Learning algorithms for Linear Feature Extraction

The aim of these Notes is to give students a brief overview of Learning Algorithms and eventually their HPC implementation. Learning algorithms have strong foundations in Linear Algebra and receive inspiration in that stems from Neuroscience. They complement classical methods of inference in Statistics and have several links with fields such as Bayesian inference. The present Notes focus on Learning methods for feature extraction from Data and highlight the relationship of Hebbian Learning with Principal Component Analysis.

1 The Perceptron

The Perceptron, introduced by Warren McCulloch and Walter Pitts (1943), is one of the earliest attempts to model biological neurons. The key idea is to create a model of associations that may develop between neurons by continuous exposure to certain signals, following ideas by Hebb. As dicussed in Hebb (The Organisation of Behaviour, 1949) : *"When an axon of cell A is near enough to excite a cell B and repeatedly and persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."*

Consider the Pavlov's dog experiment. At first, the Bell cell will fire whenever bells ring, but probably not when the salivation cells happen to be active. So, the connection between the Bell cell and the Salivation cell remains weak. But then, Pavlov intervenes and causes the Bell cell and the Salivation cell to fire at the same time by ringing the bell and presenting food at the same time (the Food detector cell already has a strong connection to the Salivation cell). Whenever the Bell cell and the Salivation cell happen to fire at the same time, the synapse between them is strengthened. Once the connection is strong enough, the Bell cell can cause the Salivation cell to fire on its own, just like the Food detector cell can.

The Perceptron is based on a linear combination of the inputs and can be formulated as

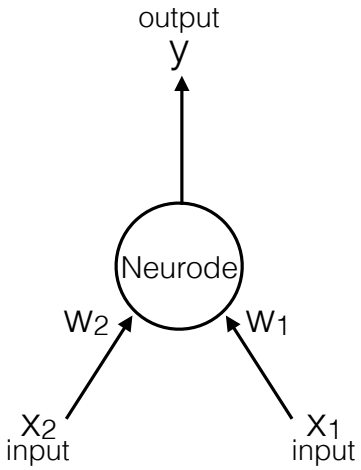
$$I = \sum_{i=1}^{n} w_i x_i \tag{1}$$

$$y = \begin{cases} +1 \text{ if } I \geq T \\ -1 \text{ if } I < T \end{cases} \tag{2}$$

where T is the threshold.

The question that was posed by Frank Rosenblatt in 1958 is whether the Perceptron can be used to perform a task. Or in other words can we use the Perceptron architecture to learn "something"?

Frank Rosenblatt (1958).
Learn a task. The task is classification: separating patterns in two categories.



Note that this task falls in the category of *supervised learning* (feature extraction).

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \beta y \mathbf{x} \tag{3}$$

where

$$\beta = \begin{cases} +1 \text{ if the answer is correct} \\ -1 \text{ if the answer is incorrect} \end{cases} \tag{4}$$

It can be shown that the Perceptron can indeed correctly classify linearly separable data. One of the problems of the Perceptron is that it cannot be extended to non-linear functions. Moreover the weights of the network may grow in an unstable manner as new data is arriving. In addition to performing tasks of classification it can be shown that Perceptron can be trained to produce the largest Principal Component of a set of Data.

1.1 Minimum error learning

The Adaline: Adaptive Linear Element, Widrow and Hoff (1960).
This is one of the first examples of supervised learning. The goal of the learning is to minimize the overall error between the Perceptron output and a set of data.

$$I = \sum_{i=1}^{n} x_i w_i \tag{5}$$

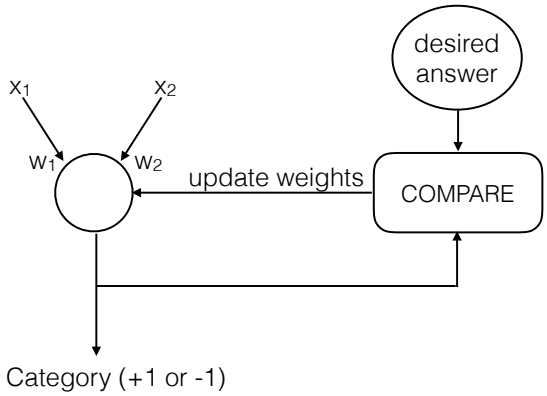
if

$$y = \begin{cases} +1 \text{ if } I \geq 0 \\ -1 \text{ if } I < 0 \end{cases} \tag{6}$$

$$\text{Error} = \text{desired\_output} - \text{actual\_output} \tag{7}$$

So Error can be $+2$, -2, 0.

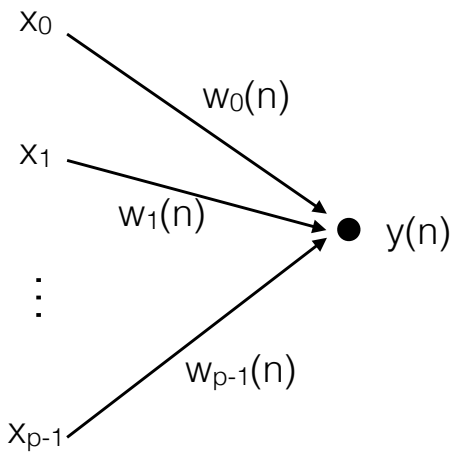
$$\mathbf{w}_{new} = \mathbf{w}_{old} + \beta \frac{Ex}{\|x\|^2} \tag{8}$$



The weights must be kept being applied until the correct answer is given.

1.2 A linear Neuron model as a maximum Eigenfilter (PCA)

A single linear neuron with Hebbian-type adaptation for its synaptic weights can evolve into a filter for the first Principal Component of the input distribution (Oja, 1982).



$$y = \sum_{i=0}^{P-1} w_i x_i \tag{9}$$

We write

$$w_i(n+1) = w_i(n) + \beta y(n) x_i(n) = w_i(n) + x_i(n) \sum_{i=0}^{P-1} w_i(n) x_i(n) \tag{10}$$

where β is the learning rate parameter.
The learning rule leads to growth of the weights:

$$y(n) = \mathbf{x}^T(n)\mathbf{w}(n) = \mathbf{w}^T(n)\mathbf{x}(n) \tag{11}$$

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) + \beta \mathbf{x}^T(n)\mathbf{w}(n)\mathbf{x}(n) \\ &= \mathbf{w}(n) + \beta \mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n) \\ &= \mathbf{w}(n) + \beta R(n)\mathbf{w}(n) \end{aligned} \tag{12}$$

The above equations can be written in continuous form replacing time for the iterations so that:

$$\frac{\mathbf{w}(n+1) - \mathbf{w}(n)}{\beta} \approx \frac{d\mathbf{w}}{dt} = R(n)\mathbf{w}(n) \tag{13}$$

We note that the covariance matrix has positive eigenvalues so (depending on the data) when an eigenvalue is larger than one the weights may increase uncontrollably.

NOTE: The exchange of vectors in the above equation is justified as follows:

$$\mathbf{w}^T\mathbf{x}\mathbf{x} = (\mathbf{x}^T\mathbf{w})\mathbf{x} =$$

$$= \left(x_1w_1 + x_2w_2 + x_3w_3 + \cdots + x_Nw_N\right) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} =$$

$$= \begin{pmatrix} x_1^2w_1 + x_1x_2w_2 + x_1x_3w_3 + \cdots + x_1x_Nw_N \\ \vdots \\ x_Nx_1w_1 + x_Nx_2w_2 + x_Nx_3w_3 + \cdots + x_N^2w_N \end{pmatrix} =$$

$$= \begin{pmatrix} x_1^2 & x_1x_2 & x_1x_3 & \dots & x_1x_N \\ \vdots & \ddots & & & \\ \vdots & & \ddots & & \\ \vdots & & & \ddots & \\ x_Nx_1 & & & & x_N^2 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix} = (\mathbf{x}\mathbf{x}^T)\mathbf{w}$$

1.3 Normalization of Learning or Saturation

Introduce competition between the synapses of neurons for limited resources which is essential for stabilization (Oja, 1982).

$$w_i(n+1) = \frac{w_i(n) + \beta y(n)x_i(n)}{\{\sum_{i=0}^{P-1}\left[w_i(n) + \beta y(n)x_i(n)\right]^2\}^{1/2}} \tag{14}$$

Assuming $\beta \ll 1$ we may expand this in power series and we will obtain

$$w_i(n+1) = w_i(n) + \beta y(n)\left[x_i(n) - y(n)w_i(n)\right] + O(\beta^2) \tag{15}$$

Ignore terms that are higher than $O(\beta^2)$

$$w_i(n+1) = w_i(n) + \beta y(n)\left[x_i(n) - y(n)w_i(n)\right] \tag{16}$$

where the term $\beta y(n)x_i(n)$ is the positive feedback term which leads to self-amplification, and the term $-\beta y(n)y(n)w_i(n)$ is the negative feedback term, used for stabilization.
Call

$$\hat{x}_i(n) = x_i(n) - y(n)w_i(n) \leftarrow \text{ defective input} \tag{17}$$

$$w_i(n+1) = w_i(n) + \beta y(n)\hat{x}_i(n) \tag{18}$$

The term $-y(n)w_i(n)$ is the forgetting factor (but more prominent for large signals).

2 Model Properties

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \beta y(n)\left[\mathbf{x}(n) - y(n)\mathbf{w}(n)\right] \tag{19}$$

and with

$$y(n) = \mathbf{x}^T(n)\mathbf{w}(n) = \mathbf{w}^T(n)\mathbf{x}(n) \tag{20}$$

then

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \beta\left[\mathbf{x}\mathbf{x}^T\mathbf{w} - \mathbf{w}^T\mathbf{x}\mathbf{x}^T\mathbf{w}\mathbf{w}\right] \tag{21}$$

Stochastic, recursive and time varying equation.

The algorithm is asymptotically stable and converges to a fixed point q_o

$$\mathbf{w}(n) \to q_o \text{ as } n \to \infty \tag{22}$$

(under certain assumptions).

- Rate of learning is slow for synaptic weights to be treated as station.
- $\mathbf{x}(n)$ is drawn from a stochastic process whose correlation matrix has distinct eigenvalues.
- $\mathbf{x}(n)$ and $\mathbf{w}(n)$ are statistically independent.

So for stationary case:

$$0 = R\mathbf{q}_o - (\mathbf{q}_o^T R\mathbf{q}_o)\mathbf{q}_o \tag{23}$$

where

$$R = E\left[\mathbf{x}(n)\mathbf{x}^T(n)\right] \tag{24}$$

So at equilibrium:

$$\begin{aligned} R\mathbf{q}_o &= \lambda\mathbf{q}_o \\ \text{and } \lambda_o &= \mathbf{q}_o^T R\mathbf{q}_o \end{aligned} \tag{25}$$

So the algorithm converges to the principal eigenvector.

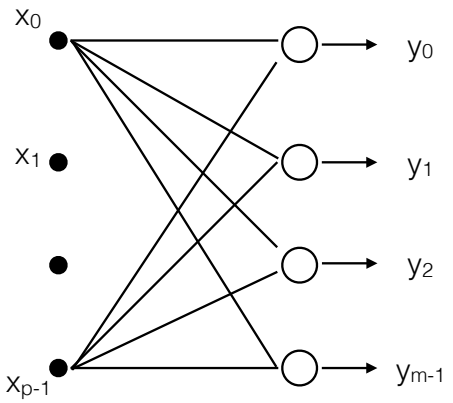
Recall **Power Iteration**

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|} \tag{26}$$

If Ab_k has an eigenvalue that is strictly greater in magnitude than its other eigenvalues then the algorithm converges to an eigenvector associated with dominant eigenvalue.

$$k_n = \begin{bmatrix} b & Ab & A^2b & A^{n-1}b \end{bmatrix} \tag{27}$$

use Gram Schmidt to orthogonalise.



- Step 1: Initialize $w_{ji} \in U[0, 0.1]$
- Step 2: $y_j(n) = \sum_{i=0}^{P-1} w_{ji}(n)x_i(n)$

$$\Delta w_{ji}(n) = \beta \left[y_j(n)x_i(n) - y_j(n) \sum_{k=0}^{l} w_{kj}(n)y_k(n) \right] \tag{28}$$

Iterate until $\Delta w_{ij} \to 0$.

w_{ji} of neuron j converges to the $i-th$ component of the eigenvector corresponding with the $j-th$ eigenvalue of the covariance matrix of input vector $\mathbf{x}(n)$.

3 Multilayer Perceptrons

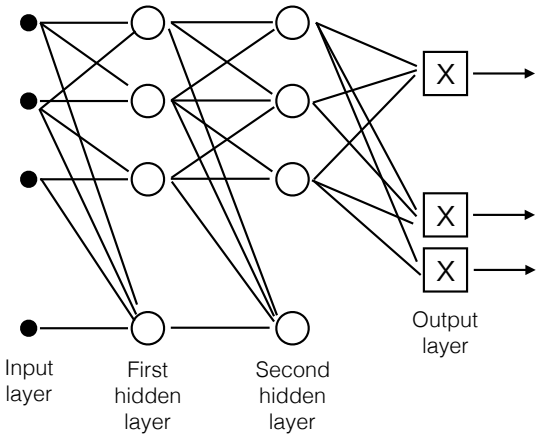
$$e_j(n) = d_j(n) - y_j(n) \tag{29}$$

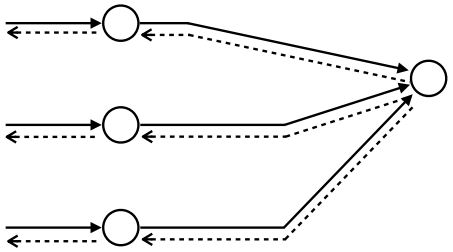
neuron j is output node.

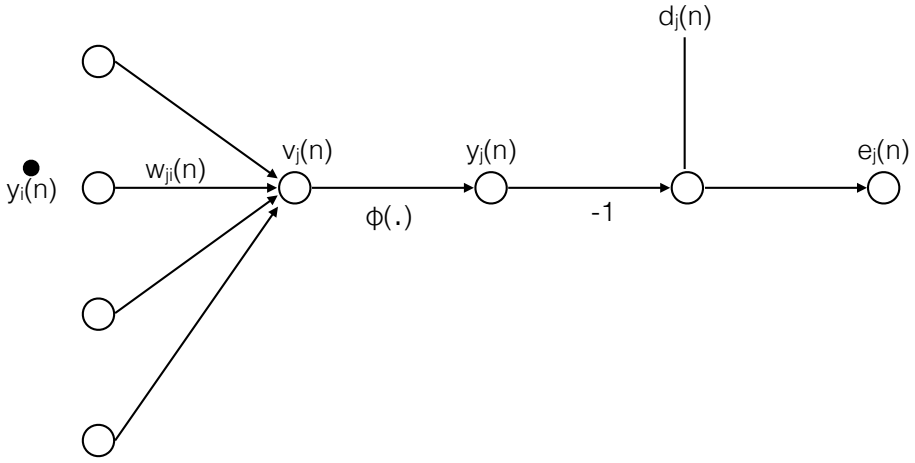
$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \text{ , C is output nodes} \tag{30}$$

$$E_{av} = \frac{1}{N} \sum_{n=1}^{N} E(n) \tag{31}$$

where N is the total number of examples in the training set.


X
X
X
Input layer
First hidden layer
Second hidden layer
Output layer




$d_j(n)$
$y_i(n)$
$w_{ji}(n)$
$v_j(n)$
$y_j(n)$
$e_j(n)$
$\phi(.)$
-1

$$v_j(n) = \sum_{i=0}^{P} w_{ji}(n) y_i(n) \tag{32}$$

where P is the total number of inputs applied to neuron j.

$$y_j(n) = \phi_j(v_j(n)) \tag{33}$$

where ϕ is a (non)-liner function.

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{34}$$

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \text{ from eq.30} \tag{35}$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{36}$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi_j^{'}(v_j(n)) \tag{37}$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{38}$$

So finally:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\phi_j^{'}(v_j(n))y_i(n) \tag{39}$$

$$\Delta w_{ji}(n) = -\beta \frac{\partial E(n)}{\partial w_{ji}} \tag{40}$$

where the minus comes from the gradient descent.

$$\Delta w_{ji}(n) = \beta \delta_j(n) y_i(n) \tag{41}$$

$$\delta_j(n) = -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \tag{42}$$

therefore:

$$\delta_j(n) = e_j(n)\phi_j^{'}(v_j(n)) \tag{43}$$

What is $e_j(n)$?

1. Neuron j is output node. $e_j(n)$ is given.
2. Neuron is hidden. No direct $e_j(n)$ so it has to be computed recursively from all other neurons.

So by redefining eq.43:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \phi_j^{'}(v_j(n)) \tag{44}$$

How to do this:

$$E(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n) \tag{45}$$

where C is output.

$$\begin{aligned}\frac{\partial E(n)}{\partial y_j(n)} &= \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \\ &= \sum_k e_k(n) \frac{\partial e_k}{\partial v_k} \frac{\partial v_k}{\partial y_j}\end{aligned} \tag{46}$$

but

$$\begin{aligned}e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \phi_k(v_k(n))\end{aligned} \tag{47}$$

So

$$\frac{\partial e_k}{\partial v_k} = -\phi_k^{'}(v_k(n)) \tag{48}$$

and note that

$$v_k(n) = \sum_{j=0}^{q} w_{kj}(n) y_j(n) \tag{49}$$

so

$$\frac{\partial v_k}{\partial y_j} = w_{kj}(n) \tag{50}$$

We substitute Eq.48 and Eq.50 into Eq.46:

$$\begin{aligned}\frac{\partial E}{\partial y_j} &= -\sum_k e_k(n) \phi_k^{'}(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \text{ , } \delta_k(n) \text{ is local gradient}\end{aligned} \tag{51}$$

So $\delta_j(n)$ for hidden neuron j is given by

$$\delta_j(n) = \phi_j^{'}(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{52}$$

$\delta_k(n)$ requires knowledge for all signals that lie to the right of j.

4 Self Organizing Maps

Let $x \in R^n$ be a stochastic vector:
One may consider that the SOM is a "non-linear" projection of the probability density function $p(x)$ of the high dimensional input data vector.

Let m_i be the nodes of the SOM.
Let

$$C = \arg\min_i \{\|x - m_i\|\} \tag{53}$$

thus

$$\|x - m_c\| = \min_i \{\|x - m_i\|\} \tag{54}$$

Learning

$$m_i(t+1) = m_i(t) + h_{ci}\left[x(t) - m_i(t)\right] \tag{55}$$

$$h_{ci}(t) = \alpha(t) \exp\left(\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}\right) \tag{56}$$

This defines the stiffness of the elastic points.

5 Appendix