 0$$

One way we can be certain of this is if can make each term of form: $w_j \times I_{(k,j)} \times Label_k$ to always be positive.

One solution to this equation is to make (in multiple steps) w_j to be equal to $I_{(k,j)} \times Label_k$. At that point the previous term will be:

$$I_{(k,j)} \times Label_k \times I_{(k,j)} \times Label_k = I_{(k,j)}^2 \times Label_k^2 = I_{(k,j)}^2 \times 1$$

since $Label_k \in \{-1|1\}$ *and as such* $Label_k^2 = 1$

Training

The solution to the previous equation is to add a small portion of $I_{(k,j)} \times Label_k$ into w_j . Other solutions are to add a small portion of the following difference $I_{(k,j)} \times Label_k - w_j$ into w_j

This is where the learning rate (α) comes into place. Usually, α is a small value with $0 < \alpha < 1$ and it works like a percentage in this case.

Training

Then, we can adjust "w" vector in the following way:

- First let's consider "$w_{(\text{iteration } t)}$" the weight vector at a specific iteration

- We can also consider "$f_t(k)$" the output of the perceptron equation for sample "k" at iteration "t"

$$f_t(k) = \left(w_{(iteration\ t)} \cdot I_k\right), Label_k \times f_t(k) \leq 0$$

- We can adjust the weight vector in the following way:

$$w_{(iteration\ t+1)} = w_{(iteration\ t)} + I_k \times \alpha \times Label_k$$

Training

As a general observation, the formula for adjusting the vector "w" can be written in different ways (but the logic is similar):

#	Formulas
1	$w_{(iteration\ t+1)} = w_{(iteration\ t)} + I_k \times \alpha \times Label_k$
2	$w_{(iteration\ t+1)} = w_{(iteration\ t)} + Error \times I_k \times \alpha,$ $Error = PredictedLabel_k - Label_k$
3	$w_{(iteration\ t+1)} = w_{(iteration\ t)} + I_k \times \alpha \times Label_k,$ $\beta_{(iteration\ t+1)} = \beta_{(iteration\ t)} + \alpha \times Label_k, \beta = thredshold$
4	$Error = \left(I_k - w_{(iteration\ t)}\right) \times \alpha$ $w_{(iteration\ t+1)} = w_{(iteration\ t)} + Error \times Label_k,$ $\beta_{(iteration\ t+1)} = \beta_{(iteration\ t)} + \alpha \times Label_k, \beta = thredshold$

Bias is included in the weights vector (for cases 1 and 2)

Training

Let’s put all of these together and build a training algorithm:

Pseudocode

```
w ← a vector of “n+1” elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
repeat
  foreach sample in trainset
    classified ← positive(sample.label x dotproduct(sample.input,w))
    if classified then continue
    w ← w + sample.input x α x sample.label
  end foreach
until exit_condition
```

Training

Let's put all of these together and build a training algorithm:

Line Pseudocode

```
w ← a vector of "n+1" elements initialized randomly or with 0
trainset ← the training set
α ← 0.0
repeat
  foreach sample in trainset
    classified ← positive(sample.label x dotproduct(sample.input,w))
    if classified then continue
    w ← w + sample.input x α x sample.label
  end foreach
until exit_condition
```

positive method is a simple function that returns **true** if its parameter is positive (>0) and **false** otherwise.

Training

Let's put all of these together and build a training algorithm:

Line	Pseudocode
1	`w ← a vector of "n+1" elements initialized randomly or with 0`
2	`trainset ← the training set`
3	`α ← 0.01 // learning rate`
4	`repeat`
	`bel x dotproduct(sample.input,w))`
	`le.label`
9	`end foreach`
10	`until exit_condition`

Exit condition can be several things:

- A number of iterations (epochs) is achieved
- Every sample is correctly classified
- A specific metric is achieved (e.g. FPR < 3%, ACC > 99%, etc)

Training

Let’s put all of these together and build a training algorithm:

Line	Pseudocode
1	`w ← a vector of “n+1” elements initialized randomly or with 0`
2	`trainset ← the training set`
3	`α ←`
4	`rep`
5	`f`
6	`product(sample.input,w))`
7	`if classified> then continue`
8	`w ← w + sample.input x α x sample.label`
9	`end foreach`
10	`until exit_condition`

If threshold (β) is being used, then this equation below changes into:

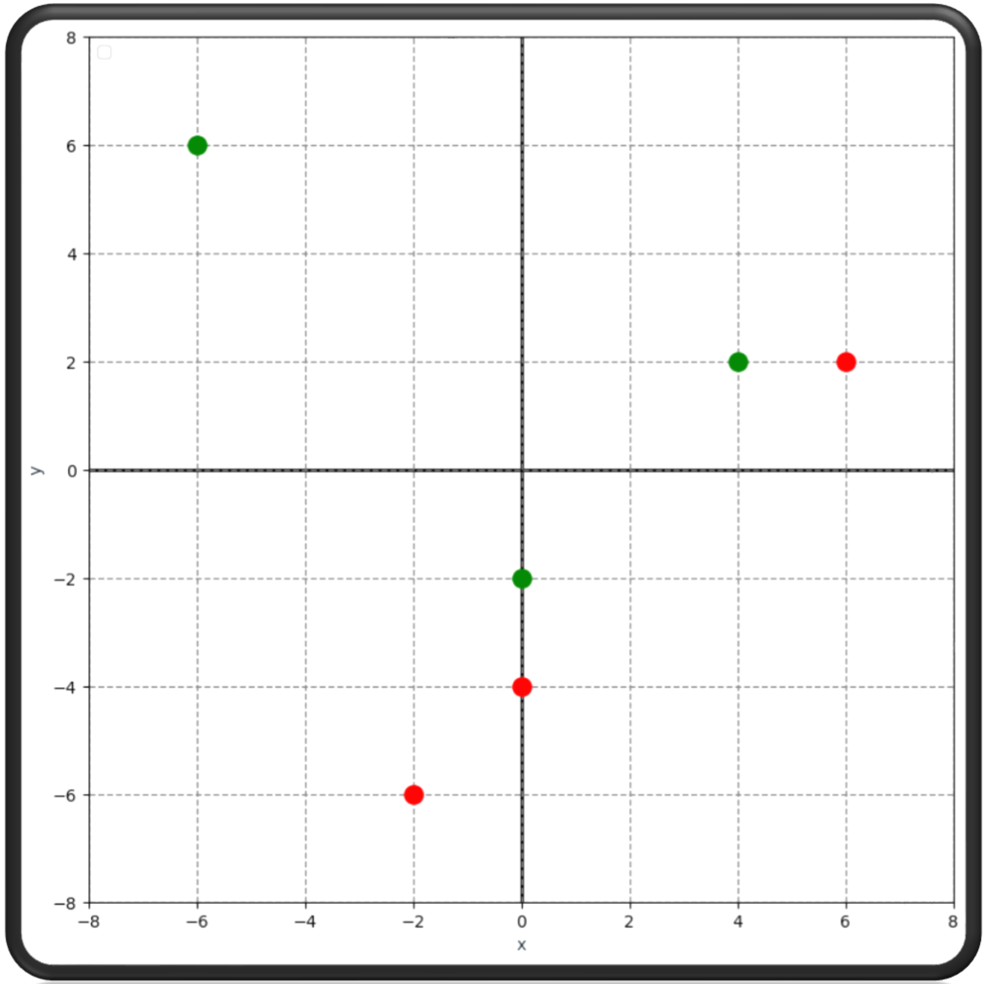
```
w ← w + sample.input x α x sample.label
β ← β + α x sample.label
```

Demo

Demo

1. Let's consider the following training dataset that consists in points in a two dimensional plane:

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)



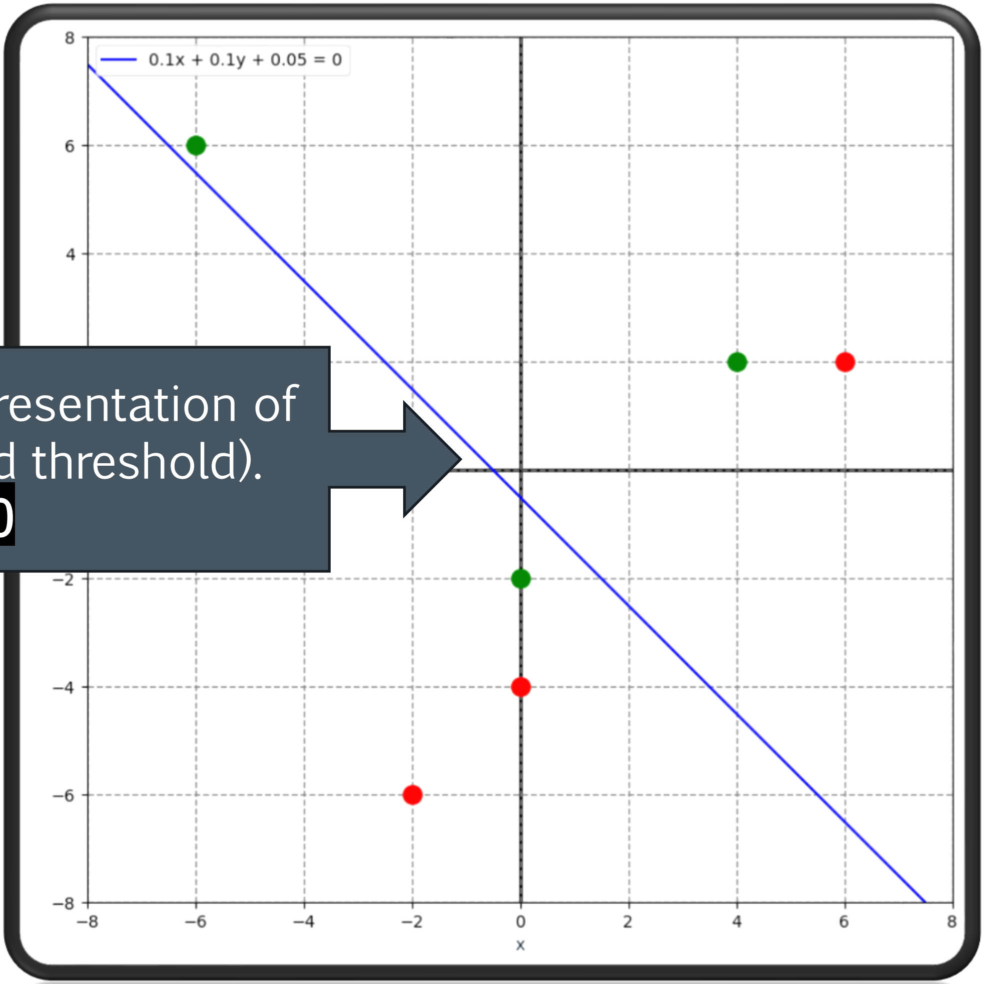
Demo

2. Now let's create a weight vector initialized with random (small values) and a threshold (β)

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	[illegible]	[illegible]	[illegible]
#4	[illegible]	[illegible]	[illegible]
#5	[illegible]	[illegible]	[illegible]
#6	6	2	-1 (■)

$Weight_1$	$Weight_2$	Threshold (β)
0.1	0.1	0.05

This blue line is the graphical representation of the model (the weight vector and threshold).
$0.1x + 0.1y + 0.05 = 0$

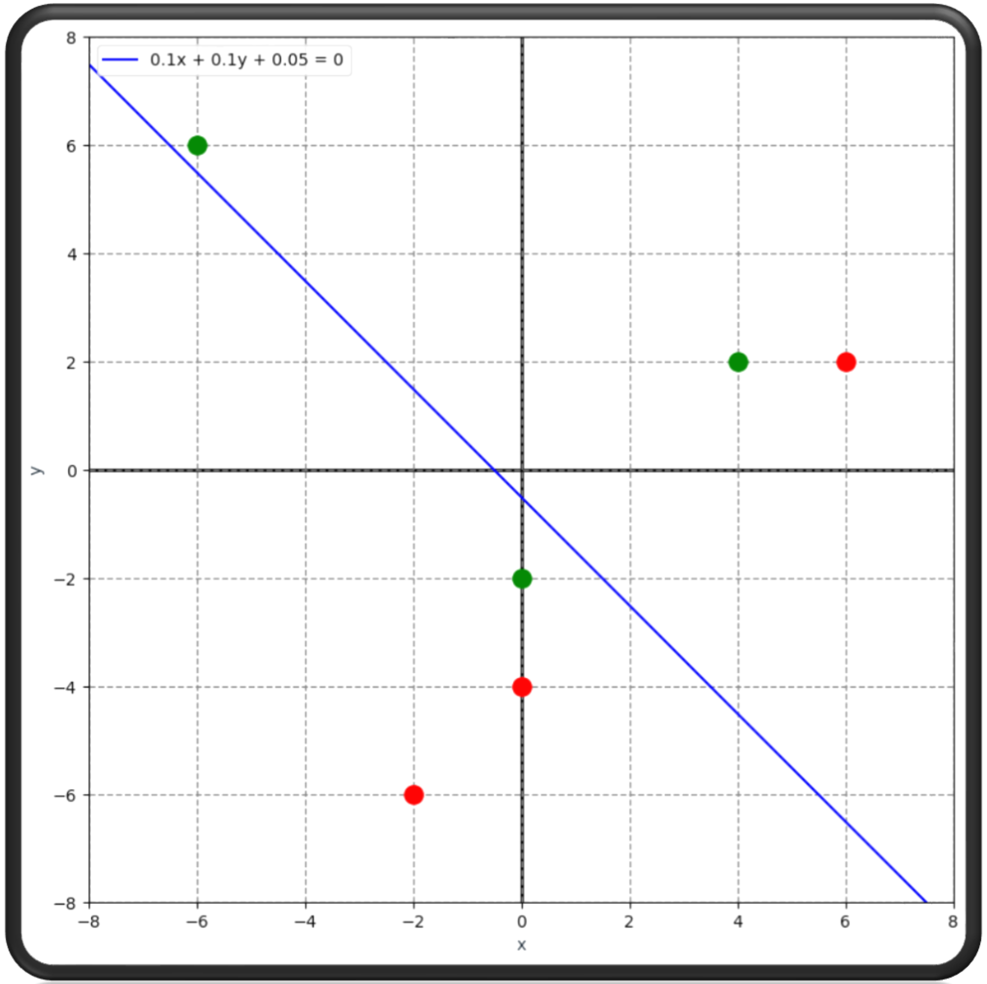


Demo

3. Let's also consider the learning rate (α) as **0.02**

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

$Weight_1$	$Weight_2$	Threshold (β)
0.1	**0.1**	**0.05**



Demo

4. For the training part, we will check each entry and see if it is correctly classified. If not, we will adjust the weight vector.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

Notice that the samples are sorted (first the ones with label 1 and then the ones with label -1). In practice, if not using batch training , it is best to shuffle them.

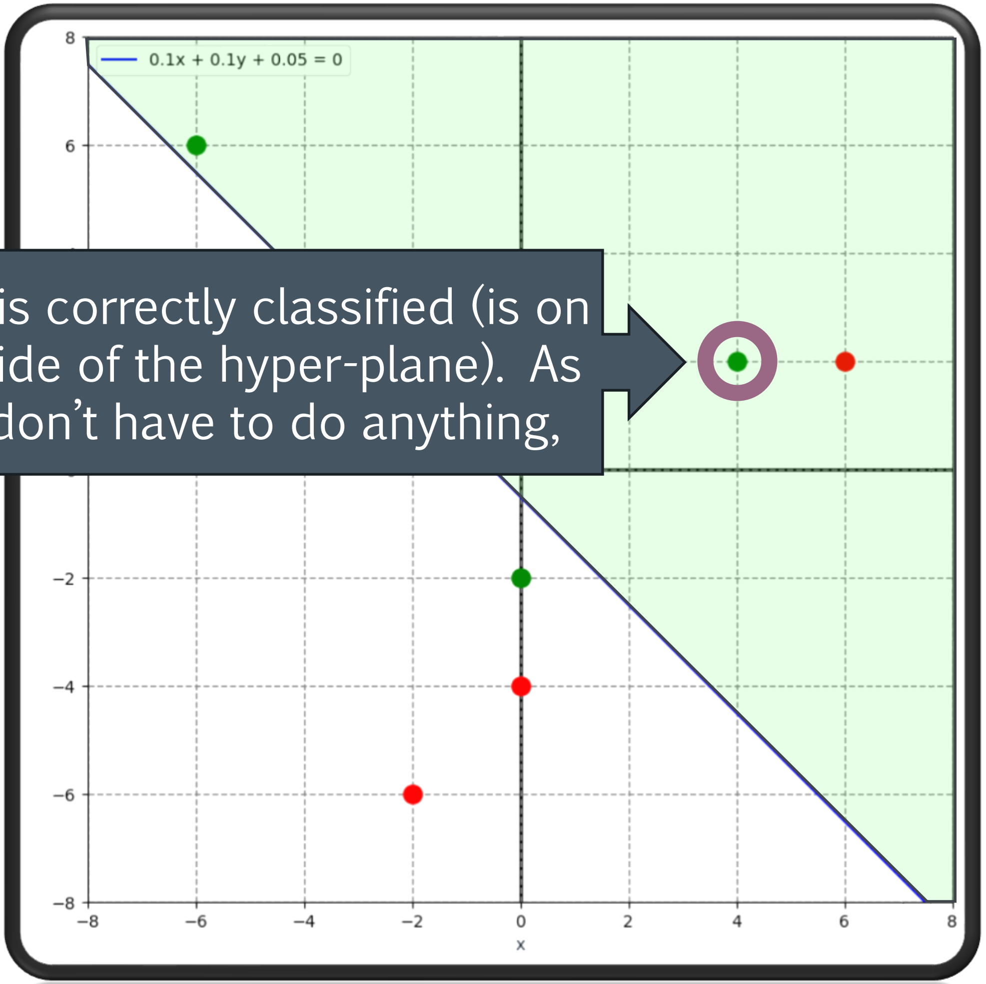
$Weight_1$	$Weight_2$	Threshold (β)
0.1	**0.1**	**0.05**

Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

$Weight_1$	$Weight_2$	Threshold (β)
0.1	0.1	0.05



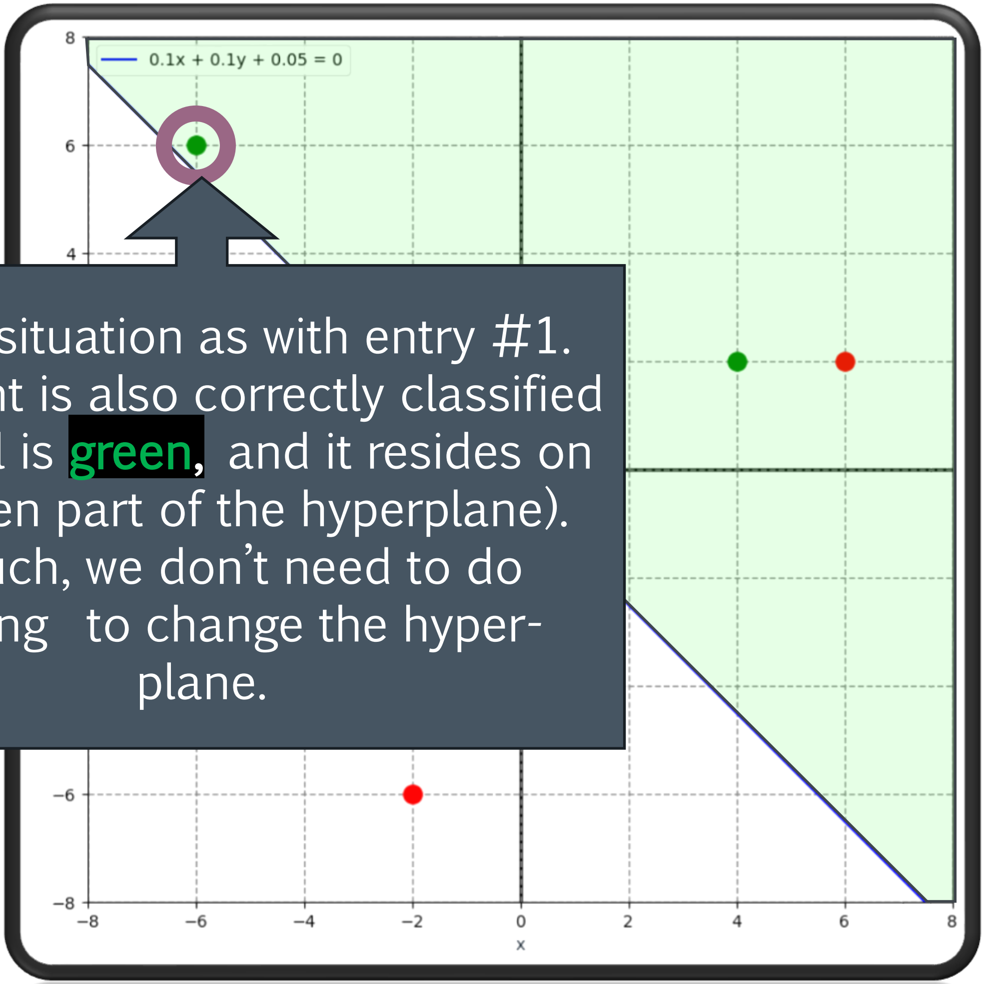
This point is correctly classified (is on the right side of the hyper-plane). As such, we don't have to do anything,

Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

$Weight_1$	$Weight_2$	Threshold (β)
0.1	0.1	0.05



Similar situation as with entry #1. This point is also correctly classified (its label is green, and it resides on the green part of the hyperplane). As such, we don't need to do anything to change the hyper-plane.

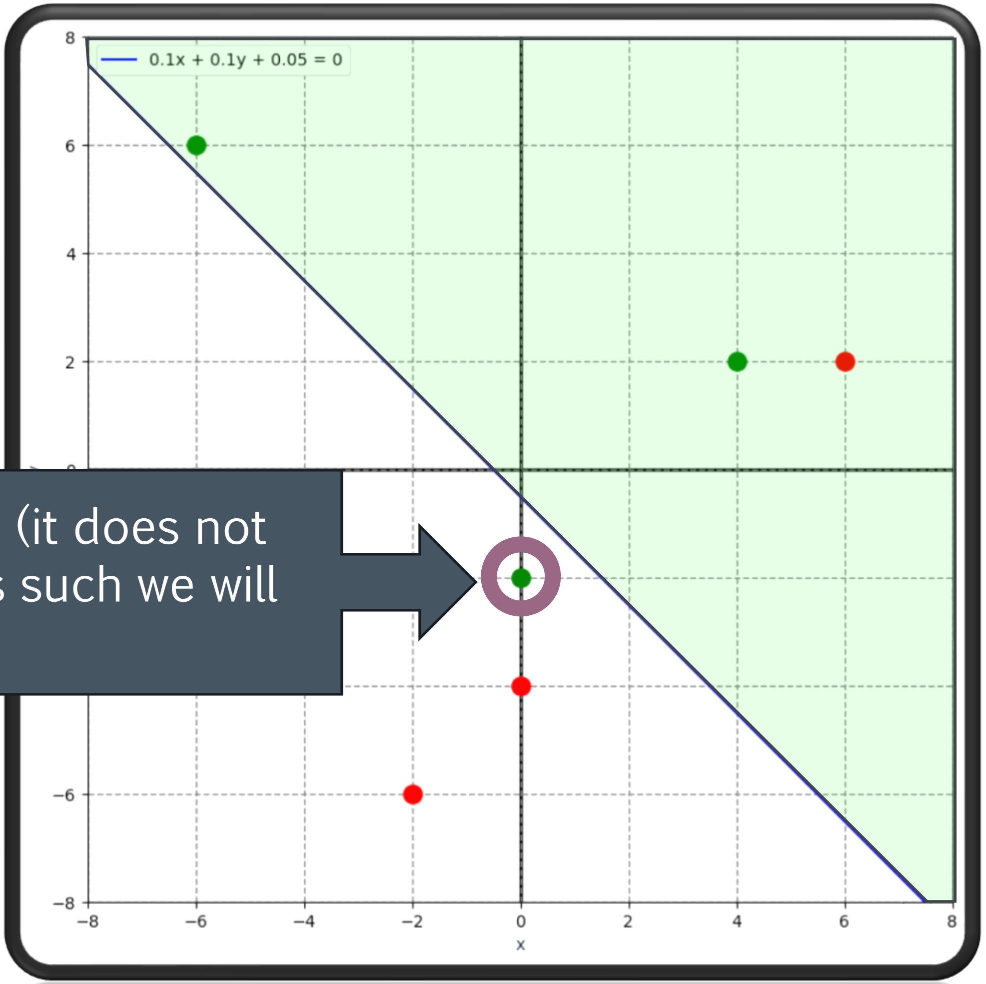
Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	Input$_1$ (X)	Input$_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)

This point however is not correctly classified (it does not reside on right part of the hyper-plan) and as such we will have to adjust the hyper plane

Weight$_1$	Weight$_2$	Threshold (β)
0.1	0.1	0.05



Demo

5. Testing each entry (from 1 to 6) against the current model.

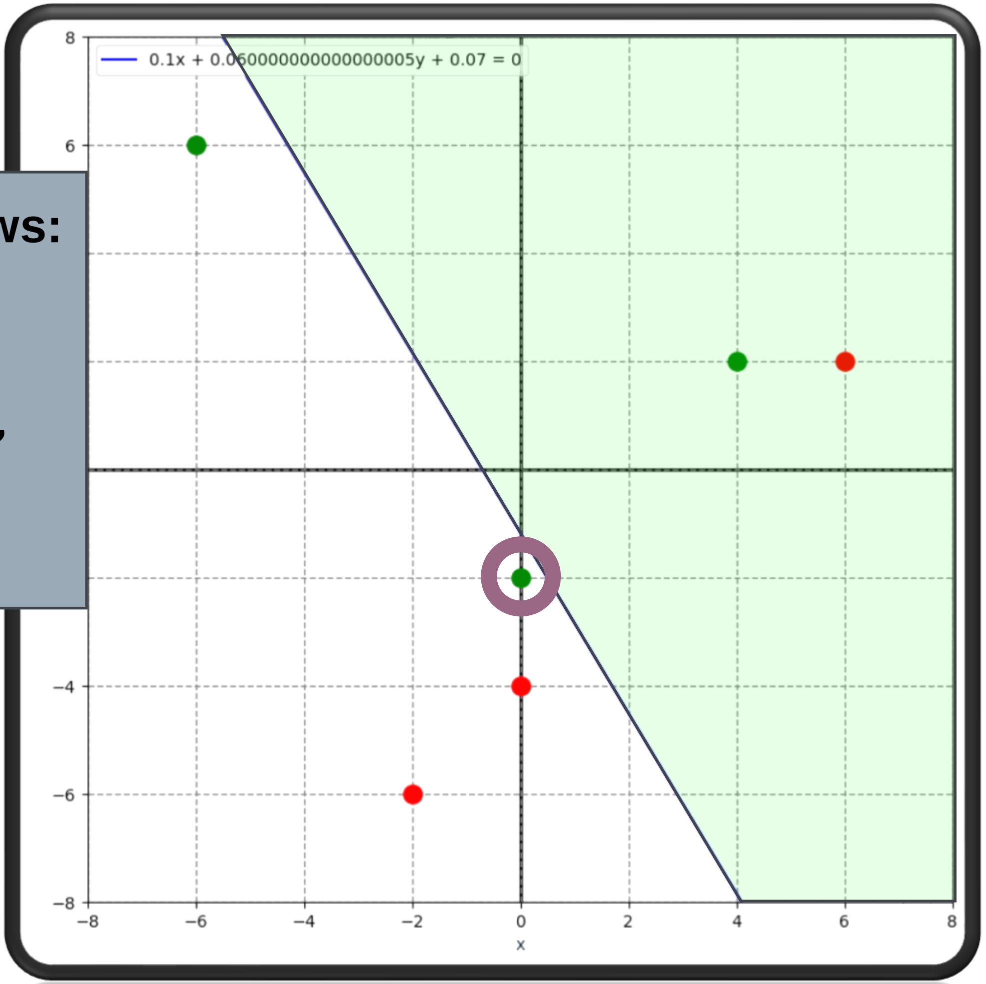
Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#6	6	2	-1 (■)

We will change vector "w" and threshold β as follows:

$$w = w + Entry_3.Input \times Entry_3.Label \times \alpha$$
$$= \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0 \\ -2 \end{bmatrix} \times 1 \times 0.02 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0 \\ -0.04 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.06 \end{bmatrix},$$

$$\beta = \beta + Entry_3.Label \times \alpha = 0.05 + 1 \times 0.02 = 0.07$$

$Weight_1$	$Weight_2$	Threshold (β)
0.1	0.06	0.07



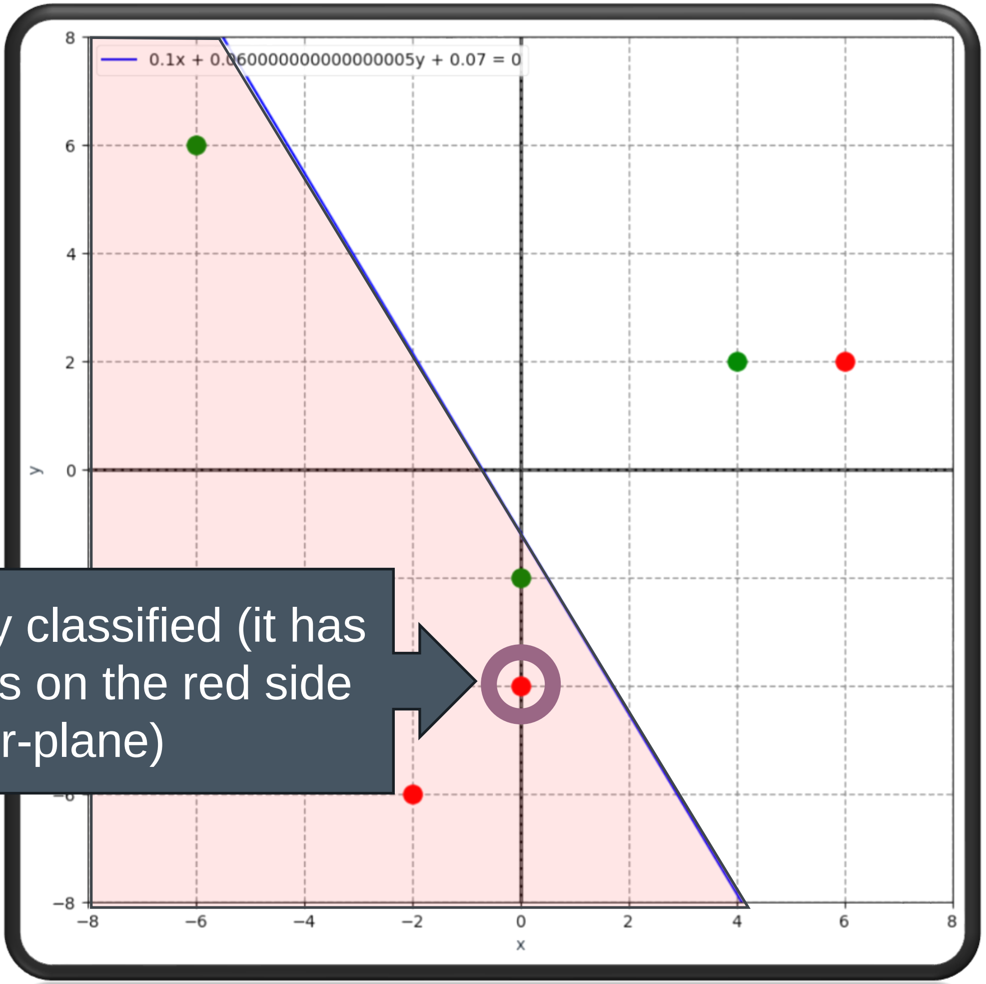
Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	Input$_1$ (X)	Input$_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	[illegible]

Weight$_1$	Weight$_2$	T… (β)
0.1	0.06	0.07

This point is correctly classified (it has the color red and it is on the red side of the hyper-plane)



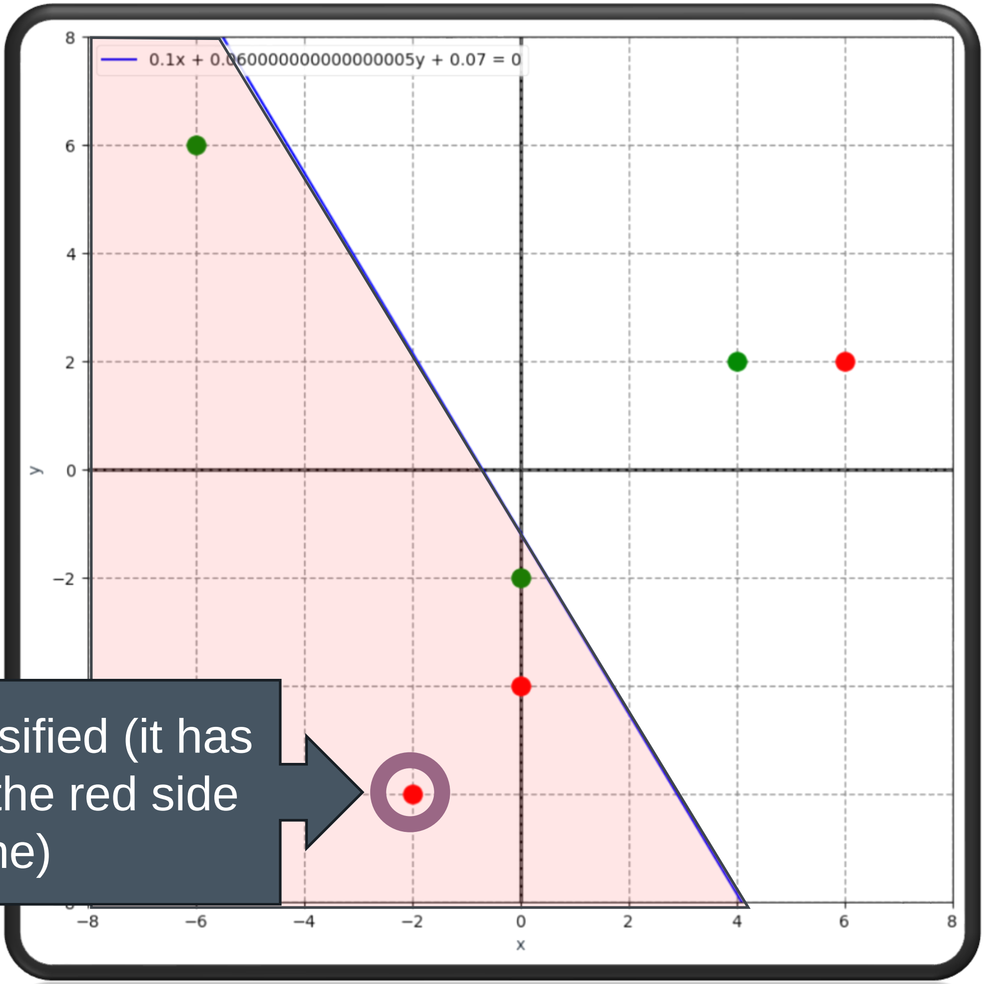
Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

$Weight_1$	Weight	
0.1	0.06	[illegible]

This point is correctly classified (it has the color red and it is on the red side of the hyper-plane)



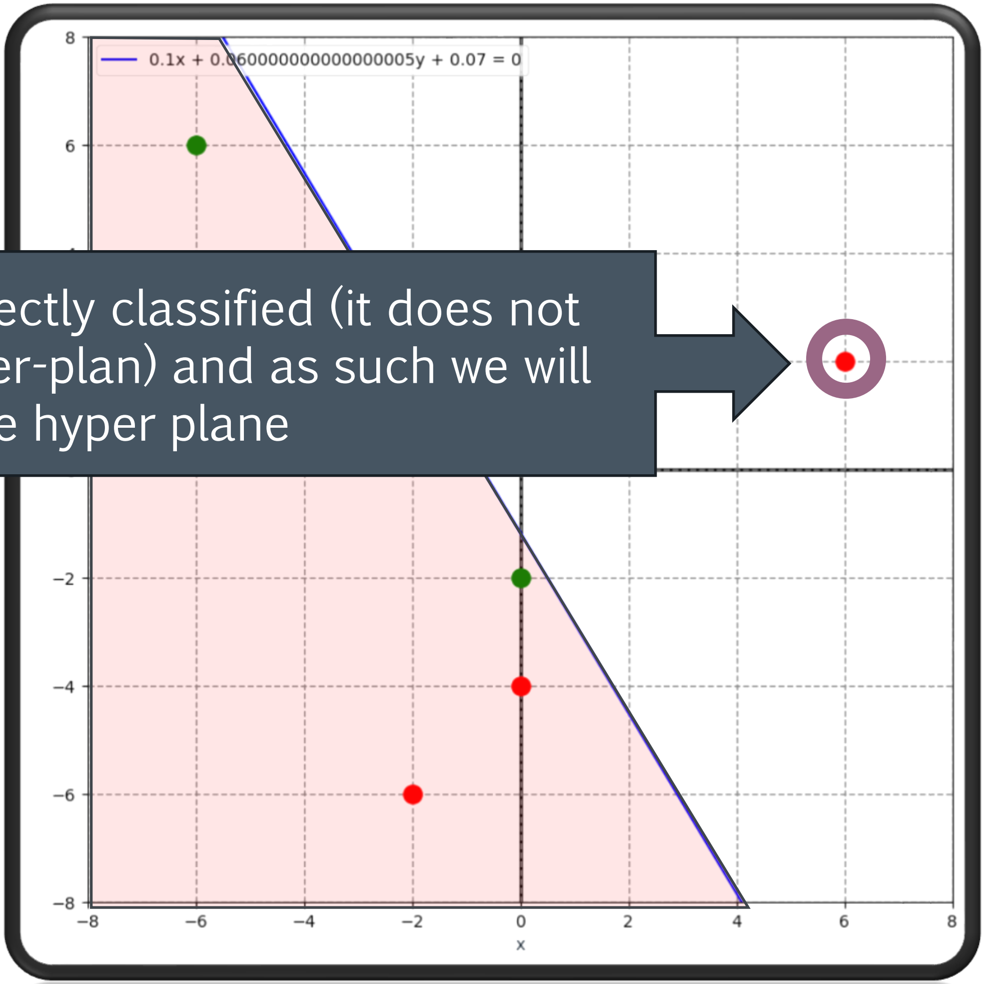
Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	[illegible]	[illegible]	[illegible]
#2	[illegible]	[illegible]	[illegible]
#3	[illegible]	[illegible]	[illegible]
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

$Weight_1$	$Weight_2$	Threshold (β)
0.1	0.06	0.07

This point however is not correctly classified (it does not reside on right part of the hyper-plan) and as such we will have to adjust the hyper plane



Demo

5. Testing each entry (from 1 to 6) against the current model.

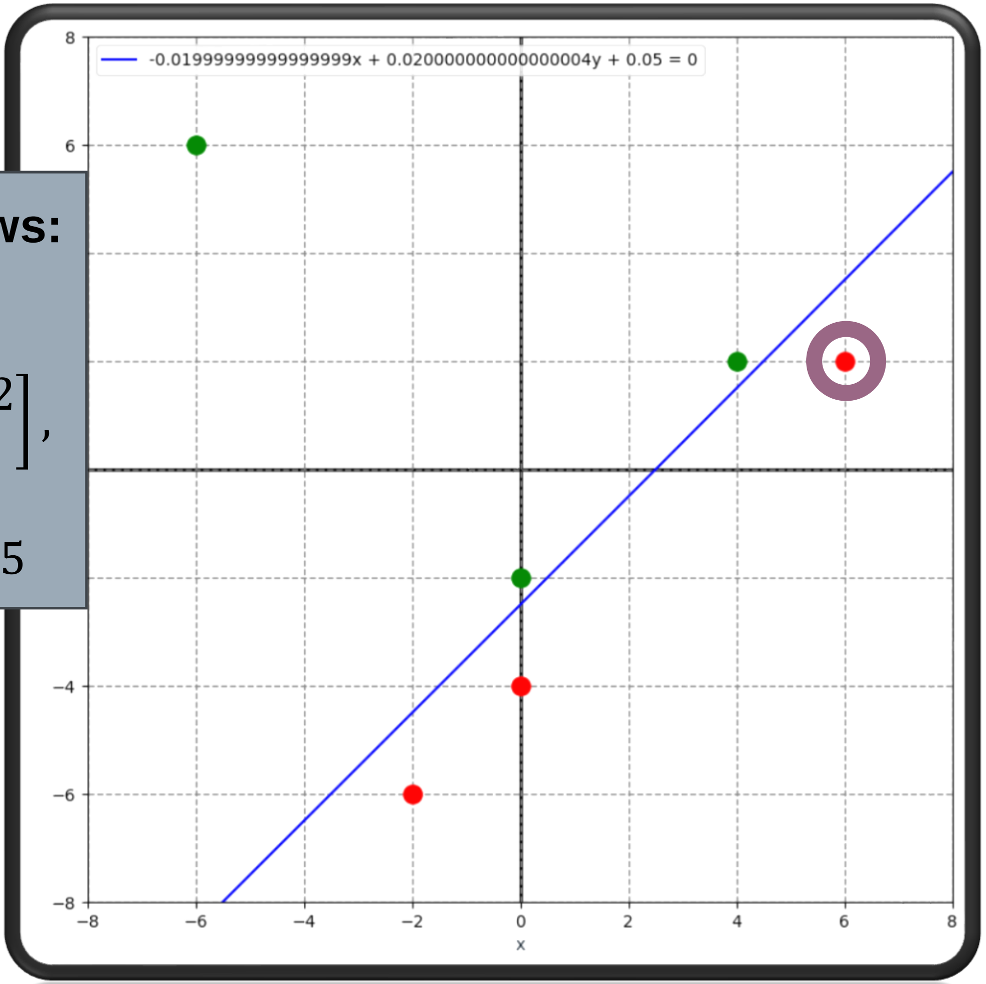
Entry	Input$_1$ (X)	Input$_2$ (Y)	Label (Color)
#6	6	2	-1 (■)

We will change vector "w" and threshold β as follows:

$$w = w + Entry_6.Input \times Entry_6.Label \times \alpha$$
$$= \begin{bmatrix} 0.1 \\ 0.06 \end{bmatrix} + \begin{bmatrix} 6 \\ 2 \end{bmatrix} \times -1 \times 0.02 = \begin{bmatrix} 0.1 \\ 0.01 \end{bmatrix} + \begin{bmatrix} -0.12 \\ -0.04 \end{bmatrix} = \begin{bmatrix} -0.02 \\ 0.02 \end{bmatrix},$$

$$\beta = \beta + Entry_6.Label \times \alpha = 0.07 + (-1) \times 0.02 = 0.05$$

Weight$_1$	Weight$_2$	Threshold (β)
-0.02	0.02	0.05



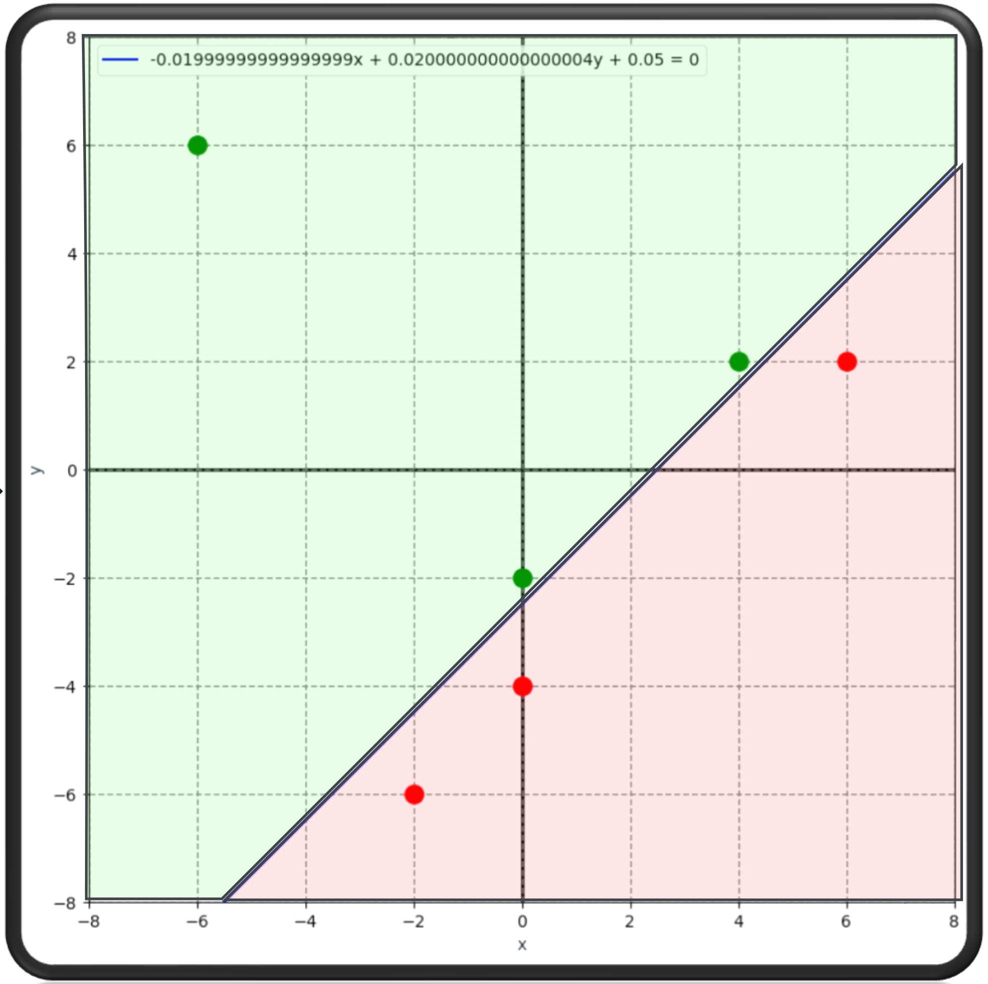
Demo

5. Testing each entry (from 1 to 6) against the current model.

Entry	$Input_1$ (X)	$Input_2$ (Y)	Label (Color)
#1	4	2	1 (■)
#2	-6	6	1 (■)
#6	6	2	-1 (■)

Notice that at this point we have found a hyper-plane that separates all green dots from the red ones. At this point we can stop the algorithm.

$Weight_1$	$Weight_2$	Threshold (β)
-0.02	**0.02**	**0.05**



Batch training

Batch training

Let's consider the previous example, but with different arrangements of the order of the elements in the dataset.

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#5	-2	-6	-1 (■)
#3	0	-2	1 (■)
#2	-6	6	1 (■)
#4	0	-4	-1 (■)
#6	6	2	-1 (■)

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

All of these databases contain the same entries just organized in different order. As such, the training will be different for each case. All of these cases will eventually reach a point where the hyperplane separates all green and red dots. For each case we will record how many iterations (epochs) it takes to find the best hyperplane, and how many changes to the original weight vector it takes.

Batch training

Let's consider the previous example, but with different arrangements of the order of the elements in the dataset.

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#2	-6	6	1 (■)
#3	0	-2	1 (■)
#4	0	-4	-1 (■)
#5	-2	-6	-1 (■)
#6	6	2	-1 (■)

Epochs=2,Changes=4

#	$Input_1$	$Input_2$	Label
#1	4	2	1 (■)
#5	-2	-6	-1 (■)
#3	0	-2	1 (■)
#2	-6	6	1 (■)
#4	0	-4	-1 (■)
#6	6	2	-1 (■)

Epochs=5,Changes=15

#	$Input_1$	$Input_2$	Label
#3	0	-2	1 (■)
#2	-6	6	1 (■)
#6	6	2	-1 (■)
#5	-2	-6	-1 (■)
#1	4	2	1 (■)
#4	0	-4	-1 (■)

Epochs=9,Changes=22

#	$Input_1$	$Input_2$	Label
#5	-2	-6	-1 (■)
#3	0	-2	1 (■)
#2	-6	6	1 (■)
#1	4	2	1 (■)
#6	6	2	-1 (■)
#4	0	-4	-1 (■)

Epochs=10,Changes=28

As seen, different shuffles of the same database produce different results.

Batch training

Considering that:

- At sample "t" from the training we have a "w_t" and that we need to change it because it is not correctly classified

- Then at sample "t+1" we will have:

$$w_{(t+1)} = w_{(t)} + I_{t+1} \times \alpha \times Label_{t+1}$$

- This means that the way current algorithm is, we can know in advance what is the value of w_{t+1} unless we have already computed w_t. Furthermore, because of this, we can not paralyze the algorithm.

Batch training

Let's write the original algorithm (this time using threshold):

```
w ← a vector of "n" elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
β ← a random value or 0
repeat
  foreach sample in trainset
    classified ← positive(sample.label x (dotproduct(sample.input,w) + β))>0
    if classified then continue
    w ← w + sample.input x α x sample.label
    β ← β + α x sample.label
  end foreach
until exit_condition
```

Batch training

Now let's see how we can modify it to work with a batch:

Pseudocode

```
w ← a vector of "n" elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
β ← a random value or 0
repeat
  Δ ← a vector filled with 0 (zeros) of size "n"
  B ← 0 // a temporary threshold
  foreach sample in trainset
    classified ← positive(sample.label x (dotproduct(sample.input,w) + β))>0
    if classified then continue
    Δ ← Δ + sample.input x α x sample.label
    B ← B + α x sample.label
  end foreach
  w ← w + Δ
  β ← β + B
until exit_condition
```

Batch training

Now let's see how we can modify it to work with a batch:

Line	Pseudocode
	w ← a vector of "n" elements initialized randomly or with 0
	trainset ← the training set
	α ← 0.01 // learning rate
	β ← a random value or 0
	repeat
	Δ ← a vector filled with 0 (zeros) of size "n"
	B ← 0 // a temporary threshold
	foreach sample **in** trainset
	(sample.input,w) + β))>0
	w ← w + Δ
	β ← β + B
	until exit_condition

Notice that we have introduced 2 new variables (Δ and B) both of them initialized with 0. They store the relative change that needs to be applied to the hyperplane.

Batch training

Now let's see how we can modify it to work with a batch:

Line Pseudocode

```
w ← a vector of "n" elements initialized randomly or with 0
tr
α
β
re
  
  B ← 0 // a temporary threshold
  foreach sample in trainset
    classified ← positive(sample.label x (dotproduct(sample.input,w) + β))>0
    if classified then continue
    Δ ← Δ + sample.input x α x sample.label
    B ← B + α x sample.label
  end foreach
  w ← w + Δ
  β ← β + B
until exit_condition
```

For every sample, we check it against the current hyper-plane (w and β). If the sample is not correctly classified, we modify the new parameters instead of the hyperplane. This makes the process be invariant to sample order.

Batch training

Now let’s see how we can modify it to work with a batch:

Line	Pseudocode
1	w ← a vector of “n” elements initialized randomly or with 0
2	trainset ← the training set
3	α ← 0.01 // learning rate
4	β ← a random value or 0
5	**repeat**
	eros) of size “n”
	d
	e.label x (**dotproduct**(sample.input,w) + β))>0
	sample.label
12	B + α x sample.label
13	**end foreach**
14	w ← w + Δ
15	β ← β + B
16	**until** exit_condition

Finally, after every sample from the training set has been validated against the current hyperplane and we have computed the differences from all samples, we adjust the hyperplane and resume the process.

Batch training

Now let's see how we can modify it to work with a batch:

Pseudocode

```
w ← a vector of "n" elements initialized randomly or with 0
trainset ← the training set
```

> To simplify the process, let's consider the following code as a separate function defined as follows:
> `train(trainingSet, weights, beta) → (Δ, B)`

```
  B ← 0 // a temporary threshold
  foreach sample in trainset
    classified ← positive(sample.label x (dotproduct(sample.input,w) + β))>0
    if classified then continue
    Δ ← Δ + sample.input x α x sample.label
    B ← B + α x sample.label
  end foreach
  w ← w + Δ
  β ← β + B
until exit_condition
```

Batch training

This is how the train function looks like:

Train function:

```
function train(trainset, w, β)
  α ← 0.01 // learning rate
  Δ ← a vector filled with 0 (zeros) of same size of w
  B ← 0 // a temporary threshold
  foreach sample in trainset
    classified ← positive(sample.label x (dotproduct(sample.input,w) + β))>0
    if classified then continue
    Δ ← Δ + sample.input x α x sample.label
    B ← B + α x sample.label
  end foreach
  return (Δ, B)
end function
```

Batch training

With this in mind we can modify the original algorithm as follows:

```
Pseudocode
w ← a vector of “n” elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
β ← a random value or 0
repeat
  Δ, B ← train (trainset, w, β)
  w ← w + Δ
  β ← β + B
until exit_condition
```

Now that we have changed the code in this way we can do some other things as well:

- Mini batch
- Parallel training

Batch training

Let's consider the following functions:

A) *split*(sampleset, number) → this method splits the sample set into multiple (disjunctive) sample sets provided by the parameter number. For example, assuming:
   - Sample set has 100 entries
   - We want to split in 4 batches
   - We will call split(sampleset,4). This will result in 4 sample sets: one that contains the first 25 entries, the second one that contains the entries from index 26 to index 50, and so on.

B) *thread::run*(command) → this will execute a specific command but on a different thread

Batch training

A mini-batch consists in the following:

Pseudocode

```
w ← a vector of “n” elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
β ← a random value or 0
batches ← split (trainset, nr_of_batches)
repeat
  foreach batch in batches
     Δ, B ← train (batch, w, β)
     w ← w + Δ
     β ← β + B
  end foreach
until exit_condition
```

Batch training

A parallel processing will look like this:

Pseudocode

```
w ← a vector of “n” elements initialized randomly or with 0
trainset ← the training set
α ← 0.01 // learning rate
β ← a random value or 0
batches ← split (trainset, number_of_threads)
repeat
  for index in 0..len(batches)
    Δ_index, B_index ← thread::run(train (batches[index], w, β))
  end for
  wait for all threads to finish
  for index in 0..len(batches)
    w ← w + Δ_index
    β ← β + B_index
  end for
until exit_condition
```

Overview training

Item	Online-training	Batch Training	Mini-batch Training
Support parallelism	No	Yes	Yes
Order is relevant	Yes	No	Yes (partial)

A couple of additional observations:

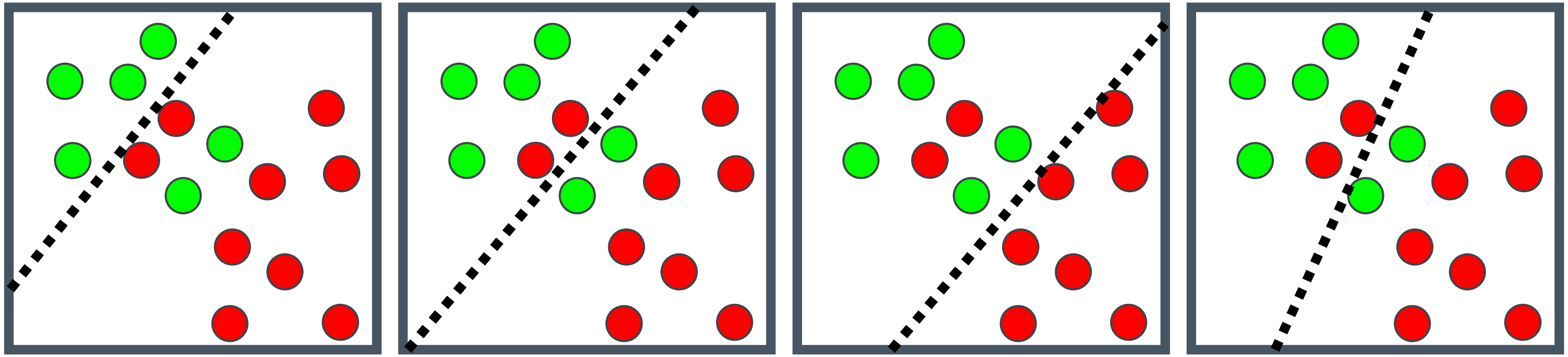
- Batch training (if executed in parallel) is mutch faster than the regular training algorithm (online training)
- It is recommended to use batch training as it will compute the overall change to the weight vector (as such it is more likely to avoid scenarios that imply various forms of numeric overflows)
- If online training or mini-batch training is being used, it is recommended to shuffle the data set before training.

Adaline perceptron

Adaline perceptron

The training algorithm as discuss up to this moment has some limitations:

1. If the data set is not linear separable, the algorithm **<u>will never be able to achieve a stable state</u>** (the hyperplane will jump from one position to another)

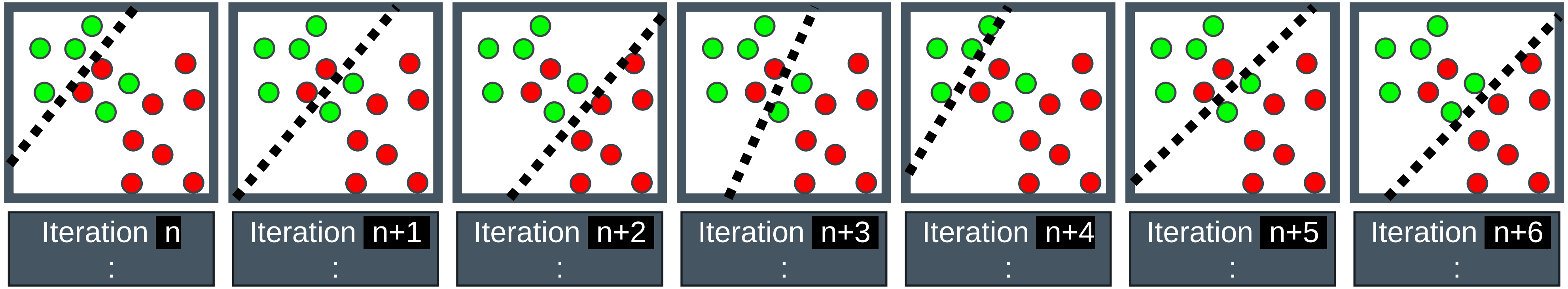


Iteration n : some green dots are not correctly classified	Iteration n+1 : some red and green dots are not correctly classified	Iteration n+2 : some red dots are not correctly classified	Iteration n+3 : some red and green dots are not correctly classified

Adaline perceptron

The training algorithm as discuss up to this moment has some limitations:

1. If the data set is not linear separable, the algorithm **will never be able to achieve a stable state** (the hyperplane will jump from one position to another)



So, in fact, the hyper plane keeps moving between a couple of points, but it never achieves a stable state.

Adaline perceptron

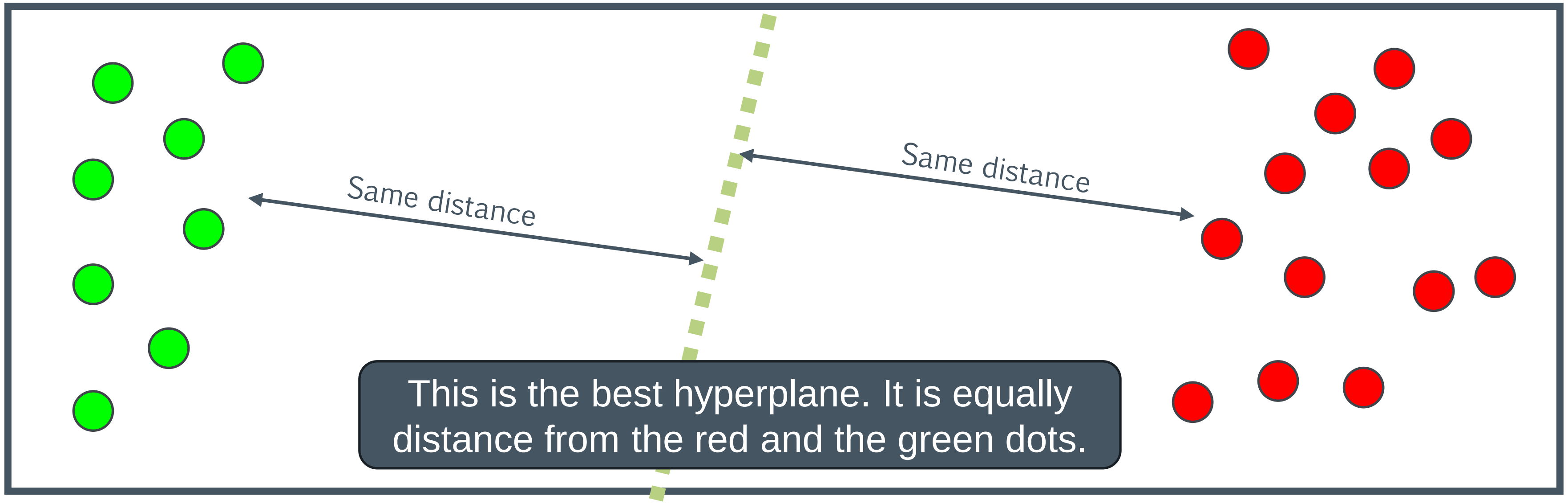
The training algorithm as discuss up to this moment has some limitations:

2. If the data is linear separable, the hyperplane obtained **is not the best one** (the one the best generalize the data) because we don't change any weight if a sample is already classified.

Adaline perceptron

The training algorithm as discuss up to this moment has some limitations:

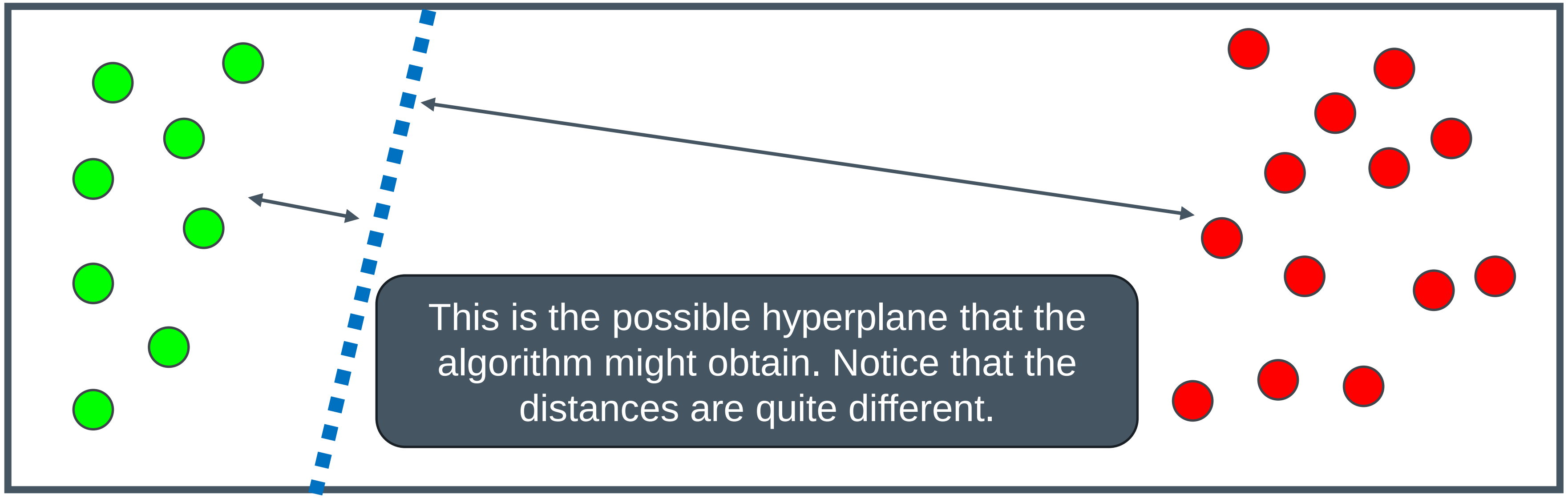
2. If the data is linear separable, the hyperplane obtained **<u>is not the best one</u>** (the one the best generalize the data) because we don’t change any weight if a sample is already classified.



Adaline perceptron

The training algorithm as discuss up to this moment has some limitations:

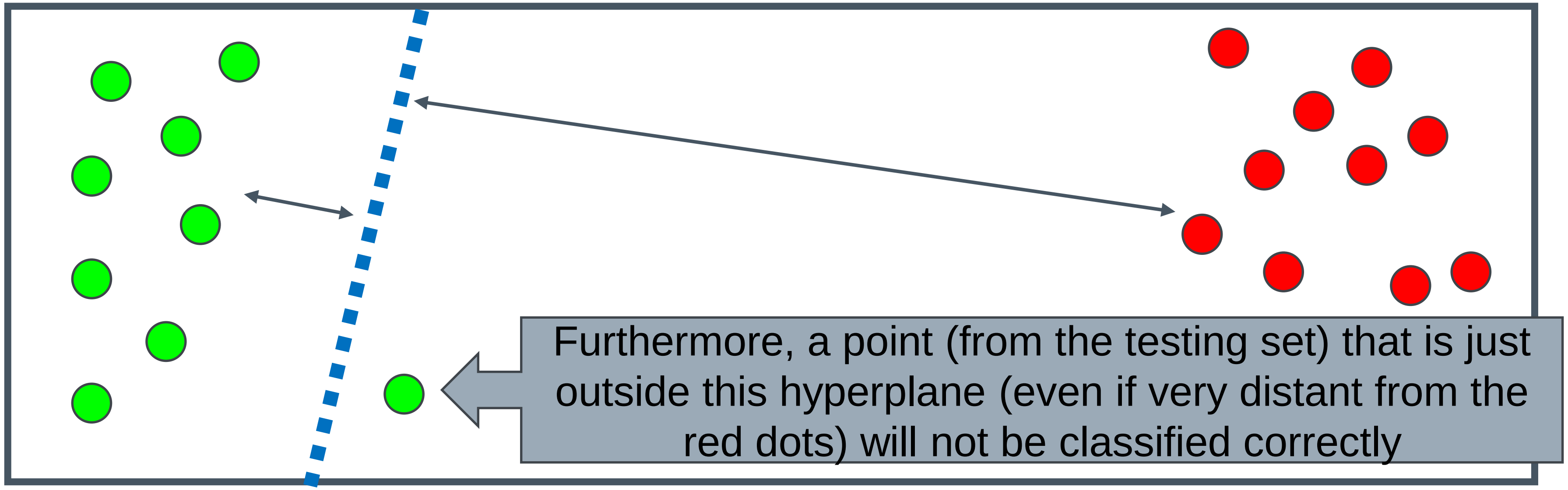
2. If the data is linear separable, the hyperplane obtained **is not the best one** (the one the best generalize the data) because we don’t change any weight if a sample is already classified.



Adaline perceptron

The training algorithm as discuss up to this moment has some limitations:

2. If the data is linear separable, the hyperplane obtained **is not the best one** (the one the best generalize the data) because we don't change any weight if a sample is already classified.



Adaline perceptron

The solution is to take into consideration how big the error is (event for the cases where a sample is correctly classified).

Line	Train function:
1	`function train(trainset, w, β)`
2	`  α ← 0.01 // learning rate`
3	`  Δ ← a vector filled with 0 (zeros) of same size of w`
4	`  B ← 0 // a temporary threshold`
5	`  foreach sample in trainset`
6	`    classified ← sign(sample.label x (dotproduct(sample.input,w) + β))>0`
7	~~if classified then continue~~
8	`    Δ ← Δ + sample.input x α x sample.label`
9	`    B ← B + α x sample.label`
10	`  end foreach`
11	`  return (Δ, B)`
12	`end function`

We just need to delete this line ☺

Q & A