# Perceptron Learning Algorithm: A Graphical Explanation Of Why It Works

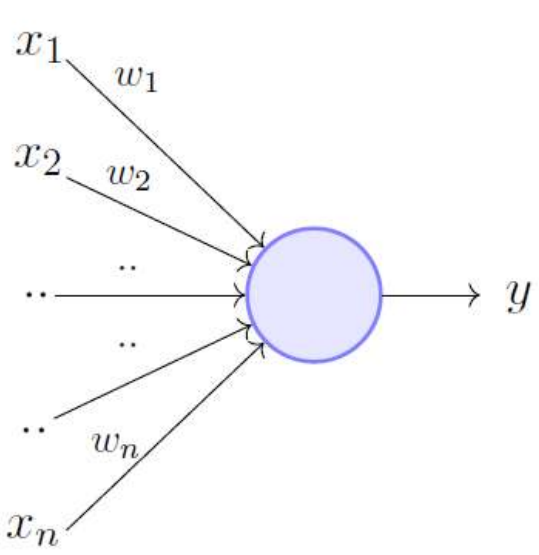
Akshay L Chandra
Aug 23, 2018 · 8 min read

This post will discuss the famous *Perceptron Learning Algorithm* proposed by Minsky and Papert in 1969. This is a follow-up post of my previous posts on the McCulloch-Pitts neuron model and the Perceptron model.

*Citation Note: The concept, the content, and the structure of this article were inspired by the awesome lectures and the material offered by Prof.* ***Mitesh M. Khapra*** *on NPTEL's Deep Learning course.*

## Perceptron

You can just go through my previous post on the perceptron model (linked above) but I will assume that you won't. So here goes, a perceptron is not the Sigmoid neuron we use in ANNs or any deep learning networks today.




$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

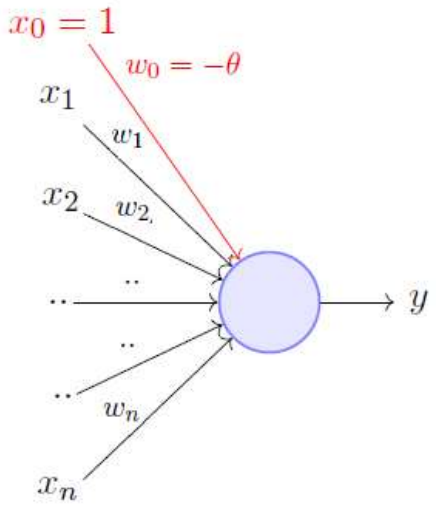
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

The perceptron model is a more general computational model than McCulloch-Pitts neuron. It takes an input, aggregates it (weighted sum) and returns 1 only if the aggregated sum is more than some threshold else returns 0. Rewriting the threshold as shown above and making it a constant input with a variable weight, we would end up with something like the following:




A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

A single perceptron can only be used to implement **linearly separable** functions. It takes both real and boolean inputs and associates a set of **weights** to them, along with a **bias** (the threshold thing I mentioned above). We learn the weights, we get the function. Let's use a perceptron to learn an OR function.

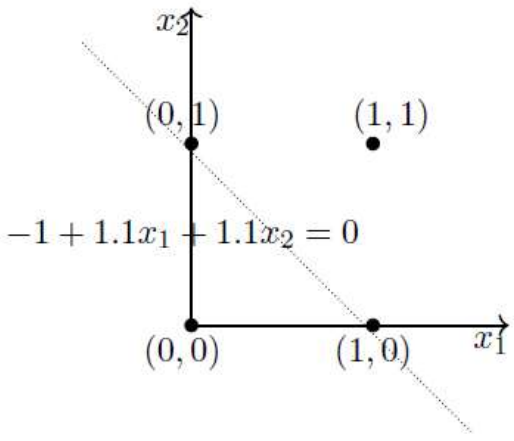
## OR Function Using A Perceptron

| $x_1$ | $x_2$ | OR | |
|---|---|---|---|
| 0 | 0 | 0 | $w_0 + \sum_{i=1}^{2} w_i x_i < 0$ |
| 1 | 0 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 0 | 1 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |
| 1 | 1 | 1 | $w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$ |

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is

$$w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$$



What's going on above is that we defined a few conditions (the weighted sum has to be more than or equal to 0 when the output is 1) based on the OR function output for various sets of inputs, we solved for weights based on those conditions and we got a line that perfectly separates positive inputs from those of negative.

Doesn't make any sense? Maybe now is the time you go through that post I was talking about. Minsky and Papert also proposed a more principled way of learning these weights using a set of examples (data). Mind you that this is NOT a Sigmoid neuron and we're not going to do any Gradient Descent.

# Warming Up — A Few Basics Of Linear Algebra

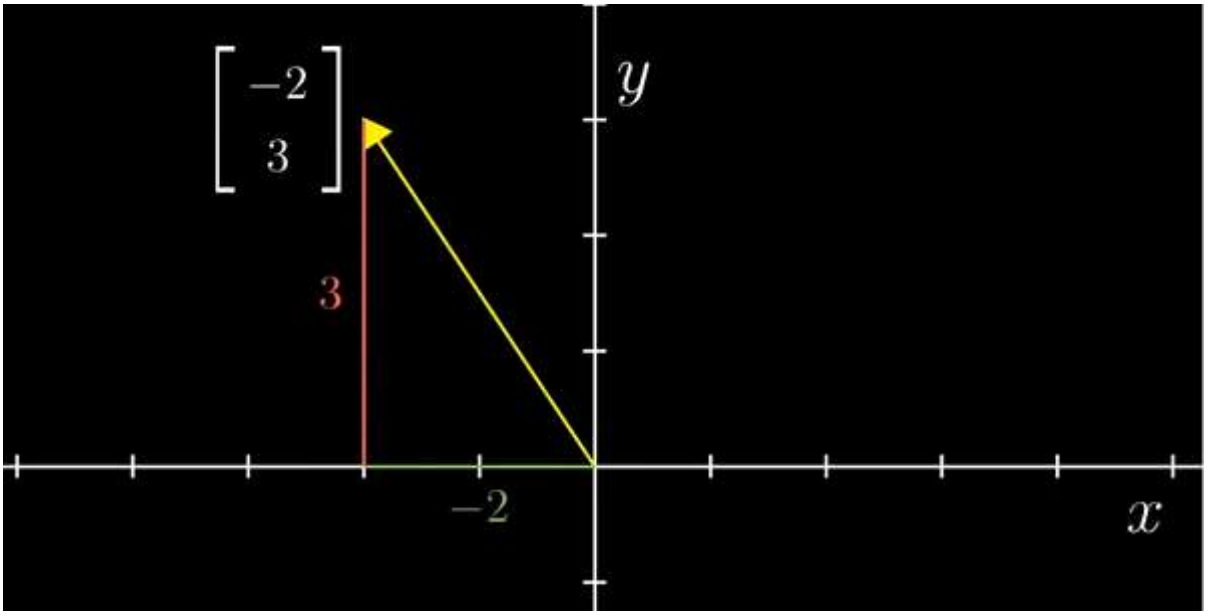
## Vector

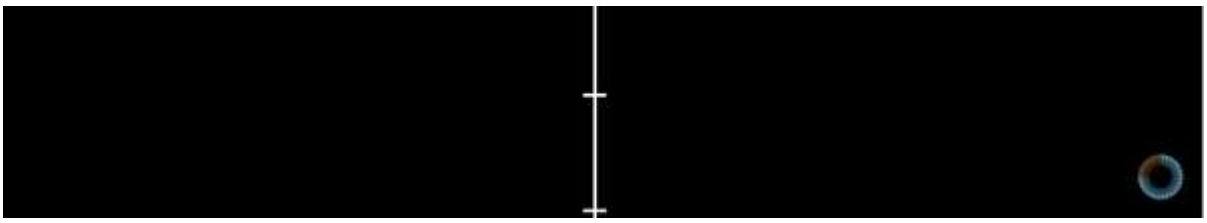
A vector can be defined in more than one way. For a physicist, a vector is anything that sits anywhere in space, has a magnitude and a direction. For a CS guy, a vector is just a data structure used to store some data — integers, strings etc. For this tutorial, I would like you to imagine a vector the Mathematician way, where a vector is an arrow spanning in space with its tail at the origin. This is not the best mathematical way to describe a vector but as long as you get the intuition, you're good to go.

*Note: I have borrowed the following screenshots from 3Blue1Brown's video on Vectors. If you don't know him already, please check his series on Linear Algebra and Calculus. He is just out of this world when it comes to visualizing Math.*

## Vector Representations

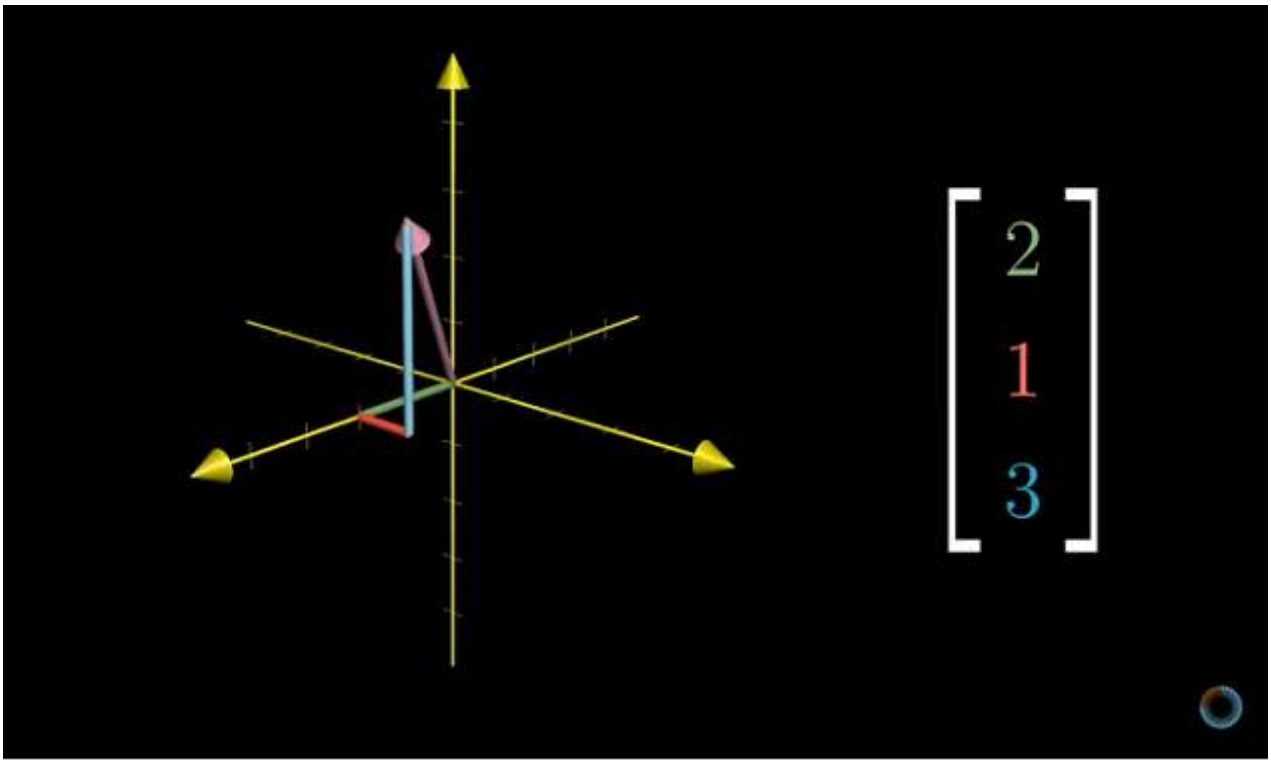
A 2-dimensional vector can be represented on a 2D plane as follows:





Source: 3Blue1Brown's video on Vectors

Carrying the idea forward to 3 dimensions, we get an arrow in 3D space as follows:




Source: 3Blue1Brown's video on Vectors

## Dot Product Of Two Vectors

At the cost of making this tutorial even more boring than it already is, let's look at what a dot product is. Imagine you have two vectors oh size **n+1**, **w** and **x**, the dot product of these vectors (***w.x***) could be computed as follows:

$$\mathbf{w} = [w_0, w_1, w_2, ..., w_n]$$
$$\mathbf{x} = [1, x_1, x_2, ..., x_n]$$
$$\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^{\mathbf{T}}\mathbf{x} = \sum_{i=0}^{n} w_i * x_i$$

The transpose is just to write it in a matrix multiplication form.

Here, **w** and **x** are just two lonely arrows in an **n+1 dimensional** space (and intuitively, their dot product quantifies how much one vector is going in the direction of the other). So technically, the perceptron was only computing a lame dot product (before checking if it's greater or lesser than 0). The decision boundary line which a perceptron gives out that separates positive examples from the negative ones is really just $\mathbf{w} \cdot \mathbf{x} = 0$.

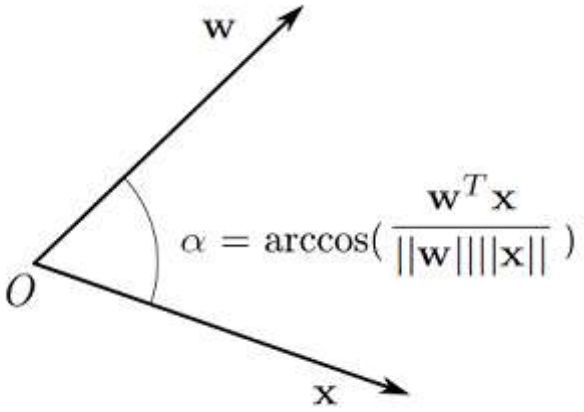
## Angle Between Two Vectors

Now the same old dot product can be computed differently if only you knew the angle between the vectors and their individual magnitudes. Here's how:

$$\mathbf{w}^T\mathbf{x} = \|\mathbf{w}\|\|\mathbf{x}\| cos\alpha$$

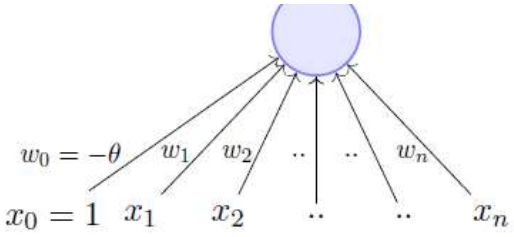
The other way around, you can get the angle between two vectors, if only you knew the vectors, given you know how to calculate vector magnitudes and their vanilla dot product.

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||}$$




When I say that the cosine of the angle between **w** and **x** is 0, what do you see? I see arrow **w** being perpendicular to arrow **x** in an n+1 dimensional space (in 2-dimensional space to be honest). So basically, when the dot product of two vectors is 0, they are perpendicular to each other.

## Setting Up The Problem




$x_3 = isDirectorNolan$
$x_4 = imdbRating(scaled\ to\ 0\ to\ 1)$
... ...
$x_n = criticsRating(scaled\ to\ 0\ to\ 1)$

We are going to use a perceptron to estimate if I will be watching a movie based on historical data with the above-mentioned inputs. The data has positive and negative examples, positive being the movies I watched i.e., 1. Based on the data, we are going to learn the weights using the perceptron learning algorithm. For visual simplicity, we will only assume two-dimensional input.

## Perceptron Learning Algorithm

Our goal is to find the **w** vector that can perfectly classify positive inputs and negative inputs in our data. I will get straight to the algorithm. Here goes:

**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs\ \ with\ \ label\ \ 1;$
$N \leftarrow inputs\ \ with\ \ label\ \ 0;$
Initialize **w** randomly;
**while** $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P\ \ and\ \ \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N\ \ and\ \ \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

We initialize **w** with some random vector. We then iterate over all the examples in the data, ($P$ U $N$) both positive and negative examples. Now if an input **x** belongs to $P$,

ideally what should the dot product **w.x** be? I'd say greater than or equal to 0 because that's the only thing what our perceptron wants at the end of the day so let's give it that. And if **x** belongs to *N*, the dot product MUST be less than 0. So if you look at the if conditions in the while loop:

```
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P and w.x < 0 then
        w = w + x ;
    end
    if x ∈ N and w.x ≥ 0 then
        w = w − x ;
    end
end
```

**Case 1:** When **x** belongs to *P* and its dot product **w.x** $< 0$
**Case 2:** When **x** belongs to *N* and its dot product **w.x** $\geq 0$

Only for these cases, we are updating our randomly initialized **w**. Otherwise, we don't touch **w** at all because Case 1 and Case 2 are violating the very rule of a perceptron. So we are adding **x** to **w** (ahem vector addition ahem) in Case 1 and subtracting **x** from **w** in Case 2.

## Why Would The Specified Update Rule Work?

But why would this work? If you get it already why this would work, you've got the entire gist of my post and you can now move on with your life, thanks for reading, bye. But if you are not sure why these seemingly arbitrary operations of **x** and **w** would help you learn that perfect **w** that can perfectly classify *P* and *N*, stick with me.

We have already established that when **x** belongs to *P*, we want **w.x** $> 0$, basic perceptron rule. What we also mean by that is that when **x** belongs to *P*, the angle between **w** and **x** should be _____ than 90 degrees. Fill in the blank.

Answer: The angle between **w** and **x** should be less than 90 because the cosine of the angle is proportional to the dot product.

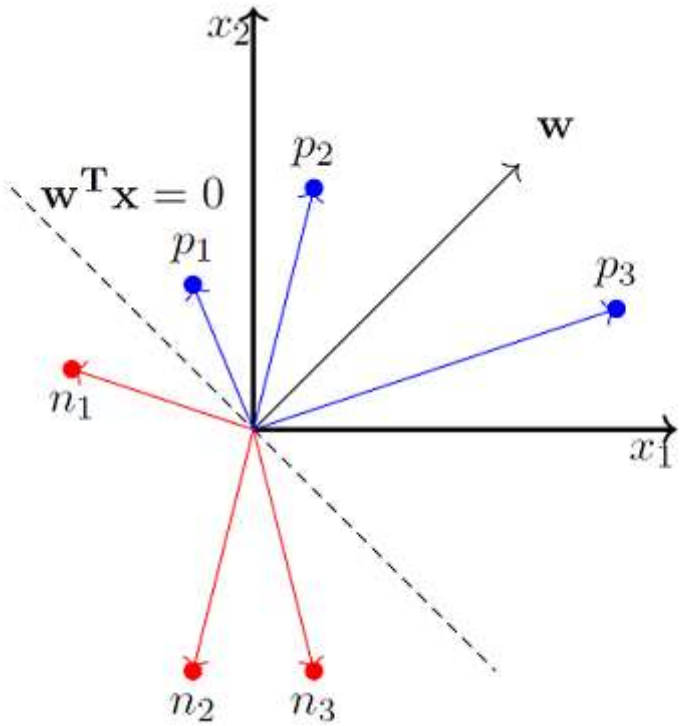
$$\mathbf{w}^T\mathbf{x} \quad | \quad T$$

$$cos\alpha = \frac{}{||\mathbf{w}||||\mathbf{x}||} \quad \Big| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \Rightarrow cos\alpha > 0 \Rightarrow \alpha < 90$$

$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \Rightarrow cos\alpha < 0 \Rightarrow \alpha > 90$$

So whatever the **w** vector may be, as long as it makes an angle less than 90 degrees with the positive example data vectors (**x** E *P*) and an angle more than 90 degrees with the negative example data vectors (**x** E *N*), we are cool. So ideally, it should look something like this:




x_0 is always 1 so we ignore it for now.

So we now strongly believe that the angle between **w** and **x** should be less than 90 when **x** belongs to *P* class and the angle between them should be more than 90 when **x** belongs to *N* class. Pause and convince yourself that the above statements are true and you indeed believe them. Here's why the update works:

$$(\alpha_{new}) \text{ when } \mathbf{w_{new}} = \mathbf{w} + \mathbf{x} \quad \Big| \quad (\alpha_{new}) \text{ when } \mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$$

$$cos(\alpha_{new}) \propto \mathbf{w_{new}}^T\mathbf{x} \quad \Big| \quad cos(\alpha_{new}) \propto \mathbf{w_{new}}^T\mathbf{x}$$

$$\propto (\mathbf{w} + \mathbf{x})^T\mathbf{x} \quad \Big| \quad \propto (\mathbf{w} - \mathbf{x})^T\mathbf{x}$$

$$\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \qquad \propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x}$$

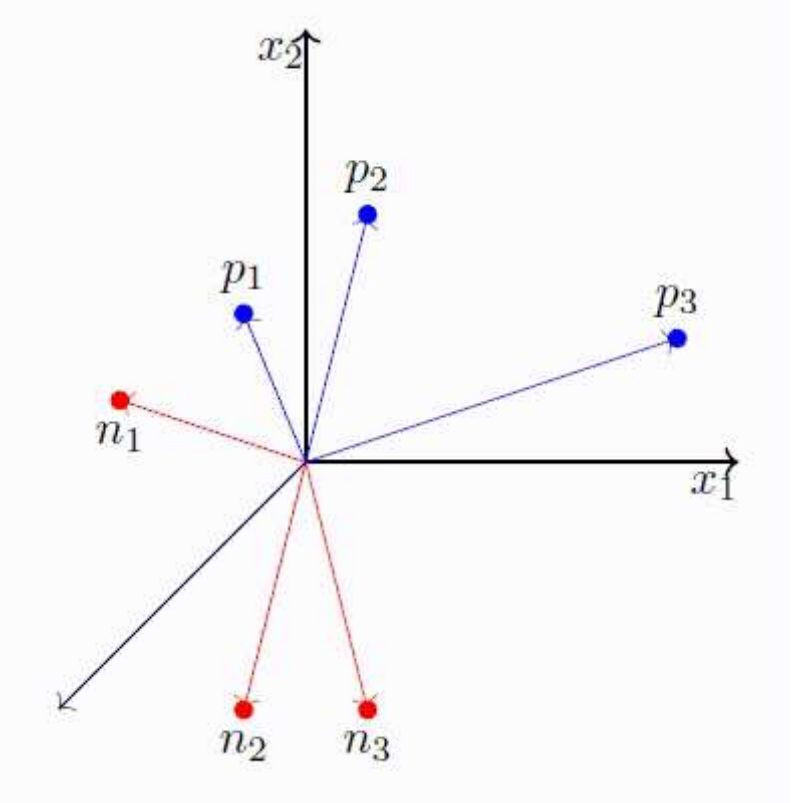
$$\propto cos\alpha + \mathbf{x}^T\mathbf{x} \qquad \propto cos\alpha - \mathbf{x}^T\mathbf{x}$$

$$cos(\alpha_{new}) > cos\alpha \qquad cos(\alpha_{new}) < cos\alpha$$

Now this is slightly inaccurate but it is okay to get the intuition.

So when we are adding **x** to **w**, which we do when x belongs to P and **w.x** < 0 (Case 1), we are essentially **increasing the *cos(alpha)*** value, which means, we are **decreasing the *alpha* value**, the angle between **w** and **x**, **which is what we desire.** And the similar intuition works for the case when **x** belongs to *N* and **w.x** ≥ 0 (Case 2).

Here's a toy simulation of how we might up end up learning **w** that makes an angle less than 90 for positive examples and more than 90 for negative examples.




We start with a random vector **w**.

## Proof Of Convergence

Now, there is no reason for you to believe that this will definitely converge for all kinds of datasets. It seems like there might be a case where the **w** keeps on moving around and never converges. But people have proved it that this algorithm converges. I am

attaching the proof, by Prof. Michael Collins of Columbia University — find the paper here.

## Conclusion

In this post, we quickly looked at what a perceptron is. We then warmed up with a few basics of linear algebra. We then looked at the *Perceptron Learning Algorithm* and then went on to visualize why it works i.e., how the appropriate weights are learned.

Thank you for reading this post.
Live and let live!
A

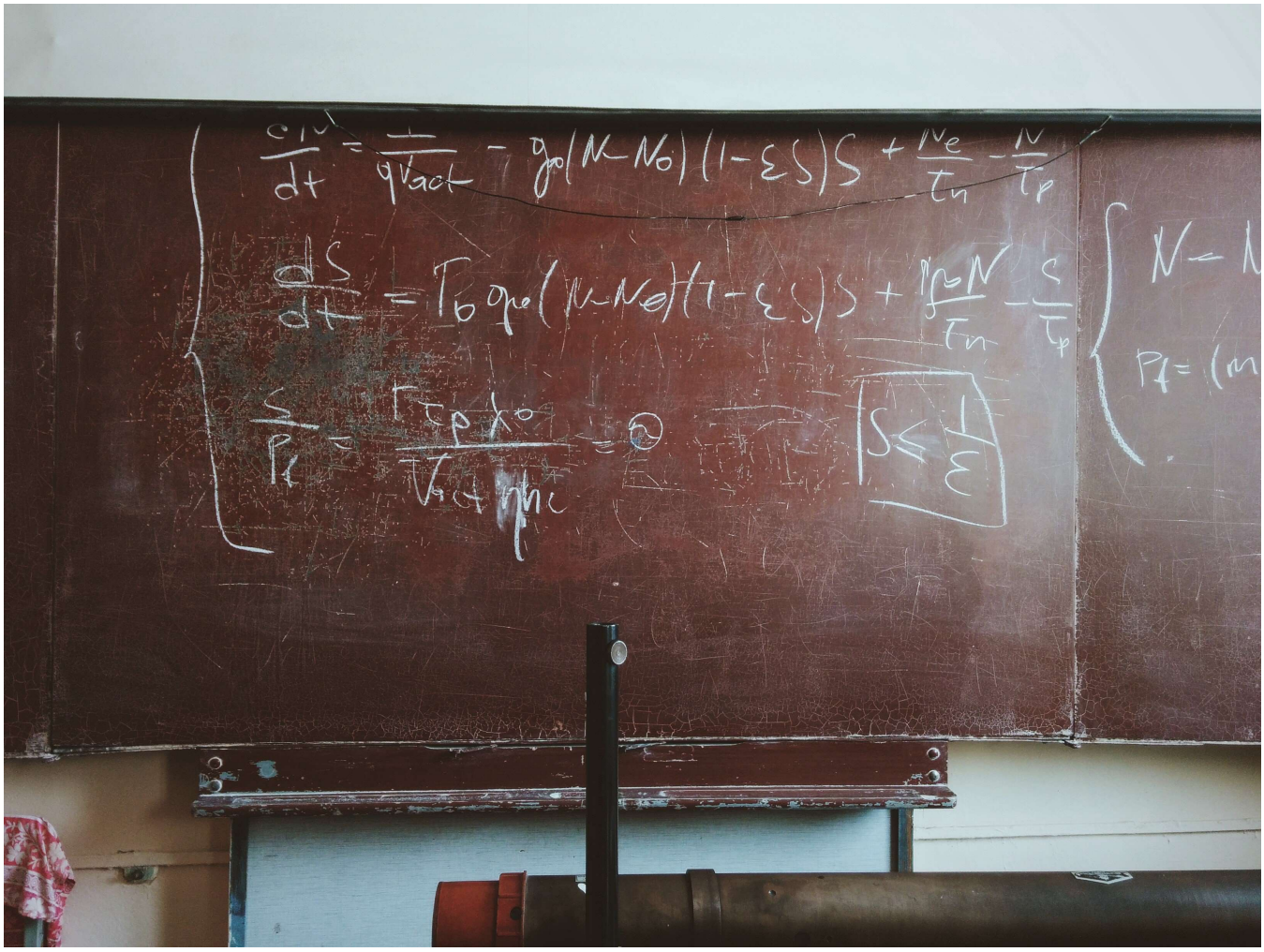


Photo by Roman Mager on Unsplash

Perceptron Artificial Neural Network Deep Learning