M-P Neuron: A binary discrete-time element

Input: a_i^t (0 or 1 only)
Weight: w^i (+1 for excitatory, -1 for inhibitory)
Excitation threshold: θ
Instant State: $S^t = \sum_i w_i^t a_i^t = f(t)$ (fixed, doesn't depend on the previous state)
Output: $x^{t+1} = 1$ iff $S^t \geq \theta$, or $x(t) = g(S^t) = g(f(t))$
Threshold activation function:

$$g(S^t) = H(S^t - \theta) = \begin{cases} 1, S^t \geq \theta \\ 0, S^t < \theta \end{cases}$$

Heaviside (unit step) function: $H(X) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$

ANN learning rule:

Adjust the weights of connections to get desirable output

Hebb's Rule

Increase weight of connection at every next instant:
$w_{ji}^{k+1} = w_{ji}^k + \Delta w_{ji}^k$, where $\Delta w_{ji}^k = C a_i^k x_j^k$ (C is learning rate)

1. Calculate $S^0 = \sum_i a_i^0 w_1^0$
2. Calculate Δw_i^0 and w_i^1
3. Calculate Δw_i^1 and w_i^2……

Supervised Learning: Classification with label

Assumption: The distribution of the training examples is identical to the distribution of test examples.

Perceptron: Error-correcting rule

Simplest architecture:
One layer of input units
One layer of output units

Input values
input layer
adjustable layer
output layer
Output values

Input: $S_j = \sum_{i=0}^n w_{ji} a_i$
with a bias input unit a_0
Threshold activation function:
$X_j = f(S_j) = \begin{cases} 1, S_j \geq \theta_j \\ 0, \ S_j < \theta_j \end{cases}$
Output vector: $X = X_0, X_1, \cdots, X_n$
Error: $e_j = (t_j - X_j)$, used to re-adjust the weights
Δw = learning rate * (teacher - output) * input

1. Calculate $e_j = (t_j - X_j)$
2. Calculate $\Delta w_{ji} = C e_j a_i = C(t_j - X_j) a_i$
3. Update weights $w_{ji} = w_{ji} + \Delta w_{ji}$

Keep training until the algorithm converges:

- The training data is linearly separable
- The learning rate is sufficiently small

Perceptron Convergence Theorem

For any data set that's linearly separable, the learning rule is guaranteed to find a solution in a finite number of steps.
Assumptions:

- At least one such set of weights, w*, exists
- There are a finite number of training patterns
- The threshold function is uni-polar (0 or 1)

Perceptron Performance: RMS

RMS (root-mean-square) error: $\sqrt{\left(\frac{\sum_{i=1}^N (x_i - \hat{x}_i)^2}{N}\right)}$
where x_i is the target output, $\hat{x}_i$ is the instant output.
The RMS error is a function of the instant output only.
Minimize the RMS error to get the best performance.

Perceptron Classifier

x_1 w_1 x_2 w_2 1 w_0
S=w₀+w₁x₁+w₂x₂ → y=g(S) → {-1, +1}
Output = class decision
View the bias as another weight from an input which is constantly on
1. Weighted sum of the inputs
2. Pass through Heaviside function: T(S)= -1 if S < 0, T(S)= 1 if S>= 0

Hyperplane decision surface: $w \cdot x^T = 0$
If two classes of patterns can be separated by a decision boundary $b + \sum_{i=1}^n x_i w_i = 0$, then they are linearly separable.
Without bias, the hyperplane will be forced to intersect origin.
Linearly inseparable: XOR; Linearly separable: AND, OR

Gradient Descent Rule

Perceptron classifier fails if the data is not linearly separable.
Minimize the error $E(w) = \frac{1}{2}\sum_e (y_e - o_e)^2$ in the steepest direction (most rapid decrease) -- in the direction opposite to the gradient:
$\Delta E(w) = [\partial E/\partial w_0, \partial E/\partial w_1, \cdots, \partial E/\partial w_n]$, $w_i = w_i + \eta\, \partial E/\partial w_i$
Weight update can be derived:

$$\partial E/\partial w = \partial\left(\frac{1}{2}\sum_e (y_e - o_e)^2\right)/\partial w_i = \sum_e (y_e - o_e)(-x_{ie})$$

➔ $w_i = w_i + \eta \sum_e (y_e - o_e) x_{ie}$, $\Delta w = -\eta \frac{\partial E_e}{\partial w}$
Repeat until termination condition is satisfied:

1. Calculate the output: $o_e = \sum_{i=0}^d w_i x_{ie}$
2. Calculate the update: $\Delta w_i = \Delta w_i + \eta(y_e - o_e) x_{ie}$
3. Update the accumulated weights: $w_i = w_i + \Delta w_i$

Cons: converges very slowly; multiple local minima in the error surface, then there is no guarantee that it will find the global min.

Incremental Gradient Descent

Difference: The gradient descent rule updates the weights after calculating the whole error accumulated from all examples, the incremental version approximates the gradient descent error decrease by updating the weights after each training example.
For each training example:

1. Calculate the network output: $o_e = \sum_{i=0}^d w_i x_{ie}$
2. Update the weights: $w_i = w_i + \eta(y_e - o_e) x_{ie}$

Sigmoidal Perceptron

$\sigma = \sigma(S) = \frac{1}{1+e^{-S}}$, where $S = \sum_{i=1}^d w_i x_i$
For each training example:

1. Calculate the output: $o_e = \sigma(\sum_{i=0}^d w_i x_{ie})$
2. Calculate the update:
$$\Delta w_i = \Delta w_i + \eta(y_e - o_e)\sigma(s)(1 - \sigma(s)) x_{ie}$$
3. Update the accumulated weights: $w_i = w_i + \Delta w_i$

Incremental Gradient Descent Version: Same as above

Perceptron Rule vs. Gradient Descent Rule

Perceptron training:

- uses thresholded unit
- converges after a finite number of iterations
- output hypothesis classifies training data perfectly
- linearly separability necessary

Gradient descent:

- uses unthresholded linear unit
- converges asymptotically toward a min error hypothesis
- termination is not guaranteed
- linear separability not necessary

Multi-layer Perceptron (MLP)

Requires differentiable, continuous nonlinear activation functions.
Sigmoid: $\sigma(S) = \frac{1}{1+e^{-S}}$, Hyperbolic tangent: $tanh(s) = \frac{e^S - e^{-S}}{e^S + e^{-S}}$
A two-layer neural network implements the function:

$$f(x) = \sigma\left(\sum_{j=1}^J w_{jk}\, \sigma\left(\sum_{i=1}^I w_{ij} x_i + w_{oj}\right) + w_{ok}\right)$$

MLP Solving XOR Problem

2 input neurons, 2 hidden neurons, 1 output neurons, weights for input-hidden {1}, for hidden-output {1, -2}, bias in hidden {0, -1}, in output 0. ReLU.

Backpropagation Learning Algorithm

Initialize weights set to small random values, set a learning rate;
Repeat for each training example (x, y):

Forward:

1. $o_j = \sigma(S_j) = \frac{1}{1+e^{-S_j}}$, $S_j = \sum_{i=0}^d w_{ij}\, o_i$, where $o_i = x_i$ (hidden)
2. $o_k = \sigma(S_k) = \frac{1}{1+e^{-S_k}}$, $S_k = \sum_{j=0}^d w_{jk}\, o_j$ (output)

Backward:

1. Calculate the benefit β_k at the node k in the output layer: $\beta_k = o_k(1 - o_k)[y_k - o_k]$ (Effects from the output nodes)
2. Calculate the changes for weights j -> k on connections to nodes in the output layer: $\Delta w_{jk} = \eta\beta_k o_j$, $\Delta w_{0k} = \eta\beta_k$ (Effects from the output of the neuron)
3. Calculate the benefit β_j for the hidden node: $\beta_j = o_j(1 - o_j)[\sum_k \beta_k w_{jk}]$ (Effects from multiple nodes in the next layer)
4. Calculate the changes for weights i -> j on connections to nodes in the hidden layer: $\Delta w_{ij} = \eta\beta_j o_i$, $\Delta w_{0j} = \eta\beta_j$
5. Update the weights by the changes: $w = w + \Delta w$

Online (Incremental) Training: Revision by example

Derivation of Backpropagation Algorithm

The BP training algo for MLP is a generalized gradient descent rule
For weights $j \to k$ on connections to nodes in the output layer:

$$\frac{\partial E_e}{\partial w_{jk}} = \frac{\partial E_e}{\partial s_k} \cdot o_j,\ \frac{\partial E_e}{\partial s_k} = \frac{\partial E_e}{\partial o_k} \cdot \frac{\partial o_k}{\partial s_k},\ \frac{\partial o_k}{\partial s_k} = \frac{\partial \sigma(s_k)}{\partial s_k} = o_k(1 - o_k)$$

$$\frac{\partial E_e}{\partial o_k} = \frac{\partial\left(\frac{1}{2}\sum_k (y_l - o_l)^2\right)}{\partial o_k} = \frac{\partial\left(\frac{1}{2}(y_k - o_k)^2\right)}{\partial o_k} = \frac{1}{2} \cdot 2 \cdot (y_k - o_k)\frac{\partial (y_k - o_k)}{\partial o_k} = -(y_k - o_k)$$

Therefore, $\frac{\partial E_e}{\partial s_k} = -(y_k - o_k) o_k (1 - o_k)$, and $\frac{\partial E_e}{\partial w_{jk}} = \frac{\partial E_e}{\partial s_k} \cdot o_j$
Substitute $\Delta w_{jk} = -\frac{\partial E_e}{\partial w_{jk}} = \eta\beta_k o_j$, $\beta_k = (y_k - o_k) o_k (1 - o_k)$
Then we have $\Delta w_i = \Delta w_i + \eta(y_e - o_e)\sigma(s)(1 - \sigma(s)) x_{ie}$
For weights $i \to j$ on connections to nodes in the hidden layer:

$$\frac{\partial E_e}{\partial s_j} = \sum_k \frac{\partial E_e}{\partial s_k} \cdot \frac{\partial s_k}{\partial s_j} = \sum_k -\beta_k \cdot \frac{\partial s_k}{\partial s_j} = \sum_k -\beta_k \cdot \frac{\partial s_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial s_j} = \sum_k (-\beta_k) \cdot w_{jk} \cdot \frac{\partial o_j}{\partial s_j} = \sum_k (-\beta_k) \cdot w_{jk} \cdot o_j (1 - o_j)$$

For the hidden units:
$\Delta w_{ij} = \eta\beta_j o_i$, $\Delta w_{0j} = \eta\beta_j$, $\beta_j = -\frac{\partial E_e}{\partial s_j} = o_j(1 - o_j)[\sum_k \beta_k w_{jk}]$
It can be generalized so that $\frac{\partial E_{total}}{\partial w_{ij}} = \sum_e \frac{\partial E_e}{\partial w_{ij}}$

Momentum: stabilize the weight change

$\Delta w(t) = -\eta \frac{\partial E_e}{\partial w(t)} + \alpha \Delta w(t-1)$, t is the index of current change
It smooths the weight changes and suppresses cross- stitching, that is cancels side-to-side oscillations across the error valley

Overcome overfitting

Early stopping, network pruning, regularization techniques, …
Weight decay: penalizes large weights to reduce variance
Cross validation (k-fold, leave-one-out)

Convolutional Neural Network

N*N layer and m*m filter → (N-m+1) * (N-m+1) layer output
Rectified Linear Unites (ReLU): $f(x) = max(0, x)$
Benefits of ReLU: much simpler computationally
– The forward/backward passes through it are just a simple if statement
– The sigmoid activation requires computing an exponent
– This advantage is huge when dealing with big networks with many neurons, and can significantly reduce both training and evaluation times
– Sigmoid activations are easier to saturate (hampers learning in deep networks), while ReLUs only saturates when the input is less than 0.
Past Exam: What is the disadvantage of a fully-connected neural network compared to a CNN with the same size layers?
- In fully-connected NN, there're too many weights to learn.

Transfer Learning

The ability of a system to recognize and apply knowledge and skills learned in previous tasks to novel tasks (in new domains)

Radial-basis Function (RBF) Networks

$F(x) = \sum_{i=1}^N w_i \phi(||x - x_i||)$, where $\phi(||x - x_i||)$ is a set of non-linear radial-basis functions, x_i are the centers of these functions, and $||\cdot||$ is the Euclidean norm. Are used to solve curve-fitting or interpolation problem.
RBF is a real-valued function whose depends only on the distance from the origin $\phi(x) = \phi(||x||)$, or the center c: $\phi(x, c) = \phi(||x - c||)$
Gaussian RBF: $F(x) = \sum_{i=1}^n w_i exp\left(-\frac{||x - t_i||^2}{2\sigma_i^2}\right)$

$$\text{Matrix Form: } \phi = \begin{bmatrix} \phi_1(||x_1 - t_1||) & \cdots & \phi_n(||x_1 - t_n||) \\ \vdots & \ddots & \vdots \\ \phi_1(||x_N - t_1||) & \cdots & \phi_n(||x_N - t_n||) \end{bmatrix}$$

$$\phi \begin{bmatrix} w_1 \\ \cdots \\ w_n \end{bmatrix} = \begin{bmatrix} d_1 \\ \cdots \\ d_N \end{bmatrix} = d, \text{ where } o(x_i) = d_i$$

RBF Training: during the training, the centers and weights are changed

1. Determine the network structure with n basis functions ϕ_i, with centers t_i using k-means clustering algorithm
2. Determine the basis function variances σ_i^2
3. Compute each output with Gaussian $\phi_{ei} = exp\left(\frac{-||x_e - t_i||^2}{2\sigma_i^2}\right)$
4. Compute the correlation matrix: $\phi^T\phi$ and pseudo inverse $(\phi^T\phi)^{-1}$, the vector $\phi^T d$, the weights $W = (\phi^T\phi)^{-1}\phi^T d$

Clustering

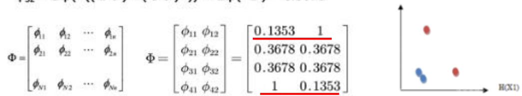
Good clustering: high intra-class similarity and low inter-class similarity
Minkowski distance: $d(i,j) = \sqrt[q]{|x_{i1} - x_{j1}|^q + |x_{i2} - x_{j2}|^q + \cdots + |x_{ip} - x_{jp}|^q}$
If q = 1, d is Manhattan distance, if q = 2, d is Euclidean distance

RBF Networks solving XOR

$\phi_i = exp\left(-||x - x_i||^2\right)$, choose $x_1 = (1,1)$, $x_2 = (0,0)$, $2\sigma_i^2 = 1$

- **Training:**

φ11 = exp(-((0-1)²+(0-1)²)) = exp(-2) = 0.1353
φ21 = exp(-((0-1)²+(1-1)²)) = exp(-1) = 0.3678
φ31 = exp(-((1-1)²+(0-1)²)) = exp(-1) = 0.3678
φ41 = exp(-((1-1)²+(1-1)²)) = exp(0) = 1
φ12 = exp(-((0-0)²+(0-0)²)) = exp(0) = 1
φ22 = exp(-((0-0)²+(1-0)²)) = exp(-1) = 0.3678
φ32 = exp(-((1-0)²+(0-0)²)) = exp(-1) = 0.3678



RBF Networks vs MLP

Similarities:

- Both are layered feedforward networks that produce nonlinear function mappings; Both proven to be universal approximators

Differences:

- RBF has only one hidden layer, while MLP has one or more hidden layers depending on the application task;
- The nodes in the hidden and output layers of MLP use the same activation function, while RBF uses different activation functions (Gaussians with different centers and variances);
- The hidden and output layers of MLP are nonlinear, while only the hidden layer of RBF is nonlinear (output linear)
- The activation functions in the RBF nodes compute the Euclidean distance between the input examples and the centers, while the activation functions of MLP compute inner products from the input examples and the incoming weights;
- MLP constructs global approximations while RBF construct local a~.

Time Series Prediction

A sequence of vectors: $\{x(t_0), x(t_1), \cdots, x(t_{i-1}), x(t_i), x(t_{i+1}), \cdots\}$
MLP & RBF networks are static networks
Dynamic networks: Elman network, RNN, LSTM
Elman network adopts BP training, the error function $E = \sum_{k=1}^n [y(k) - d(k)]^2$, where $d(k)$ is the expected output (target). The network structure includes an input layer, a hidden layer (recurrent connections) and an output layer. Back propagation through time (BPTT) training.
In the prediction, the Elman network uses the input data at the current time step and the hidden state to predict the output at the current time step.
RNN: self-connected networks, BPTT learning, suffers from vanishing gradient and long memory problems
LSTM: memory block (cell), non-decaying error backpropagation, solves the vanishing gradients and the long memory limitations

Principal Component Analysis (PCA)

Eigenvectors and eigenvalues: $Sv = \lambda v \Leftrightarrow (S - \lambda I)v = 0$
Singular Value Decomposition (SVD): $A = U \sum V^T$, $\sigma_i = \sqrt{(\lambda_i)}$
Used for dimension reduction, PC may not be interpretable

Hebbian Learning: Unsupervised, intrinsically unstable

When a neuron repeatedly excites another neuron, then the threshold of the latter neuron is decreased, or the synaptic weight between the neurons is increased, in effect increasing the likelihood of the second neuron to excite.

$$y = w^T x = x^T w, \quad \Delta w_{ji} = \eta y_j x_i, \quad y = |w||x|cos(\alpha)$$

- The simple Hebbian rule causes the weights to increase (or decrease) without bounds

Oja's Rule (normalized Hebbian rule): involves "forgetting term"
Normalized to $w_{ji}(n+1) = \frac{w_{ji}(n) + \eta x_i(n) y_j(n)}{\sqrt{\left(\sum_i [w_{ji}(n) + \eta x_i(n) y_j(n)]^2\right)}}$
For $\eta \ll 1$, $w_{ji}(n+1) = w_{ji}(n) + \eta y_j(n)[x_i(n) - y_j(n) w_{ji}(n)]$
In PCA, assume that the first component is already obtained, compute the projection of the first eigenvector on the input $y = w_1^T x$
Then generate the modified input as $\hat{x} = x - w_1 y = x - w_1 w_1^T x$

Auto-encoders: information compression, dimensionality reduction

Back-propagation algorithm can be used for unsupervised learning to discover significant features that characterise input patterns. This can be achieved by learning the identity mapping, passing the data through a bottleneck: auto-encoders. input-to-hidden: encoder; hidden-to-output: decoder

Unsupervised Competitive Learning: Winner-takes-all (WTA)
Simple competitive learning: $h_j = \sum_i w_{ji} x_i$, $w_{j*} x \geq w_j x\ \forall x$
Winner = output node whose incoming weights are the shortest Euclidean distance from the input vector
Update rule for all neurons: $\Delta w_{j*i} = \eta y_j (x_i - w_{j*i})$, $\begin{cases} y_{j*} = 1 \\ y_j = 0, if\ j \neq j* \end{cases}$
Leaky learning: modifying weights of both winning and losing units but at different learning rates $w(t+1) = w(t) + \eta(x - w(t))$, where $\eta_w \gg \eta_L$
Maxnet: a specific competitive net that performs WTA competition
Lateral inhibition between competitors: output of each node feeds to others through inhibitory connections (with negative weights)

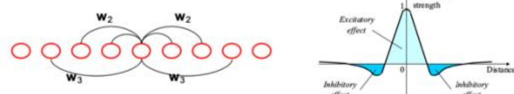
weights: $w_{ji} = \begin{cases} \theta & \text{if } i = j \\ -\varepsilon & \text{otherwise} \end{cases}$
node function: $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

- Competition: iterative process until the net stabilizes (at most one node with positive activation) 0<ε <1/m, where m is the # of competitors

- ε too small: takes too long to converge
- ε too big: may suppress the entire network (no winner)

Mexican Hat Network: Multiple winners WTA

For a given node in the output layer,
Close neighbors: cooperative (mutually excitatory, w > 0)
Distant neighbors: competitive (mutually inhibitory, w < 0)
Too far away neighbors: irrelevant (w = 0)



$$w_{ij} = \begin{cases} c_1 & \text{if} \quad \text{distance}(i,j) < k \ (c_1 > 0) \\ c_2 & \text{if} \quad \text{distance}(i,j) = k \ (0 < c_2 < c_1) \\ c_3 & \text{if} \quad \text{distance}(i,j) > k \ (c_3 \le 0) \end{cases}$$

activation function

$$f(x) = \begin{cases} 0 & \text{if} \quad x < 0 \\ x & \text{if} \quad 0 \le x \le \max \\ \max & \text{if} \quad x > \max \end{cases} \quad \text{ramp function:}$$

Important application of competitive learning

Vector quantization: categorize a given set of input vectors into M classes using competitive learning algorithms, represent any vector just by the class into which it falls.

- Divides entire pattern space into a number of separate subspaces
- Set of M units represent set of prototype vectors: CODEBOOK
- New pattern x is assigned to a class based on its closeness to a prototype vector using Euclidean distances

Associative Memories

An associative memory is a content- addressable structure that maps a set of input patterns to a set of output patterns.
Two types: auto-associative and hetero-associative
Goal: obtain a set of weights w_{ij} from a set of training pattern pairs $s:t$, such that when s in the input layer, t in the output layer
Simple AM: single layer, similar to Hebbian in classification
Algo: For each training samples $s:t$, $\Delta w_{ij} = s_i \cdot t_j$:
If $\Delta w_{ij} = 0$ initially, then after updates for all P training patterns
$w_{ij} = \sum_{p=1}^{P} s_i(p)t_j(p)$, $W = w_{ij}$ → Calculate the outer product
Example 1: Hetero-associative
Given:

Binary pattern pairs $s:t$ with |s|=4 and |t|=2, total weighted input t output units: $y_{in_j} = \sum_i x_i w_{ij}$,

activation: $y_j = \begin{cases} 1, & y_{in_j} > 0 \\ 0, & y_{in_j} \le 0 \end{cases}$,

weights $W = \sum_{p=1}^{P} s_i^T(p)t_j(p)$

	s(p)	t(p)
p=1	(1 0 0 0)	(1, 0)
p=2	(1 1 0 0)	(1, 0)
p=3	(0 0 0 1)	(0, 1)
p=4	(0 0 1 1)	(0, 1)

Compute:

$$S^T(1) \otimes t(1) = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} [1 \quad 0] = \begin{bmatrix}1&0\\0&0\\0&0\\0&0\end{bmatrix} \quad S^T(2) \otimes t(2) = \begin{bmatrix}1\\1\\0\\0\end{bmatrix} [1 \quad 0] = \begin{bmatrix}1&0\\1&0\\0&0\\0&0\end{bmatrix}$$

$$S^T(3) \otimes t(3) = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} [0 \quad 1] = \begin{bmatrix}0&0\\0&0\\0&0\\0&1\end{bmatrix} \quad S^T(4) \otimes t(4) = \begin{bmatrix}0\\0\\1\\1\end{bmatrix} [0 \quad 1] = \begin{bmatrix}0&0\\0&0\\0&1\\0&1\end{bmatrix}$$

Compute the weights: $W = \sum_{p=1}^{P} s_i^T(p)t_j(p)$, $W = \begin{bmatrix}2&0\\1&0\\0&1\\0&2\end{bmatrix}$

x=(1 0 0 0) **x=(0 1 0 0)** similar to S(1) and S(2)

$$(1 \ 0 \ 0 \ 0)\begin{pmatrix}2&0\\1&0\\0&1\\0&2\end{pmatrix} = (2 \ 0) \qquad (0 \ 1 \ 0 \ 0)\begin{pmatrix}2&0\\1&0\\0&1\\0&2\end{pmatrix} = (1 \ 0)$$

$y_1 = 1, \ y_2 = 0$ $\qquad$ $y_1 = 1, \ y_2 = 0$

Example 2: Auto-associative

For a single pattern $s = (1,1,1,-1)$ in $W = \begin{bmatrix}1&1&1&-1\\1&1&1&-1\\1&1&1&-1\\-1&-1&-1&1\end{bmatrix}$

training pat. (111 −1)·W = (4 4 4 − 4)→ (111 −1)
noisy pat. (−111 −1)·W = (2 2 2 − 2)→ (111 −1)
missing info (001−1)·W =(222−2)→(111−1)
more noisy (−1 −11 −1)·W = (0 0 0 0) not recognized
Replace diagonal elements by zero: (-1 -1 1 -1) wrong

Hopfield Network

A fully connected, symmetrically weighted network where each node functions act as both input and output node.
Can be used to restore incomplete or noisy input patterns.
Randomly select one unit, asynchronous

$H_i(t+1) = \sum_{j=1, j \neq i}^{n} w_{ij} \, v_j(t) + I_i$

$v_i(t+1) = sgn[H_i(t+1)] = \begin{cases} 1, H_i(t+1) \ge 0 \\ -1, H_i(t+1) < 0 \end{cases}$

Example 1: Given 4 node network, 2 patterns (1111) (-1-1-1-1), weights $w_{l,j} = 1$ for $l \neq j$, and $w_{j,j} = 0$ for all j.
Recover the input pattern: $I = (I_1, I_2, I_3, I_4) = (111-1)$
For node 2: $w_{2,1}x_1 + w_{2,3}x_3 + w_{2,4}x_4 + I_2 = 2 \ge 0$ => $(111-1)$
For node 4: $w_{4,1}x_1 + w_{4,2}x_2 + w_{4,3}x_3 + I_4 = 2 \ge 0$ => (1111)
Recover the input pattern: $I = (I_1, I_2, I_3, I_4) = (11-1-1)$
For node 2: net = 0, no change (1 1 -1 -1)
For node 3: net = 0, change state from -1 to 1 (1 1 1 -1)
For node 4: net = 0, change state from -1 to 1 (1 1 1 1)
Example 2: Calculate weights matrix $w = \sum_{k=1}^{p} x^k (x^k)^T - pI$
Example 3: Spurious State
Given 4 node network, 3 patterns (1 1 -1 -1) (1 1 1 1) (-1 -1 1 1)

- Recover pattern (-1 -1 -1 -1):
  If node 4 is randomly selected,
  no change of state for node 4
  Same for all other nodes
  net stabilized at (-1 -1 -1 -1) ➔ spurious state

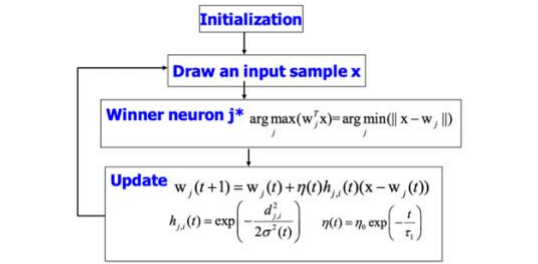
  $w = \begin{pmatrix}0&1&-1/3&-1/3\\1&0&-1/3&-1/3\\-1/3&-1/3&0&1\\-1/3&-1/3&1&0\end{pmatrix}$
- Recover another pattern (-1 -1 -1 0): if the node selection sequence is 1,2,3,4, the net stabilizes at state (-1 -1 1 1) → correct

Limitations of Hopfield Network

The number of patterns that can be stored and accurately recalled is severely limited (net may converge to a novel spurious pattern)
Exemplar pattern will be unstable if it shares many bits in common with another exemplar pattern

Kohonen's Self-organizing Map (SOM) for dimension reduction

The idea in an SOM is to transform an input of arbitrary dimension into a 1 or 2 dimensional discrete map.
Competition, Cooperation, and Synaptic Adaptation:
Larger neighborhood: good global ordering and bad local fit



Learning Vector Quantizer (LVQ)

LVQ is a supervised learning technique that uses class information to move the Voronoi vectors slightly, so as to improve the quality of the classifier decision regions.

1. Randomly select an input vector x
2. If the winner belongs to the right class, $w^{new} = w^{old} + \eta(x - w)$
3. If the winner belongs to the wrong class, $w^{new} = w^{old} - \eta(x - w)$

MATLAB Code

```
% Create a Self-Organizing Map
dimension1 = 10;
dimension2 = 10;
net = selforgmap([dimension1
dimension2]);
% Train the Network
[net,tr] = train(net,x);
% Test the Network
y = net(x);
% View the Network
view(net)
% Plot
figure, plotsomtop(net)
```

```
%Create and train 2D-SOM
% SOM parameters
dimensions = [10 10];
coverSteps = 100;
initNeighbor = 4;
topologyFcn = 'hextop';
distanceFcn = 'linkdist';
% define net
net2 =
selforgmap(dimensions,coverSte
ps,initNeighbor,topologyFcn,dist
anceFcn);
% train
[net2,Y] = train(net2,P);
%plot 2D-SOM results
% plot input data and SOM
weight positions
plotsompos(net2,P);
grid on
% plot SOM neighbor distances
plotsomnd(net2)
% plot for each SOM neuron the
number of input vectors that it
classifies
figure
plotsomhits(net2,P)
```

```
% Define the training inputs and
targets
p = [0 0 1 1; 0 1 0 1];
t = [0 0 0 1];
% Create the backpropagation
network
net = newff(minmax(p), [4 1],
{'logsig', 'logsig'}, 'traingdx');
% Train the bp network
net.trainParam.epochs = 500; %
training stops if epochs reached
net.trainParam.show = 1; % plot
the performance function at
every epoch
net = train(net, p, t);
% Testing the performance of the
trained backpropagation network
a = sim(net, p)
>> a = 0.0002 0.0011 0.0001
0.9985
>> t = 0 0 0 1
```

```
%1D and 2D Self Organized Map
%Define 4 clusters of input data
close all; clear all; clc; format compact
% number of samples of each cluster
K = 200;
% offset of classes
q = 1.1;
% define 4 clusters of input data
P = [rand(1,K)-q rand(1,K)+q
rand(1,K)+q rand(1,K)-q;
rand(1,K)+q rand(1,K)+q rand(1,K)-q
rand(1,K)-q];
% plot clusters
plot(P(1,:),P(2,:),'g.')
hold on
grid on
```

```
%Create and train 1D-SOM
% SOM parameters
dimensions = [100];
coverSteps = 100;
initNeighbor = 10;
topologyFcn = 'gridtop';
distanceFcn = 'linkdist';
% define net
net1 =
selforgmap(dimensions,coverSteps,
initNeighbor,topologyFcn,distanceFcn);
% train
[net1,Y] = train(net1,P);
% plot 1D-SOM results
% plot input data and SOM weight
positions
plotsompos(net1,P);
grid on
```

```
% Load the data points into
Workspace...
% Assign training inputs and targets
P = Points; % inputs
T = Group; % targets
% Construct a two-input, single-output
perceptron
net = newp (minmax (P), 1);
% Train the perceptron network with
training inputs (p) and targets (t)
net = train (net, P, T);
% Simulate the perceptron network
with same inputs again
a = sim (net, P);
% Querying the perceptron with inputs
it never seen before P9 = [-2; -3];
P10 = [0.5; 4];
a_P9 = sim (net, P9)
a_P10 = sim (net, P10);
```

```
% Initialize a multi-layer network  with
4 hidden, 2 output units and sigmoid
activation functions.
net = newff (minmax (p), [4, 2],
{'tansig', 'logsig'});
```