A Report on “Forming sparse representations by local anti-Hebbian Learning” by P. Foldiak

Yashee Sinha[1]

[1]*Birla Institute of Technology and Science, Pilani, Goa, India*

Hebbian learning largely involves the adjustment of weight vectors based on frequency of patterns. This process is fundamental in creating neural networks capable of associative learning. However, there are no distinctions between different neurons of the network and hence different features can not be picked out effectively. Peter Földiák proposed an anti-Hebbian learning that decorrelates neurons using inhibitory connections and pushes them away from firing together[1]. This also allows one to pick out individual features of recurring patterns in different units of the network allowing the network to simultaneouly learn all possible combination of frequent features of the patterns.

I. INTRODUCTION

A singular Hebb unit is given by the following activation rule: if x_i denotes the i-th input component, y denotes the activity of the unit, and q_i is the connection weight between the i-th input and the output, then

$$y = \begin{cases} 1 & \text{if } \sum_j q_j x_j > t \\ 0 & \text{else} \end{cases} \tag{1}$$

where t is an arbitrary threshold.

More specifically, a Hebbian unit functions as a basic pattern matcher. The weighted sum $\sum_j q_j x_j$ is maximized when the input pattern matches the weight vector precisely. Hebb proposed a weight modification rule such that on each iteration, $\Delta q_j = x_j y$, and so the weight vector slowly moves towards the input vector[2]. If a unit fires upon pattern presentation, the weights from the active inputs strengthen, enhancing future responses to that pattern. Thus, if provided a random sample of input vectors, the weight vector will eventually find the average of all inputs.

Multiple Hebb units can be strung together in a neural layer to detect different features using competitive learning. In this scheme, only the unit with the highest weighted sum activates, suppressing all others. This division of input space results in a local, grandmother-cell representation that can be used by a supervised layer to associate outputs with single trials. However, this type of representation has limited capacity and generalization ability. A new pattern with different combination of popular features, which would be easily recognizable by a human brain, would rarely produce any reliable outputs in such a network.

Földiák proposed a sparse coding scheme called anti-Hebbian learning for producing better results using Hebb units[1]. Each input state is represented by a small set of the units which can learn the various features of frequent patterns separately. An inhibitory connection between different units ensures that they do not learn the same features. This offers a higher representational capacity and desirable properties such as noise resistance and pattern generalization. Sparse coding strikes a balance between local and distributed representations, controlling the proportion of active units.

If we denote $\vec{y}$ as the vector of activities of all N units in the network, $\vec{x}$ as the vector of M inputs, $\mathbf{q}$ as the matrix of weights between the inputs and the units, and $\mathbf{w}$ as the matrix of inhibitory connections between the units, then the activation rule for this scheme is given by:

$$y_i = \begin{cases} 1 & \text{if } f\left(\sum_j q_{ij} x_j + \sum_j w_{ij} y_j - t_i\right) > 0.5 \\ 0 & \text{else} \end{cases} \tag{2}$$

where f is the nonlinear activation function to introduce smoothness in learning. The weight modification rule for inhibitory connections is then given by $\Delta w_{ij} = -y_i y_j$. Thus, connections between simultaneously active units become more inhibitory, discouraging future joint activity and reducing correlation. This allows for a sparse representation of input patterns by a statistically uncorrelated set of features.

The output cannot be calculated in a single step due to the feedback from other units. However, it is proven that the network settles in a stable state after an initial transient if feedback weights are symmetric ($w_{ij} = w_{ji}$). The transient and the stable state can be simulated smoothly using the following nonlinear dynamical equation:

$$\frac{dy_i^*}{dt} = -y_i^* + f\left(\sum_j q_{ij} x_j + \sum_j w_{ij} y_j^* - t_i\right) \tag{3}$$

The output y_i is binary, given by:

$$y_i = \begin{cases} 1 & \text{if } y_i^* > 0.5 \\ 0 & \text{else} \end{cases} \tag{4}$$

The network's learning rules involve adjusting the feedforward and feedback weights and the thresholds:

$$\Delta w_{ij} = -\alpha(y_i y_j - p^2) \tag{5}$$

$$\Delta q_{ij} = \beta y_i (x_j - q_{ij}) \tag{6}$$

$$\Delta t_i = \gamma (y_i - p) \tag{7}$$

Here, α, β, and γ are positive constants, and p is the target firing rate of the units. The Hebbian rule includes a weight decay term to keep the feedforward weights bounded, while the anti-Hebbian rule inherently remains stable without normalization.

This combination of Hebbian and anti-Hebbian mechanisms allows the network to learn representations that maximize information retention while minimizing redundancy, leading to efficient, sparse coding of input patterns.

II. IMPLEMENTATION

The explicit algorithm used to generate the images in this report is given by Algorithm 1.

Algorithm 1 Implementation of anti-Hebbian Learning

```
1: **Input:** $X \in \mathbb{R}^{M\times 100}$, the data matrix containing 100 samples where each column is an input vector.
2: **Parameters:** $\alpha, \beta, \gamma, p, \lambda, \Theta$ (learning rates, target firing rate, nonlinearity parameter, number of iterations)
3: Initialize weights $Q \in \mathbb{R}^{N\times M}$ and $W \in \mathbb{R}^{N\times N}$ with small random values around 0.
4: Initialize thresholds $T \in \mathbb{R}^{N}$ with zeros.
5: **for** $i = 1$ to 100 **do**
6:     Set $\alpha = \beta = 0, \gamma = 0.1$.
7:     Perform steps 10 to 26.
8: **end for**
9: Reset $\alpha, \beta, \gamma$ to their original values.
10: **for** $\theta = 1$ to $\Theta$ **do**
11:     Compute the initial outputs $Y$ for all input vectors in $X$ using the feedforward weights and thresholds:
12:         $Y_{ik} = f\left(\sum_{j=1}^{m} Q_{ij}X_{jk} - T_i\right)$ for $i = 1, \ldots, n$
13:     **repeat**
14:         $Y_{ik} \leftarrow dt\cdot\left(f\left(\sum_{j=1}^{m} Q_{ij}X_{jk} + \sum_{j=1}^{n} W_{ij}Y_{jk} - T_i\right)\right)$
15:    $-Y_{ik}$ for $i = 1, \ldots, n$
16:     **until** convergence
17:     Update the feedforward weights $Q$ using Hebbian learning:
18:         $\Delta Q_{ij} = \beta y_i(x_j - q_{ij})$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$
19:         $Q_{ij} \leftarrow Q_{ij} + \Delta Q_{ij}$
20:     Update the feedback weights $W$ using anti-Hebbian learning:
21:         $\Delta W_{ij} = -\alpha(y_iy_j - p^2)$ for $i, j = 1, \ldots, n$
22:         $W_{ij} \leftarrow W_{ij} + \Delta w_{ij}$
23:     Update the thresholds $t_i$:
24:         $\Delta t_i = \gamma(y_i - p)$ for $i = 1, \ldots, n$
25:         $t_i \leftarrow t_i + \Delta\theta_i$
26: **end for**
27: **Output:** The learned feedforward weights $Q$ and feedback weights $W$.
```

III. EXAMPLE I

Here we will consider how a Hebbian network learns patterns of random lines on an 8×8 matrix. The patterns are represented by binary values where a 1 denotes a point on the line segment.

We start by defining a set of random line patterns. For simplicity, let's assume each pattern consists of only horizontal and vertical lines, and are sampled independently from a uniform probability distribution. The input to the network is a vectorized form of these 8×8 matrices, resulting in a 64-dimensional input vector for each pattern.

A. Training the Hebbian Network

The Hebbian network is initialized with random weights $Q \in \mathbb{R}^{n\times 64}$ and thresholds $T \in \mathbb{R}^{n}$. The network's task is to learn the sparse representation of the input patterns. We use the following parameters for the learning algorithm: $\alpha = 0.3, \beta = 0.1, \gamma = 0.1, p = 0.125, \lambda = 10, T = 100$.

Then Algorithm 1 is used to train the network.

B. Results

After training, the network produces a sparse representation of the line patterns. The feed-forward weights Q can be reshaped and visualized as 8×8 matrices to observe the features learned by each unit. For example, if a unit has learned to detect horizontal lines, its corresponding weight matrix will show stronger connections along the rows. It is found that the weights of each unit selectively identifies one of the 16 possible horizontal or vertical lines as seen in Fig. 1.

IV. EXAMPLE II

A more realistic example is a set of binary images of characters of an English alphabet. For this example, I used an 8×8 matrix instead of the 8x15 matrix as used in the paper to reduce computation costs. The fonts are drawn from the same starting position in each of the frames. The input is sampled from the probability distribution of letters occurring in the alphabet. The scheme and training is similar to the previous example with the following parameters: $\alpha = 0.1, \beta = 0.1, \gamma = 0.2, p = 0.125, \lambda = 10, T = 100$. Due to lack of time, the font chosen for this is not ideal and the binary images are flawed. I believe choosing an appropriate font and editing the function to produce more consistent letter patterns can improve the results of my report.

A. Results

The network successfully provides a sparse representation of the letters and shows a 93% retention of information as one can see from Table I and Table II. Fig. 2

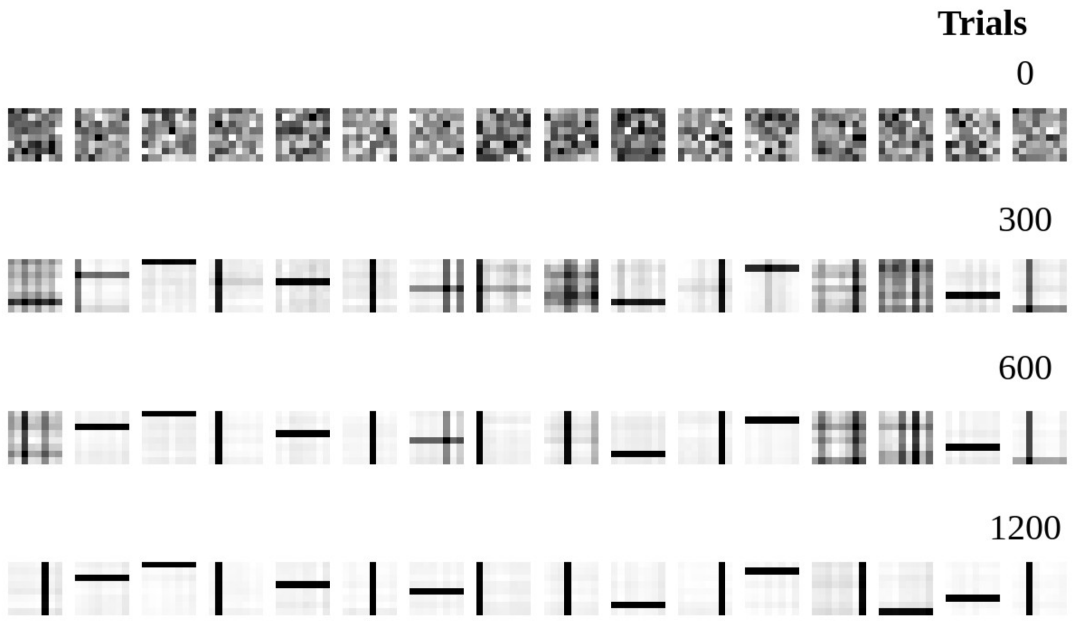

Figure 1. Weight vectors as a function of the training trials in Example I for the parameters, $\alpha = 0.3, \beta = 0.1, \gamma = 0.1, p = 0.125, \lambda = 10, T = 100$. Each 8×8 matrix corresponds to a unit's weight vector where white represents the bit value `0`, and black represents the bit value `1`

Table I. The code generated by the network after the presentation of 1200 letters. The rows indicate the output of the 16 units for the input patterns indicated on the righthand side ($\alpha = 0.001, \beta = 0.001, \gamma = 0.01, p = 0.125, \lambda = 10, T = 100$). Iz = 10, p = 0.1)

Letter	Network Outputs
a	0000001000000000
b	0000000000000001
c	0000010000001000
d	0010000000111001
e	0000010000001000
f	0000000100001000
g	0000101000000000
h	0101000000000001
i	0000000000000000
j	1000000000100100
k	1000000100000000
l	0000000000000000
m	0101000000110001
n	0001000000000000
o	0010000001000000
p	0000000000001001
q	0010000000000001
r	0000000100000000
s	1000100000000000
t	0000000100001000
u	0010001000110000
v	1000100000000000
w	1001100000000000
x	1000000001000000
y	1000100000000000
z	0010100000000000

Table II. The results from Example II

	inputs	**outputs**
number of units	64 (8 x 8)	16
Entropy (E)	4.18	3.917 (93% information retained)
sum of bit entropies (e)	24.14	7.26
Redundancy (R)	477%	85%

shows the weights of the network for this example as a function of training trials.

V. ASSUMPTIONS AND CRITICAL ANALYSIS

While the anti-Hebbian network seems to learn the important features of the patterns really well in the first example, it is important to note that the parameters and the setup are chosen cleverly to optimize the performance of this scheme. The parameters that are specialized for Example I are detailed as follows:

1. The distribution of the lines were artificially chosen to ensure that the firing rates of the neuron matches perfectly with it.
2. The number of the units in the network also correspond to the exact number of independent line structures in the patterns.

The simplicity of the data structure, therefore, allows the code to efficiently learn in surprisingly few trials.

If one were to design a problem statement where the number of distinct features outnumber the number of units, the outputs of the network will become less reliable in many cases. For example, in Figure 3 the network is run for the exact same problem as Example I except for inputs of 9×9 dimensions. We find that the 3rd unit could not select a truly distinct feature of the output. Hence many different patterns would inevitably end up having the same output representation. However, the most frequent features will usually have distinct outputs. This property of the network could be useful in many tasks like Example II.

Even though the second example is a more realistic scenario, it still assumes that the letters are centered in the exact same positions on the frame. That may seem like a decent assumption, however it trains the network to believe that the features of a pattern are always found at the same position and hence the sparse representation is not a true generalization of the patterns.

In real life, the samples of characters may not necessarily overlap with each other and the distinct features (like the top curve of an 'e') may not occur in the same position everytime. This significantly reduces the ability of this network to learn the weight distribution for a constant number of units. The number of units required to recognize the individual features in different positions would grow exponentially for an anti-Hebbian network as described in the paper.

Any learning scheme that aims to generalize recognition of features must be able to identify local information spikes. If one were to apply the same scheme in an MNIST-like dataset, one possible modification I would suggest would be to calculate local conditional information over the input patterns provided. Once areas of interest are found using spikes in this local information, an anti-Hebbian or even a simple Hebbian layered network can be applied to learn the specific feature of that pattern.

I would expect the time complexity of such an algorithm to be high and I recognize that there exist many better learning schemes for the task. However, this idea could provide a more interpretable machine learning setup that could be useful in studying more efficient networks further.

Github: https://github.com/Bluish-y/Anti-Hebbian-Learning/tree/main

[1] P. Földiák, Biological Cybernetics **69**, 165 (1990).

[2] D. Hebb, *The Organization of Behaviour* (Wiley, NY,

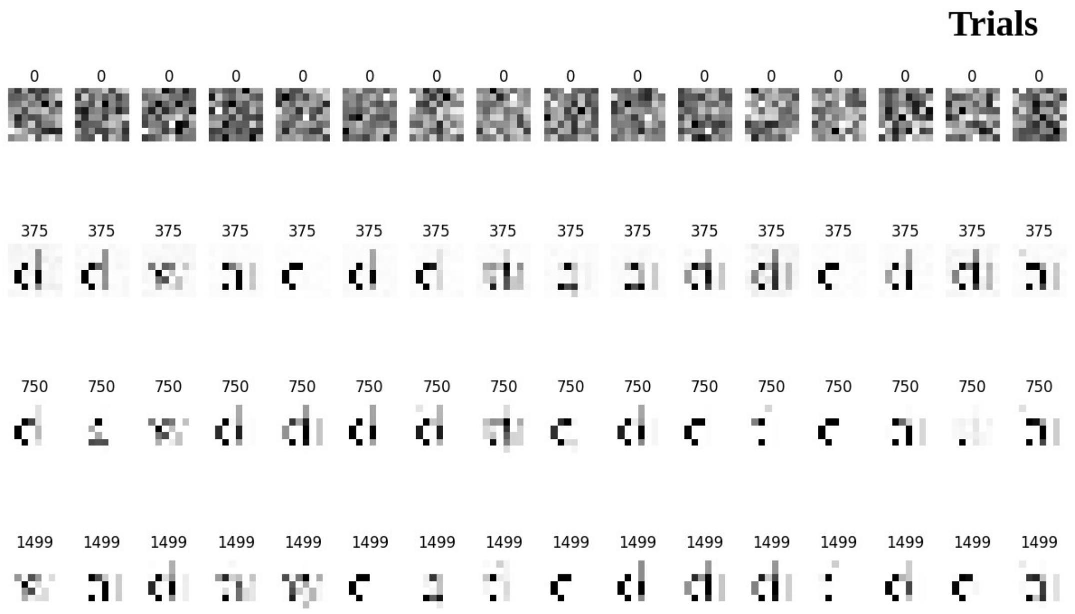

Figure 2. Weight vectors as a function of the training trials in Example II for the parameters, $\alpha = 0.1, \beta = 0.1, \gamma = 0.2, p = 0.125, \lambda = 10, T = 100$. Each 8×8 matrix corresponds to a unit's weight vector where white represents the bit value `0`, and black represents the bit value `1`

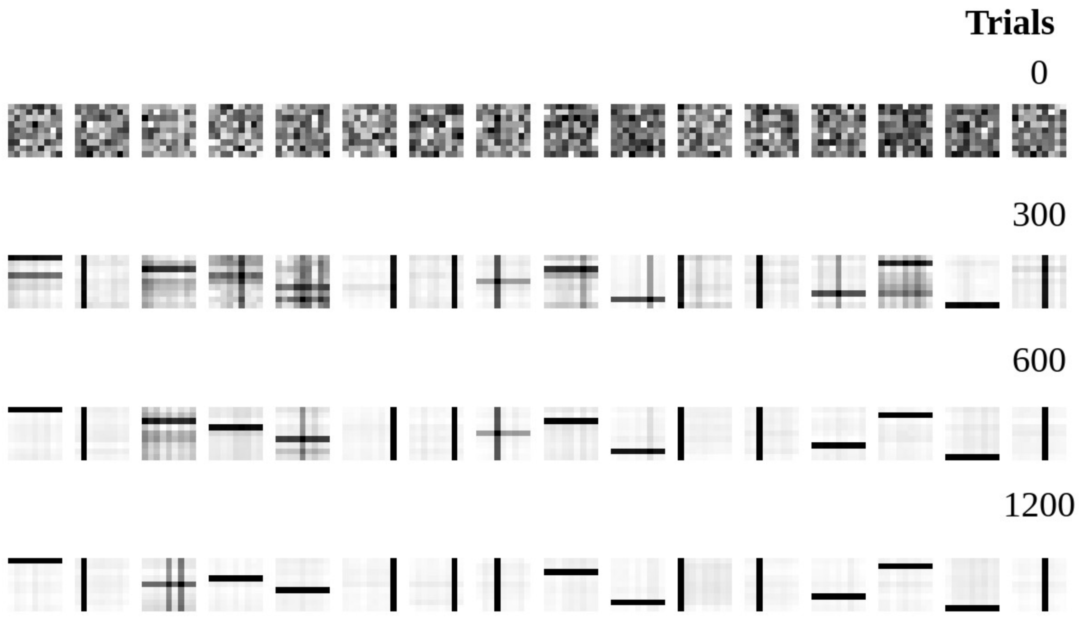

Figure 3. Weight vectors as a function of the training trials for the parameters, $\alpha = 0.1, \beta = 0.1, \gamma = 0.2, p = 0.125, \lambda = 10, T = 100$. The inputs are similar to Example I except with dimensions 9×9. Each 9×9 matrix corresponds to a unit's weight vector where white represents the bit value `0`, and black represents the bit value `1`

1949).