**Perceptron Learning Algorithm**

[[](https://medium.com/@francescofranco_39234?source=post_page-----1e4eaef887--------------------------------)](https://medium.com/@francescofranco_39234?source=post_page-----1e4eaef887--------------------------------)

[[AI Mind](https://pub.aimind.so/?source=post_page-----1e4eaef887--------------------------------)](https://pub.aimind.so/?source=post_page-----1e4eaef887--------------------------------)

[Francesco Franco](https://medium.com/@francescofranco_39234?source=post_page-----1e4eaef887--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ffe44f5fcf1d3&operation=register&redirect=https%3A%2F%2Fpub.aimind.so%2Fperceptron-learning-algorithm-1e4eaef887&user=Francesco+Franco&userId=fe44f5fcf1d3&source=post_page-fe44f5fcf1d3----1e4eaef887---------------------post_header-----------)

Published in

[AI Mind](https://pub.aimind.so/?source=post_page-----1e4eaef887--------------------------------)

·

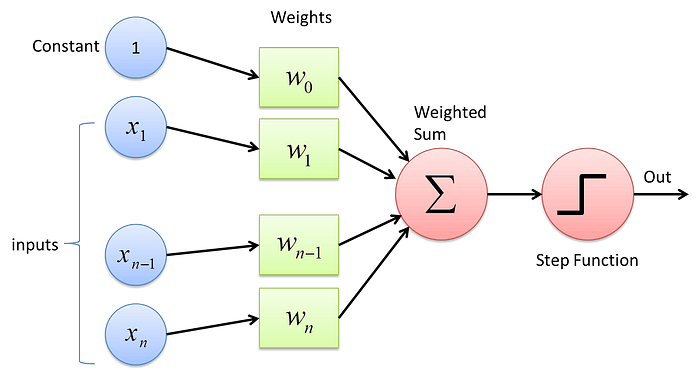
9 min read

·

Aug 25

222

1



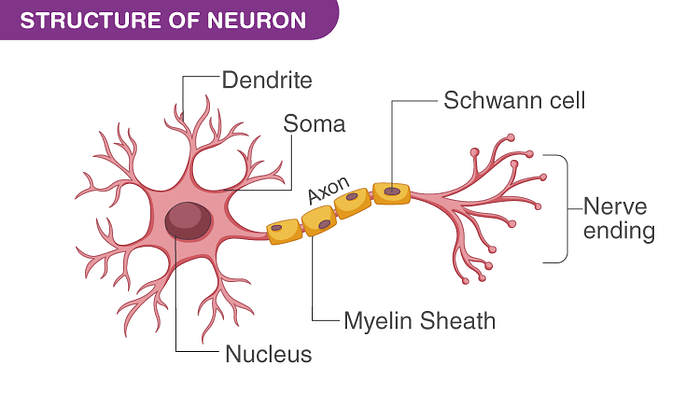
Basic perceptron Credit:[DeepAI.org](https://deepai.org/machine-learning-glossary-and-terms/perceptron)

So far we have discussed two types of learning models: in decision trees, only a small number of features are used to make decisions; in nearest neighbor algorithms, all features are used equally. Neither of these extremes is always desirable. In some problems, we might want to use most of the features, but some more than others.

Something like that can be done by using the perceptron algorithm for learning **weights** for features**.**It turns out that learning weights for features amounts to learning a**hyperplane**classifier: that is, basically a division of space into two halves by a straight line, where one half is “positive” and the other half is “negative.” In this sense, the perceptron can be seen as explicitly finding a good**linear decision boundary.**

**Biology Inspired Learning**

Folk biology tells us that our brains are made up of a bunch of little units called **neurons,** that send electrical signals to one another.



credit: [BYJU’s](https://byjus.com/biology/neurons/)

The rate of firing tells us how “activated” a neuron is. A single neuron, like that shown above, might have three incoming neurons. These incoming neurons are firing at different rates (i.e. have different **activations)**

Based on how much these incoming neurons are firing, and how strong the neural connections are, our main neuron will “decide” how strongly it wants to fire. And so on through the whole brain. Learning in the brain happens by neurons becoming connected to other neurons, and the strengths of connections adapting over time.

The real biological world is much more complicated than this, of course. However, our goal is not to build a brain, but to simply be *inspired*by how they work.

We can think of our learning algorithm as a single neuron. It receives input from D many other neurons, one for each input feature. The strength of these inputs are the feature values.

Each incoming connection has a weight and the neuron simply sums up all the weighted inputs. Based on this sum, it decides whether to “fire” or not. Firing is interpreted as being a “positive” example and not firing is interpreted as being a “negative” example. In particular, if the weighted sum is positive, it “fires” and otherwise it doesn’t fire.

Mathematically, an input vector **x= <**x1, x2,…xD> arrives. The neuron stores D-many weights, w1,w2,…,wD. The neuron computes the sum:



to determine its amount of activation. If this activation is positive (i.e. a >0) it predicts that this example is a positive example. Otherwise it predicts a negative example.

The weights of this neuron are fairly easy to interpret. Suppose that a feature, for example “is this object orange?” gets a zero weight. Then the activation is the same regardless of the value of this feature.

So features with zero weight are ignored. Features with positive weights are indicative of positive examples because they cause the activation to increase. Features with negative weights are indicative of negative examples because the cause the activation to decrease.

It is often convenient to have a non-zero **threshold.**In other words, we might want to predict positive if a > θ for some value of θ. The way that is most convenient to achieve this is to introduce a **bias**term into the neuron, so that the activation is always increased by some fixed value b. Thus, we compute:



This is the complete neural model of learning. The model is parametrized by D-many weights, w1 , w 2 , . . . , wD, and a single scalar bias value.

**The Perceptron Algorithm**

The perceptron is a classic learning algorithm for the neural model of learning.

PerceptronTrain(D,MaxIter)   
  
w\_d <- 0 for all d= 1 to D // initialize weights  
b <- 0 // initialize bias  
for iter=1 to MaxIter do  
 for all (x,y) ∈ D do  
 a ← ∑ w\_d \* x\_d + b // compute activation for this example  
 if ya <= 0 then  
 w\_d ← w\_d + yx\_d, for all d = 1...D // update weights  
 b <- b + y // update bias   
 endif  
 endfor  
endfor  
return w\_1,w\_2,w\_3,...,w\_D,b

PerceptronTest(w\_0,w\_1,...,w\_D,b,x̂ )  
  
a ← ∑ w\_d \* x̂ d + b // compute activation for the test example  
return sing(a)

The algorithm is actually quite different than either the decision tree algorithm or the KNN algorithm. First, it is **online.**This means that instead of considering the entire data set at the same time, it only ever looks at one example.

It processes that example and then goes on to the next one. Second, it is **error-driven.**This means that, so long as it is doing well, it doesn’t bother updating its parameters.

The algorithm maintains a “guess” at good parameters (weights and bias) as it runs. It processes one example at a time.

For a given example, it makes a prediction. It checks to see if this prediction is correct (recall that this is *training data,*so we have access to true labels). If the prediction is correct, it does nothing.

Only when the prediction is incorrect does it change its parameters, and it changes them in such a way that it would do better on this example the next time around. It then goes on to the next example. Once it hits the last example in the training set, it loops back around for a specified number of iterations.

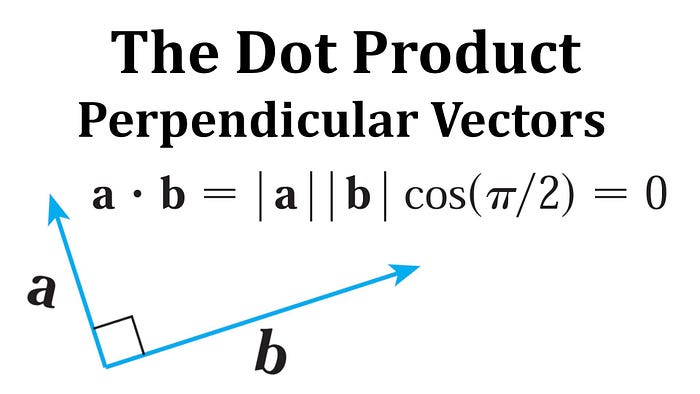
**Geometric Interpretation**

A question that arises now is: what does the decision boundary of a perceptron look like? You can actually answer the question mathematically. For a perceptron, the decision boundary is precisely where the sign of the activation, a, changes from -1 to +1.

In other words, it is the set of points x that achieve *zero*activation. The points that are not clearly positive nor clearly negative. For simplicity, we’ll first consider the case where there is no “bias” term (or, equivalently, the bias is zero). Formally, the decision boundary beta is:



We can now apply some linear algebra. Recall that ∑ w\_d · x\_d is just the**dot product** between the vectors **w =**<w1,w2,…wD> and the vector**x.**This is written as**w** ·**x**. Two vectors have a zero dot product only if they are perpendicular. Thus, if we think of the weights as a vector **w,**then the decision boundary is simply the plane perpendicular to **w.**



Credit: [Math Easy Solutions (YT channel)](https://www.youtube.com/watch?v=1qimuAKpgZU)

From a geometric perspective, the role of the bias is to shift the decision boundary away from the origin, in the direction of**w.**It isshifted exactly -b units. So if b is positive, the boundary is shifted away from **w,**and if b is negative, the boundary is shifted toward**w.**This makes intuitive sense: a positive bias means that more examples should be classified positive. By moving the decision boundary in the negative direction, more space yields a positive classification.

**Perceptron Convergence and Linear Separability**

We already have an intuitive feeling for why the perceptron works: it moves the decision boundary in the direction of the training example. An important question is: does the perceptron converge? If so, what does it converge to? And how long does it take?

What does it mean for the perceptron to converge? It means that it can make an entire pass through the training data without making any more updates. In other words, it has correctly classified every training example. Geometrically, this means that it was found some hyperplane that correctly segregates the data into positive and negative examples.

In this case, this data is **linearly separable**. This means that there  
exists some hyperplane that puts all the positive examples on one side  
and all the negative examples on the other side. If the training is not  
linearly separable then the perceptron has no hope of converging. It could never possibly classify each point correctly.

The somewhat surprising thing about the perceptron algorithm is that if the data is linearly separable, then it will converge to a weight vector that separates the data. (And if the data is inseparable, then it will never converge). This is great news. It means that the perceptron converges whenever it is even remotely possible to converge.

The second question is: how long does it take to converge? By “how long,” what we really mean is “how many updates?” As is the case for much learning theory, you will not be able to get an answer of the form “it will converge after 5293 updates.” This is asking too much. The sort of answer we can hope to get is of the form “it will converge after at most 5293 updates.”

What we might expect to see is that the perceptron will converge more quickly for easy learning problems than for hard learning problems. This certainly fits intuition. The question is how to define “easy” and “hard” in a meaningful way. One way to make this definition is through the notion of **margin**. If I give you a dataset and hyperplane that separates it, then the *margin* is the distance between the hyperplane and the nearest point. Intuitively, problems with large margins should be easy (there’s a lot of “wiggle room” to find a separating hyperplane) ; and problems with small margins should be hard (you really have to get a very specific well tuned weight vector).

Formally, the margin is only defined if **w,**b actually separate the data (otherwise it is -∞). In the case that it separates the data, we find the point with the minimum activation, after the activation is multiplied by the label.

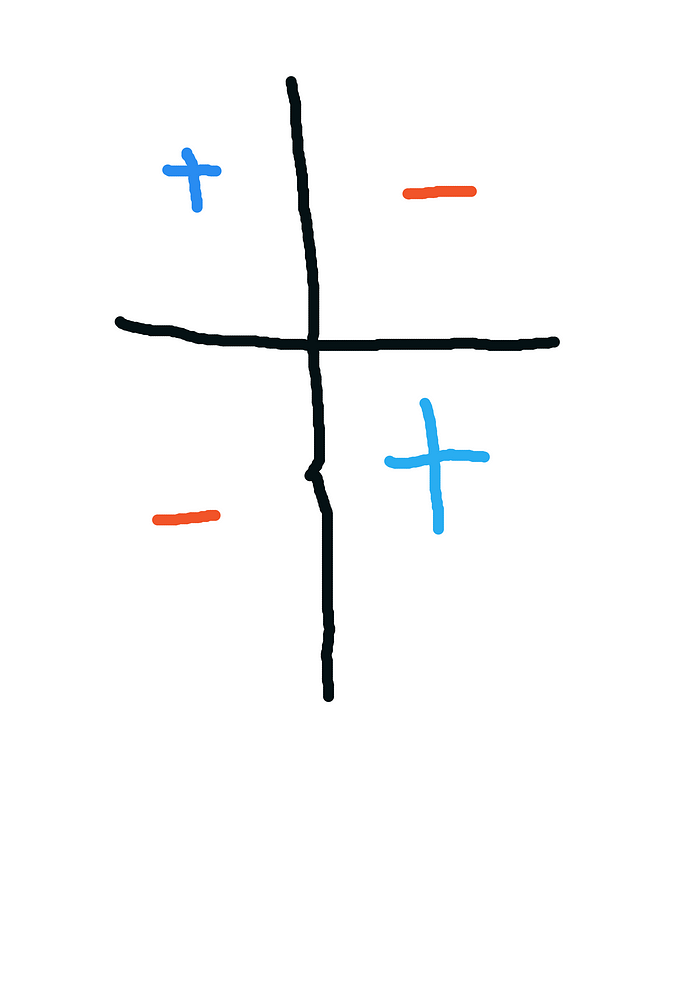
For some historical reason, margins are always demoted by the Greek letter γ (gamma). Often one talks about the**margin of the dataset.**The margin of a dataset is the largest attainable margin on this data. Formally,

margin (D) = max (margin(D, w, b))

In words, to compute the margin of a dataset, you “try” every possible **w,**b pair. For each pair, you compute its margin. We then take the largest of these as the overall margin of the data. If the data is not linearly separable, then the value of the max, and therefore the value of the margin, is −∞.

**Limitations of the Perceptron**

Although the perceptron is very useful, it is fundamentally limited in a way that neither decision trees nor KNN are. Its limitation is that its decision boundaries can only be linear. The classic way of showing this limitation is through the XOR problem (XOR = exclusive or).



The image above consists of four data points, each at a corner of the unit square. The labels for these points are the same, along the diagonals. You can try, but you will not be able to find a linear decision boundary that perfectly separates these data points.

One question you might ask is: do XOR-like problems exist in the real world? Unfortunately for the perceptron, the answer is yes. Consider a sentiment classification problem that has three features that simply say whether a given word is contained in a review of a course. These features are: excellent, terrible and not. The excellent feature is indicative of positive reviews and the terrible feature is indicative of negative reviews. But in the presence of the not feature, this categorization flips.

In fact, the “XOR problem” is so significant that it basically killed  
research in classifiers with linear decision boundaries for a decade  
or two. In some future post, I will discuss two alternative approaches to  
taking key ideas from the perceptron and generating classifiers with  
non-linear decision boundaries. One approach is to combine multiple perceptrons in a single framework: this is the**neural network** approach. The second approach is to find computationally efficient ways of doing feature mapping in a computationally and statistically efficient way: this is the kernels approach.

**Implementation**

The Python implementation of single-layer perceptron learning algorithm using NumPy and Pandas can be found [here.](https://github.com/Francesco601/Perceptron_Learning_Algo/blob/main/perceptron_algorithm.py)

Please click clap and follow if you would like more posts like this. Comments are also welcome. Thanks for reading!

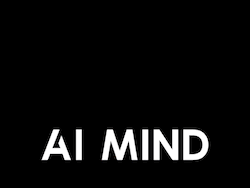
**References**

[1] Simeone, Asvaldo. [A Brief Introduction to Machine Learning for Engineers](https://arxiv.org/pdf/1709.02840.pdf) (2018)

[2] Welling, Max. [A First Encounter With Machine Learning](https://arxiv.org/pdf/1709.02840.pdf). (2011) Wayback Machine.

[3] Pinker, Steven. How the Mind Works. (1998) Penguin Press Science

**A Message from AI Mind**



Thanks for being a part of our community! Before you go:

* 👏 Clap for the story and follow the author 👉
* 📰 View more content in the [AI Mind Publication](https://pub.aimind.so/)
* 🧠 Improve your [AI prompts effortlessly and FREE](https://www.aimind.so/prompt-generator?utm_source=pub&utm_medium=message)
* **🧰 Discover**[**Intuitive AI Tools**](https://www.aimind.so/?utm_source=pub&utm_medium=message)