1 Introduction

Hebbian learning is a fundamental principle in neuroscience and artificial intelligence, where synaptic weights are updated based on the correlation of neuron activations. This principle is summarized by the phrase: *'Neurons that fire together, wire together."* This particular set of notes will focus on SUPERVISED Hebian learning, which will use the target output in its learning.

Thiss paper will also focus on the Universal Approximation Theorem that states that a neural network with at least one hidden layer and a non-linear activation function can approximate any continuous function given sufficient neurons and training data. This theorem highlights the power of deep learning models in function representation.

2 Mathematical Formulation of Hebbian Learning

Consider a neural network where an input vector $p \in \mathbb{R}^{R\times 1}$ is transformed through a weight matrix $W \in \mathbb{R}^{S\times R}$ to produce an output activation vector $a \in \mathbb{R}^{S\times 1}$.

$$a = Wp \tag{1}$$

Each element of the output a is computed as:

$$a_i = \sum_{j=1}^{R} W_{ij} p_j \tag{2}$$

where W_{ij} represents the weight connecting input p_j to output a_i.

2.1 Weight Update Rule

The weight update rule in Hebbian learning follows:

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}, \tag{3}$$

where t_{iq} is the ith element of the qth target vector $\mathbf{t}_q$.

This can be written in vector notation as:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T. \tag{4}$$

3 Hebbian Learning Algorithm

The Hebbian learning algorithm's pseudo code follows these steps:

1. Initialize all weights W_{ij} to 0.
2. For each training example (x_j, y_j):
   (a) Compute neuron activation: $a_i = \sum_j W_{ij} x_j$.
   (b) Update the weights:
   $$W_{ij}^{new} = W_{ij}^{old} + x_j y_j \tag{5}$$
   (c) Update the bias:
   $$b^{new} = b^{old} + y_j \tag{6}$$

Through this, we are supervising the network to match the target output eventually.

4 Example: Learning AND with Hebbian Learning

We can enumerate the input for an AND gate with the following truth table:

x_1	x_2	b	Target
-1	-1	1	-1
-1	1	1	-1
1	-1	1	-1
1	1	1	1

In this dataset, the output/target is 1 only when both inputs are 1. We use these inputs in the Hebbian learning rule to adjust the weights accordingly.

4.1 Step 1: Initialize Weights

We initialize the weight vector W to zero:

$$W^{(0)} = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{7}$$

Note: The bias term is treated as an additional input with a fixed value of 1, i.e., $x_b = 1$ for all training samples. This allows the bias weight to be updated using the same Hebbian learning rule as the other weights.

4.2 Step 2: Apply Hebbian Learning Rule

The Hebbian learning rule is given by:

$$W^{(new)} = W^{(old)} + xy \tag{8}$$

where:

- $x = [x_1, x_2, b]$ (input including bias)
- y is the target output

We update the weights for each input step by step.

4.3 Step 3: Iterative Weight Updates

First example: $(-1, -1, 1) \rightarrow y = -1$

$$W^{(1)} = W^{(0)} + (-1, -1, 1)(-1) \tag{9}$$

$$W^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \tag{10}$$

$$W^{(1)} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \tag{11}$$

Second example: $(-1, 1, 1) \rightarrow y = -1$

$$W^{(2)} = W^{(1)} + (-1, 1, 1)(-1) \tag{12}$$

$$W^{(2)} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{13}$$

$$W^{(2)} = \begin{bmatrix} 2 \\ 0 \\ -2 \end{bmatrix} \tag{14}$$

Third example: $(1, -1, 1) \rightarrow y = -1$

$$W^{(3)} = W^{(2)} + (1, -1, 1)(-1) \tag{15}$$

$$W^{(3)} = \begin{bmatrix} 2 \\ 0 \\ -2 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \tag{16}$$

$$W^{(3)} = \begin{bmatrix} 1 \\ 1 \\ -3 \end{bmatrix} \tag{17}$$

Fourth example: $(1, 1, 1) \rightarrow y = 1$

$$W^{(4)} = W^{(3)} + (1, 1, 1)(1) \tag{18}$$

$$W^{(4)} = \begin{bmatrix} 1 \\ 1 \\ -3 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{19}$$

$$W^{(4)} = \begin{bmatrix} 2 \\ 2 \\ -2 \end{bmatrix} \tag{20}$$

4.4 Final Weight Vector

After training on all four inputs, the final weight vector is:

$$W^{(final)} = \begin{bmatrix} 2 \\ 2 \\ -2 \end{bmatrix} \tag{21}$$

4.5 Step 4: Classifying Inputs

To verify that the trained perceptron correctly classifies the AND function, we compute the weighted sum for each input:

$$a = W^T x = 2x_1 + 2x_2 - 2 \tag{22}$$

Applying the symmetric hard limit activation function as an activation:

$$f(a) = \begin{cases} 1, & \text{if } a \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{23}$$

Computing for each input:

- For $(-1, -1, 1)$: $2(-1) + 2(-1) - 2 = -6 \rightarrow f(-6) = -1$
- For $(-1, 1, 1)$: $2(-1) + 2(1) - 2 = -2 \rightarrow f(-2) = -1$
- For $(1, -1, 1)$: $2(1) + 2(-1) - 2 = -2 \rightarrow f(-2) = -1$
- For $(1, 1, 1)$: $2(1) + 2(1) - 2 = 2 \rightarrow f(2) = 1$

4.6 Conclusion

As we can see, the trained perceptron using Hebian learning was able to correctly classify inputs with respect to the AND function.
For this dataset, the first three inputs are negative, while the fourth is positive. We use these inputs in the Hebbian learning rule to adjust the weights accordingly.

5 Representational Power of Neural Networks

Neural networks possess strong representational power, allowing them to approximate various functions:

- Boolean/discrete function approximation
- Continuous function approximation

For example, in classification problems, perceptrons create decision boundaries to distinguish between classes. From the ones we have seen before, we usually cases where a single line is used as a boundary. But there could be casees where more complex boundaires need to be used, like in Figure 1. As you can see in this image, thee boundary takes shape of a hexagon, and the model needs to accurately place the green input data inside the shape or inside the boundary, and everything else outside. We could use multiple perceptrons (for this case 6) and train a model to accomplish this task.

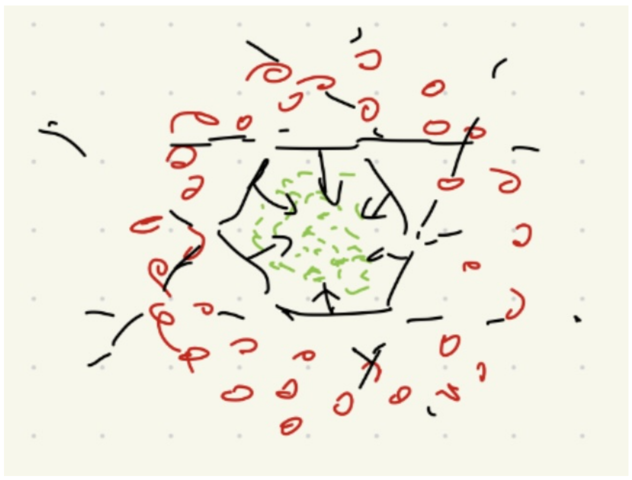

Figure 1: Example of a classification problem solved using perceptrons

5.1 Universal Approximation Theorem

A feedforward network with a single hidden layer containing a sufficient number of neurons and a sigmoidal activation function can approximate any continuous function arbitrarily well.
A sigmoidal function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-ax}} \tag{24}$$

where a controls the steepness.
For any function $f(x)$, it can be approximated as:

$$f(x) \approx \sum_{i=1}^{N} d_i \sigma(w_i x + \theta_i) \tag{25}$$

where d_i are learned coefficients.

6 Mathematical Proof of the Universal Approximation Theorem

The Universal Approximation Theorem states that a feedforward neural network with a single hidden layer, given a non-constant, bounded, and continuous activation function, can approximate any continuous function on a compact subset of $\mathbb{R}^n$ arbitrarily well. This result was rigorously proven by Cybenko (1989) and further expanded upon by Hornik et al.

6.1 Step 1: Approximation by Superpositions of a Sigmoidal Function

Cybenko's proof is based on the idea that a function $f(x)$ can be approximated by finite linear combinations of compositions of a fixed, univariate function σ (activation function) and a set of affine functionals. Specifically, for any continuous function $f : \mathbb{R}^n \to \mathbb{R}$ defined on a compact domain and any $\epsilon > 0$, there exist parameters $w_i \in \mathbb{R}^n$, $b_i \in \mathbb{R}$, and $c_i \in \mathbb{R}$ such that:

$$\sup_{x\in\mathbb{R}^n} \left| f(x) - \sum_{i=1}^{N} c_i \sigma(w_i^T x + b_i) \right| < \epsilon \tag{26}$$

where σ is any sigmoidal activation function satisfying:

$$\lim_{t\to\infty} \sigma(t) = 1, \quad \lim_{t\to-\infty} \sigma(t) = 0. \tag{27}$$

6.2 Step 2: Density in Continuous Function Space

To show that such superpositions are dense in the space of continuous functions, consider the function space $C(I^n)$ of all continuous functions on the unit hypercube $I^n = [0,1]^n$. Define a linear subspace S of $C(I^n)$ consisting of finite linear combinations of the form:

$$G(x) = \sum_{i=1}^{N} c_i \sigma(w_i^T x + b_i). \tag{28}$$

If S were not dense in $C(I^n)$, then by the Hahn-Banach Theorem, there would exist a nonzero signed measure μ such that:

$$\int_{I^n} G(x) d\mu(x) = 0, \quad \forall G \in S. \tag{29}$$

By showing that σ is discriminatory (i.e., it does not satisfy such an integral equation unless μ is the zero measure), it follows that the closure of S is all of $C(I^n)$. Hence, any continuous function can be approximated arbitrarily well by such neural networks.

6.3 Step 3: Approximation of Arbitrary Decision Regions

An important consequence of the theorem is that a neural network can approximate any decision function corresponding to a partition of the input space. Suppose we wish to classify points in $\mathbb{R}^n$ into two sets, represented by an indicator function:

$$f(x) = \begin{cases} 1, & x \in D \\ 0, & x \notin D \end{cases} \tag{30}$$

where D is a measurable subset of $\mathbb{R}^n$. By Lusin's theorem, there exists a continuous function $h(x)$ that agrees with $f(x)$ on a large subset of the domain. Since $h(x)$ can be approximated by a neural network, we conclude that a neural network can approximate $f(x)$ to arbitrary accuracy.

6.4 Conclusion

The Universal Approximation Theorem highlights the fundamental power of neural networks. The ability to approximate arbitrary continuous functions makes neural networks a cornerstone of modern machine learning. However, this result does not address the efficiency of such approximations, which depends on the number of neurons and the complexity of function $f(x)$. Further research explores optimal network architectures and training methodologies to achieve efficient approximations.