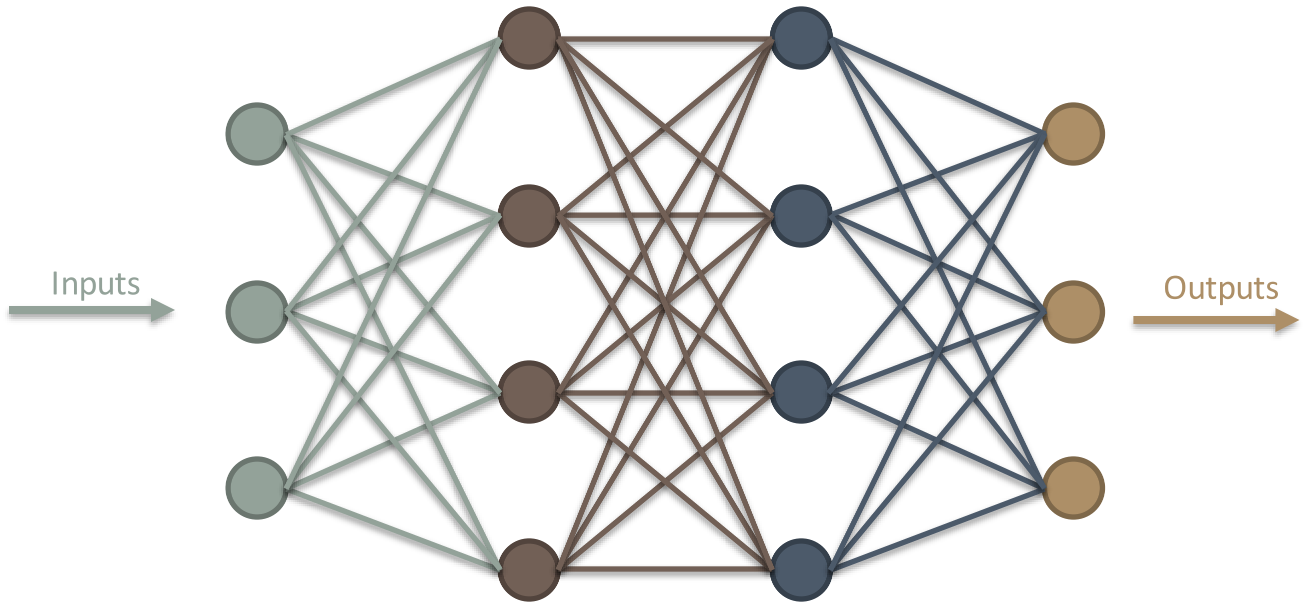
Hopfield Nets and Boltzmann Machine

Jiahui Chen
Department of Mathematical Sciences
University of Arkansas

References: Bhiksha Raj's Lecture and Wikipedia

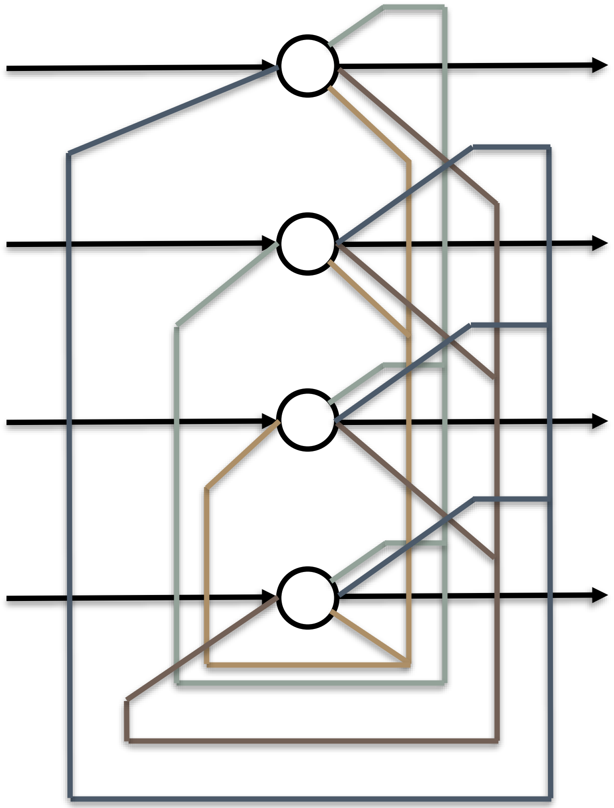
Recap

So for, we meet the network like



Input Layer Hidden Layer Output Layer

But what about

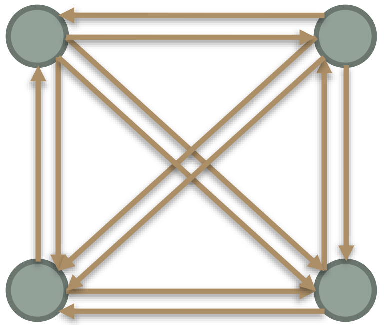


UNIVERSITY OF ARKANSAS

Loopy network

$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \le 0 \end{cases}$$

$$y_i = F\left(\sum_{j \ne i} w_{ij} y_j + b_i\right)$$



The output of a neuron affects the input to the neuron

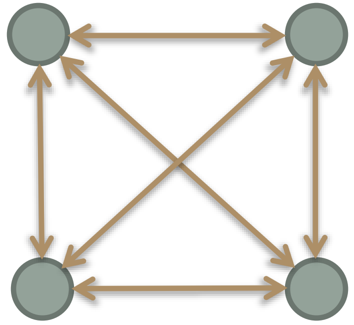
- Each neuron is a perceptron with +1/-1 output
- Every neuron receives input from every other neuron
- Every neuron outputs signals to every other neuron

Hopfield Net

$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

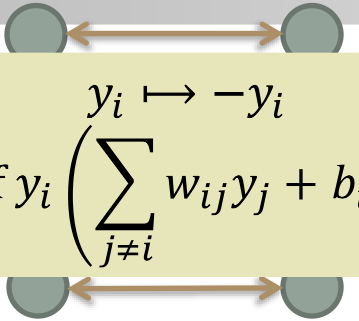
A neuron “flip” if weighted sum of other neurons’ outputs is of the opposite sign



A symmetric network: $w_{ij} = w_{ji}$

- Each neuron is a perceptron with +1/-1 output
- Every neuron receives input from every other neuron
- Every neuron outputs signals to every other neuron

Hopfield Net



$$y_i \longmapsto -y_i$$

$$\text{if } y_i\left(\sum_{j\neq i} w_{ij}y_j + b_i\right)$$

$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

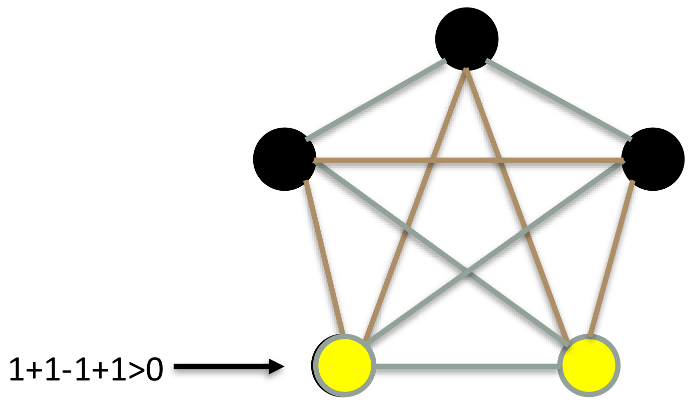
$$y_i = F\left(\sum_{j\neq i} w_{ij}y_j + b_i\right)$$

- At each time each neuron receives a "field"

$$\sum_{j\neq i} w_{ij}y_j + b_i$$

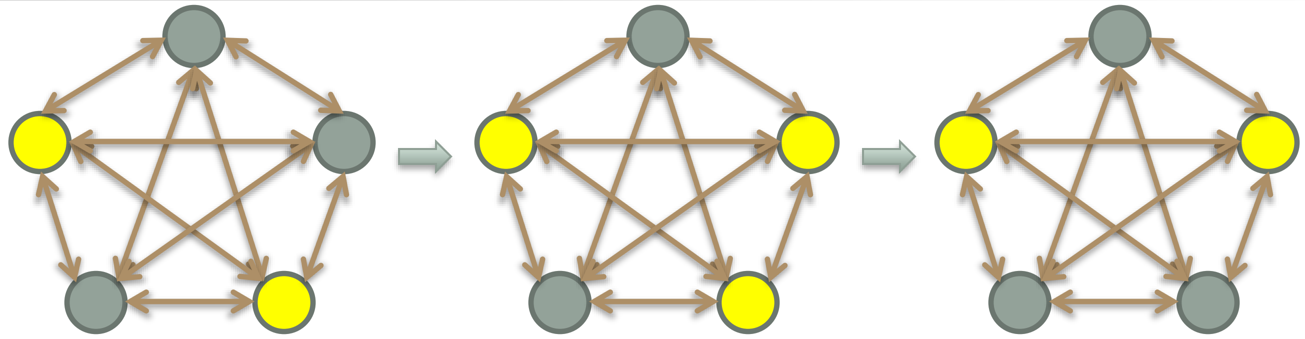
- If the sign of the field matches its own sign, it does not respond
- If the sign of the field opposes its own sign, it "flips" to match the sign of the field

Example



- Red edges are -1, blue edges are +1
- Yellow nodes are +1, black nodes are -1

Hopfield Net



- If the sign of the field at any neuron opposes its own sign, it “flips” to match the field
    - Which will change the field at other nodes
        - Which may then flip
            - …
                - And so on…

Hopfield Net

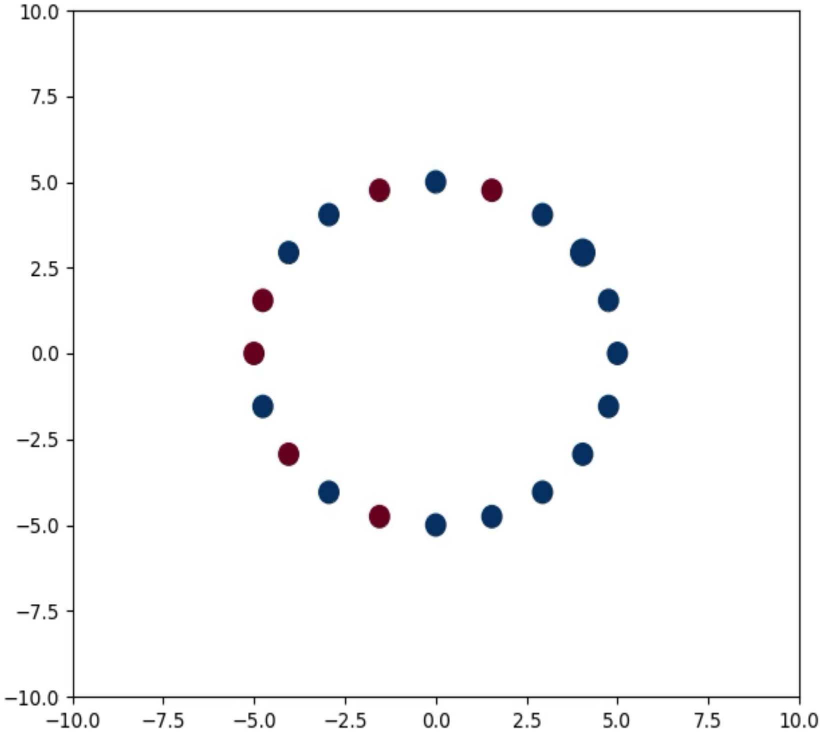
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \le 0 \end{cases}$$

$$y_i = F\left(\sum_{j \ne i} w_{ij} y_j + b_i\right)$$

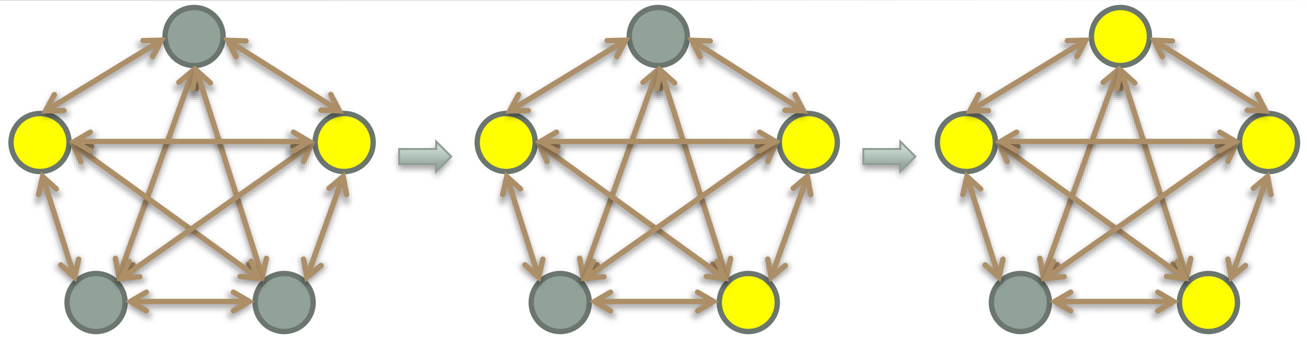
A neuron "flips" if weighted sum of other neuron's outputs is of the opposite sign

But this may cause other neurons to flip!

All neurons which do not "align" with the local field "flip"



Hopfield Net

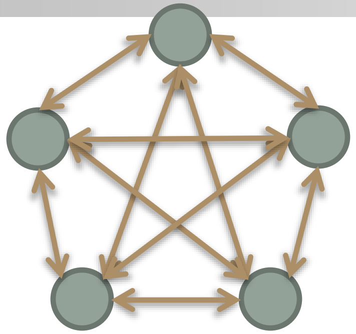


- If the sign of the field at any neuron opposes its own sign, it "flips" to match the field
  - Which will change the field at other nodes
    - Which may then flip
      - …

- Will this continue for ever??

UNIVERSITY OF ARKANSAS

Hopfield Net



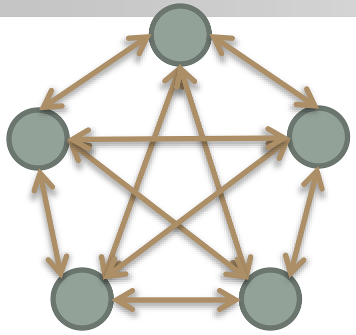
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

- Let y_i^- be the output of the i-th neuron just before it responds to the current field
- Let y_i^+ be the output of the i-th neuron just after it responds to the current field
- If $y_i^- = \text{sign}\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$, then $y_i^+ = y_i^-$
  - If the sign of the field matches its own sign, it does not flip

$$y_i^+\left(\sum_{j \neq i} w_{ij} y_j + b_i\right) - y_i^-\left(\sum_{j \neq i} w_{ij} y_j + b_i\right) = 0$$

Hopfield Net



$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

- If $y_i^- \neq \text{sign}\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$, then $y_i^+ = y_i^-$

$$y_i^+\left(\sum_{j \neq i} w_{ij} y_j + b_i\right) - y_i^-\left(\sum_{j \neq i} w_{ij} y_j + b_i\right) = 2y_i^+\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

This term is always positive!

- Every flip of a neuron is guaranteed to locally increase

$$y_j\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

Globally

- Consider the following sum across all nodes

$$D(y_1, y_2, \ldots, y_N) = \sum_i y_i \left(\sum_{j<i} w_{ij} y_j + b_i \right)$$

$$= \sum_{i,j<i} w_{ij} y_i y_j + \sum_i b_i y_i$$

- For any unit k that "flips" because of the local field

$$\Delta D(y_k) = D(y_1, \ldots, y_k^+, \ldots, y_N) - D(y_1, \ldots, y_k^-, \ldots, y_N)$$

- Expanding

$$\Delta D(y_k) = (y_k^+ - y_k^-) \sum_{j \neq i} w_{ij} y_j + (y_k^+ - y_k^-) b_k$$

- This is positive!
- Every flip of a unit results in an increase in D

Hopfield Net

- Flipping a neuron will result in an increase (non-decrease) of

$$D = \sum_{i,j<i} w_{ij} y_i y_j + \sum_i b_i y_i$$

- D is bounded, $D_{\max} = \sum_{i,j<i} |w_{ij}| + \sum_i |b_i|$
- The minimum increment of D in a flip is

$$\Delta D_{\min} = \min_{i,\{y_i, i=1,\dots,N\}} 2 \left| \sum_{j<i} w_{ij} y_j + b_i \right|$$

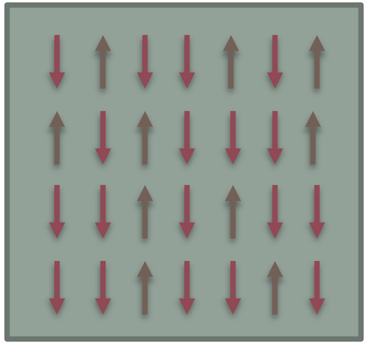
- Any sequence of flips must converge in a finite number of steps
- Define the energy of the Hopfield network as

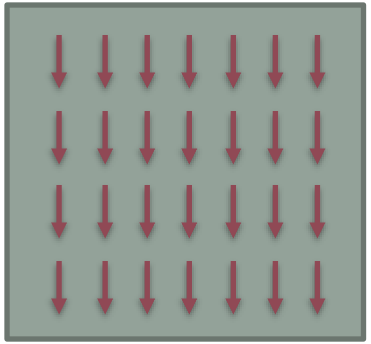
$$E = -D = -\sum_{i,j<i} w_{ij} y_i y_j - \sum_i b_i y_i$$

- The evolution of a Hopfield network constantly decreases its energy
- Where did this “energy” concept suddenly sprout from?

Analogy: Spin Glass

Impure Magnet

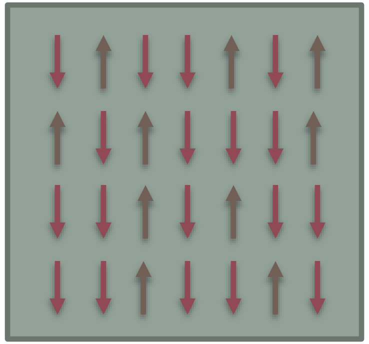




Pure Magnet

- Magnetic diploes in a disordered magnetic material
- Each dipole tries to align itself to the local field
  - In doing so it may flip
- This will change fields at other dipoles
  - Which may flip
- Which changes the field at the current dipole…

Analogy: Spin Glass



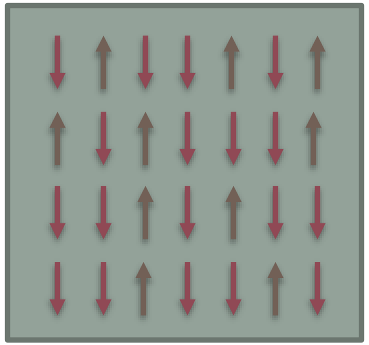
Total field at current dipole:

$$f(x_i) = \sum_{j \neq i} J_{ij} x_j + b_i$$

intrinsic external

- x_i is vector position of i-th dipole
- The field at any dipole is the sum of the field contributions of all other dipoles
- The contribution of a dipole to the field at any point depends on interaction J
  - Derived from the "Ising" model for magnetic materials (Ising and Lenz, 1924)

Analogy: Spin Glass



Total field at current dipole:

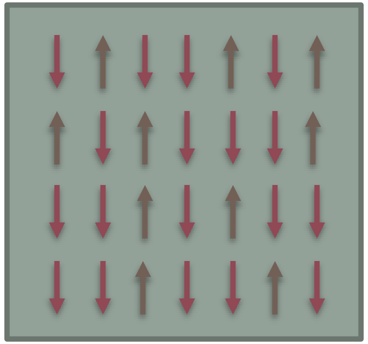
$$f(x_i) = \sum_{j \neq i} J_{ij} x_j + b_i$$

Response of current dipole

$$x_i = \begin{cases} x_i & \text{if } \operatorname{sign}\big(x_i f(x_i)\big) = 1 \\ -x_i & \text{otherwise} \end{cases}$$

- A dipole flips if it is misaligned with the field in its location
- Dipoles will keep flipping
  - A flipped dipole changes the field at other dipoles
    - Some of which will flip
  - Which will change the field at the current dipole
    - Which may flip
  - Etc….
- When will it stop???

Analogy: Spin Glass



Total field at current dipole:

$$f(x_i) = \sum_{j \neq i} J_{ij} x_j + b_i$$

Response of current dipole

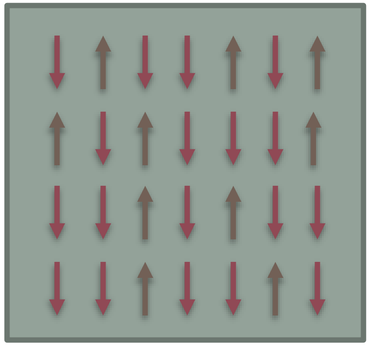
$$x_i = \begin{cases} x_i & \text{if } \operatorname{sign}\big(x_i f(x_i)\big) = 1 \\ -x_i & \text{otherwise} \end{cases}$$

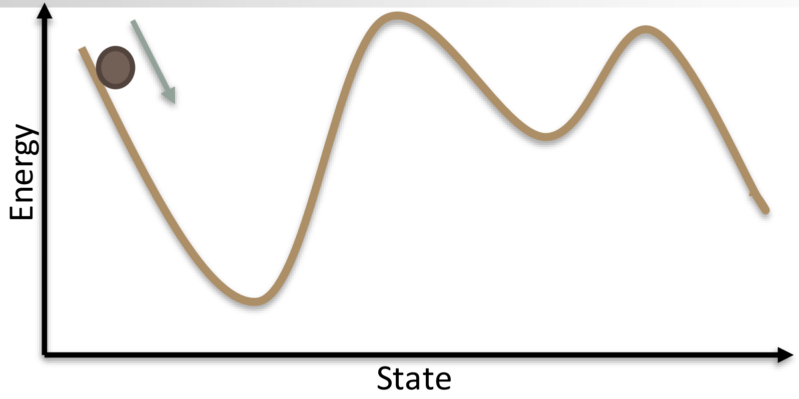
- The "Hamiltonian" (total energy) of the system

$$E = -\frac{1}{2} \sum_i x_i f(x_i) = -\frac{1}{2} \sum_i \left(\sum_{j \neq i} J_{ij} x_j + b_i \right)$$

- The system evolves to minimize the energy
  - Dipoles stop flipping is any flips result in increase of energy

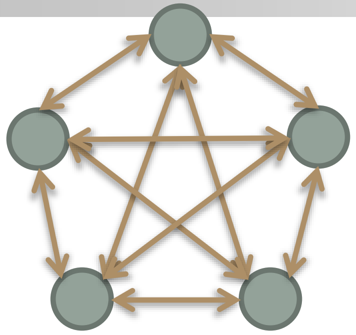
Analogy: Spin Glass





- The system stops at one of its stable configurations
  - Where energy is a local minimum
- Any small jitter from this stable configuration returns it to the stable configuration
  - i.e. the system remembers its stable state an returns to it

Hopfield Net



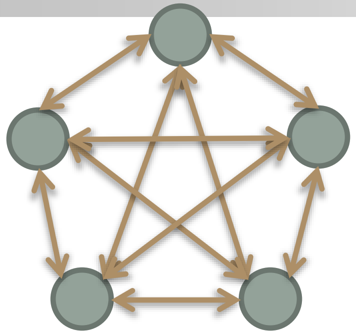
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \le 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

$$E = -\frac{1}{2} \sum_i y_i \left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

- This is analogous to the potential energy of a spin glass
  - The system will evolve until the energy hits a local minimum
- Typically will not utilize bias: the bias is similar to having a single extra neuron that is pegged to 1.0
- Removing the bias term does not affect the rest of the discussion in any manner, but it will be back later.

Hopfield Net



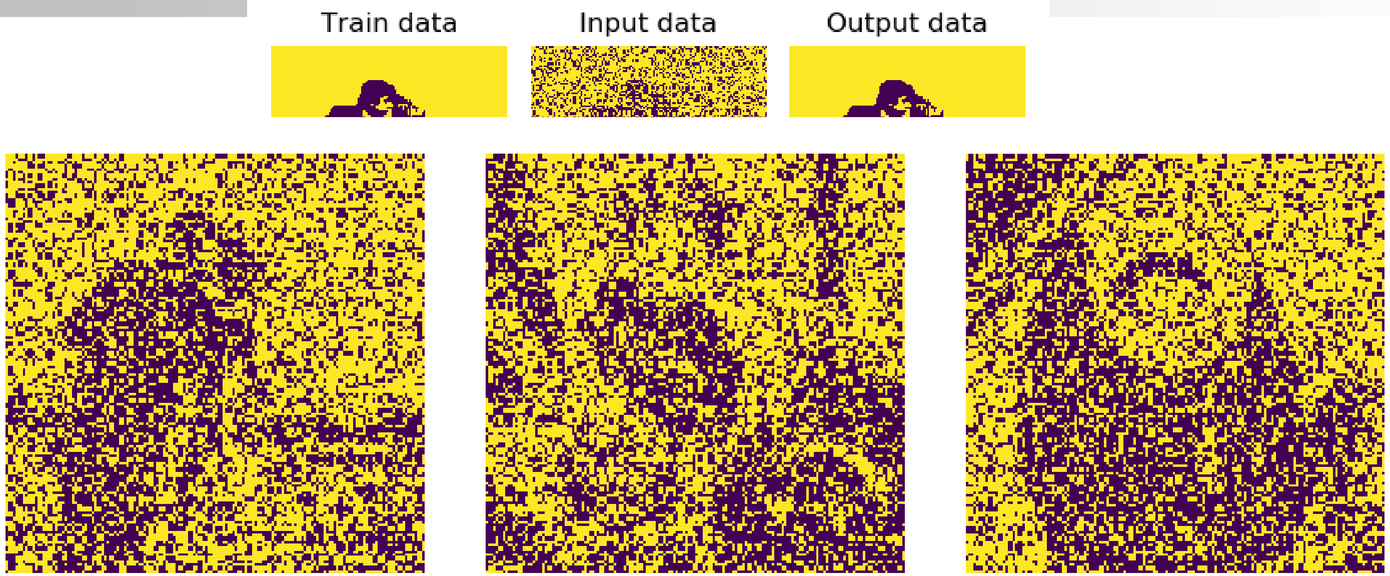
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

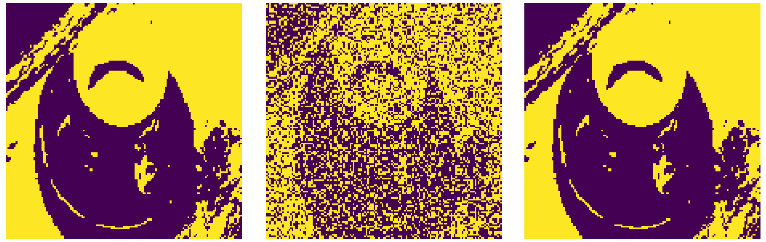
$$E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} y_i y_j$$

- This is analogous to the potential energy of a spin glass
  - The system will evolve until the energy hits a local minimum
- Each of the minima is a “stored” pattern
  - If the network is initialized close to a stored pattern, it will inevitably evolve to the pattern
- This is a content addressable memory
  - Recall memory content from partial or corrupt values
- Also called associative memory

Examples: Content addressable memory



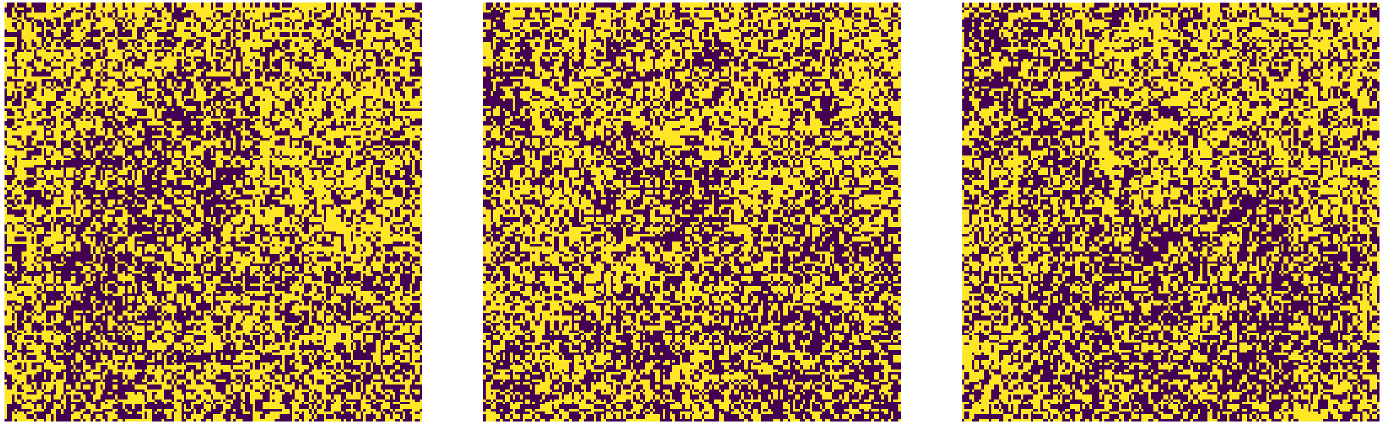
coffee



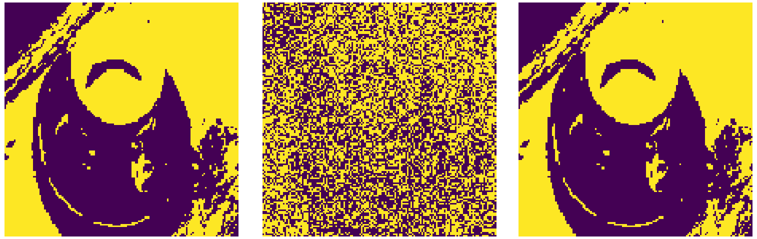
[images from skimage]

Examples: Content addressable memory

Train data
Input data
Output data



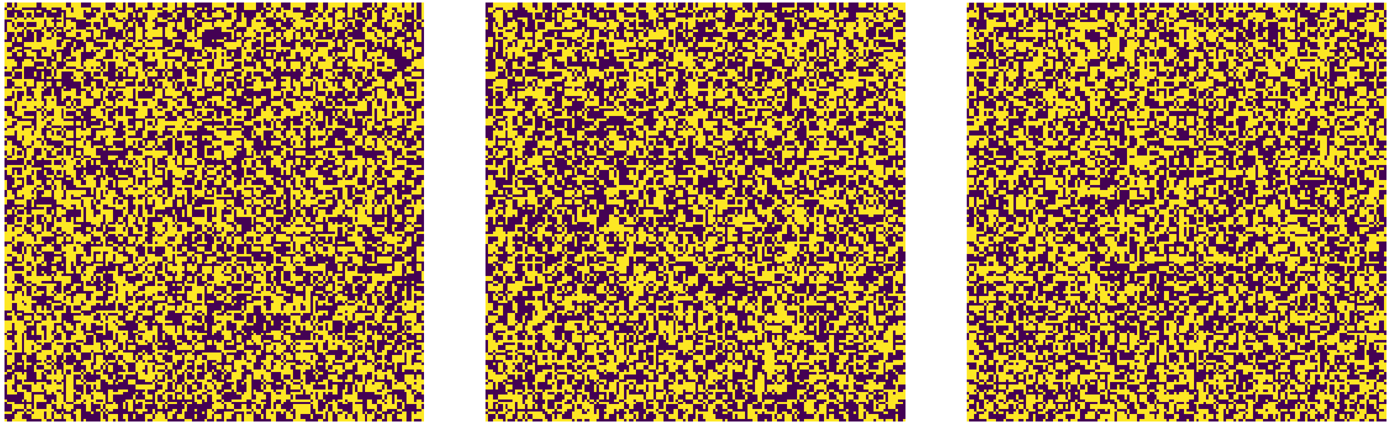
coffee



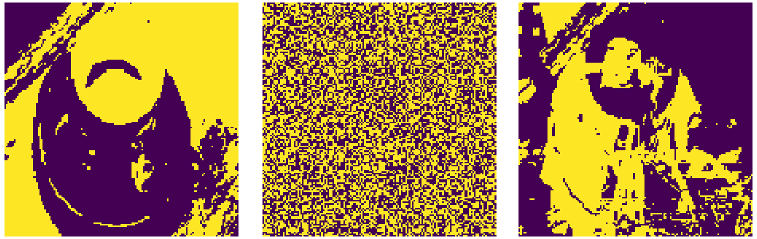
[images from skimage]

Examples: Content addressable memory

Train data
Input data
Output data

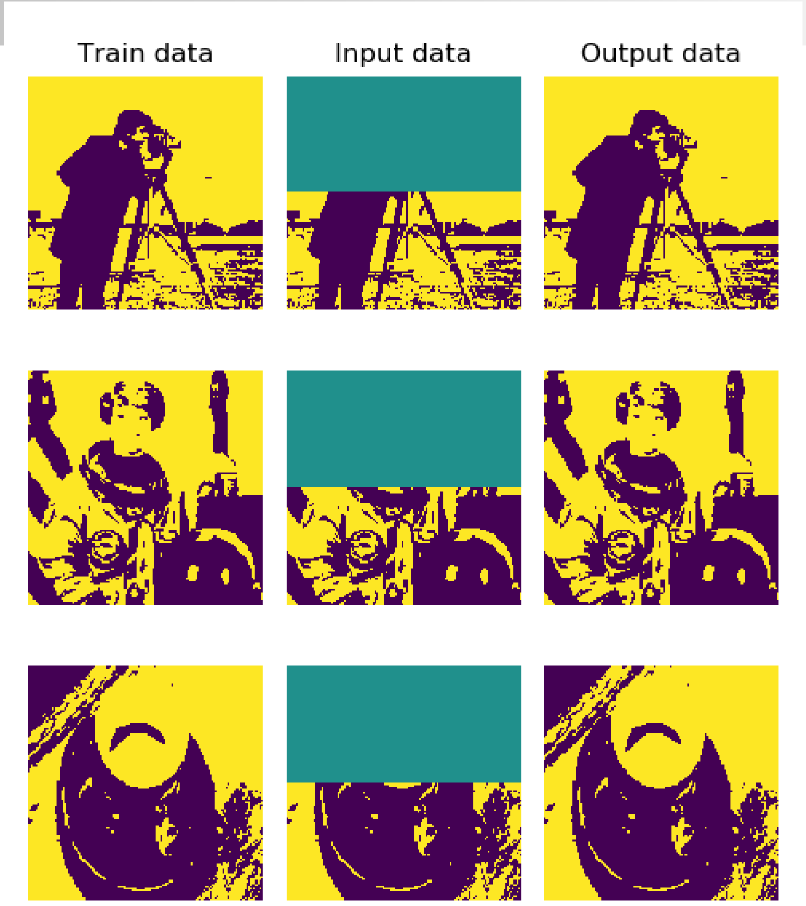


coffee



[images from skimage]

Examples: Content addressable memory



camera

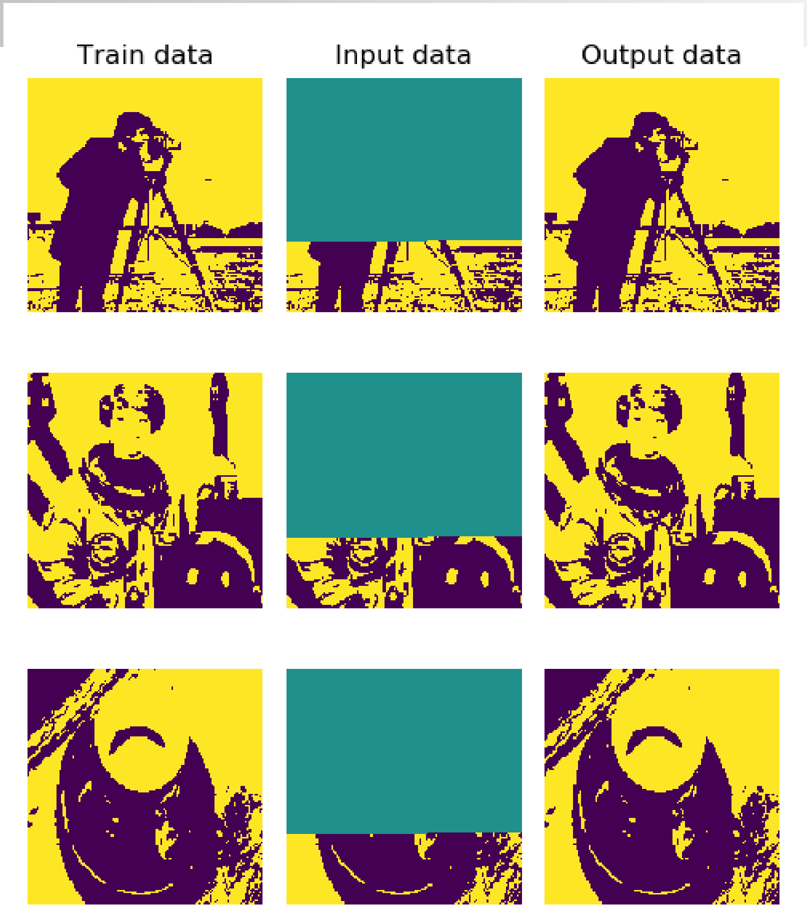
astronaut

coffee

Percentage: 50%

[images from skimage]

Examples: Content addressable memory



camera

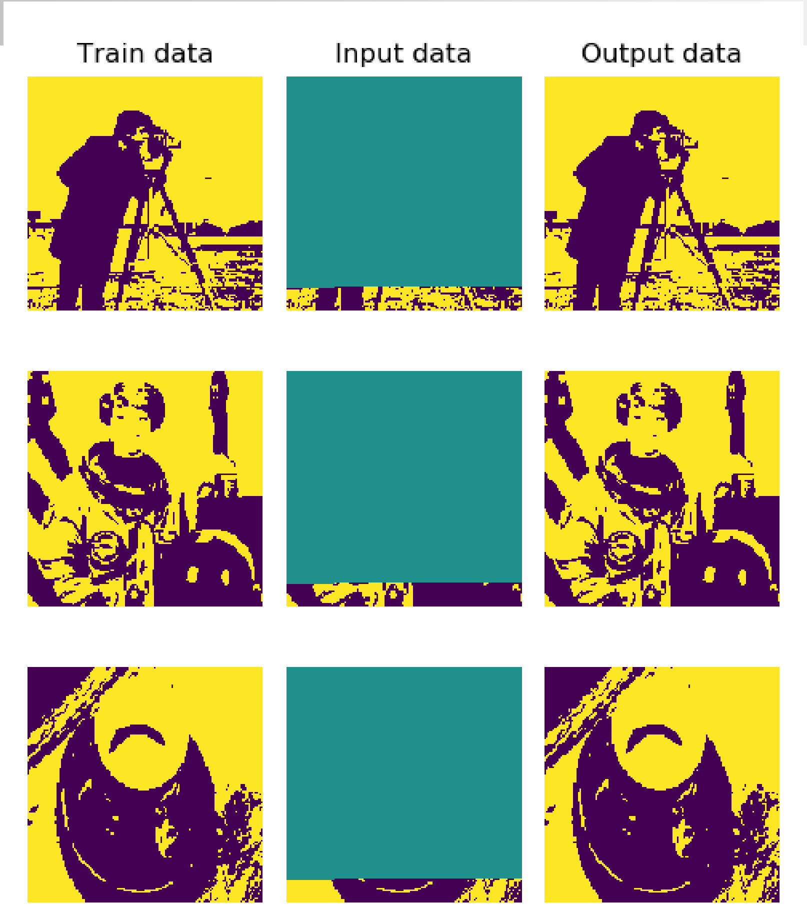
astronaut

coffee

Percentage: 70%

[images from skimage]

Examples: Content addressable memory



camera

astronaut

coffee

Percentage: 90%

[images from skimage]

UNIVERSITY OF ARKANSAS

Computational algorithm

1. Initialize network with initial pattern

$$y_i(0) = x_i, \qquad 0 \le i \le N-1$$

2. Iterate until convergence

$$y_i(t+1) = F\left(\sum_{j \neq i} w_{ij} y_j\right), 0 \le i \le N-1$$

- Very simple
- Updates can be done sequentially, or all at once
- Convergence

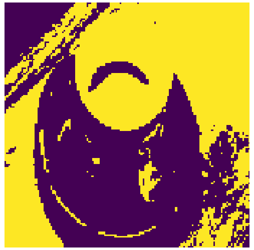
$$E = -\frac{1}{2}\sum_{i}\sum_{j \neq i} w_{ij} y_i y_j$$

Issues

- How do we make the network store a specific pattern or set of patterns?
- How many patterns can we store?
- How to “retrieve” patterns better?

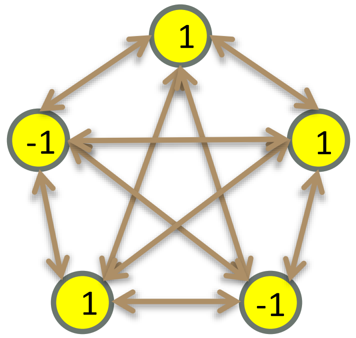
How to remember a specific pattern?

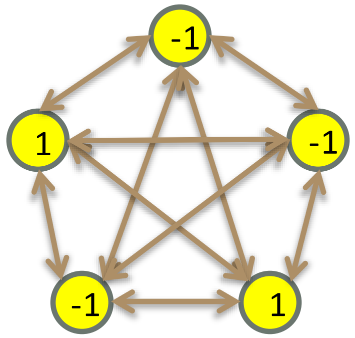
- How to teach a network to "remember" these images



- For an image with N pixels we need a network with N neurons
- Every neuron connects to every other neuron
- Weights are symmetric (not mandatory)
- N(N-1)/2 weights in all

Storing patterns: training a network



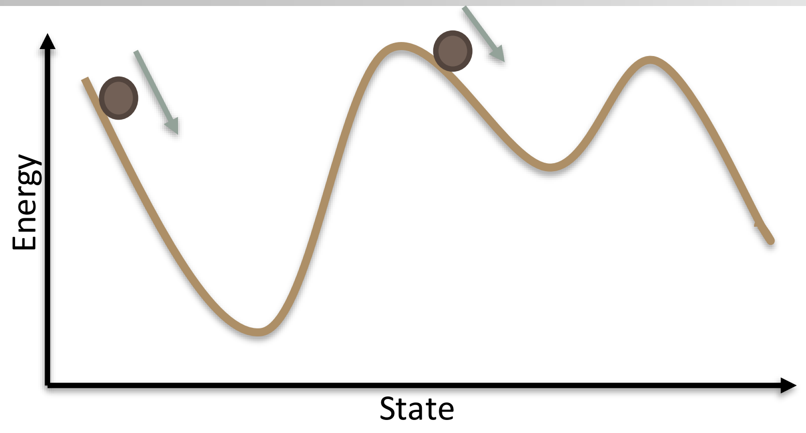


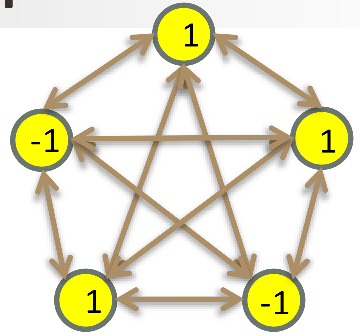
- A network that stores pattern P also naturally stores –P
  - Symmetry E(P) = E(-P) since E is a function as following

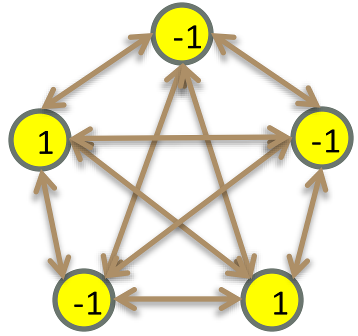
$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}y_iy_j$$

UNIVERSITY OF ARKANSAS

A network can store multiple patterns







- Every stable point is a stored pattern
- So we could design the net to store multiple patterns
  - Remember that every stored pattern P is actually two stored patterns, P and -P

Storing a pattern

- Design $\{w_{ij}\}$ such that the energy is a local minimum at the desired $P = \{y_i\}$
- Storing 1 pattern: $\mathrm{sign}\left(\sum_{j\neq i} w_{ij} y_j\right) = y_i \quad \forall i$
- This is a stationary pattern
- Hebb law (also called as Hebbian learning): $w_{ij} = y_i y_j$

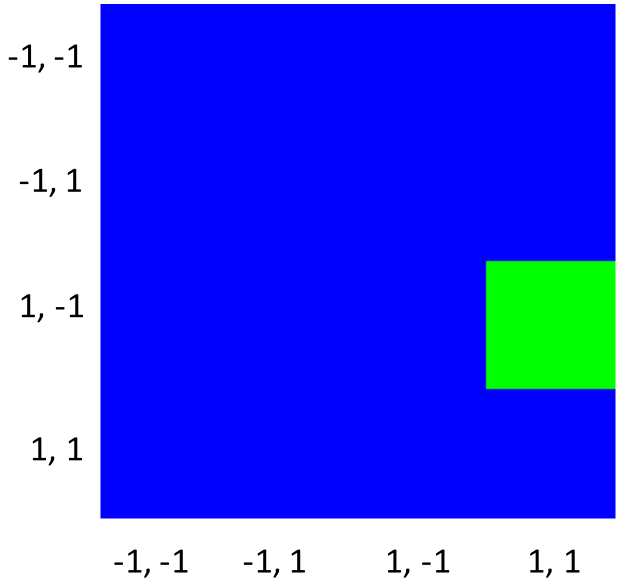
$$\begin{aligned}\mathrm{sign}\left(\sum_{j\neq i} w_{ij} y_j\right) &= \mathrm{sign}\left(\sum_{j\neq i} y_i y_j y_j\right)\\ &= \mathrm{sign}\left(\sum_{j\neq i} y_i y_j^2\right)\\ &= \mathrm{sign}(y_i)\\ &= y_i\end{aligned}$$

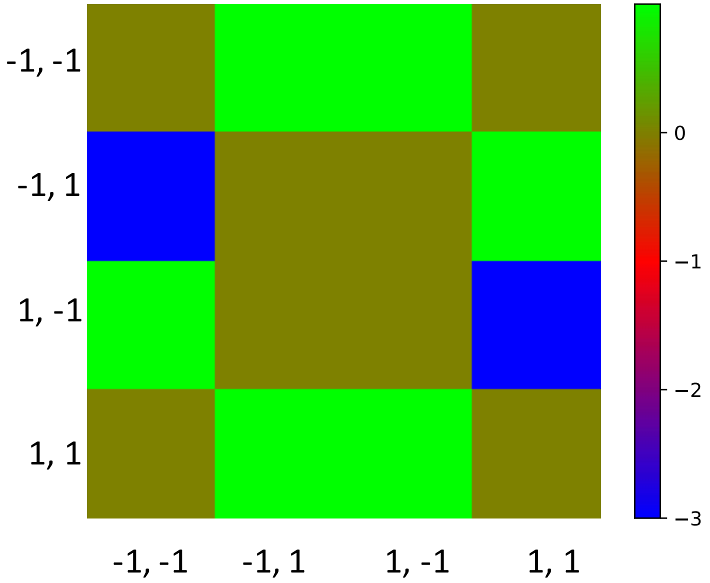
Storing a pattern

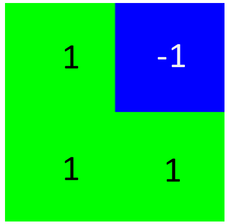
- By Hebb law, the pattern is stationary
- The lowest possible energy value for the network

$$
\begin{aligned}
E &= -\sum_{i}\sum_{j\neq i} w_{ij} y_i y_j \\
&= -\sum_{i}\sum_{j\neq i} y_i^2 y_j^2 \\
&= -0.5N(N-1)
\end{aligned}
$$

Hebbian learning: Storing a 4-bit pattern

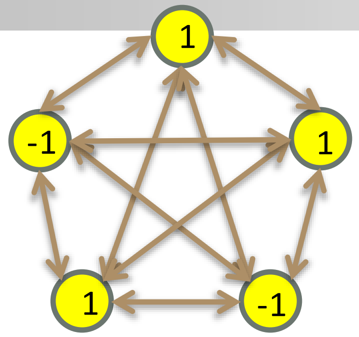


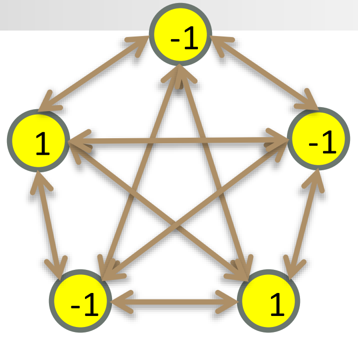




- Left: Pattern stored; right: energy map
- Stored pattern has lowest energy
- Gradation of energy ensures stored pattern (or its ghost) is recalled from everywhere

Storing multiple patterns



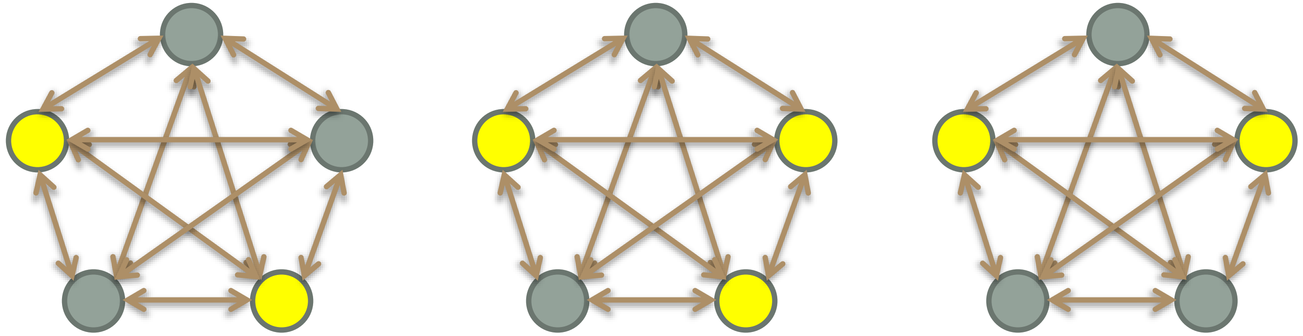


- To store more than one pattern

$$w_{ij} = \sum_{y^p \in \mathbf{Y}} y_i^p y_j^p$$

- $\mathbf{Y} = \{y^p\}$ is the set of patterns to store
- Super/subscript p represents the specific pattern

How many patterns can we store?



- Hopfield: For a network of N neurons can store up to ~0.14N patterns through Hebbian learning
  - Provided they are “far” enough
- Where did this number come from?

The limits of Hebbian Learning

- Consider the following: we must store K N-bit patterns of the form

$$\mathbf{y}_k = \left[y_1^k, y_2^k, y_3^k, \dots, y_N^k\right], \qquad k = 1, \dots, K$$

- Hebbian learning (scaling by 1/N for normalization, this does not affect actual pattern storage):

$$w_{ij} = \frac{1}{N}\sum_{y^p \in \mathbf{Y}} y_i^p y_j^p$$

- For any pattern y_p to be stable:

$$y_i^p \frac{1}{N}\sum_j w_{ij} y_j^p > 0 \ \forall i$$

$$y_i^p \frac{1}{N}\sum_j \sum_k y_i^k y_j^k \, y_j^p > 0 \ \forall i$$

The limits of Hebbian Learning

- For any pattern $\mathbf{y}_p$ to be stable:

$$y_i^p \frac{1}{N} \sum_j \sum_k y_i^k y_j^k \, y_j^p > 0 \;\; \forall i$$

$$y_i^p \frac{1}{N} \sum_j y_i^p y_j^p y_j^p + y_i^p \frac{1}{N} \sum_j \sum_k y_i^k y_j^k \, y_j^p > 0 \;\; \forall i$$

- Note that the first term equals 1 (because $y_j^p y_j^p = y_i^p y_i^p = 1$)
  - For $\mathbf{y}_p$ to be stable the requirement is that the second crosstalk term:

$$y_i^p \frac{1}{N} \sum_j \sum_{k \neq p} y_i^k y_j^k \, y_j^p > (-1) \;\; \forall i$$

- The pattern will fail to be stored if the crosstalk

$$y_i^p \frac{1}{N} \sum_j \sum_{k \neq p} y_i^k y_j^k \, y_j^p \leq (-1) \;\; \forall i$$

The limits of Hebbian Learning

- For any random set of K patterns to be stored the probability of the following must be low

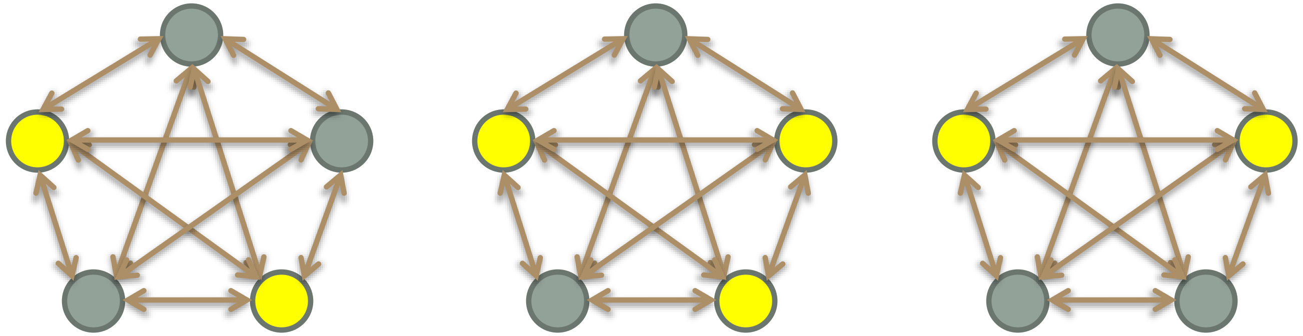
$$C_i^p = y_i^p \frac{1}{N} \sum_j \sum_{k \neq p} y_i^k y_j^k \, y_j^p \leq (-1)$$

- For large N and K the probability distribution of C_i^p approaches a Gaussian with 0 mean, and variance K/N
  - Considering that individual bits $y_i^l \in \{-1, +1\}$ and have variance 1
- For a Gaussian, $C \sim N(0, K/N)$

$$P\left(C < -1 \middle| \mu = 0, \sigma^2 = \frac{K}{N}\right) < 0.004 \text{ for } K/N < 0.14$$

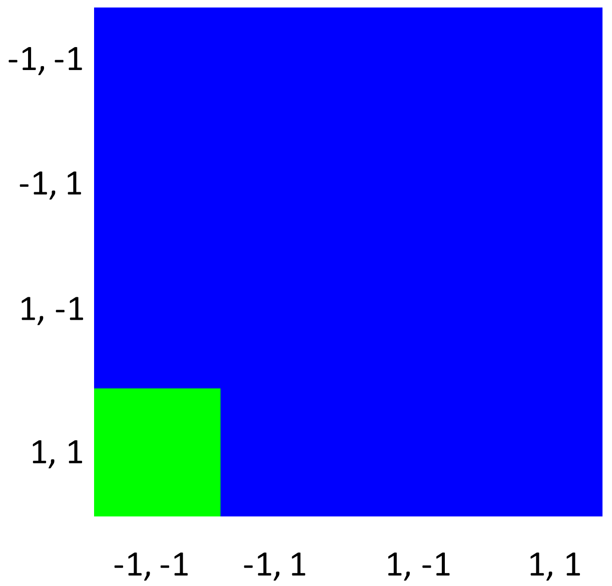
- To have less than 0.4% probability that stored patterns will not be stable, K < 0.14N

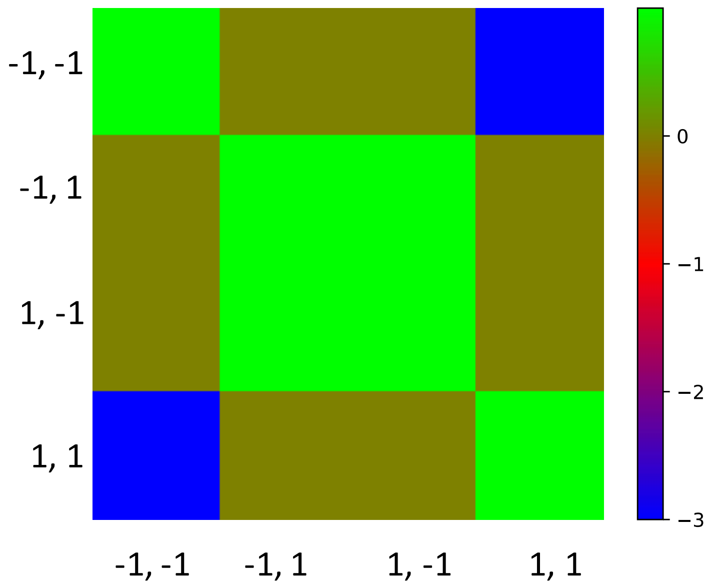
How many patterns can we store?

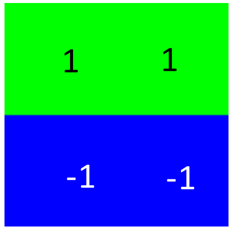


- A network of N neurons trained by Hebbian learning can store up to ~0.14N patterns with low probability of error
  - Computed assuming prob(bit=1)=0.5
    - On average # of matched bits in any pair = # of mismatched bits
    - Patterns are “orthogonal”-maximally distant-from one another
  - Expected behavior for non-orthogonal patterns?
- To get some insight into what is stored, lets see some examples

Hebbian learning: Storing a 4-bit pattern







- Left: Pattern stored; right: energy map
- Stored pattern has lowest energy
- Gradation of energy ensures stored pattern (or its ghost) is recalled from everywhere

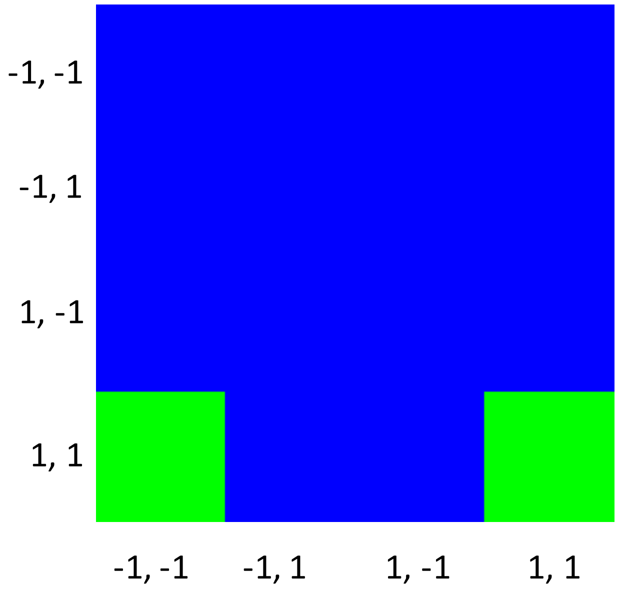
Storing multiple patterns: Orthogonality

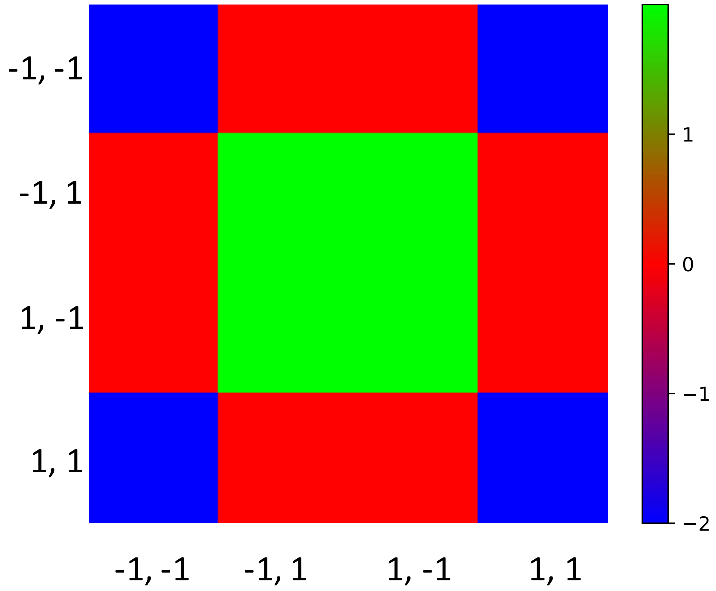
- The maximum Hamming distance between two N-bit patterns is N/2
  - Because any pattern Y = -Y for our purpose

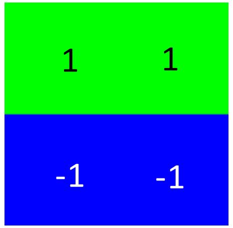
- Two patterns y_1 and y_2 that differ in N/2 bits are 2^M orthogonal
  - Because $y_1^T y_2 = 0$

- For $N = 2^M L$, where L is an odd number, there are at most orthogonal binary patterns
  - Others may be almost orthogonal

Two orthogonal 4-bit pattern

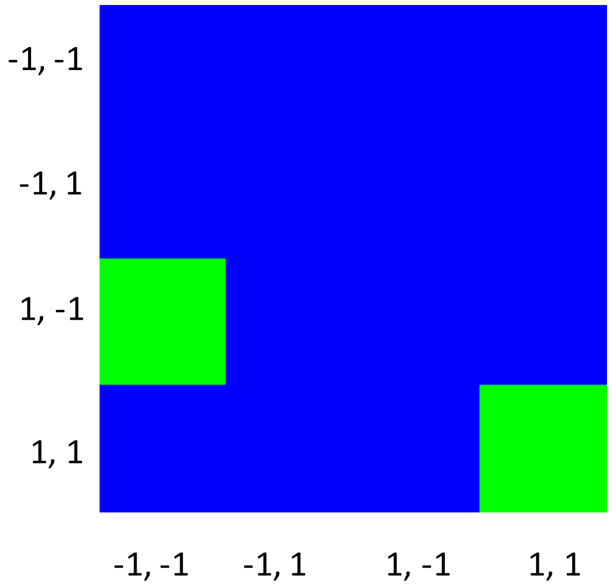


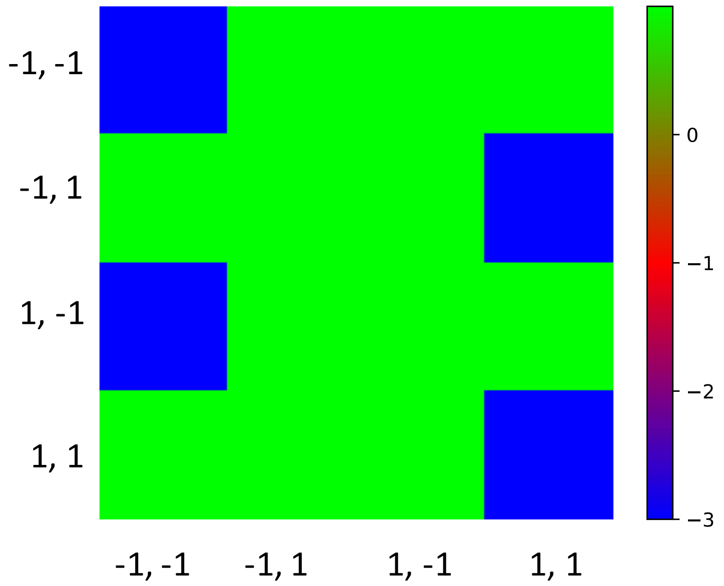


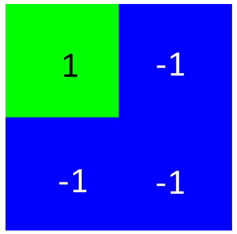


- Patterns are local minima (stationary and stable)
    - No other local minima exit
    - But patterns perfectly confusable for recall

Two non-orthogonal 4-bit pattern

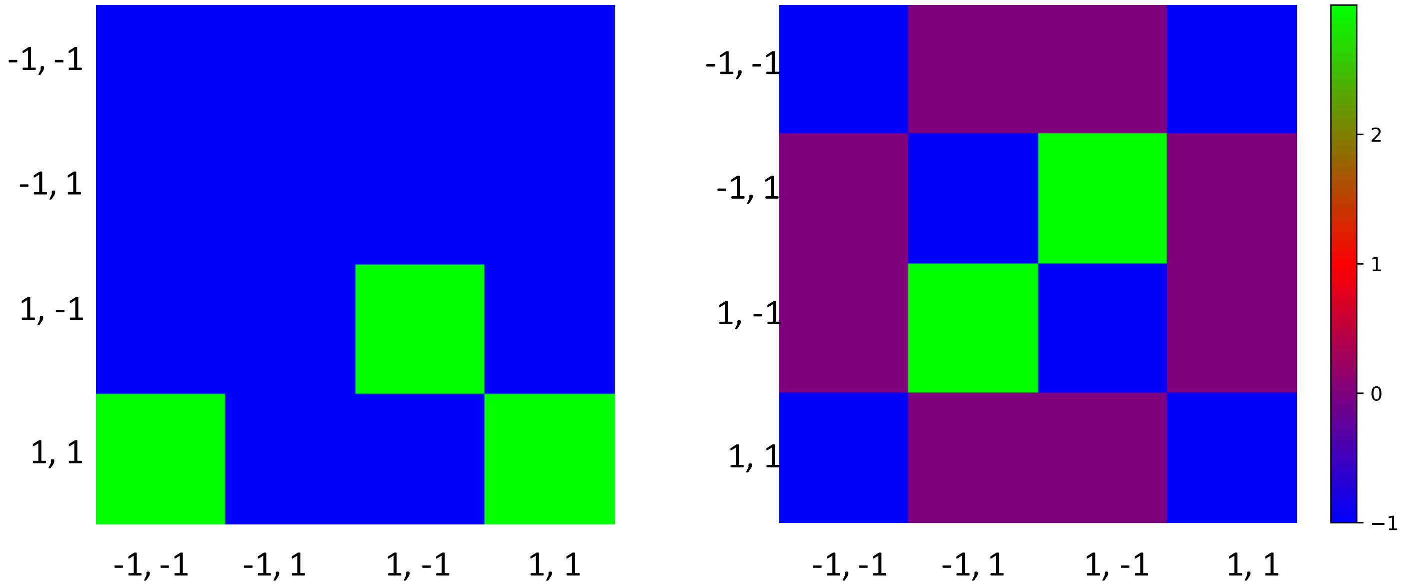






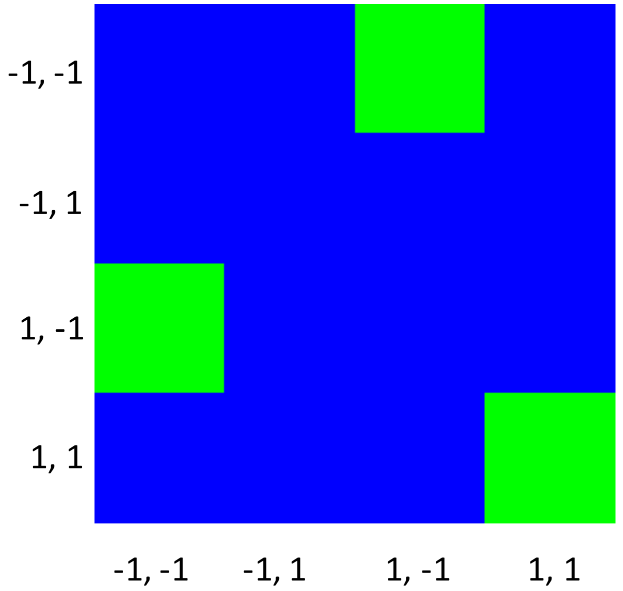
- Patterns are local minima (stationary and stable)
  - No other local minima exit
  - Patterns may be perfectly recalled
  - Note K > 0.14N

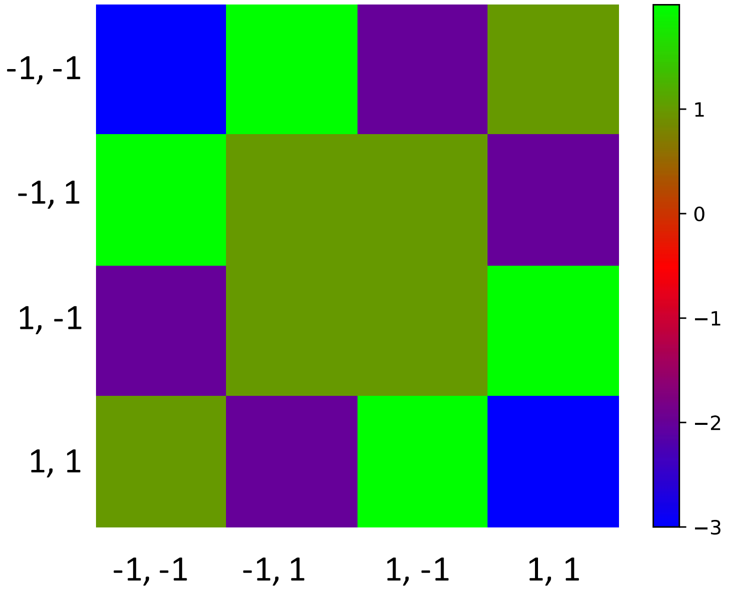
Three orthogonal 4-bit pattern



All patterns are local minima (stationary and stable)
But recall from perturbed patterns is random

Three non-orthogonal 4-bit pattern





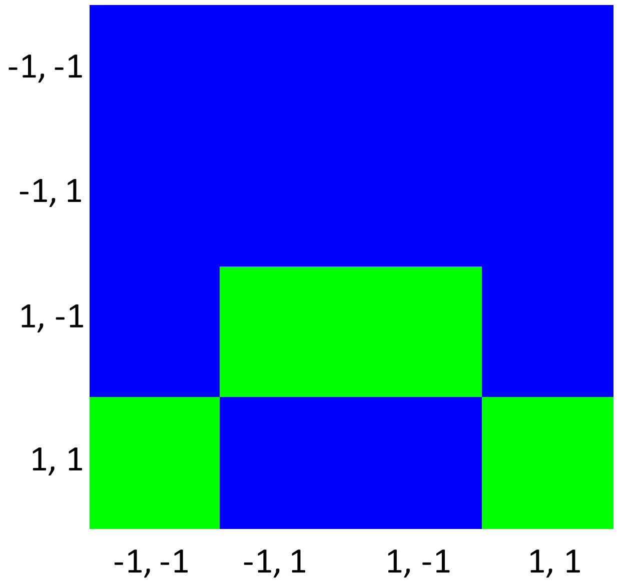
All patterns are local minima and recalled

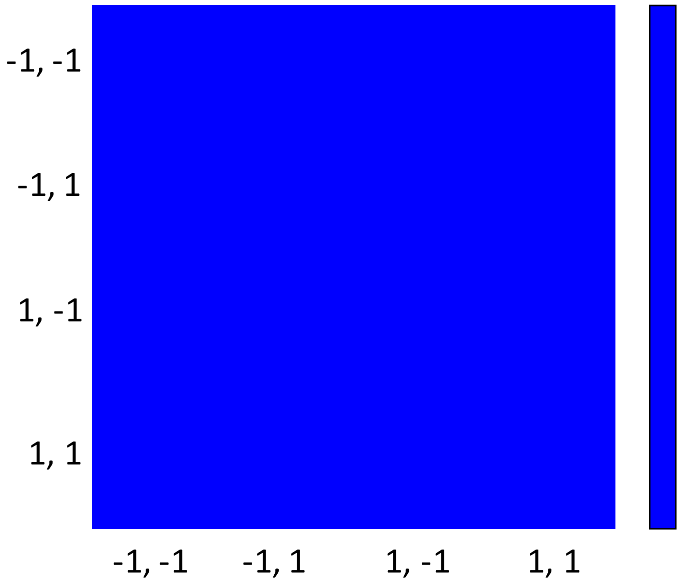
Note K>0.14N

Note some “ghosts” ended up in the “well” of other patterns

So one of the patterns has stronger recall than the other two

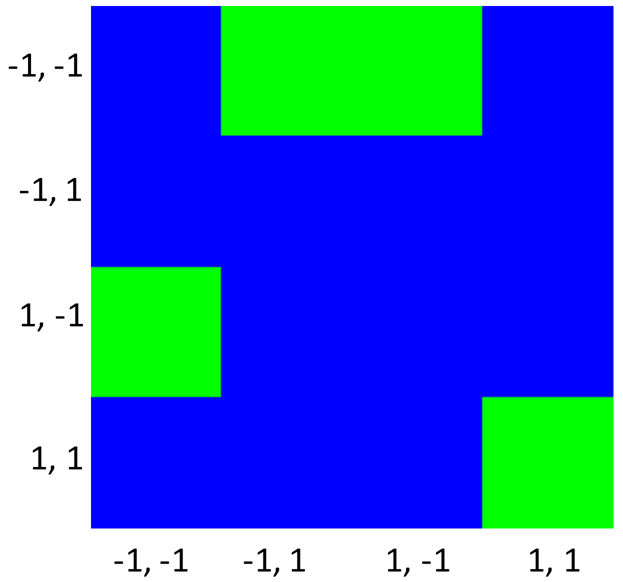
Four orthogonal 4-bit pattern

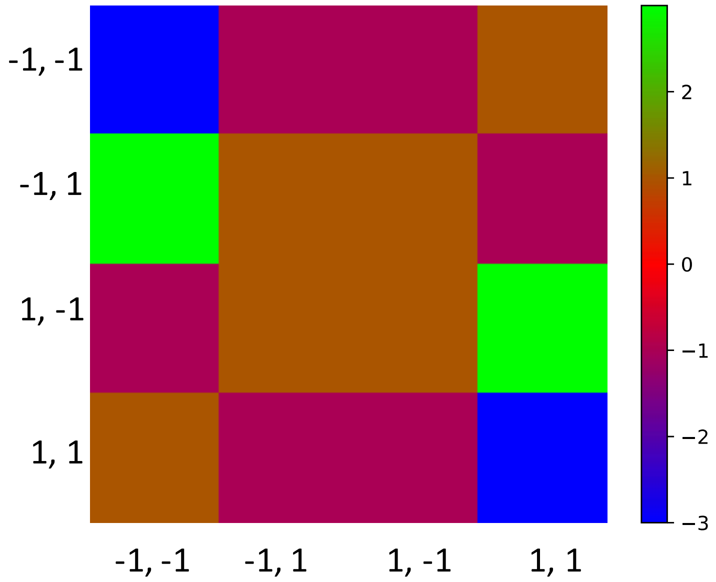




All patterns are stationary, but none are stable
Total wipe out

Four non-orthogonal 4-bit pattern

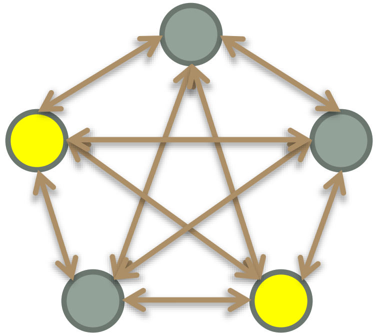


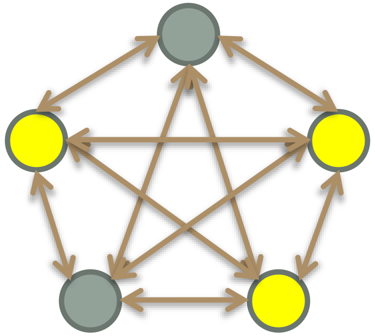


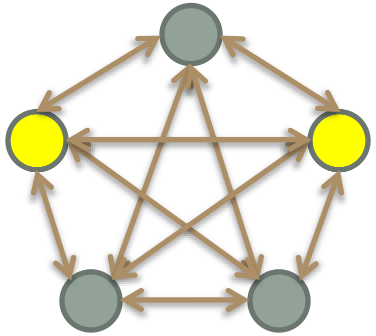
One stable pattern, rest are not stationary

“Collisions” when the ghost of one pattern occurs next to another

How many patterns can we store?





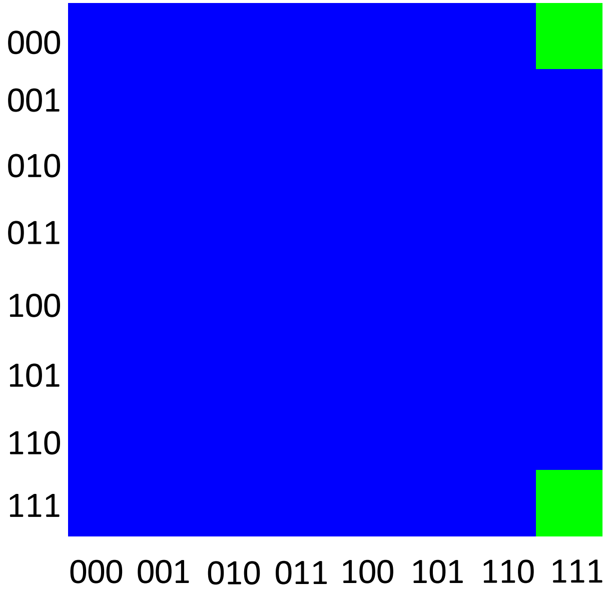


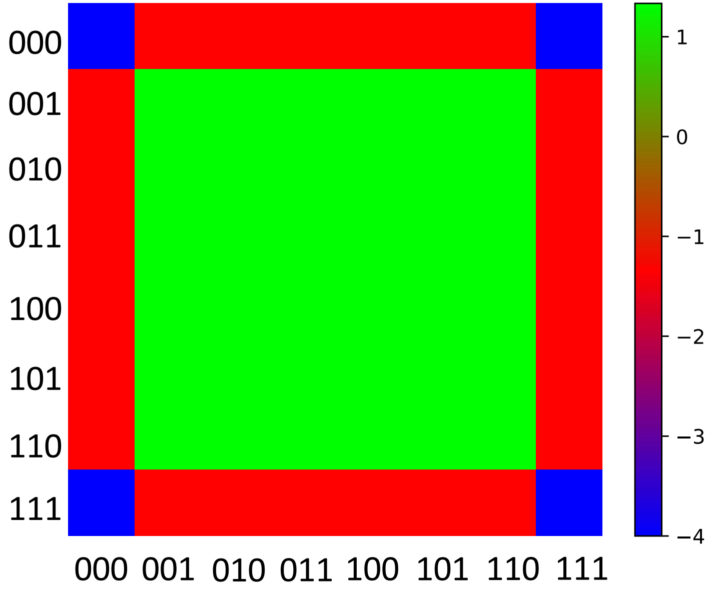
- Hopfield: For a network of N neurons can store up to 0.14N patterns

- Apparently a fuzzy statement
  - What does it really mean to say “stores” 0.14N patterns?
    - Stationary? Stable? No other local minima?

- N=4 may not be a good care (N too small)

Two orthogonal 6-bit pattern



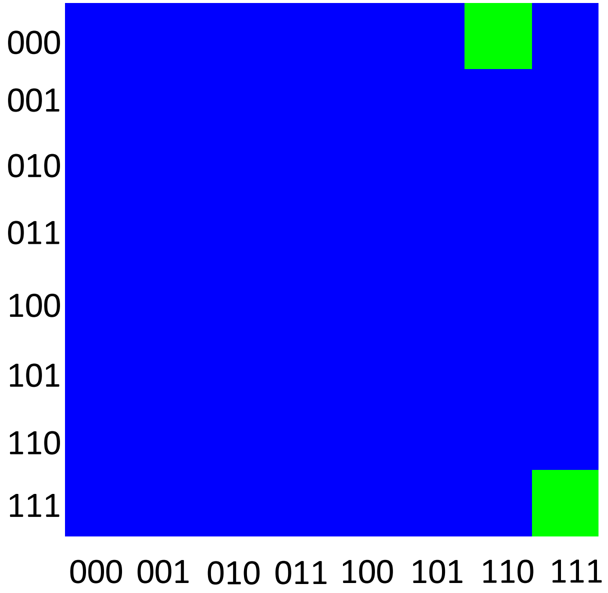


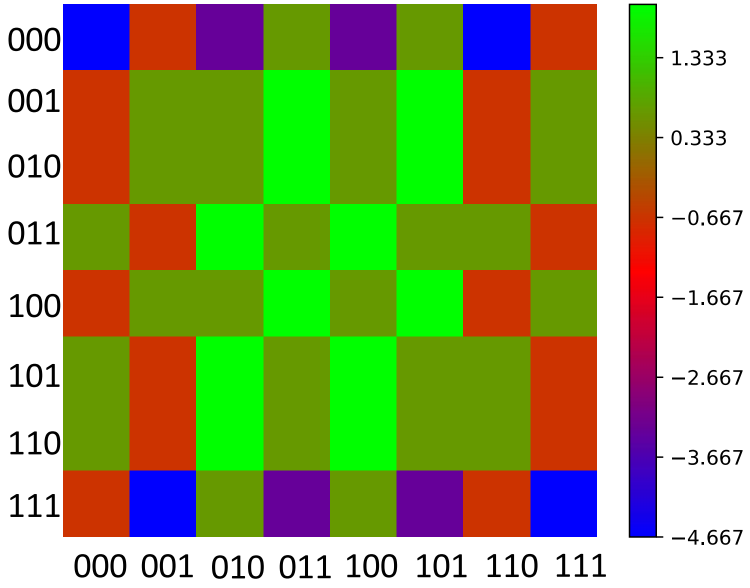
Perfectly stationary and stable

Several spurious "fake-memory" local minima..

Figure over-states the problem: actually a 3-D Kmap

Two non-orthogonal 6-bit pattern



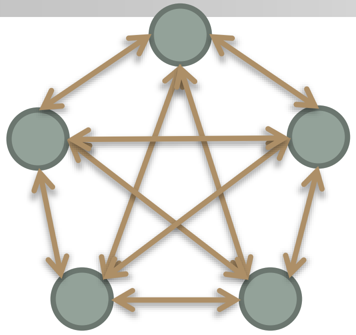


Perfectly stationary and stable
Several spurious "fake-memory" local minima..
Figure over-states the problem: actually a 3-D Kmap

Observations

- Many "parasitic" patterns
  - Undesired patterns that also become stable or attractors
- Apparently a capacity to store more than 0.14N patterns
- **Parasitic patterns** can occur because sums of odd numbers of stored patterns are also stable for Hebbian learning
- They are also from other random local energy minima from the weights matrices themselves
- Seems possible to store K>0.14N patterns
  - Obtain a weight matrix W such that K>0.14N patterns are stationary
  - Possible to make more than 0.14N patterns at-least 1-bit stable
- Patterns that are non-orthogonal easier to remember
  - Patterns that are closer are easier to remember than patterns that are farther!!
- Can we attempt to get greater control on the process that Hebbian learning give us?

Review: Hopfield Net

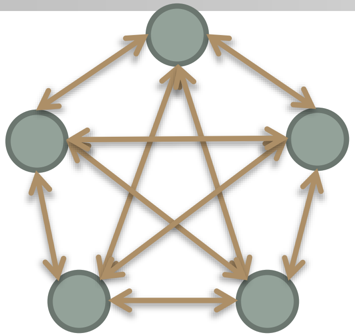


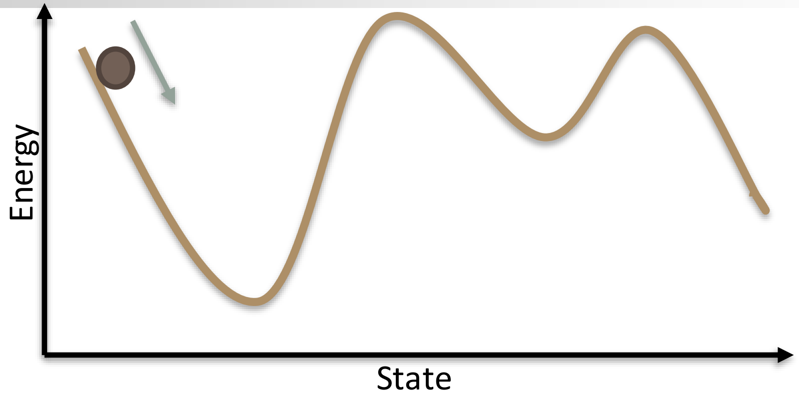
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

- If the sign of the field at any neuron opposes its own sign, it “flips” to match the field
    - Which will change the field at other nodes
        - Which may then flip
            - …
                - And so on…

Review: Evolution





- This is analogous to the potential energy of a spin glass
    - The system will evolve until the energy hits a local minimum
- Each of the minima is a “stored” pattern
    - If the network is initialized close to a stored pattern, it will inevitably evolve to the pattern
- This is a content addressable memory
    - Recall memory content from partial or corrupt values
- Also called associative memory

UNIVERSITY OF ARKANSAS

Review: Computational algorithm

1. Initialize network with initial pattern

$$y_i(0) = x_i, \qquad 0 \le i \le N-1$$

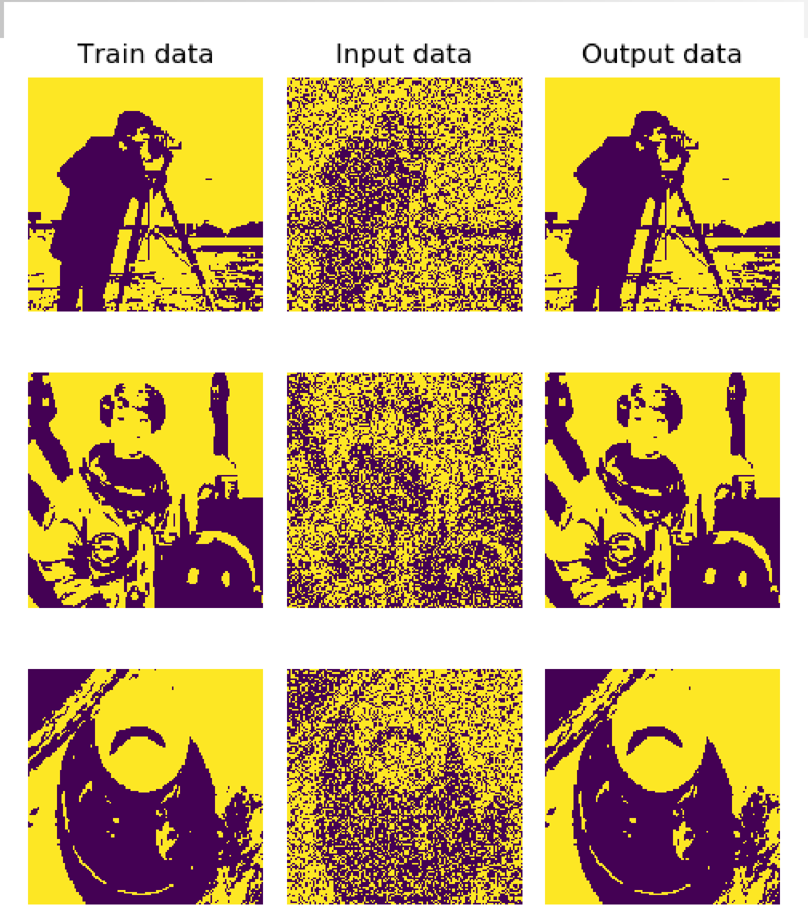
2. Iterate until convergence

$$y_i(t+1) = F\left(\sum_{j \neq i} w_{ij} y_j\right), 0 \le i \le N-1$$

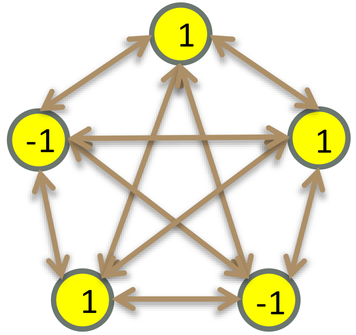
- Very simple
- Updates can be done sequentially, or all at once
- Convergence

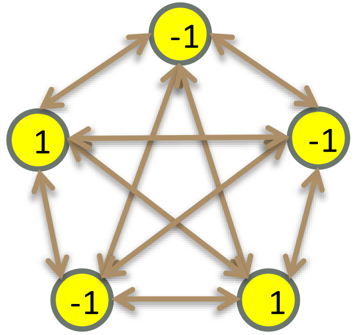
$$E = -\frac{1}{2} \sum_{i} \sum_{j \neq i} w_{ij} y_i y_j$$

Review: Content addressable memory



Review: Hebbian learning to store multiple patterns





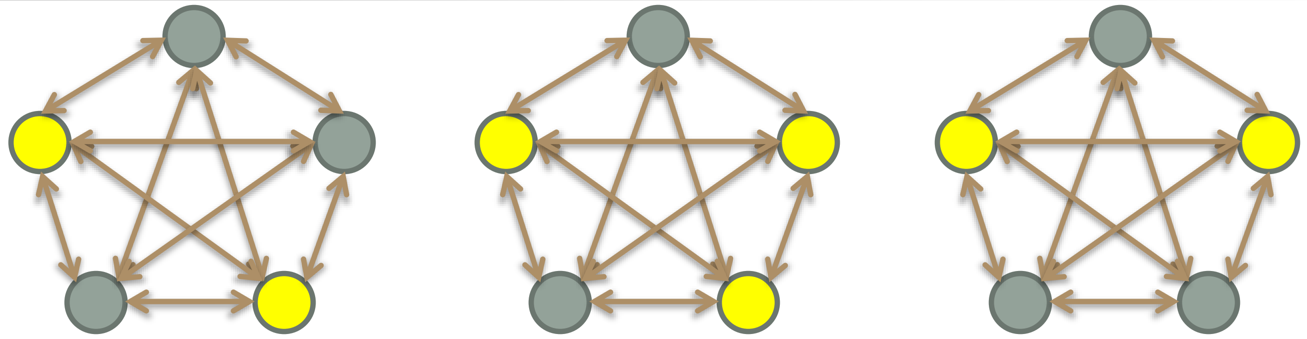
- To store more than one pattern

$$w_{ij} = \sum_{\mathbf{y}^p \in \mathbf{Y}} y_i^p y_j^p \qquad \mathbf{W} = \sum_p (\mathbf{y}_p \mathbf{y}_p^T - I) = \mathbf{Y}\mathbf{Y}^T - N_p \mathbf{I}$$

- $\mathbf{Y} = \{\mathbf{y}^p\}$ is the set of patterns to store

- Super/subscript p represents the specific pattern

Review: patterns we can store



- Hopfield: For a network of *N* neurons can store up to ~0.14*N* patterns through Hebbian learning
  - Provided they are “far” enough
- Where did this number come from?

Claim and questions

- We can always store (up to) *N* orthogonal patterns such that they are stationary! (may not be stable)
- How to let the network store a specific pattern or a set of patterns?
  - Hebbian learning
  - <u>Geometric approach</u>
  - Optimization
- How many patterns can we store?

Back to the energy function

- Note behavior of $\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y}$ with

$$\mathbf{W_1} = \mathbf{YY}^T - N_p\mathbf{I}$$

- Is identical to behavior with

$$\mathbf{W_2} = \mathbf{YY}^T$$

Same eigenvectors

- Since the inner productions are different up to a constant NN_p.

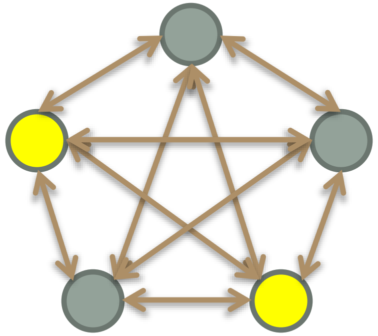
$$\mathbf{y}^T\left(\mathbf{YY}^T - N_p\mathbf{I}\right)\mathbf{y} = \mathbf{y}^T\mathbf{YY}^T\mathbf{y} - NN_p$$

$$< \mathbf{y}, \mathbf{y} >_{\mathbf{W_1}} = < \mathbf{y}, \mathbf{y} >_{\mathbf{W_2}} - NN_p$$

- $\mathbf{W} = \mathbf{YY}^T$ is easier to analyze. Hence in the following slides we will use $\mathbf{W} = \mathbf{YY}^T$

Positive semidefinite matrix

Energy function



$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$

This is a quadratic!

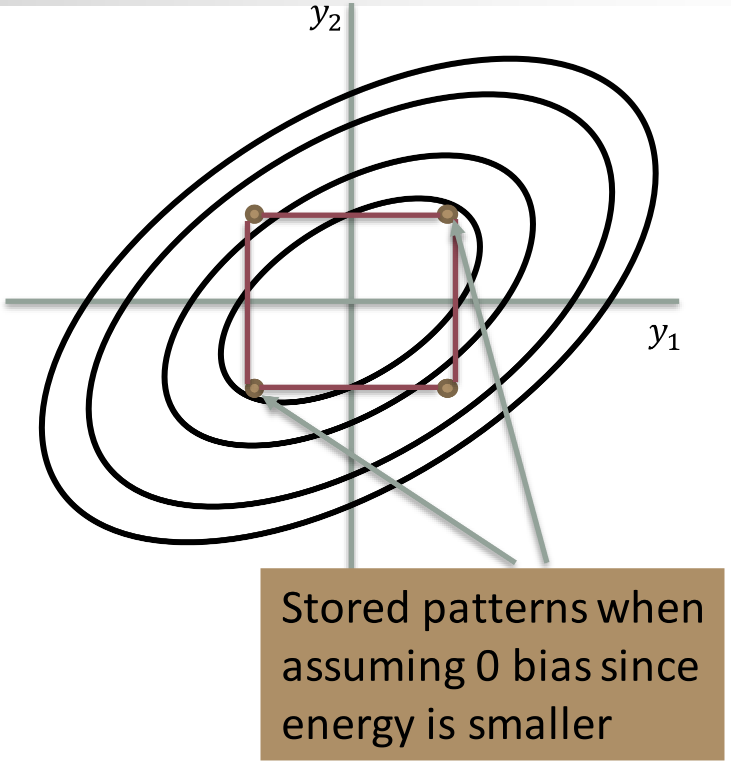
For Hebbian learning, $\mathbf{W}$ is positive semidefinite

Reinstating the bias term for completeness sake

Energy function (2D)

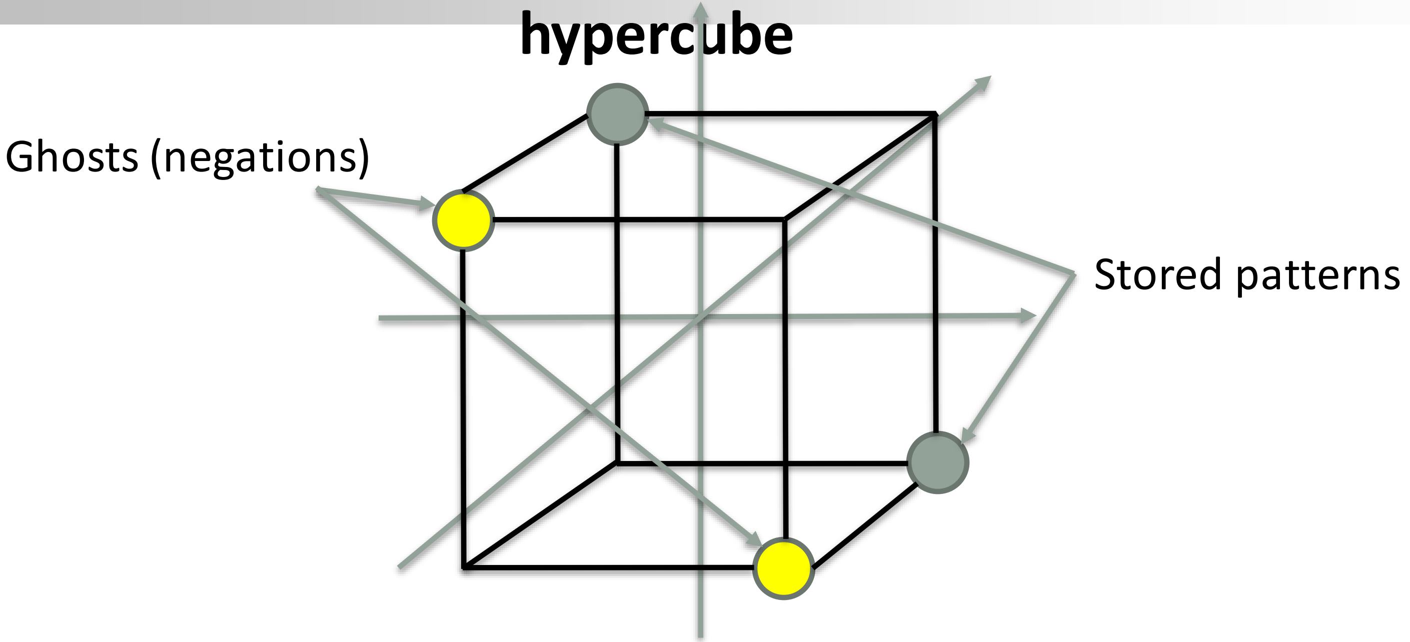
$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$

- $\mathbf{E}(\mathbf{y})$ is a quadratic and $\mathbf{W}$ is positive semidefinite matrix
- But components of $\mathbf{y}$ can only take values ± 1
  - $\mathbf{y}$ lies on the corners of the unit hypercube



Stored patterns when assuming 0 bias since energy is smaller

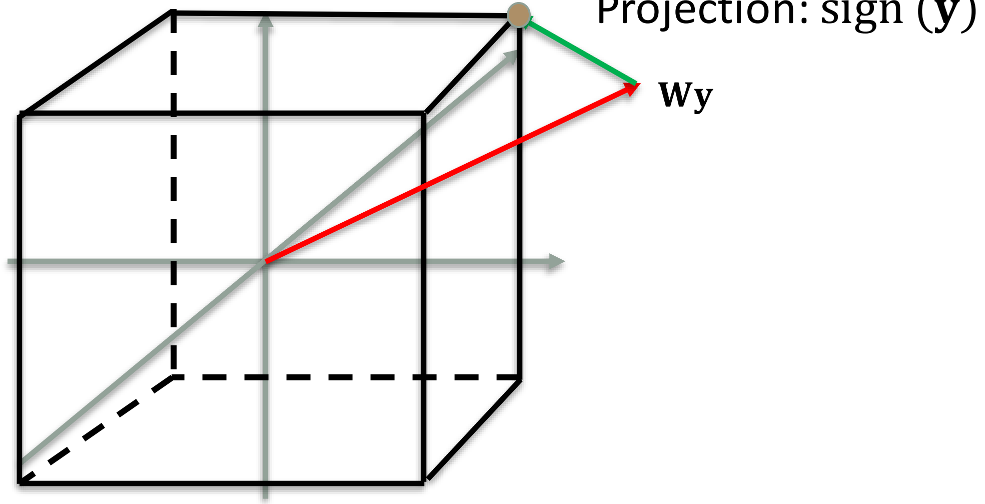
3D hypercube



- All patterns are on the corners of a hypercube
  - If a pattern is stored, it's the "ghost" that is stored as well
  - Intuitively, patterns must ideally be maximally far apart
    - Though this does not seem to hold for Hebbian learning

Evolution of the network

- Note: for binary vectors $\text{sign}\,(\mathbf{y})$ is a projection
  - Projects $\mathbf{y}$ onto the nearest corner of the hypercube
  - It "quantizes" the space into orthants
- Response to field: $\mathbf{y} \leftarrow \text{sign}\,(\mathbf{W}\mathbf{y})$
  - Each step rotates the vector $\mathbf{y}_p$ and then projects it onto the nearest corner



Storing patterns

- A pattern $\mathbf{y}_p$ is stored if:
  - $\text{sign}\left(\mathbf{W}\mathbf{y}_p\right) = \mathbf{y}_p$ for all target patterns
- Training: Design $\mathbf{W}$ such that this holds
  - Simple solution: $\mathbf{y}_p$ is an eigenvector of $\mathbf{W}$
    - And the corresponding eigenvalue is positive

$$\mathbf{W}\mathbf{y}_p = \lambda\mathbf{y}_p$$

    - More generally $\text{orthant}\left(\mathbf{W}\mathbf{y}_p\right) = \text{orthant}\ (\mathbf{y}_p)$
- How many such $\mathbf{y}_p$ can we have?
  - Patterns that differ in $N/2$ bits are orthogonal
  - N orthogonal vectors in an N-dimensional space

Eigenvectors and eigenvalues

- The eigenvectors of any symmetric matrix $\mathbf{W}$ are orthogonal
- The eigenvalues may be positive or negative
- Requirement: given $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_p$
  - Design $\mathbf{W}$ such that
    - $\operatorname{sign}\left(\mathbf{W}\mathbf{y}_p\right) = \mathbf{y}_p$ for all target patterns
    - There are no other binary vectors for which this holds
- What is the largest number of patterns that can be stored?

Storing K orthogonal patterns

- Simple solution: Design $\mathbf{W}$ such that $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$ are the eigenvector of $\mathbf{W}$
  - Let $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K]$
    $$\mathbf{W} = \mathbf{Y}\mathbf{\Lambda}\mathbf{Y}^T$$
  - $\lambda_1, \dots, \lambda_K$ are positive
  - For $\lambda_1 = \lambda_2 = \cdots = \lambda_K = 1$ this is exactly the Hebbian rule
- The patterns are provably stationary

Hebbian rule

- In reality
  - Let $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K, \mathbf{r}_{K+1}, \dots, \mathbf{r}_N]$
    $$\mathbf{W} = \mathbf{Y}\mathbf{\Lambda}\mathbf{Y}^T$$
  - $\mathbf{r}_{K+1}, \dots, \mathbf{r}_N$ are orthogonal to $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$
  - $\lambda_1 = \lambda_2 = \cdots = \lambda_K = 1$
  - $\lambda_{K+1} = \lambda_{K+2} = \cdots = \lambda_N = 0$
- All patterns orthogonal to $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$ are also stationary
  - Although not stable

Storing N orthogonal patterns

- When we have N orthogonal (or near orthogonal) patterns $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N$
  - $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N]$

$$\mathbf{W} = \mathbf{Y\Lambda Y}^T$$

  - $\lambda_1 = \lambda_2 = \cdots = \lambda_N = 1$
- The eigenvectors of $\mathbf{W}$ span the space
- Also, for any $\mathbf{y}_p$

$$\mathbf{W}\mathbf{y}_p = \mathbf{1}\mathbf{y}_p$$

- The N orthogonal patterns $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N$ span the space

Storing N orthogonal patterns

- Any pattern $\mathbf{y}$ can be written as

$$\mathbf{y} = a_1\mathbf{y}_1 + a_2\mathbf{y}_2 + \cdots + a_N\mathbf{y}_N$$

$$\mathbf{Wy} = a_1\mathbf{Wy}_1 + a_2\mathbf{Wy}_2 + \cdots + a_N\mathbf{Wy}_N$$

$$= a_1\mathbf{y}_1 + a_2\mathbf{y}_2 + \cdots + a_N\mathbf{y}_N = \mathbf{y}$$

- All patterns are stable
  - Remembers everything
  - Completely useless network

Storing K orthogonal patterns

- Even if we store fewer than $\boldsymbol{N}$ patterns
  - Let $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K, \mathbf{r}_{K+1}, \dots, \mathbf{r}_N]$
  $$\mathbf{W} = \mathbf{Y}\boldsymbol{\Lambda}\mathbf{Y}^T$$
  - $\mathbf{r}_{K+1}, \dots, \mathbf{r}_N$ are orthogonal to $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$
  - $\lambda_1 = \lambda_2 = \cdots = \lambda_K = 1$
  - $\lambda_{K+1} = \lambda_{K+2} = \cdots = \lambda_N = 0$
- All patterns orthogonal to $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$ are stationary
- Any pattern that is entirely in the subspace spanned by $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$ is also stable
- Only patterns that are partially in the subspace spanned by $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$ are unstable
  - Get projected onto subspace spanned by $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_K$

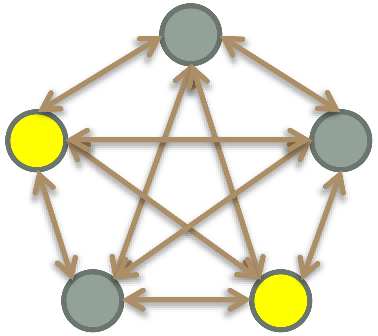
Problem with Hebbian Rule

- With an network of *N* units (i.e. *N*-bit patterns)
- The maximum number of stable patterns is actually exponential in *N*
  - McElice and Posner (1984)
  - E.g., when we had the Hebbian net with *N* orthogonal base patterns, all patterns are stable
- For a specific set of *K* patterns, we can always build a network for which all *K* patterns are stable provided $K \leq N$ (But this may come with many "parasitic" memories

Claim and questions

- We can always store (up to) *N* orthogonal patterns such that they are stationary! (may not be stable)
- How to let the network store a specific pattern or a set of patterns?
  - Hebbian learning
  - Geometric approach
  - <u>Optimization</u>
- How many patterns can we store?

Consider the energy function



$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$

- This must be maximally low for target patterns
- Must be maximally high for all other patterns
  - So that they are unstable and evolve into one of the target patterns
- Estimate $\mathbf{W}$ (and $\mathbf{b}$) such that
  - E is minimized for $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_p$
  - E is maximized for all other $\mathbf{y}$

Optimizing W (and b)

$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{y}$$

The bias can be captured by another fixed-value component

Minimize total energy of target patterns

$$\widehat{\mathbf{W}} = \mathrm{argmin}_{\mathbf{W}} \sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{E}(\mathbf{y})$$

Maximize the total energy of all non-target patterns

$$\widehat{\mathbf{W}} = \mathrm{argmin}_{\mathbf{W}} \sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{E}(\mathbf{y}) - \sum_{\mathbf{y}\notin\mathbf{Y}_p} \mathbf{E}(\mathbf{y})$$

Optimizing W

Simple gradient descent:

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T \right)$$

"emphasize" the importance of a pattern by repeating

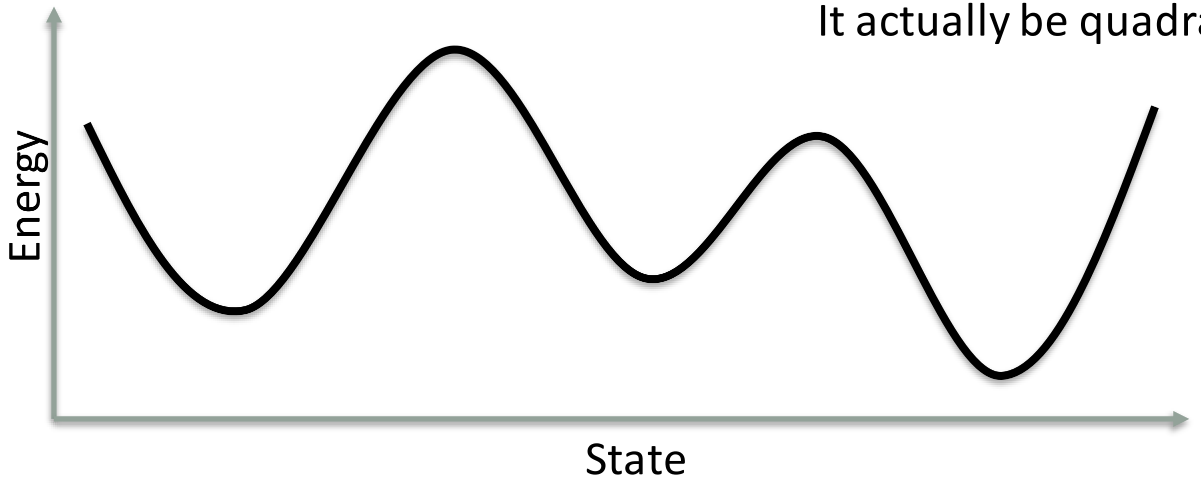
Not important patterns

Optimization process

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T \right)$$

Note the energy contour of a Hopfield network for any weight $\mathbf{W}$

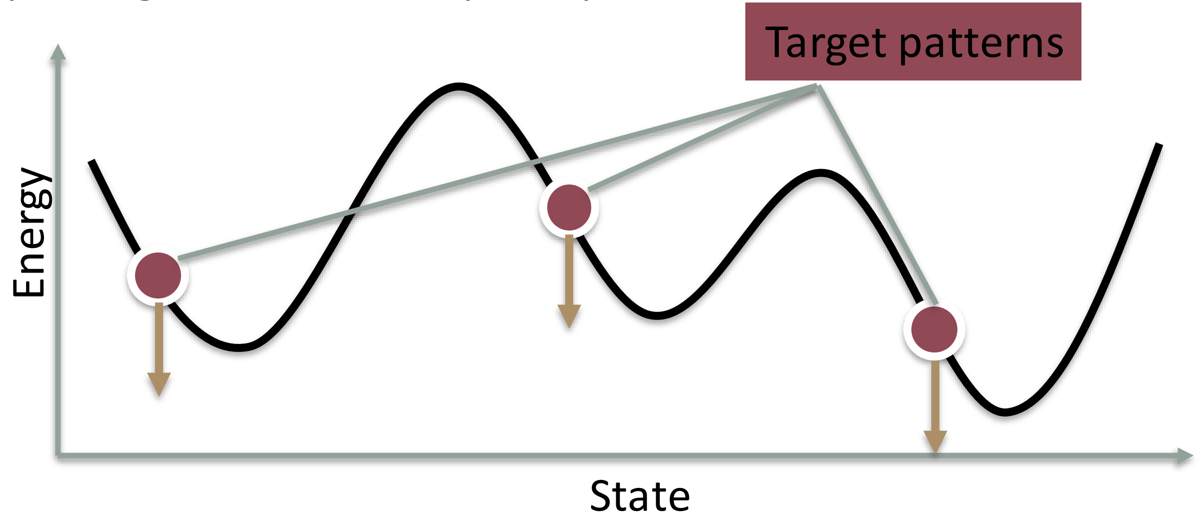
It actually be quadratic



Optimization process

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T \right)$$

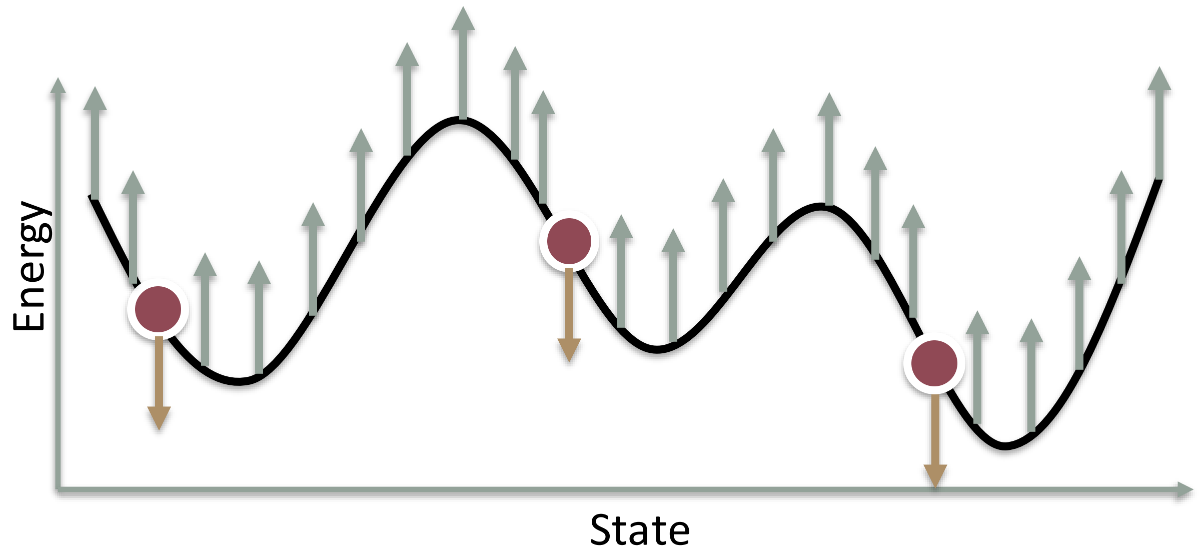
The first term tries to minimize the energy at target patterns
Make them local minima, emphasize more "important" memories by repeating them more frequently



Optimization process

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p}\mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p}\mathbf{y}\mathbf{y}^T\right)$$

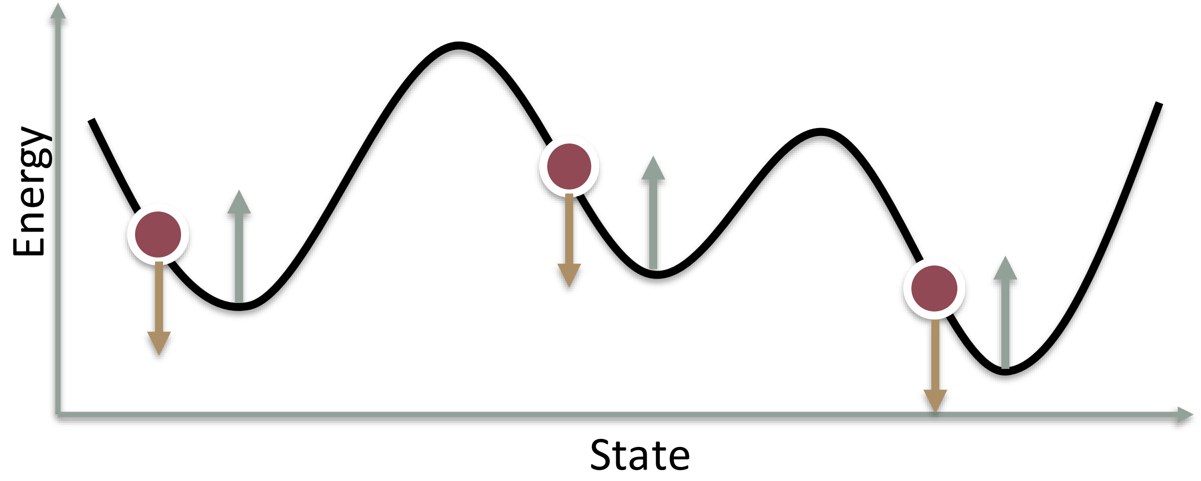
The second terms tries to “raise” all non-target patterns



Focus on the valleys

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p \& \mathbf{y} = \text{valley}} \mathbf{y}\mathbf{y}^T \right)$$

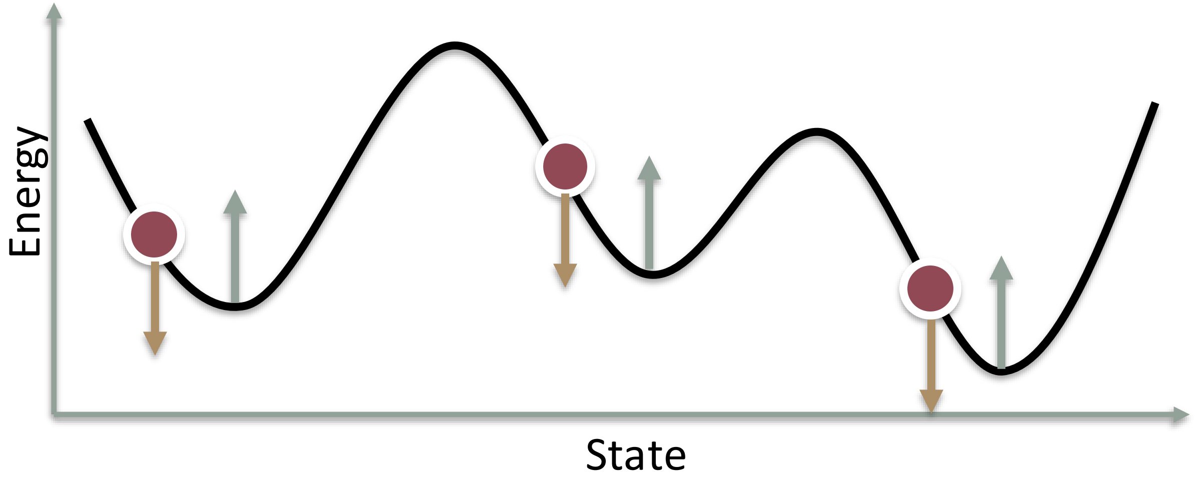
Focus on raising the valleys: if you raise every valley, eventually they'll all move up above the target patterns, and many will even vanish



Focus on the valleys

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p \& \mathbf{y} = \text{valley}} \mathbf{y}\mathbf{y}^T \right)$$

Problem: How do you identify the valleys for the current **W**?



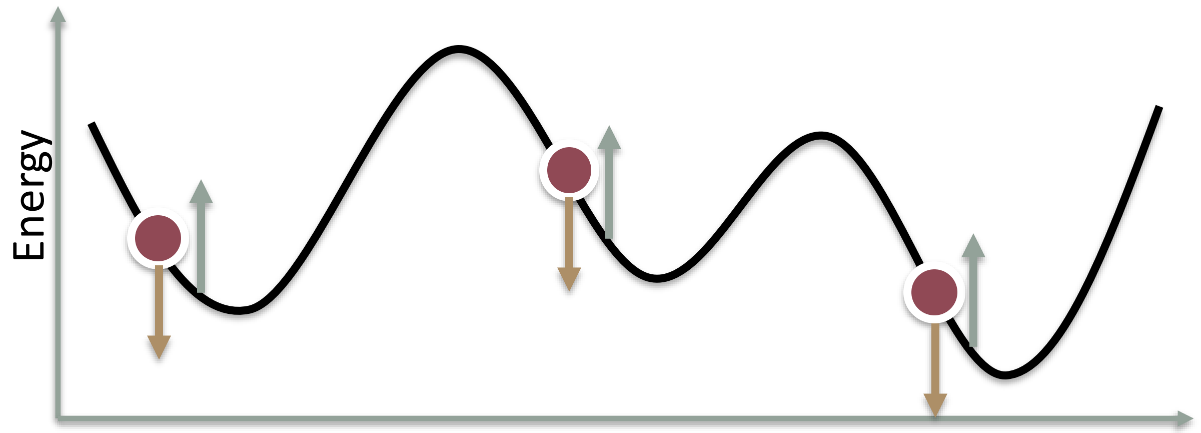
Training the Hopfield network

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p \& \mathbf{y} = \text{valley}} \mathbf{y}\mathbf{y}^T \right)$$

- Initialize $\mathbf{W}$ (randomly, and let it evolve. It will settle in a valley)
- Compute the total outer product of all target patterns. More important patterns presented more frequently.
- Initialize the network with each target pattern $\mathbf{y}_p$ and let it evolve
  - And settle at a valley
- Compute the total outer product of valley patterns
- Update weights

Efficient training

- Really no need to raise the entire surface, or even every valley
- Raise the neighborhood of each target memory (Sufficient to make the memory a valley, and the broader the neighborhood considered, the broader the valley



Training the Hopfield network: SGD version

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p \& \mathbf{y}=\text{valley}} \mathbf{y}\mathbf{y}^T\right)$$

- Initialize $\mathbf{W}$
- Do until convergence, satisfaction, or death from boredom:
  - Sample a target pattern $\mathbf{y}_p$
    - Sampling frequency of pattern must reflect importance of pattern
  - Initialize the network at $\mathbf{y}_p$ and let it evolve a few steps (2-4)
    - Settle at a valley $\mathbf{y}_v$
  - Update weights

$$\mathbf{W} = \mathbf{W} + \eta(\mathbf{y}_p\mathbf{y}_p^{\mathbf{T}} - \mathbf{y}_v\mathbf{y}_v^{\mathbf{T}})$$

A probabilistic interpretation

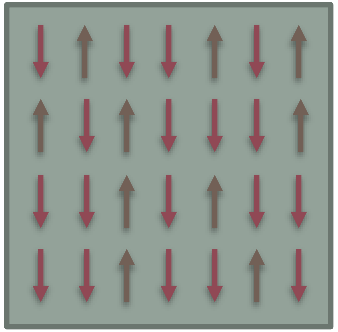
$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} \qquad P(\mathbf{y}) = C\exp\left(\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y}\right)$$

$$C = \frac{1}{\sum_{\mathbf{y}} P(\mathbf{y})}$$

- For continuous $\mathbf{y}$, the energy of a pattern is a perfect analog to the negative log likelihood of a Gaussian density
- For binary y, it is the analog of the negative log likelihood of a Boltzmann distribution
- Minimizing energy maximizes log likelihood

The Boltzmann Distribution

$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y}$$



$$P(\mathbf{y}) = C\exp\left(\frac{-E(\mathbf{y})}{kT}\right)$$

$$C = \frac{1}{\sum_{\mathbf{y}} P(\mathbf{y})}$$

- k is the Boltzmann constant
- T is the temperature of the system
- The energy terms are like the loglikelihood of a Boltzmann distribution at $T = 1$ and $k = 1$
- The system probabilistically selects states with lower energy

Optimizing W

$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} \qquad \widehat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} \sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{E}(\mathbf{y}) - \sum_{\mathbf{y}\notin\mathbf{Y}_p} \mathbf{E}(\mathbf{y})$$

Simple gradient descent:

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p} \alpha_{\mathbf{y}}\mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p} \beta(E(\mathbf{y}))\mathbf{y}\mathbf{y}^T\right)$$

More importance to more frequently presented memories

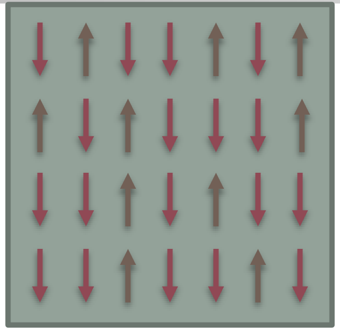
More importance to more attractive spurious memories

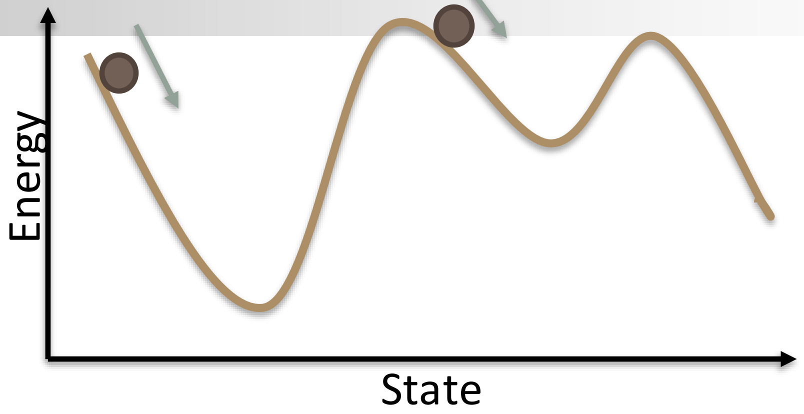
Like an expectation

$$\mathbf{W} = \mathbf{W} + \eta\left(E_{\mathbf{y}\sim\mathbf{Y}_p}\mathbf{y}\mathbf{y}^T - E_{\mathbf{y}\sim\mathbf{Y}}\mathbf{y}\mathbf{y}^T\right)$$

The Hopfield network is a distribution

Thermodynamic phenomena





- Is the system actually in a specific state at any time?
- No – the state is actually continuously changing
    - Based on the temperature of the system (at higher temperatures, state changes more rapidly
- What is actually being characterized is the

The Helmholtz Free Energy

- A thermodynamic system at temperature *T* can exist in one of many states (potentially infinite states, the probability of finding the system in state s at temperature *T* is $P_T(s)$)
- At each state s it has a potential energy E_s
- The internal energy of the system, representing its capacity to do work, is the average:

$$U_T = \sum_s P_T(s) E_s$$

- The capacity to do work is counteracted by the internal disorder of the system, i.e. its entropy

$$H_T = -\sum_s P_T(s) \log P_T(s)$$

The Helmholtz Free Energy

- The Helmholtz free energy is:

$$F_T = U_T + kTH_T$$

$$= \sum_s P_T(s) E_s - kT \sum_s P_T(s) \log P_T(s)$$

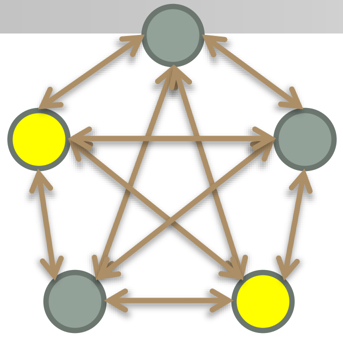
- The probability distribution of the states at steady state is known as the Boltzmann distribution

$$P_T(s) = \frac{1}{Z} \exp\left(\frac{-E_s}{kT}\right)$$

Z is a normalizing constant

  - Temperature depended T
  - At $T = 0$, the system will always remain at the lowest energy configuration with prob = 1

The Energy of the Network

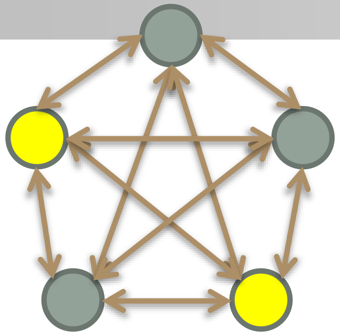


$$E(S) = -\sum_{i<j} w_{ij}s_i s_j - b_i s_i$$

$$P(S) = \frac{\exp(-E(S))}{\sum_{S'} \exp(-E(S'))}$$

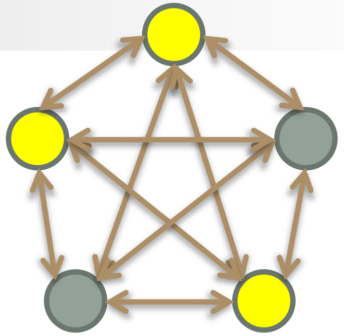
- Define the energy of the system as before
- Since neurons are stochastic, there is entropy (with *T*=1)
- The equilibrium probability distribution over states is the Boltzmann distribution at *T=1*
- The stochastic Hopfield network modes a **probability distribution** over states
- The probability that (at equilibrium) the network will be in any state is $P(S)$

The field at a single node



$$P(S) = P\left(s_i = 1 \middle| s_{j\neq i}\right) P\left(s_{j\neq i}\right)$$

$$P(S') = P\left(s_i = -1 \middle| s_{j\neq i}\right) P\left(s_{j\neq i}\right)$$



- Let S and S' be otherwise identical state that only differ in the i-th bit. S has i–th bit = +1 and S' has i-th bit = -1

$$\log P(S) - \log P(S') = \log P\left(s_i = 1 \middle| s_{j\neq i}\right) - \log P\left(s_i = -1 \middle| s_{j\neq i}\right)$$

$$\log P(S) - \log P(S') = \log \frac{P\left(s_i = 1 \middle| s_{j\neq i}\right)}{1 - P\left(s_i = 1 \middle| s_{j\neq i}\right)}$$

The field at a single node

$$\log P(S) = -E(S) + C$$

$$E(S) = -\frac{1}{2}\left(E_{\neq i} + \sum_{i\neq j} w_j s_j + b_i\right) \quad E(S') = -\frac{1}{2}(E_{\neq i} - \sum_{i\neq j} w_j s_j - b_i)$$

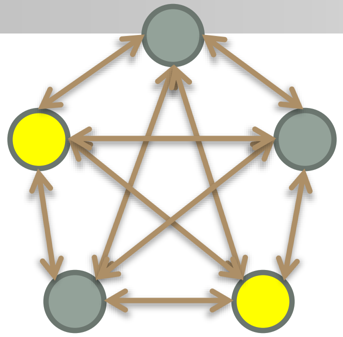
$$\log P(S) - \log P(S') = E(S) - E(S') = \sum_{i\neq j} w_j s_j - b_i$$

$$\log P(S) - \log P(S') = \log \frac{P\left(s_i = 1 \middle| s_{j\neq i}\right)}{1 - P\left(s_i = -1 \middle| s_{j\neq i}\right)} = \sum_{i\neq j} w_j s_j - b_i$$

- The probability of any node taking value 1 given order node values is a logistic

$$P\left(s_i = 1 \middle| s_{j\neq i}\right) = \frac{1}{1 + e^{-(\sum_{i\neq j} w_j s_j - b_i)}}$$

Redefining the network

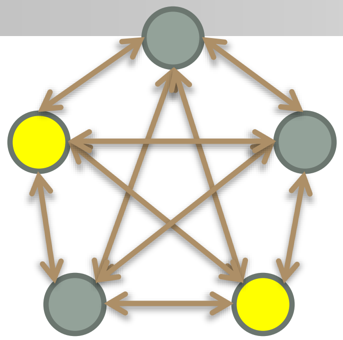


$$Z_i = \sum_{i \neq j} w_j s_j - b_i$$

$$P\left(s_i = 1 \middle| s_{j \neq i}\right) = \frac{1}{1 + e^{-Z_i}}$$

- First try: redefine a regular Hopfield net as a stochastic system
- Each neuron is now a stochastic unit with a binary state s_i, which can take value 0 or 1 with a probability that depends on the local field
    - Note the slight change from Hopfield nets
    - Not actually necessary
- The conditional distribution of individual bits in the sequence is logistic

Training the network



$$E(S) = -\sum_{i<j} w_{ij}s_is_j - b_is_i$$

$$P(S) = \frac{\exp(-E(S))}{\sum_{S'}\exp(-E(S'))}$$

$$P(S) = \frac{\exp(\sum_{i<j} w_{ij}s_is_j + b_is_i)}{\sum_{S'}\exp(\sum_{i<j} w_{ij}{s'}_i{s'}_j + b_i{s'}_i)}$$

- As in Hopfield nets, in order to train the network, we need to select weights such that those states are more probable than other states
- Maximize the likelihood of the “stored” states

Maximum Likelihood Training

$$\log P(S) = \log \frac{\exp(\sum_{i<j} w_{ij} s_i s_j + b_i s_i)}{\sum_{S'} \exp(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i)}$$

$$\log P(S) = \left(\sum_{i<j} w_{ij} s_i s_j + b_i s_i\right) - \log \sum_{S'} \exp\left(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i\right)$$

$$< \log P(\mathbf{S}) > = \frac{1}{N} \sum_{S \in \mathbf{S}} \log P(S)$$

$$= \frac{1}{N} \left(\sum_{S \in \mathbf{S}} \sum_{i<j} w_{ij} s_i s_j + b_i s_i\right) - \log \sum_{S'} \exp\left(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i\right)$$

- Maximize the average log likelihood of all “training” vectors $\mathbf{S} = \{S_{\mathbf{1}}, S_{\mathbf{2}}, \dots, S_{\boldsymbol{N}}\}$

Maximum Likelihood Training

$$< \log P(\mathbf{S}) >= \frac{1}{N}\left(\sum_{S\in\mathbf{S}}\sum_{i<j} w_{ij}s_is_j + b_is_i\right) - \log\sum_{S'}\exp\left(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i\right)$$

$$\frac{d < \log P(\mathbf{S}) >}{dw_{ij}} = \frac{1}{N}\sum_{S\in\mathbf{S}} s_is_j - \ \ldots$$

- We take the gradient descent, but the second term is summed over all states, which can be an exponential number.

$$\frac{d\log\sum_{S'}\exp(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i)}{dw_{ij}} = \sum_{S'}\frac{\exp(\sum_{i<j} w_{ij}s_is_j + b_is_i)}{\sum_{S'}\exp(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i)}s'_is'_j$$

$$= \sum_{S'} P(S')s'_is'_j$$

- The second term is simply the expected value of s_is_j, over all possible values of the state
- We cannot compute it exhaustively, but by sampling.

The simulation solution

- Initialize the network randomly and let it “evolve”
- After many epochs, take a snapshot of the state
- Repeat this many times
- Let the collection of states be $\mathbf{S}_{\boldsymbol{snap}} = \{S_{\mathbf{1}}, S_{\mathbf{2}}, \ldots, S_{\boldsymbol{M}}\}$

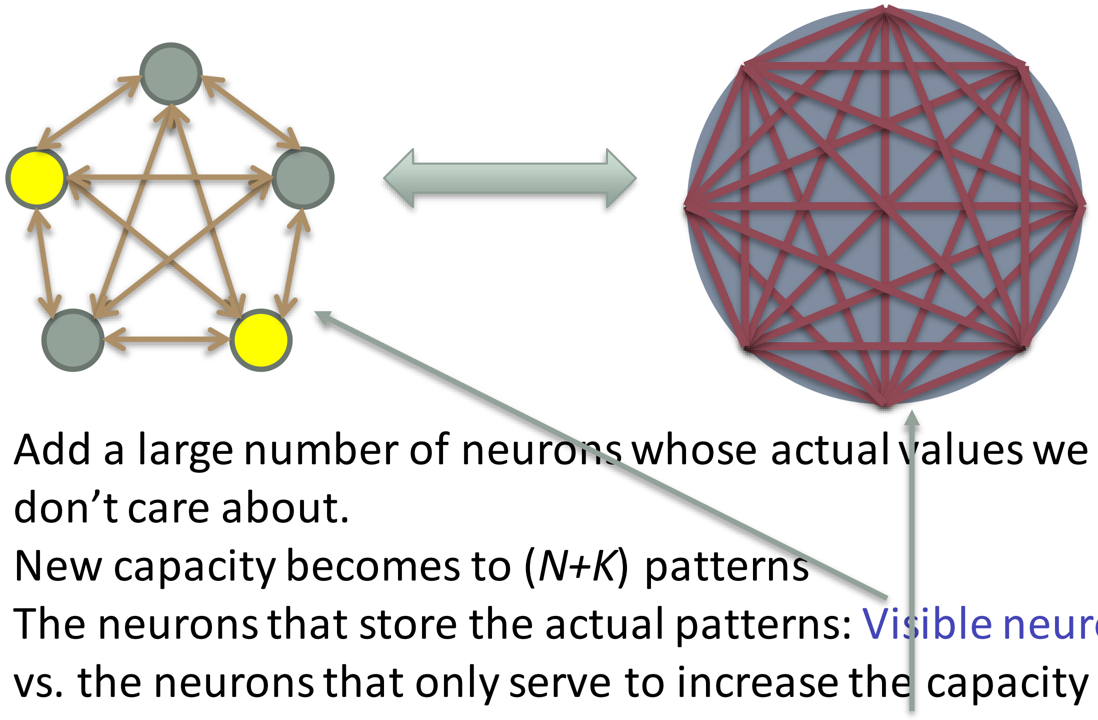
$$\sum_{S'} P(S')s'_i s'_j \approx \frac{1}{M} \sum_{S' \in \mathbf{S}_{\boldsymbol{snap}}} s'_i s'_j$$

The second term in the derivative is computed as the average of sampled states when the network is running “freely”.
The overall gradient ascent rule

$$w_{ij} = w_{ij} + \eta \frac{d < \log P(\mathbf{S}) >}{dw_{ij}}$$

Storing more than *N* patterns

- The memory capacity of an *N*-bit network is most *N*
- Expanding *N* neurons to *K* neurons

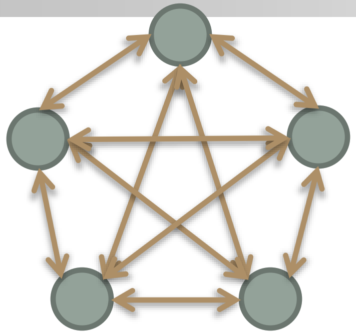


- Add a large number of neurons whose actual values we don't care about.
- New capacity becomes to (*N*+*K*) patterns
- The neurons that store the actual patterns: Visible neurons vs. the neurons that only serve to increase the capacity but whose actual values are not important: Hidden neurons

The patterns

- For a given (*N*+*K*) neurons (*N* visible neurons), there are many number of hidden patterns (2^K)
- Ideally choose the one that results in the lowest energy
- But that is an exponential search space
- In fact, we could have multipole hidden patterns coupled with any visible pattern
- These would be multiple stored patterns that all give the same visible output
- But, how many do we permit?
- Do we need to specify one or more particular hidden patterns?

Review: Hopfield Net



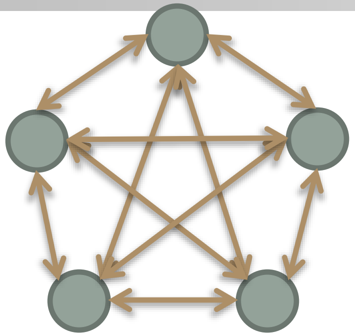
$$F(x) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x \le 0 \end{cases}$$

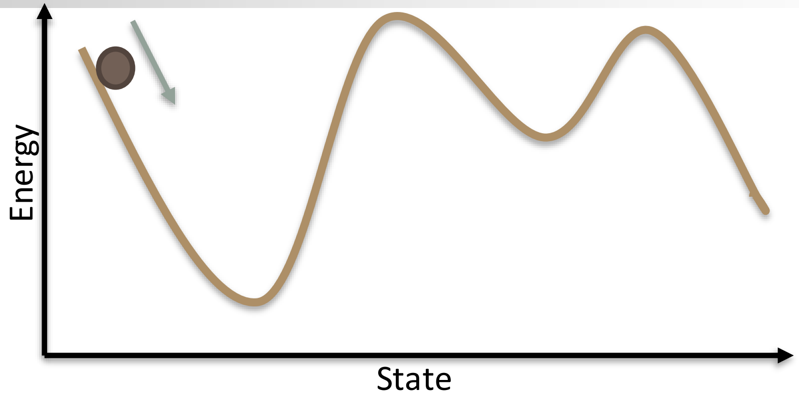
$$y_i = F\left(\sum_{j \neq i} w_{ij} y_j + b_i\right)$$

- At each time each neuron receives a “field”
- If the sign of the field matches its own sign, no respond
- If the sign of the field opposes its own sign, flips
- The system will evolve until the energy hits a local min

$$E = -\frac{1}{2}\sum_{i}\sum_{j \neq i} w_{ij} y_i y_j$$

Review: Content-addressable memory





- This is analogous to the potential energy of a spin glass
    - The system will evolve until the energy hits a local minimum
- Each of the minima is a “stored” pattern
    - If the network is initialized close to a stored pattern, it will inevitably evolve to the pattern
- This is a content addressable memory
    - Recall memory content from partial or corrupt values
- Also called associative memory

Training

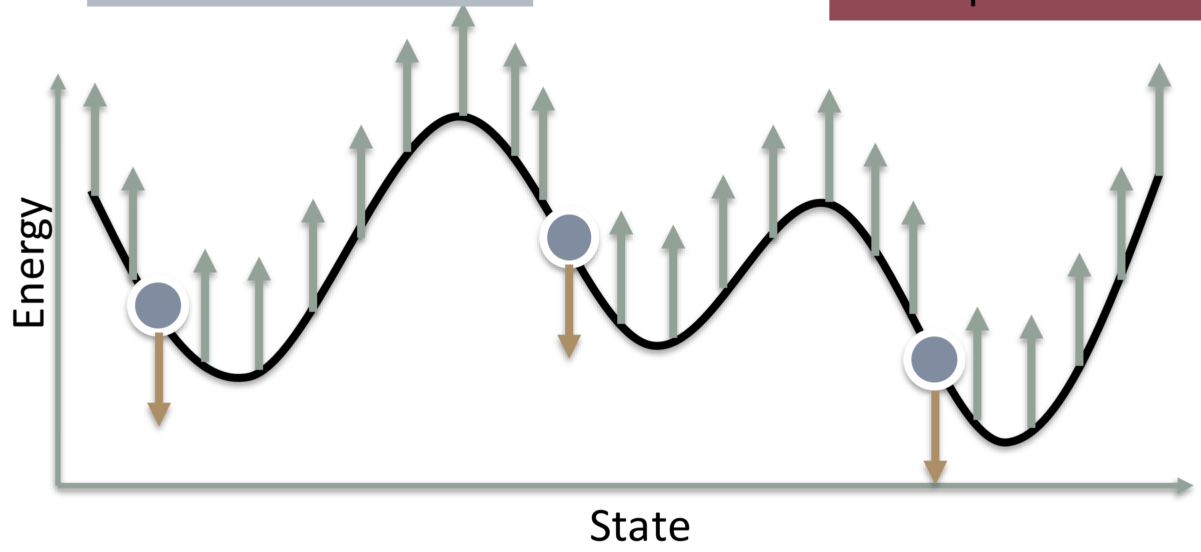
- The Hopfield network can be trained to remember specific "target" patterns
- This can be done by setting the weights $\mathbf{W}$ appropriately (Hebb rule)
- A Hopfield net with N neurons can designed to store up to N target N-bit memories (However, it can store an exponential number of unwanted "parasitic" memories along with the target patterns)
- Training the network: Design weights matrix $\mathbf{W}$ such that the energy …
  - Target patterns is minimized
  - Other untargeted potentially parasitic patterns is maximized so that they don't become parasitic

Training

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p} \mathbf{y}\mathbf{y}^T\right)$$

Minimize energy of target patterns

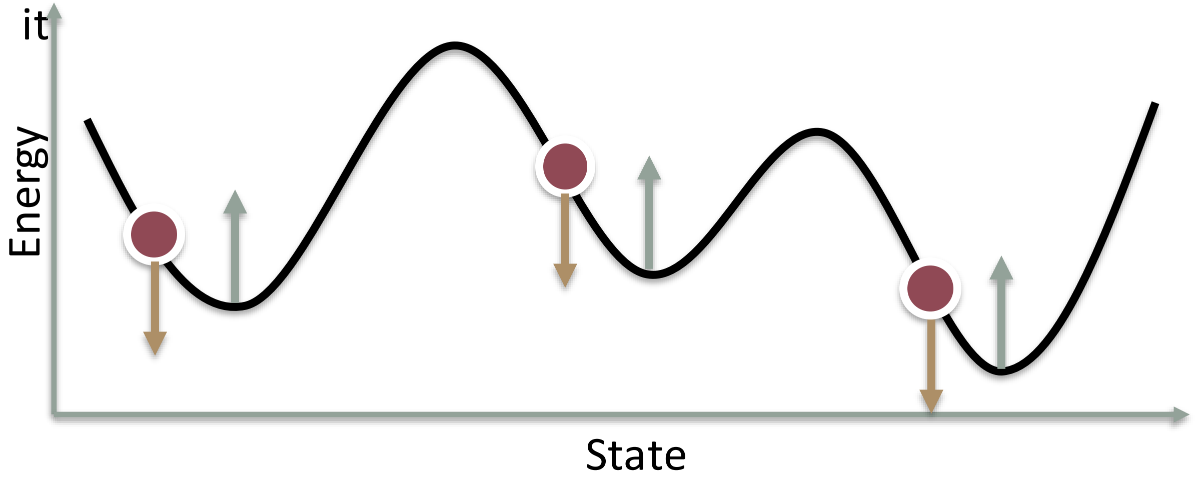
Maximize energy of all other patterns



Optimizing training

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p}\mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p\&\mathbf{y}=\text{valley}}\mathbf{y}\mathbf{y}^T\right)$$

Lower energy at valid memories
Initialize the network at valid memories and let it evolve
It will settle in a valley. If this is not the target patterns, raise it



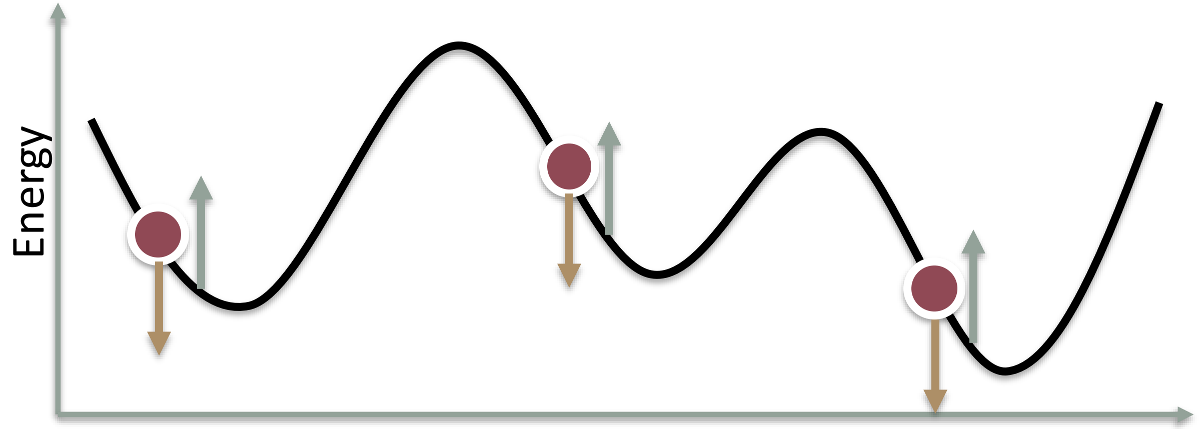
Training the Hopfield network

$$\mathbf{W} = \mathbf{W} + \eta \left(\sum_{\mathbf{y} \in \mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y} \notin \mathbf{Y}_p \& \mathbf{y} = \text{valley}} \mathbf{y}\mathbf{y}^T \right)$$

- Initialize $\mathbf{W}$ (randomly, and let it evolve. It will settle in a valley)
- Compute the total outer product of all target patterns. More important patterns presented more frequently.
- Initialize the network with each target pattern $\mathbf{y}_p$ and let it evolve
    - And settle at a valley
- Compute the total outer product of valley patterns
- Update weights

Efficient training

- Really no need to raise the entire surface, or even every valley
- Raise the neighborhood of each target memory (Sufficient to make the memory a valley, and the broader the neighborhood considered, the broader the valley



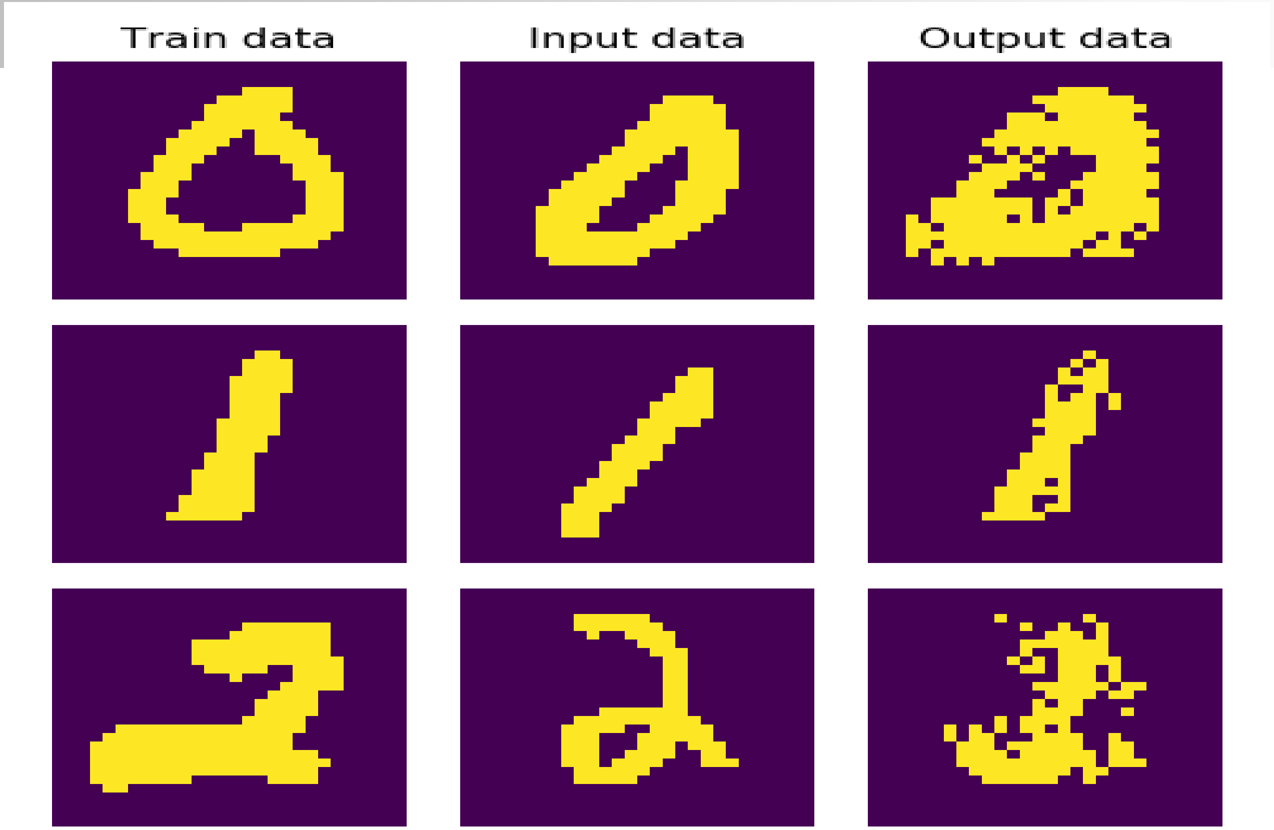
Training the Hopfield network: SGD version

$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_p} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_p \& \mathbf{y}=\text{valley}} \mathbf{y}\mathbf{y}^T\right)$$

- Initialize $\mathbf{W}$
- Do until convergence, satisfaction, or death from boredom:
  - Sample a target pattern $\mathbf{y}_p$
    - Sampling frequency of pattern must reflect importance of pattern
  - Initialize the network at $\mathbf{y}_p$ and let it evolve a few steps (2-4)
    - Settle at a valley $\mathbf{y}_v$
  - Update weights

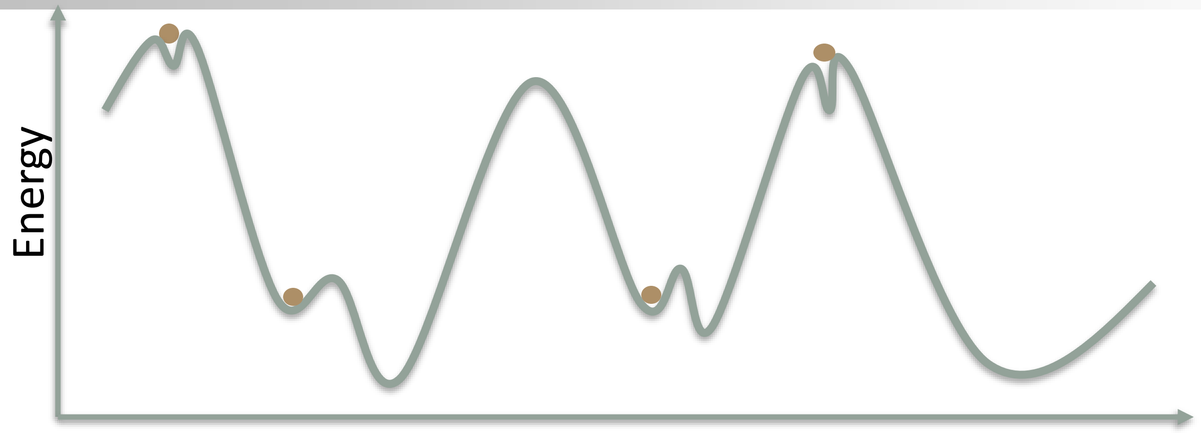
$$\mathbf{W} = \mathbf{W} + \eta(\mathbf{y}_p\mathbf{y}_p^{\mathbf{T}} - \mathbf{y}_v\mathbf{y}_v^{\mathbf{T}})$$

Problems of Hopfield net



Why is the recalled pattern not perfect?

A problems of Hopfield net



- Many local minima
  - Parasitic memories
- May be escaped by adding some noise during evolution
  - Permit changes in state even if energy increases..
    - Particularly if the increase in energy is small

Stochastic Hopfield Net

$$z_i = \frac{1}{T}\sum_{j\neq i} w_{ji}y_j\,, P(y_i = 1) = \sigma(z_i), P(y_i = 0) = 1 - \sigma(z_i)$$

- The evolution of the Hopfield net can be made stochastic
- Instead of deterministically responding to the sign of the local field, each neuron responds probabilistically

Field quantifies the energy difference obtained by flipping the current unit

If the difference is not large, the probability of flipping approaches

T is a "temperature" parameter: increasing it moves the probability of bits towards 0.5
At T=1, we get the traditional definition of field and energy
At T=0, we get deterministic Hopfield behavior

Evolution of stochastic Hopfield Net

1. Initialize network with initial pattern

$$y_i(0) = x_i, \qquad 0 \le i \le N-1$$

2. Iterate $0 \le i \le N$, assuming T=1

$$P = \sigma\left(\sum_{j \ne i} w_{ji} y_j\right)$$

$$y_i(t+1) \sim \text{Binomial}\,(P)$$

- When do we stop?
- What is the final state of the system?

Evolution of stochastic Hopfield Net

```
1. Initialize network with initial pattern
       $y_i(0) = x_i, \quad 0 \le i \le N-1$
2. For $T = T_0$ down to $T_{\min}$
    i. For iteration $1 \ldots L$
        a) For $0 \le i \le N-1$
            $P = \sigma\left(\frac{1}{T}\sum_{j \ne i} w_{ji} y_j\right)$
            $y_i(t+1) \sim \text{Binomial distribution } (P)$
```

- Let the system evolve to "equilibrium"
- Let $\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_L$ be the sequence of values (L large)
- Final prediction configuration: from the average of the final few iterations $\mathbf{y} = \left(\frac{1}{M}\sum_{t=L-M+1}^{L} \mathbf{y}_t\right)$ Final state

Evolution of stochastic Hopfield Net

1. Initialize network with initial pattern

$$y_i(0) = x_i, \qquad 0 \le i \le N-1$$

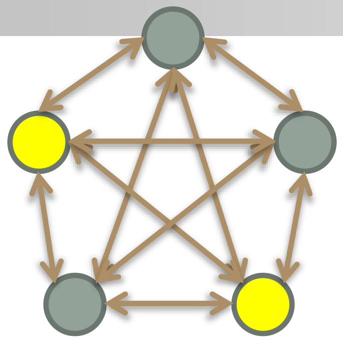
2. Iterate $0 \le i \le N$, assuming T=1

$$P = \sigma\left(\sum_{j \neq i} w_{ji} y_j\right)$$

$$y_i(t+1) \sim \text{Binomial}(P)$$

- When do we stop?
- What is the final state of the system?

The Boltzmann Machine



$$z_i = \frac{1}{T}\sum_{i \neq j} w_j s_j$$

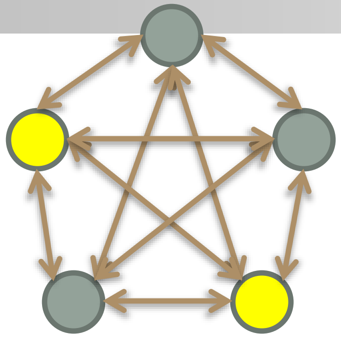
$$P(s_i = 1 | s_{j \neq i}) = \frac{1}{1 + e^{-z_i}}$$

- **The Boltzmann Distribution**

$$\mathbf{E}(\mathbf{y}) = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} \qquad P(\mathbf{y}) = C\exp\left(\frac{-E(\mathbf{y})}{T}\right)$$

- The parameter of the distribution is the weights matrix $\mathbf{W}$
- The conditional distribution of individual bits in the sequence is a logistic
- We will call this a Boltzmann machine

Training the network



$$E(S) = -\sum_{i<j} w_{ij} s_i s_j - b_i s_i$$

$$P(S) = \frac{\exp(-E(S))}{\sum_{S'} \exp(-E(S'))}$$

$$P(S) = \frac{\exp(\sum_{i<j} w_{ij} s_i s_j + b_i s_i)}{\sum_{S'} \exp(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i)}$$

- As in Hopfield nets, in order to train the network, we need to select weights such that those states are more probable than other states
- Maximize the likelihood of the “stored” states

Maximum Likelihood Training

$$\log P(S) = \log \frac{\exp(\sum_{i<j} w_{ij} s_i s_j + b_i s_i)}{\sum_{S'} \exp(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i)}$$

$$\log P(S) = \left(\sum_{i<j} w_{ij} s_i s_j + b_i s_i \right) - \log \sum_{S'} \exp\left(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i \right)$$

$$< \log P(\mathbf{S}) >= \frac{1}{N} \sum_{S \in \mathbf{S}} \log P(S)$$

$$= \frac{1}{N} \left(\sum_{S \in \mathbf{S}} \sum_{i<j} w_{ij} s_i s_j + b_i s_i \right) - \log \sum_{S'} \exp\left(\sum_{i<j} w_{ij} s'_i s'_j + b_i s'_i \right)$$

- Maximize the average log likelihood of all “training” vectors $\mathbf{S} = \{S_\mathbf{1}, S_\mathbf{2}, \dots, S_\mathbf{N}\}$

Maximum Likelihood Training

$$< \log P(\mathbf{S}) >= \frac{1}{N}\left(\sum_{S\in\mathbf{S}}\sum_{i<j} w_{ij}s_is_j + b_is_i\right) - \log\sum_{S'}\exp\left(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i\right)$$

$$\frac{d < \log P(\mathbf{S}) >}{dw_{ij}} = \frac{1}{N}\sum_{S\in\mathbf{S}} s_is_j - \ \ldots$$

- We take the gradient descent, but the second term is summed over all states, which can be an exponential number.

$$\frac{d\log\sum_{S'}\exp(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i)}{dw_{ij}} = \sum_{S'}\frac{\exp(\sum_{i<j} w_{ij}s_is_j + b_is_i)}{\sum_{S'}\exp(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i)}s'_is'_j$$

$$= \sum_{S'} P(S')s'_is'_j$$

- The second term is simply the expected value of s_is_j, over all possible values of the state.
- We cannot compute it exhaustively, but by sampling.

UNIVERSITY OF ARKANSAS

The simulation solution

- Initialize the network randomly and let it "evolve"
- After many epochs, take a snapshot of the state
- Repeat this many times
- Let the collection of states be $\mathbf{S}_{snap} = \{S_1, S_2, \dots, S_M\}$

$$\sum_{S'} P(S') s'_i s'_j \approx \frac{1}{M} \sum_{S' \in \mathbf{S}_{snap}} s'_i s'_j$$

The second term in the derivative is computed as the average of sampled states when the network is running "freely".
The overall gradient ascent rule

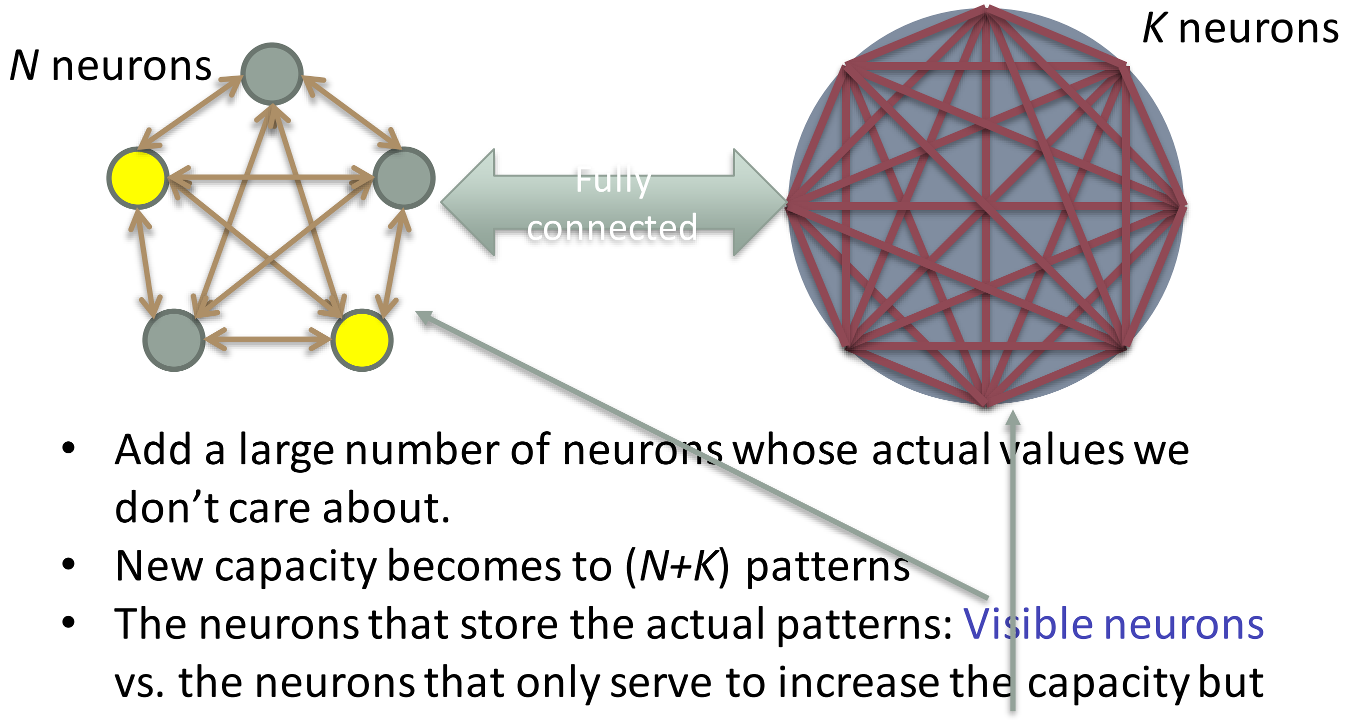
$$w_{ij} = w_{ij} + \eta \frac{d < \log P(\mathbf{S}) >}{d w_{ij}}$$

Adding capacity to the Hopfield Network / Boltzmann Machin

- The network can store up to *N*-bit patterns
- How do we increase the capacity

Expanding the network

- The memory capacity of an *N*-bit network is most *N*
- Expanding *N* neurons to *K* neurons

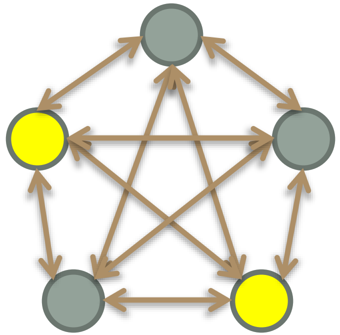


- Add a large number of neurons whose actual values we don't care about.
- New capacity becomes to (*N*+*K*) patterns
- The neurons that store the actual patterns: Visible neurons vs. the neurons that only serve to increase the capacity but whose actual values are not important: Hidden neurons

The patterns

- For a given (*N*+*K*) neurons (*N* visible neurons), there are many number of hidden patterns (2^K)
- Ideally choose the one that results in the lowest energy
- But that's an exponential search space
- In fact, we could have multipole hidden patterns coupled with any visible pattern
- These would be multiple stored patterns that all give the same visible output
- But, how many do we permit?
- Do we need to specify one or more particular hidden patterns?

Boltzmann machine without hidden units

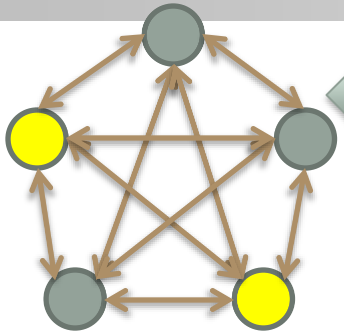


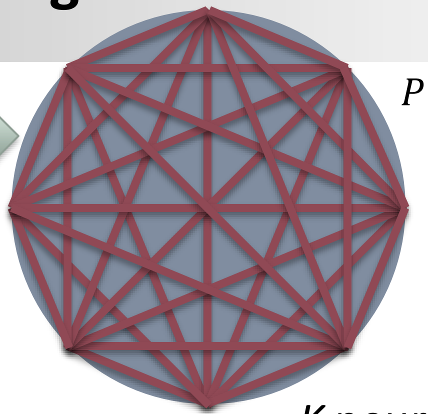
$$\frac{d < \log P(\mathbf{S}) >}{dw_{ij}} = \frac{1}{N}\sum_{S\in\mathbf{S}} s_i s_j - \frac{1}{M}\sum_{S'\in\mathbf{S}_{snap}} s'_i s'_j$$

$$w_{ij} = w_{ij} + \eta \frac{d < \log P(\mathbf{S}) >}{dw_{ij}}$$

- This basic framework has no hidden units
- Extended to have hidden units

Expanding the network





$$P(S) = \frac{\exp(-E(S))}{\sum_{S'} \exp(-E(S'))}$$

$$P(V) = \sum_{H} P(S)$$

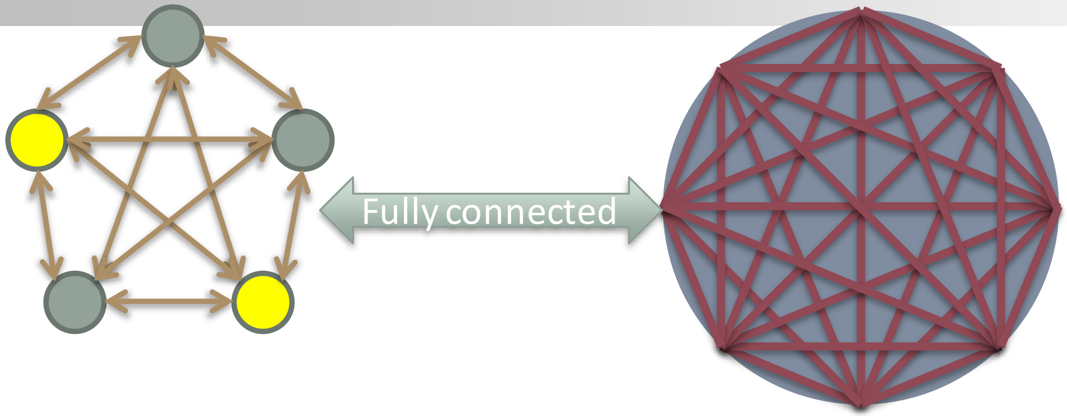
N neurons

K neurons

- Now, with hidden neurons the complete state pattern for even the training patterns is unknown (Since they are only defined over visible neurons)
- We are interested in the marginal probabilities over visible bits
- $S = (V, H)$, V: visible bits, H: hidden bits
- Maximizing the marginal probability of V requires summing over all values of H. An exponential state space.

Training

$$P(S) = \frac{\exp(-E(S))}{\sum_{S'} \exp(-E(S'))}$$



$$P(V) = \sum_{H} P(S)$$

- For each training pattern V_i, fix the visible units to V_i, let the hidden neurons evolve from a random initial point to generate H_i, generate $S_i = [V_i, H_i]$
- Repeat K times to generate synthetic training

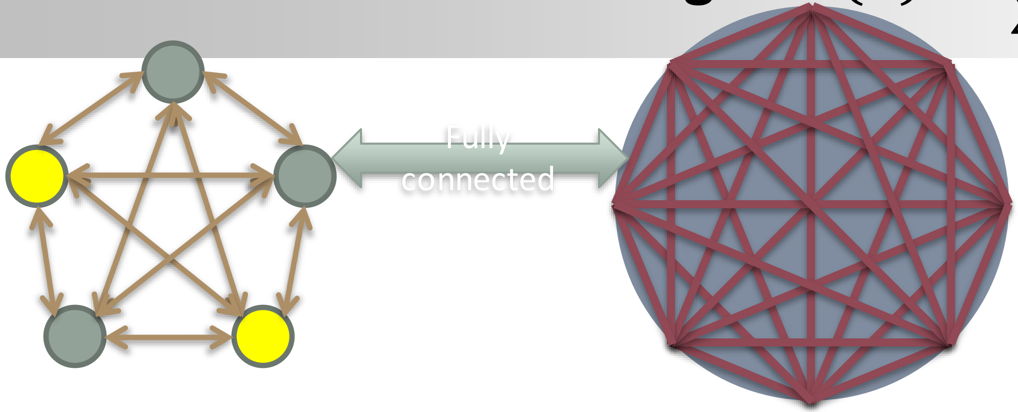
$$\mathbf{S} = \{S_{1,1}, S_{1,2}, \dots, S_{1,K}, S_{2,1}, \dots, S_{N,K}\}$$

- Now unclamp the visible units and let the entire network evolve several times to generate

$$\mathbf{S}_{\boldsymbol{snap}} = \{S_{snap,1}, S_{snap,2}, \dots, S_{snap,M}\}$$

Training

$$P(S) = \frac{\exp(-E(S))}{\sum_{S'} \exp(-E(S'))}$$

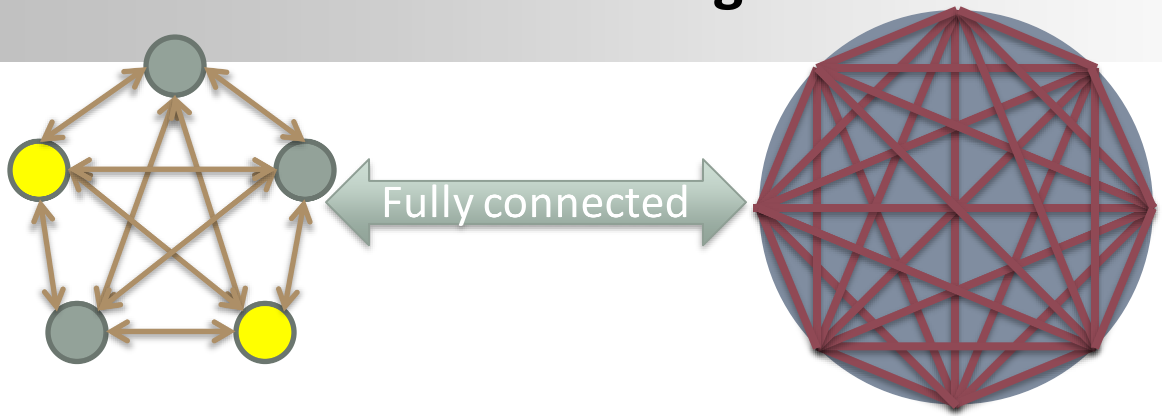


$$P(V) = \sum_{H} P(S)$$

$$\frac{d < \log P(\mathbf{S}) >}{dw_{ij}} = \frac{1}{NK} \sum_{S \in \mathbf{S}} s_i s_j - \frac{1}{M} \sum_{S' \in \mathbf{S}_{snap}} s'_i s'_j$$

- Gradients are computed as before, except that the first term is now computed over the expanded training data

Training



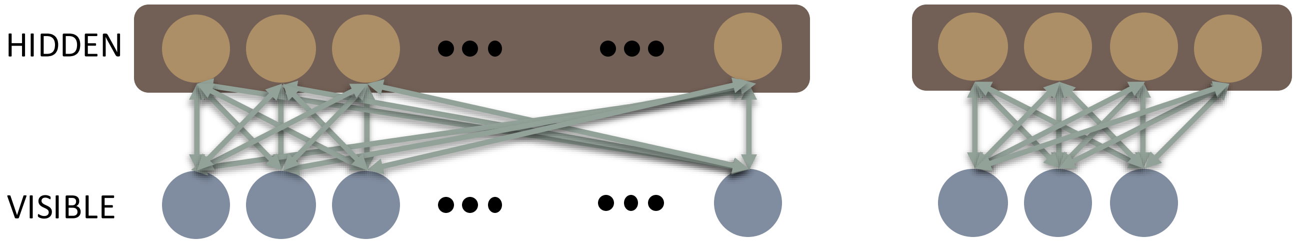
$$w_{ij} = w_{ij} + \eta \frac{d < \log P(\mathbf{S}) >}{dw_{ij}}$$

- Initialize weights
- Run simulations to get clamped and unclamped training sample
- Compute gradient and update
- Iteration

Boltzmann machines

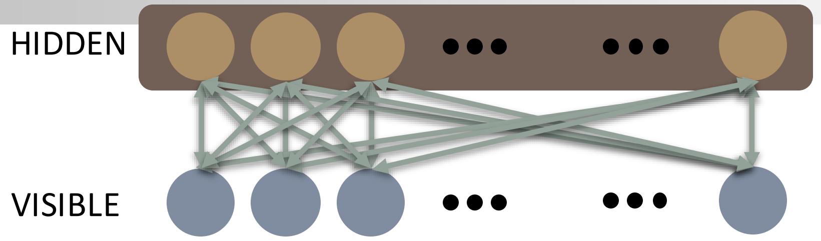
- Stochastic extension of Hopfields nets
- Enables storage of many more patterns than Hopfield nets
- But also enables computation of probabilities of patterns, and completion of pattern
- Applications
  - Filling out patterns
  - Denoising patterns
  - Computing conditional probabilities of patterns
  - Classification
- Drawbacks: forever training time, and doesn’t really work for large problems (a small number of training instances over a small number of bits)

Solution: Restricted Boltzmann Machines



- Partition visible and hidden units
  - Visible units ONLY talk to hidden units
  - Hidden units ONLY talk to visible units
- Restricted Boltzmann machine (Paul Smolensky)

Restricted Boltzmann Machines



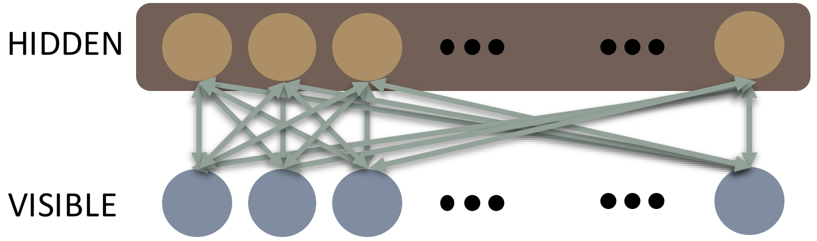
$$z_i = \sum_{i \neq j} w_j s_j + b_i \qquad P\left(s_i = 1 \middle| s_{j \neq i}\right) = \frac{1}{1 + e^{-z_i}}$$

- Still obeys the same rules as a regular Boltzmann machine
- But the modified structure adds a big benefit

HIDDEN $$z_i = \sum_{i \neq j} w_{ji} v_i + b_i \qquad P(h_i = 1) = \frac{1}{1 + e^{-z_i}}$$

VISIBLE $$y_i = \sum_{i \neq j} w_{ji} h_i + b_i \qquad P(v_i = 1) = \frac{1}{1 + e^{-y_i}}$$

Restricted Boltzmann Machines



$$z_i = \sum_{i \neq j} w_{ji} v_i + b_i$$

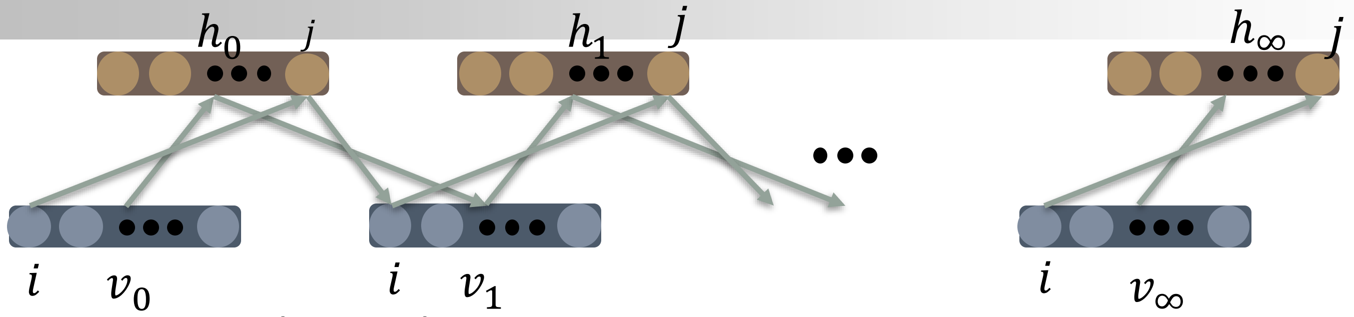
$$P(h_i = 1) = \frac{1}{1 + e^{-z_i}}$$

$$y_i = \sum_{i \neq j} w_{ji} h_i + b_i$$

$$P(v_i = 1) = \frac{1}{1 + e^{-y_i}}$$

- For each sample:
  - Iteratively sample hidden and visible units for a long time
  - Draw final sample of both hidden and visible units

Pictorial representation of RBM training



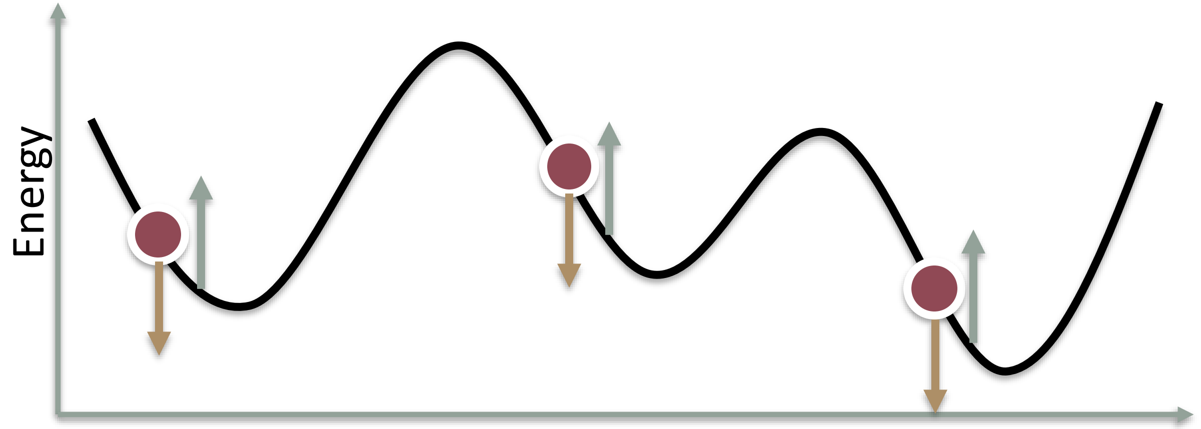
- For each sample:
  - Initialize V_0 (visible) to training instance value
  - Iteratively generate hidden and visible units
    - For a very long time
- Gradient (showing only one edge from visible node i to hidden node j)

$$\frac{d < \log p(v) >}{dw_{ij}} = < v_i \ , h_j \ >^0 - < v_i \ , h_j \ >^\infty$$

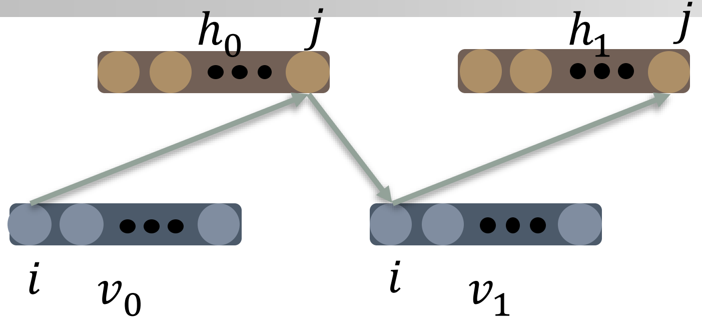
- $< v_i \ , h_j \ >$ represents average over many generated training samples

Recall: Efficient training

- Really no need to raise the entire surface, or even every valley
- Raise the neighborhood of each target memory (Sufficient to make the memory a valley, and the broader the neighborhood considered, the broader the valley



A shortcut: Contrastive Divergence



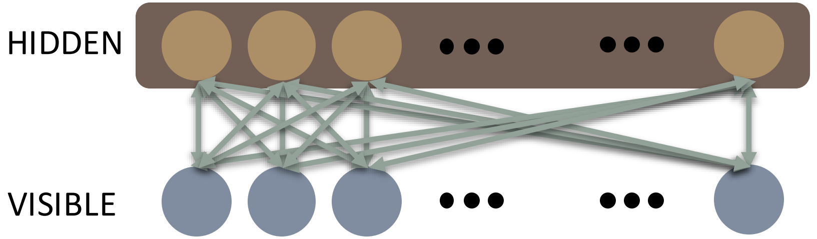
- Sufficient to run one iteration!

$$\frac{d < \log p(v) >}{dw_{ij}} = < v_i \ , h_j \ >^0 - < v_i \ , h_j \ >^1$$

This is sufficient to give you a good estimate of the gradient

Restricted Boltzmann Machines

- Excellent generative models for binary data
- Can also be extended to continuous-valued data
- Use for classification and regression
- Continuous-values RBMs



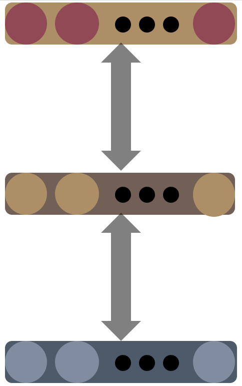
$$z_i = \sum_{i \neq j} w_{ji} v_i + b_i$$

$$P(h_i = 1) = \frac{1}{1 + e^{-z_i}}$$

$$y_i = \sum_{i \neq j} w_{ji} h_i + b_i$$

$$P(v_i) = \lambda e^{-\lambda y_i}, y_i \geq 0$$

Deep Boltzmann Machines



- The potential of learning internal representations that become increasingly complex (speech recognition problem)
- High-level representations can be built from a large supply of unlabeled sensory inputs and very limited labeled data can be used to only slightly fine-tune the model for a specific task
- The approximate inference procedure can incorporate top-down feedback (unlike deep belief networks)

Discussion

- There are other algorithms for learning and inference over RBMs such as Mean field approximations
- RBMs as feature extractors: Pre training
- More structured BDMs and RBMs