my note about

Basic (and Maybe advanced) Deep Learning stuff

theory and samples

Valgi0 (me as always)

March 21, 2020

Contents

List of Figures

1Why this documents

I feel the exigence to put together all my notes and my experience in this wide word of Deep Neural Network. I want them in one place with a good organization allowing me and the reader to access to what I need quickly. This thought came to my mind while I was working to my dissertation thesis, and for the haste, I used to write notes every where and when I needed them they were hard to find. Btw I don't care to write why I'm doing this so

1.1 Where Information come from

I'm going to use some books freely distributed and one day I'll put the links here. Stay tuned

1.2 Chapters organization

Each chapter contains arguments to one topic and information are organized following this schema:

- Introduction to the topic
- Theory about topic
- Examples maybe
- Related links or papers (Idk)

1.3 Notation

- a,b,c are scalars
- **a,b,c** are vectors
- **A,B,C** are matrices
- **0** it is a zero vector

2Has someone said Math?

In this chapter we are going to explore all math stuff we need to fully understand the concepts. If God Exists he must be a mathematician.

2.1 Linear Algebra OMG

Let's start by defining what is a vector.. a vector is a vector. $\mathbf{v} = [i_0, i_1, i_2, ..., i_n]$. In this case v is a vector composed by elements $i \in \mathbb{I}$ with size n. So vector comes from $\mathbb{R}^n$ space. This is the n-dimensional Euclidean Space. A vector with all zeros is called **zero vector** and it will be represented as $\mathbf{0}$

2.1.1 Vector Space

The Vector Space $\mathbb{X}$ is a set of vectors that satisfies the following conditions:

- **Addition Closure**. This means that:
  - if $\mathbf{x}, \mathbf{y} \in \mathbb{X}$ so $\mathbf{x} + \mathbf{y} = \mathbf{j}$ and $\mathbf{j} \in \mathbb{X}$ for each **x,y**
  - $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$
  - $\mathbf{x} + (\mathbf{y} + \mathbf{c}) = (\mathbf{x} + \mathbf{y}) + \mathbf{c}$
  - $\mathbf{x} + \mathbf{0} = \mathbf{x}$
  - For each $x \in \mathbb{X}$ exists a unique vector called **-x** such that $\mathbf{x} +$ (**-x**) $= \mathbf{0}$
- **Multiplication Closure**. This means that if $\mathbf{x} \in \mathbb{X}$ and y a scalar so $\mathbf{x} * y = \mathbf{j}$ and $\mathbf{j} \in \mathbb{X}$ for each **x,y**
  - $\mathbf{x} * 1 = \mathbf{x}$
  - For any a, b scalars $(ab) * \mathbf{x} = a(b * \mathbf{x})$
  - $(a + b) * \mathbf{x} = a * \mathbf{x} + b * \mathbf{x}$
  - $a(\mathbf{x} + \mathbf{y}) = a * \mathbf{x} + a * \mathbf{y}$

From the above conditions we can extract some considerations. The first is that each $\mathbb{R}^n$ is a vector space. A subset of this euclidean space can be a vector space only if it his unbounded. For example:

$$\mathbb{X} = ((x, y) : x < 5y < 5, (x, y) \in \mathbb{R}^2)$$

$$v_0 = (4, 4), v_1 = (3, 3) \in \mathbb{X}$$

$$v_0 + v_1 = v_2 = (7,7) not \in \mathbb{X}$$

So $\mathbb{X}$ is not a vector space.

2.1.2 Linear Independence

Linear combination is a very cool stuff. Let's have a vector $\mathbf{x}$ and some different vectors **y,z,k**. If there exists a set of three scalars such that $a_0 * \mathbf{y} + a_1 * \mathbf{z} + a_2 * \mathbf{k} = \mathbf{x}$ so $a_0 * \mathbf{y} + a_1 * \mathbf{z} + a_2 * \mathbf{k}$ is a linear combination. Linear indipendence is a propery between more vectors. $\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n$ are linearly dependent just if there exist n scalars at least one of which is not zero, that:

$$a_0\mathbf{x}_0 + a_1\mathbf{x}_1 + ... + a_n\mathbf{x}_n = \mathbf{0}$$

Otherwise, if all a_i are 0, vectors are *linearly independent.* **Attention** if a set of vectors is linearly dependent so one or more that tis vectors can be obtained by a linear combination of others.
Let's do some examples

$$\mathbf{v}_0 = [1,-1], \mathbf{v}_1 = [-1,1], \mathbf{v}_2 = [-1,-1]$$

These are three vectors. Are they linearly independent?

$$a_0 * \mathbf{v}_0 + a_1 * \mathbf{v}_1 + \mathbf{v}_2 * a_2 = 0 \rightarrow \begin{bmatrix} a_0 - a_1 - a_2 = 0 \\ -a_0 + a_1 - a_2 = 0 \end{bmatrix}$$

Now we need to solve those equations:

$$a_0 = a_1, a_2 = 0$$

So they are a linear dependent vectors. This means that we can select one of them and express it as a linear combination of the others:

$$\mathbf{v}_0 = \mathbf{v}_1 * a_1 + \mathbf{v}_2 * a_2$$

That is true for $a_1 = -1$ and $a_2 = 0$.

2.1.3 Spanning a Vectors space

To create a vector space we can grab some vectors and create this space as the set of all linear combinations of these vectors. It the given vectors are linearly independent they are called **Basis set** for that vector space. The number of vectors in the basis set is the dimension of the vector space. This means that we can express each vector inside that space using a number of independent vectors equal to the dimension of the space.
For Example $\mathbb{R}^2$ can have as basis $(1,0),(0,1)$. Using these two vectors we can create each vector inside the space. Try to believe.

A vector space can have infinite basis. But the basis composed by binary vectors

2.1.4 Some functions

Some functions can be defined between vectors. the first is the **Inner Product** that is the product between two vector and it is represented with $(\mathbf{x}, \mathbf{y})$. This function must follow these properties:

- $(\mathbf{x}, \mathbf{y}) = (\mathbf{y}, \mathbf{x})$
- $\mathbf{x}(a\mathbf{y} + b\mathbf{j}) = a(\mathbf{x}, \mathbf{y}) + b(\mathbf{x}, \mathbf{j})$
- $(\mathbf{x}, \mathbf{y}) \geq 0$ if $\mathbf{x}$ is not the zero vector.

The standard inner product adopted the respect the above properties is:

$$\mathbf{xy} = x_0 y_0 + x_1 y_1 + ... + x_n y_n \tag{2.1}$$

Another function that can be defined is the **Norm** represented by $\parallel \mathbf{x} \parallel$. This function must respect these properties:

- $\parallel \mathbf{x} \parallel \geq 0$
- $\parallel \mathbf{x} \parallel = 0$ if $\mathbf{x}$ is the zero vector
- $\parallel a\mathbf{x} \parallel = \mid a \mid \parallel \mathbf{x} \parallel$
- $\parallel \mathbf{x} + \mathbf{y} \parallel \leq \parallel \mathbf{x} \parallel + \parallel \mathbf{y} \parallel$

There are many function that satisfy these conditions. The most common **norm** is:

$$\parallel \mathbf{x} \parallel = (\mathbf{xx})^{\frac{1}{2}} = \sqrt{x_0^2 + x_1^2 + ... + x_n^2} \tag{2.2}$$

Using both of the previous function we can define the *angle* between two vectors. In particular we can calculate the *cosine*:

$$cosine(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{xy}}{\parallel \mathbf{x} \parallel \parallel \mathbf{y} \parallel} \tag{2.3}$$

2.2 Linear Transformation

For **Transformation** we identify the operation that relate a value x to a value y. Let's define:

- *Domain* the set of $x_i \in \mathbb{X}$ values.
- *Range* or Co-domain the set of $y_i \in \mathbb{Y}$ values
- *Transformation* as the rule relating each x_i to an y_i

A transformation A is *Linear* if:

- for all x_0, x_1 $A(x_0 + x_1) = A(x_0) + A(x_1)$
- for all x_0 and for all a scalars $A(x_0 * a) = A(x_0) * a$

IMPORTANT
For any transformation between two finite dimensions vector space can be represented by a Matrix. This Matrix is build using the basis of the vector spaces so changing them the matrix change

2.2.1 Eigenvalues eigenvectors

All vectors from a Vector space not 0 and scalars that make this equation true:

$$A((z)) = a(z) \tag{2.4}$$

are called Eigenvalues and eigenvectors. What that means?? Graphically the eingenvector of a Linear transformation is the direction for which vectors in that direction that after the transformation will have the same direction but a different length.

How to compute them

In order to compute them let's take a look to the formula:

$$A(\mathbf{z}) = a\mathbf{z}$$

We said that a linear transformation can be represented by a matrix so the formula becomes:

$$\mathbf{A}\mathbf{z} = a\mathbf{z}$$

$$(\mathbf{A} - a\mathbf{I}) * \mathbf{z} = 0$$

Let's define **A** matrix as:

$$\begin{vmatrix} -1 & 1 \\ 0 & -2 \end{vmatrix}$$

so $(\mathbf{A} - a\mathbf{I}) = 0$ (Now we look for determinant that is 0,.. dont ask why) is:

$$\begin{vmatrix} -1-a & 1 \\ 0 & -2-a \end{vmatrix} = 0$$

The solutions are found solving:

$$a^2 + 3a + 2 = (a+1)(a+2) = 0 \Rightarrow a = -1a = -2$$

3 Let's start

In this chapter we are going to see the very basic of Neural Network like *Neurons*, *Links*, ...

3.1 Single Neuron

Neuron is the basic unit of Neural Network and is just a mathematical function applied to an input. Let's define an input a scalar i a scalar weight w, a scalar bias b and a function *f*. Neuron function is:

$$a = f(iw + b) \tag{3.1}$$

Function f is called **Activation Function** or sometimes **Transfer function** and usually is a non linear function.

3.1.1 Example of neuron

$i = 5$, $w = 0.5$, $b = 2$ and *f(x)* $= x + 1$

$$f(5 * 0.5 + 2) = f(4,5) = 5.5 \tag{3.2}$$

3.2 Transfer functions

Or Activation function are chosen during model creation in fact they are Hyper-Parameters. They can be linear or non-linear. Now some of the most important linear functions are listed:

- **Hard Limit Transfer Function**. This function produce 0 if input value is negative or 1 if value is equal or greater that 0. It creates a step and it is used for binary classification problems
- **Linear Function**. $f(x) = x * m + q$. It is the function of the line.

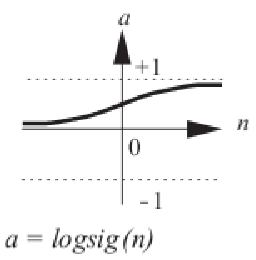
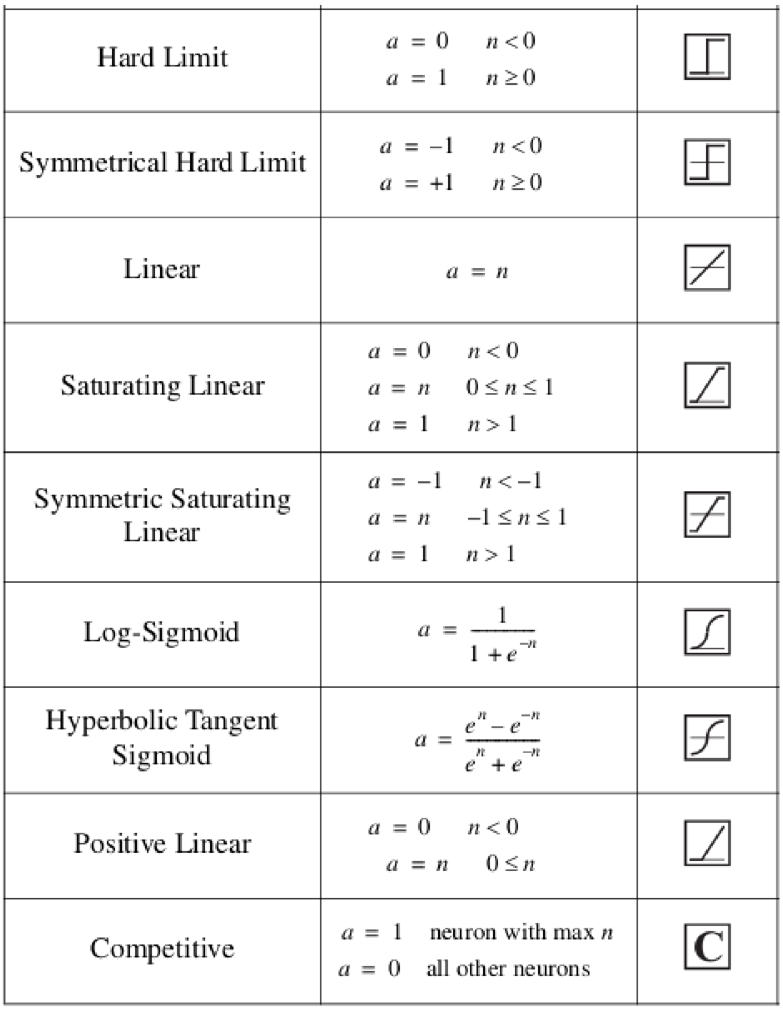


Figure 3.1: log-sigmoid function



Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$	
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$	
Linear	$a = n$	
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$	
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$	
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$	
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$	
Positive Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n$	
Competitive	$a = 1$ neuron with max n $a = 0$ all other neurons	C

Figure 3.2: Overview of basic activation functions

- **log-sigmoid transfer function**. $f(x) = \frac{1}{1+e^{-x}}$. This function produce a value always positive and you can see the graphic at the figure 3.1

3.3 Multi-input Neuron

Usually neuron has more than one input at time. Each of them want a private w while b the bias is one for all. Under this light we can say that input $\mathbf{i}$ is a column vector of $i_0, .., i_n$ elements and neuron contains a scalar b and a vectors of $\mathbf{w}$ of n weights. Using f as linear function $f(x) = x + 1$ the neuron perform this equation:

$$a = f(\mathbf{wi} + b) = f((\sum_{j=0}^{n} w_j * i_j) + b) \tag{3.3}$$

3.3.1 Example

$\mathbf{i} = 5, 6, 7$ $\mathbf{w} = 1, 2, -1$ $b = 2$ and $f(x) = x + 1$

$$f(5 * 1 + 6 * 2 + 7 * -1 + 2) = f(12) = 13 \tag{3.4}$$

3.4 Multi-neurons

Often Input vector goes to multiple neurons and these neurons are a Layer. It is important keep in mind that:

- All Neurons take all input. So w weights is a vector with the same dimension of the input vector
- All Neuron produce a scalar so the number of neurons in a layer is the number of outputs.
- Each neuron has its bias so a layer has a vector of bias one per neurons

Let's define input $\mathbf{i}$ a column vector and weight $\mathbf{W}$ as a matrix $m * n$ where m is the number of the neuron and n the number of the input elements in $\mathbf{i}$. f is the activation function:

$$\mathbf{a} = f(\mathbf{W}\mathbf{i} + b) \tag{3.5}$$

From the above equation we can see that a matmul is performed between weights matrix and input column vector. So row j, that is the weight vector of the neuron j is multiplied and summed with the whole input column vector.

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,0} & \dots & w_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{bmatrix} * \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_n \end{bmatrix} \tag{3.6}$$

So a neural layer can be seen as a matrix of weight and a Bias vector:

$$Layer_w = \begin{bmatrix} n_0 \\ n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,0} & \dots & w_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{bmatrix} \tag{3.7}$$

$$Layer_b = [b_0, b_1, b_2, \dots, b_m] \tag{3.8}$$

Mathematically a layer can be represents just by $\mathbf{W}$ the weight matrix and $\mathbf{b}$ the bias vector.

3.5 Multi Layers

A neural network has more layers. The first layer is called **Input layer** the last in called **Output layer** and the ones in the middle are **Hidden Layers**. Each layers can have different number of neurons and different activation function. So a basic multi-layer network with l layers is defined by $f_0, ..., f_l$ activation functions, $\mathbf{W}_0, ..., \mathbf{W}_l$ weight matrices and $\mathbf{b}_0, ..., \mathbf{b}_l$ or just $\mathbf{B}$ bias (matrix contains in row j the bias vector for the layer j). This network take inputs with the first layer, generate output and then gives this output to the second layer and so on. A Multi layer Network performs:

$$\mathbf{a} = f_l(f_{l-1}(\ldots * \mathbf{W}_{l-1} + \mathbf{b}_{l-1}) * \mathbf{W}_l + \mathbf{b}_l) \tag{3.9}$$

This Multi-layer neural Network is called **Feed Forward Neural Network**

3.6 Recurrent Neural Network

Feed forward neural network has layers and layer has neurons. Each Neuron take a input vector **i** and produce a scalar a. Differently, Recurrent Neural Network takes two input vector the input vector **i** and some *a'* that is the output of the previous input. Let's imagine we have a time series of data $i_0, ..., i_t$. This data must be given to the model respecting the sequence but we want model somehow remember previous data. For this kind of task Recurrent Neural Network was created. They have layers and neurons but each neuron has two vectors and a bias: u weights for the input, w weights for its layer output at the previous step and b the bias. Let me explain in a intuitive way before using math: the input is a series of vectors. First vector i_0 is taken and is given to the first layer of the model. Each neurons of this layer get the input vector i_0 and produce a scalar a_0 and with other neurons output of that layer it creates the layer output a Nothing new until now. Output goes to the second layer and so on until last layer. Second input is taken i_1 and neuron receives it and the previous layer output a. Using both of them it compute the next scalar output $nexta_0$. And so on!

Let's define its behavior using some cool math. Our network is composed by 1 layer with 1 neuron:

$$inputsequence = \mathbf{i}_0, \ldots, \mathbf{i}_k \tag{3.10}$$

$$f_{neuron} = ActivationFunc(b + \mathbf{w} * a'_{t-1} + \mathbf{u} * \mathbf{i}_t) = a_t \tag{3.11}$$

The neuron takes at time t in input $a'_{0,t-1}$ and $i_{0,t}$ and produce a_t. In this case with just one neuron the layer output coincide with the neuron output. After that the following equation is performed to compute a'_t

$$a'_t = c + \mathbf{v}a_t \tag{3.12}$$

Let's imagine to have a Recurrent Neural Network with more layers and more neurons for each layers. The above equations become:

$$\mathbf{a}_t = ActivationFunc(\mathbf{b} + \mathbf{W} * \mathbf{a'}_{l,t-1} + \mathbf{U} * \mathbf{i}_t) \tag{3.13}$$

$a'_{l,t-1}$ means the column vector output from the layer l at time step $t-1$

$$\mathbf{a'}_{l,t} = \mathbf{c} + \mathbf{V}\mathbf{a}_{l,t} \tag{3.14}$$

That is all my dear.. Not true, we are going to expand this kind of network later

3.6.1 Keep in mind

Some tips to keep in mind when you work with Recurrent Neural Network

- Externally are similar to feed forward network.. it has layers and neurons
- The inputs are sequence of inputs vectors
- Output is a series of vectors one for each inputs vector but if a prediction is what u want last output must be used.
- At any time neuron takes exactly 2 input vectors in input and they are the output from prior layer and output for its layer but a time t-1.
- For each input vectors neurons has a weight vector this means that a layer has a **W** and **U** weight. The first for its own output and the second for the output of the previous layer.
- Output of a layer is turned in a input performing $c + \mathbf{V}a_l$ where c is a bias and **V** is a weight matrix.

3.7 Some Applications

In this section we are going to see how the previously explained network are used in a very easy problem: *We have some fruits composed by oranges and apples and we need a model capable to tell us if a given fruit is one or it is the other. Fruits is defined by size, dimension, weight.*

Let be $i_a = \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}$ the vector that represent an apple and $i_o = \begin{vmatrix} 1 & 1 & -1 \end{vmatrix}$ the one that represent an orange.

3.7.1 Perceptron

Perceptron is a binary classifier based on single layer **Feed Forward Network** with **Hard Limit** as activation function. So if the output is minor than 0 the function produces 0 otherwise 1.

$$output = HardLims(\mathbf{W}\mathbf{i} + \mathbf{b}) \tag{3.15}$$

W is a matrix nxm where n is the dimension of the input vector (3 in this case) and m the number of neurons in the layer.

Let's define $W = [0, 1, 0]$ for a single neuron layer. This is called **Single-Neuron Perceptron**. Now we can test it against our data.
Apple defined as $i_a = \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}$ is multiplied by **W**

$$[0, 1, 0] * \begin{vmatrix} 1 & -1 & -1 \end{vmatrix}^T = 0 * 1 + 1 * -1 * 0 * 1 = -1 \tag{3.16}$$

For simplicity input vector is not in column but it should be. Don't judge me

$$hardlims(-1) = -1 \tag{3.17}$$

Ok! model classify apple as -1 let's check if orange are classified as 1

$$[0, 1, 0] * \begin{vmatrix} 1 & 1 & -1 \end{vmatrix}^T = 0 * 1 + 1 * 1 * 0 * 1 = 1 \tag{3.18}$$

$$hardlims(1) = 1 \tag{3.19}$$

OMG!!!! That is amazing... or not but anyway with a weight vector $[0, 1, 0]$ model is able to classify the fruits. Now try with banana and tell me if it works! COME ON!

4Basic Concepts about learning rules

In this chapter we are going to study in deep the basic concepts about learning rule. With *learning rule* it means all the rules and algorithms used to modify the weight and the bias of the network in order to improve the results. There are three main approaches:

- *Supervised learning.* In this case the net is trained by using a set of examples composed by (p_0, t_0) where p_0 is the input for the network and t_0 the corresponding correct result (target). The result produced is compared to the target. The learning rules are used to modify weights to make model produce the right result.

- *Reinforcement Learning.* The algorithm produce a score or a grade according to the network performance and the learning rule seek to adjust weights in order to get higher scores. The network continues to learn also during its application.

- *Unsupervised Learning.* The data does not have the target but they are somehow clustered. The cluster becomes the target to predict like the one in the supervised learning.

Probably, we will speak about all this approaches later.

4.1 Learning rules for Perceptron Architecture

4.1.1 Perceptron Network again

The perceptron network has a single layer of S neurons and it use as activation function the **Hardlim** (it produce 0 or 1 according to the output of the neuron). The weights of the network are:

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_0 \\ \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_s \end{bmatrix} \tag{4.1}$$

So each neuron has a list of $w_{n0}, ..., w_{nm}$ weights where n it the neuron and m is the number of weight and it is the same for the number of inputs.

Each neuron produce a result o_n and the list with all results from all neurons is given to the activation function.

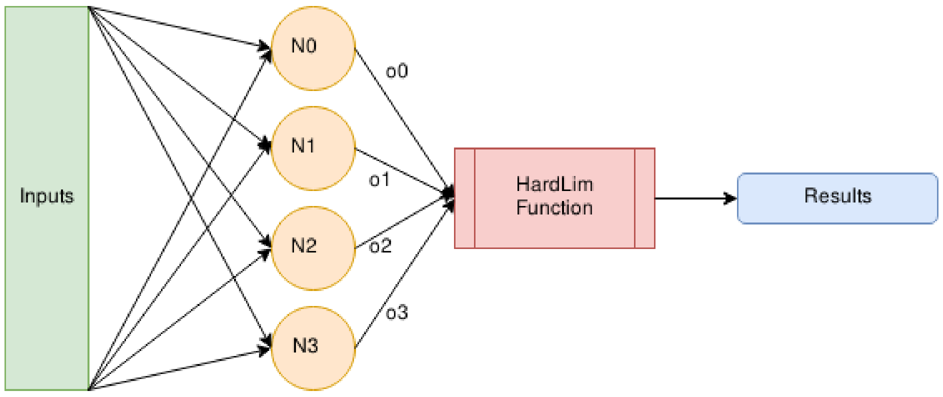


Figure 4.1: Overview of a Perceptron Network

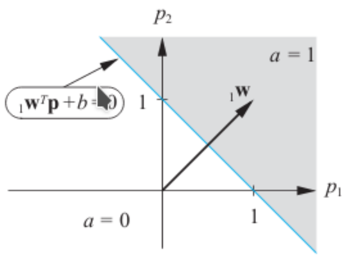


Figure 4.2: Graphic representation of the decision boundary

the decision boundary

Let's use a Perceptron Network with one neuron and 2 inputs. The result is obtained by this formula:

$$o = hardlim((w_0 * p_0 + w_1 * p_1) + b) \tag{4.2}$$

The decision boundary of the network is determined by the sub set of inputs that make the neurons produce zeros.
Example: Let's have $w_0 = 1$, $w_1 = 1$ and $b = -1$. So the decision boundary will be:

$$(w_0 * p_0 + w_1 * p_1) + b = p_0 + p_1 - 1 = 0 \Rightarrow p_0 + p_1 = 1 \tag{4.3}$$

The decision boundary define a line in the input space. On one side of that line, the network output will be 0, on the other side will be one. Modifying the weights and the bias of the network this line is moved up and down changing also its orientation.

4.1.2 Decision Boundary for Multiple Neurons

Network with S neurons has S lines, one for each of them. So each neuron can classify each input in just two categories. A network with one neuron has one decision boundary so it can produce just 2 category, while a network with S neurons can produce 2^s different output vectors so it can be used to classify an input to these categories.

4.2 Learning rules for Perceptron Network

There is a precise algorithm that tells us how to update the weights and it guarantees the best solution is found. The idea behind is if the model predict the wrong result weight must be modified to let the model classify that input. For this case let's define $e = t - a$ where t is the target and a is the output of the model. Each weight is updated according to this formula:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} \tag{4.4}$$

We can do the same for bias: Bias are just single weight for an input that is always one.

$$b^{new} = b^{old} + e \tag{4.5}$$

So, the perceptron rule can be written for the matrix of weight in case the network has more than one neuron.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{ep} \tag{4.6}$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e} \tag{4.7}$$

That is all by now, but there is a proof of convergence that demonstrates this algorithm always find the best solution.

4.2.1 Example

Let's have a singular-neuron perceptron network for 3-input vectors: $i_0 = [1, -1, 0]$, $i_1 = [1, 1, -1]$, $i_2 = [1, 0, 0]$. For all of them we have the corresponding targets $t_0 = 1, t_1 = 0, t_2 = 1$. Let's initialize our net with random values: $w = [0, 0.5, -0.2], b = 0.3$. First, it can be useful find out the decision boundary:

$$(0 * i_{n0} + 0.5 * i_{n1} - 0.2 * i_{n2}) + 0.3) = 0 \Rightarrow \frac{i_{n1}}{2} - \frac{i_{n2}}{5} = 0.3 \tag{4.8}$$

Now we can proceed giving the first input to the network and see what happend:

$$hardlim((0*1+0.5*-1-0.2*0)+0.3) = hardlim(0-0,5+0+0.3) = hardlim(-0,2) = 0 \tag{4.9}$$

So $e = t - a = 1 - 0 = 1$. Let's apply the updating formula :

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} = [0, 0.5, -0.2] + (1)[1, -1, 0] = [1, -0.5, -0.2] \tag{4.10}$$

$$b^{new} = b^{old} + e = 0.3 + (1) = 1.3 \tag{4.11}$$

Let's do it again but with the second input. (I gonna skip some calculus)

$$hardlim((1 * 1 + -0.5 * 1 - 0.2 * -1) + 1.3) = hardlim(2) = 1 \tag{4.12}$$

Again the model mistakes. $e = t - a = -1$. So let's repeat the update phase:

$$\mathbf{w}^{new} = \mathbf{w}^{old} + e\mathbf{p} = [1, -0.5, -0.2] + (-1)[1, 1, -1] = [0, -1.5, 0.8] \tag{4.13}$$

$$b^{new} = b^{old} + e = 1.3 + (-1) = 0.3 \tag{4.14}$$

Let's do it again but with the third input. (I gonna skip some calculus)

$$hardlim((0 * 1 - 1.5 * 0 + 0.8 * 0) + 0.3) = hardlim(0.3) = 1 \tag{4.15}$$

This time the prediction is good and $e = 0$
Let's test the new weights with the first input:

$$hardlim((0 * 1 - 1.5 * -1 + 0.8 * 0) + 0.3) = hardlim(1.8) = 1 \tag{4.16}$$

Good, it is correct. And at last, let's check for second input:

$$hardlim((0 * 1 - 1.5 * 1 + 0.8 * -1) + 0.3) = hardlim(-2) = 0 \tag{4.17}$$

That is cool! Now model can predict the right result for all of them!

4.3 Perceptron Network Limitation

This network can find the best solution of classification problem in a finite number of steps, however it can cut the input space using lines (hyperplane). It can't model create decision boundary not-linear, so if data are a long some curve they can't be classified. For example this network can learn to produce AND but it can't reproduce XOR. Try to think why by yourself.