The Maths behind Neural Networks

Alex Punnen
© All Rights Reserved

Contents

- Chapter 1: The simplest Neural Network - Perceptron using Vectors and Dot Products
- Chapter 2: Perceptron Training via Feature Vectors & HyperPlane split
- Chapter 3: Gradient Descent and Optimization
- Chapter 4: Back Propagation - Pass 1 (Chain Rule)
- Chapter 5: Back propagation - Pass 2 (Scalar Calculus)
- Chapter 6: A Simple NeuralNet with Back Propagation
- Chapter 7: Back Propagation Pass 3 (Matrix Calculus)
- Chapter 8: Back Propagation in Full - With Softmax & CrossEntropy Loss

The Maths behind Neural Networks

Alex Punnen
© All Rights Reserved

Contents

Chapter 1

The simplest Neural Network - Perceptron using Vectors and Dot Products

The aim of this set of articles is to unravel the mathematical concepts used in Deep Learning in as simple a way.

Deep Learning is about neural networks and their structures and designs; and training those networks.

Even the most complex Neural network is based on vectors, matrices and using the concept of cost function and algorithms like gradient descent to find a reduced cost; and then propagating the cost back to all constituents of the network proportionally via a method called back-propagation.

Have you held an integrated circuit or chip in hand or seen. It looks overwhelmingly complex. But its base is the humble transistor and Boolean logic. To understand something complex we need to understand the simpler constituents.

The earliest neural network - the `Rosenblatt's Perceptron` was the first to introduce the concept of using vectors and the property of `vector dot product`, to split hyperplanes of input feature vectors. These are the fundamentals that are still in used today.

First a short refresher.

Vectors

A vector is an object that has both a magnitude and a direction. Example Force and Velocity. Both have magnitude as well as direction.

However we need to specify also a context where this vector lives -Vector Space. For example when we are thinking about something like Force vector, the context is usually 2D or 3D Euclidean world.

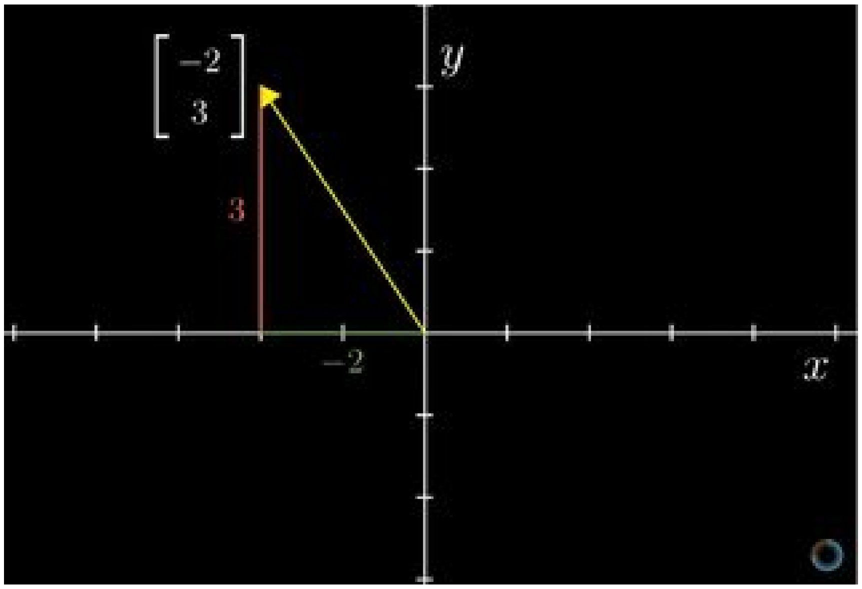

Figure 1: 2D Vector

(Source: 3Blue1Brown)

The easiest way to understand the Vector is in such a geometric context, say 2D or 3D cartesian coordinates, and then extrapolate it for other Vector spaces which we encounter but cannot really imagine.

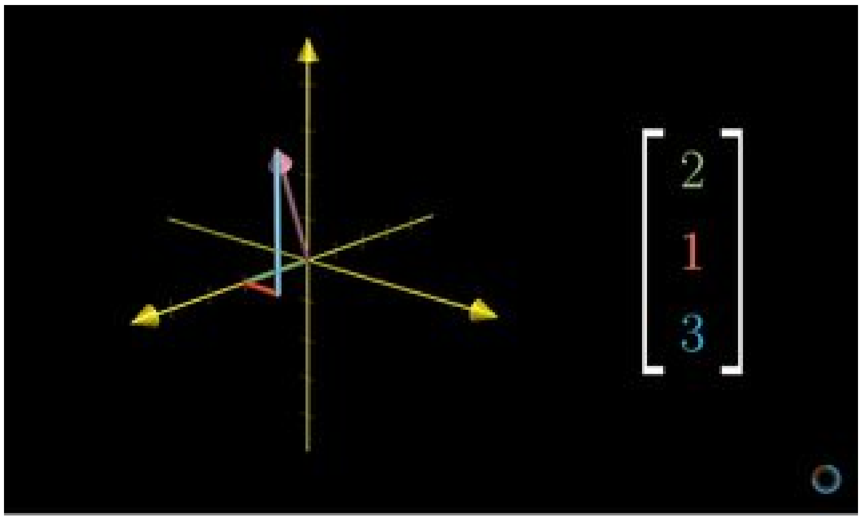

Figure 2: 3D Vector

Matrices - A way to represent Vectors (and Tensors)

Vectors are represented as matrices. A Vector is a one dimensional matrix.A matrix is defined to be a rectangular array of numbers. Example here is a Euclidean Vector in three-dimensional Euclidean space (or R^3) with some magnitude and direction (from (0,0,0) origin in this case).

A vector is represented either as column matrix (m*1)or as a row matrix (1*m).

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$$

a_1, a_2, a_3 are the component scalars of the vector. A vector is represented as $\vec{a}$ in the **Vector notation** and as a_i in the **Index Notation**.

Two dimensional matrices can be thought of as one dimensional vectors stacked on top of each other. This intuition is especially helpful when we use dot products on neural network weight matrices.

Note: - the index notation of matrices is important later on when we start deriving out formulas

In classical machine learning, example for linear regression -(line or curve fitting methodology), matrices play an important part in representing the set of modelled linear equations in matrix form and then using the matrix properties -matrix triangulation methods, to solve the linear equations. See matrix decomposition methods, example LU, QR decomposition.

The other option is to find the optimal solution set by gradient descent, as it is computationally efficient.

In Deep learning, the concept is not of linear regression or set of linear equations; but matrices still are heavily used to model the neural network. Here gradient descent is heavily used to find optimal solution.

Tensors

Since we will be dealing soon with **multidimensional matrices**, it is as well to state here what Tensors are. Easier is to define how they are represented, and that will suit our case as well. We have seen that a Vector is a one dimensional matrix.

Higher dimension matrices are used for Tensors. Example is the multidimensional weight matrices we use in neural network. They are **weight tensors**. A Vector is a Tensor of Rank 1 and technically a Scalar is also a Tensor of Rank 0.

Dot product

Algebraically, the dot product is the sum of the products of the corresponding entries of the two sequences of numbers.

if $\vec{a} = \langle a_1, a_2, a_3 \rangle$ and $\vec{b} = \langle b_1, b_2, b_3 \rangle$, then

$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 = a_i b_i \quad$ in index notation

Geometrically, it is the product of the Euclidean magnitudes of the two vectors and the cosine of the angle between them

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \ \left\|\vec{b}\right\| \cos \theta$$

Note- These definitions are equivalent when using Cartesian coordinates. Here is a simple proof that follows from trigonometry 8 and 9

Dot Product for checking Vector Alignment

If two Vectors are in the same direction the dot product is positive and if they are in the opposite direction the dot product is negative. This can be visualized geometrically putting in the value of the Cosine angle.

So we could use the dot product as a way to find out if two vectors are aligned or not.

Dot Product and Splitting the hyper-plane -the crux of the Perceptron

Image source

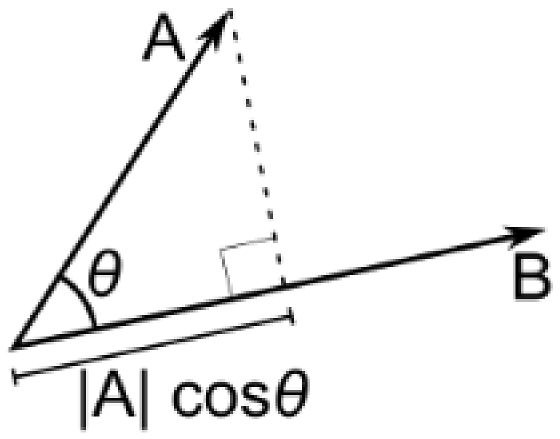

Figure 3: dotproduct

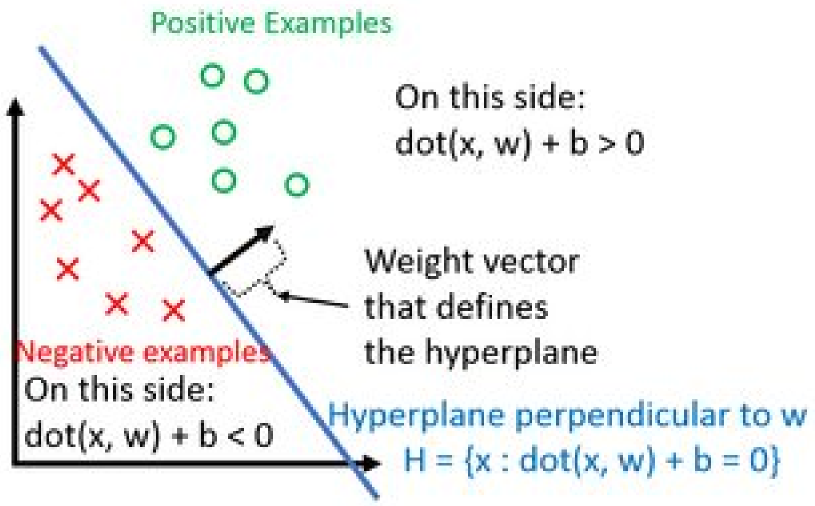

Figure 4: hyperplane1

Imagine we have a problem of classifying if a leaf is healthy or not based on certain features of the leaf. For each leaf we have some feature vector set.

For any **input feature vector** in that vector space, if we have a **weight vector**, whose dot product with one feature vector of the set of input vectors of a certain class (say leaf is healthy) is positive, and with the other set is negative, then that weight vector is splitting the feature vector hyper-plane into two.

Or in a better way, which shows the vectors properly

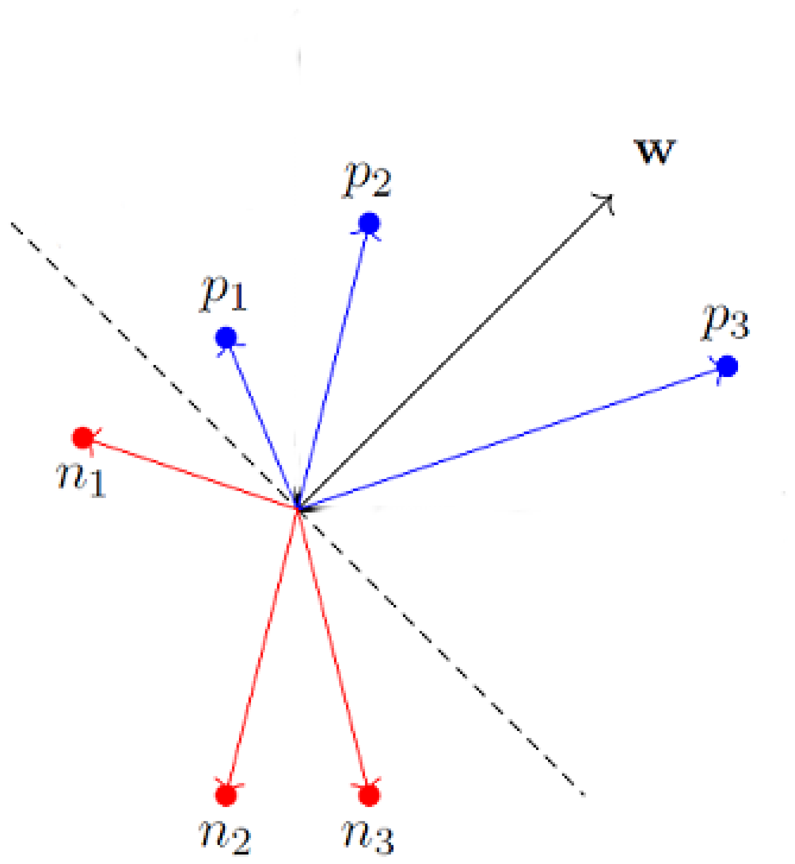

Figure 5: weightvector

In essence, we are using the weight vectors to split the hyper-plane into two distinctive sets.

For any new leaf, if we only extract the same features into a feature vector; we can *dot* it with the *trained* weight vector and find out if it falls in healthy or deceased class.

Not all problems have their feature set which is linearly seperable. So this is a constraint of this system.

The Perceptron

The initial neural network - the **Frank Rosenblatt's perceptron** was basically splitting a linearly seperable feature set into distinct sets. Here is how the Rosenblatt's perceptron is modelled

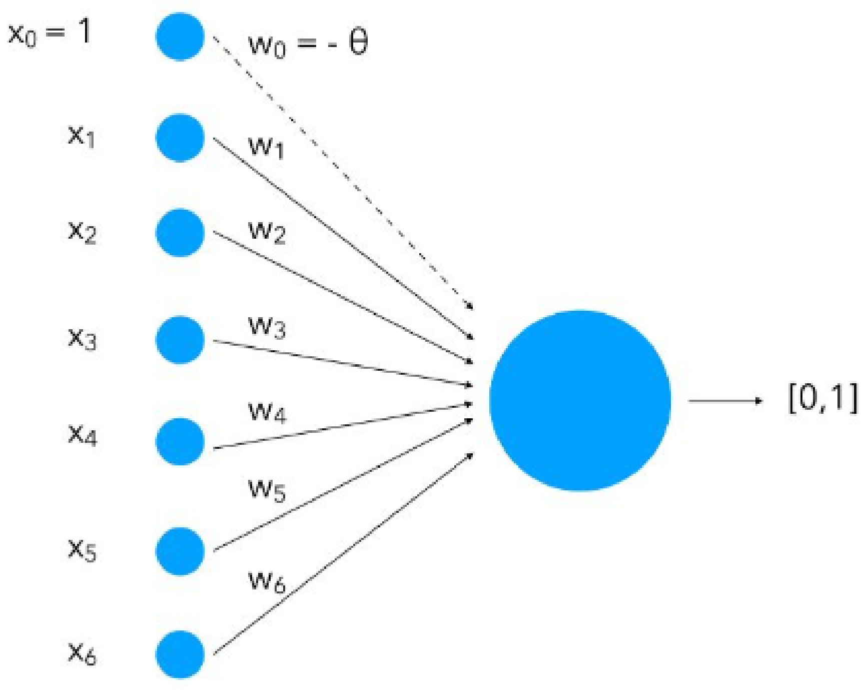

Figure 6: perceptron2

Image source

Inputs are x_1 to x_n , weights are some values that are learned w_1 to w_n. There is also a bias (b) which in above is -θ

The bias can be modelled as a a weight w_0 connected to a dummy input x_0 set to 1.

If we ignore bias term, the output y can be written as the sum of all inputs times the weights; thresholded by zero.

$$y = 1 \text{ if } \sum_i w_i x_i \geq 0 \text{ else } y = 0 \qquad \rightarrow \quad (Eq1)$$

The Perceptron will fire if the sum of its inputs is greater than zero; else it will not. It will be *activated* if the sum of its inputs is greater than zero; else it will not.

The Activation Function of the Nerual Network

The big blue circle is the primitive brain of the primitive neural network - the perceptron brain. This is what is called as an **Activation Function** in Neural Networks. We will see that later.

In Perceptron, the activation function is a `step function`, output is non continuous (and hence non-differentiable) and is either 1 or 0.

If the inputs are arranged as a column matrix and weights also arranged likewise then both the input and weights can be treated as vector and $\sum_i w_i x_i$ is same as the dot product $\mathbf{w} \cdot \mathbf{x}$.

Hence the activation function can also be written as

$$\sigma(x) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0, & \text{otherwise} \qquad \rightarrow \quad (Eq2) \end{cases}$$

Note that dot product of two matrices (representing vectors), can be written as the transpose of one multiplied by another, $w \cdot x = w^T x$

$$\sigma(w^T x + b) = \begin{cases} 1, & \text{if } w^T x + b \geq 0 \\ 0, & \text{otherwise} \qquad \rightarrow \quad (Eq3) \end{cases}$$

All three equations (Eq 1,2 &3) are the same, just that in different reference it will be written in one of these forms.

The equation $w.x > b$ defines all the points on one side of the hyperplane, and $w \cdot x \geq b$ all the points on the other side of the hyperplane and on the hyperplane itself.

This happens to be the very definition of "linear separability"

Thus, the Perceptron allows us to separate our feature space in two convex half-spaces

Ref (12)

If we can calculate or train the weight matrix used here, then we can have a weight vector, which splits the input feature vectors to two regions by a hyperplane.

This is the essence of the Perceptron, the initial artificial neuron.

In simple terms, it means that an unknown feature vector of an input set belonging to say Dogs and Cats, when done a Dot product with a **trained weight vector**, will fall into either the Dog space of the hyperplane, or the Cat space of the hyperplane. This is how neural networks do classifications.

Concept of Hyperplane

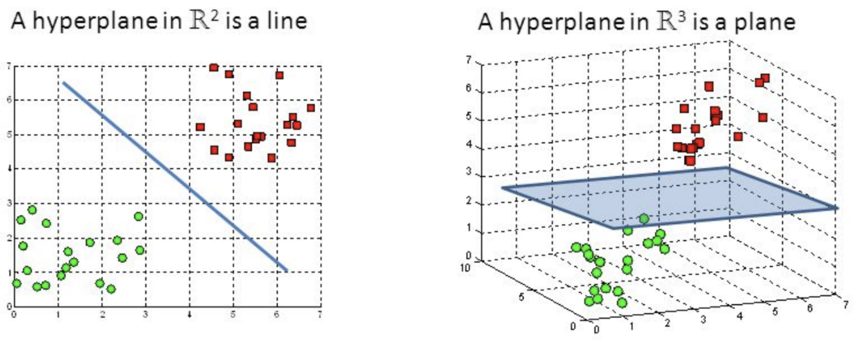

Figure 7: hyperplane2

Next let's see how Perceptron is trained next.

Next: Perceptron Training via Feature Vectors & HyperPlane split