Neural Networks

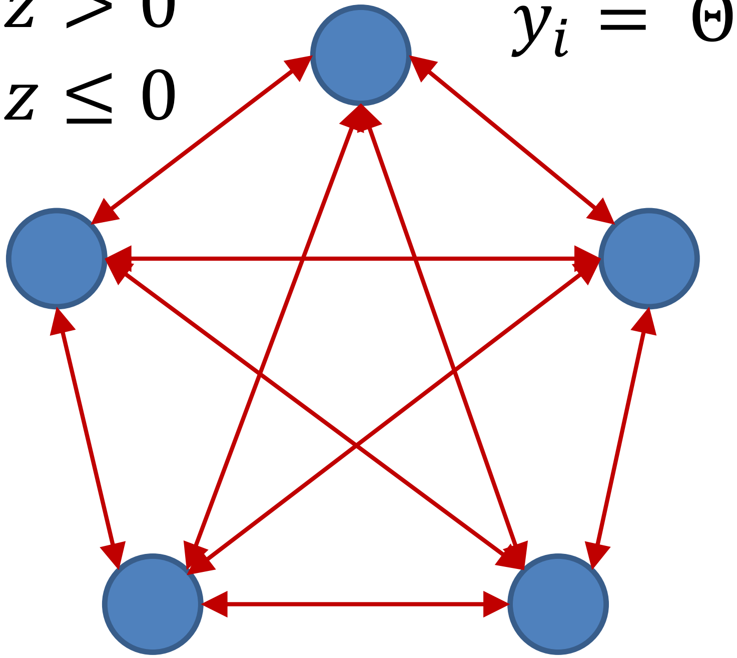
Hopfield Nets and Boltzmann Machines

Spring 2018

Recap: Hopfield network

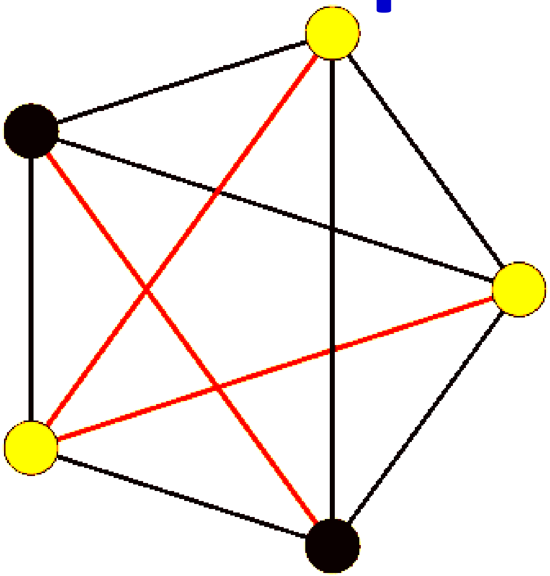
$$\Theta(z) = \begin{cases} +1 \; if \; z > 0 \\ -1 \; if \; z \leq 0 \end{cases}$$

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



- ***Symmetric loopy network***
- Each neuron is a perceptron with +1/-1 output

Recap: Hopfield network

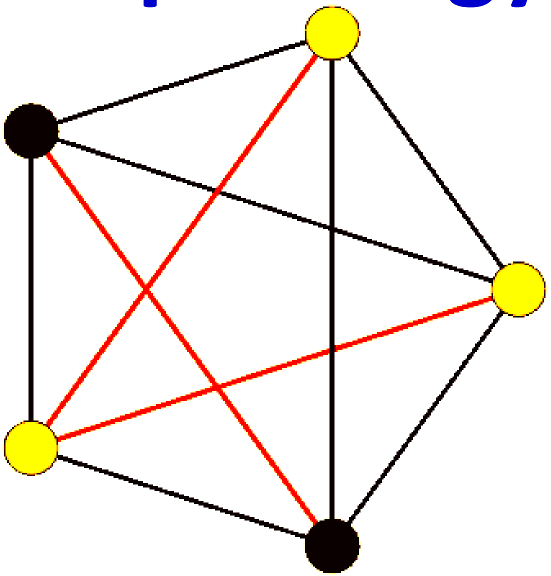


$$y_i = \Theta\left(\sum_{j \neq i} w_{ji} y_j + b_i\right)$$

$$\Theta(z) = \begin{cases} +1 \; if \; z > 0 \\ -1 \; if \; z \leq 0 \end{cases}$$

- At each time each neuron receives a “field” $\sum_{j \neq i} w_{ji} y_j + b_i$
- 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

Recap: Energy of a Hopfield Network



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

$$\Theta(z) = \begin{cases} +1 \; if \; z > 0 \\ -1 \; if \; z \leq 0 \end{cases}$$

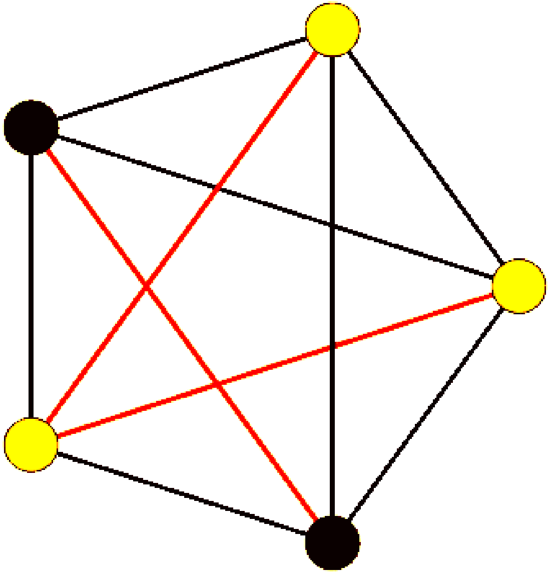
Not assuming node bias

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

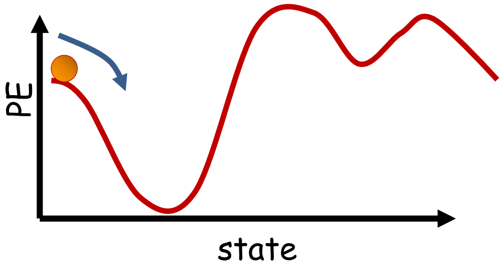
- The system will evolve until the energy hits a local minimum
- In vector form, including a bias term (not typically used in Hopfield nets)

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

Recap: Evolution

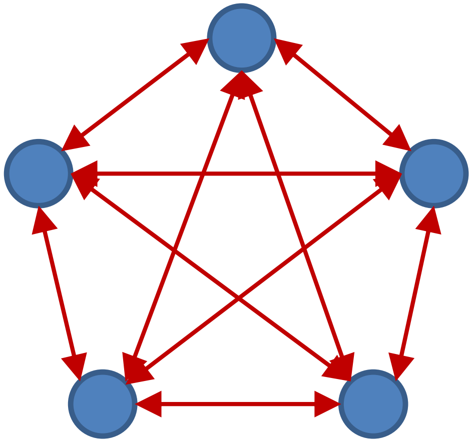


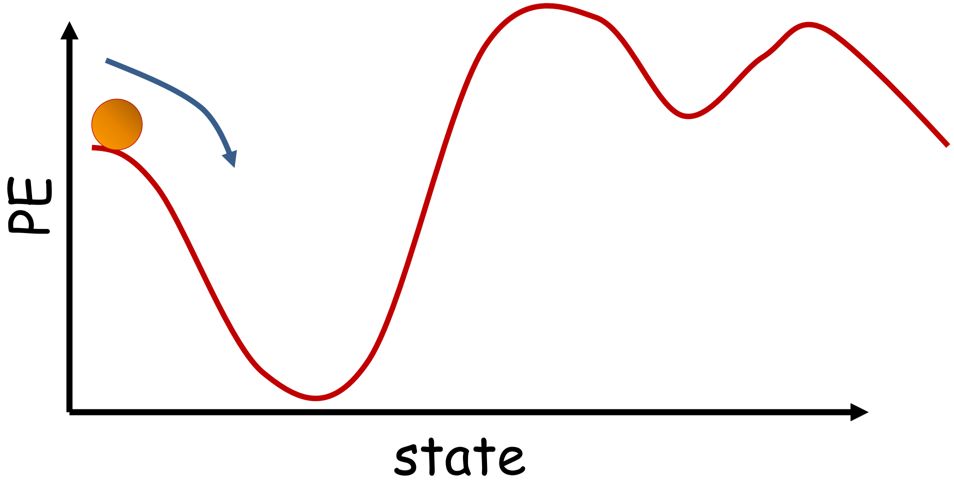
$$E = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y}$$



- The network will evolve until it arrives at a local minimum in the energy contour

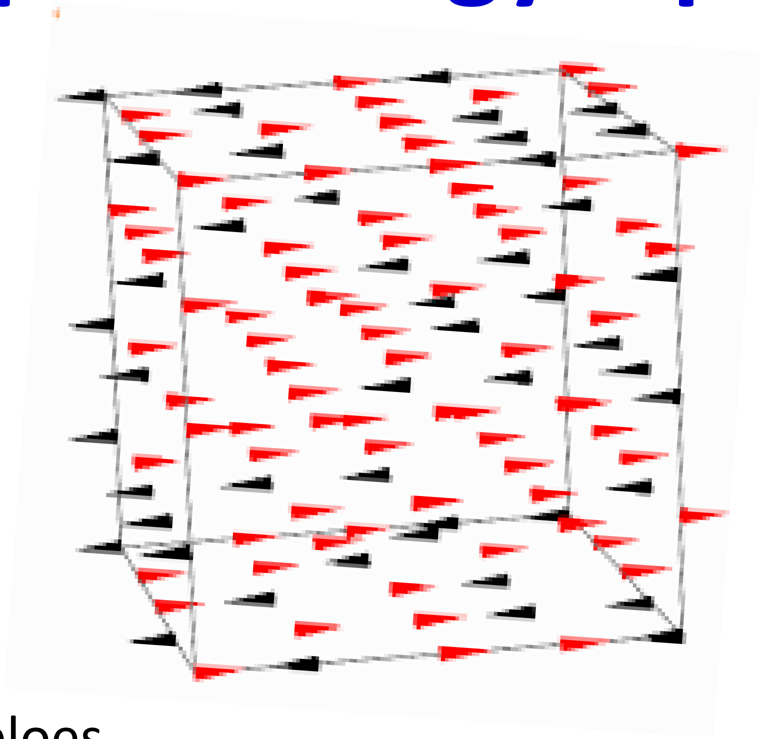
Recap: Content-addressable memory





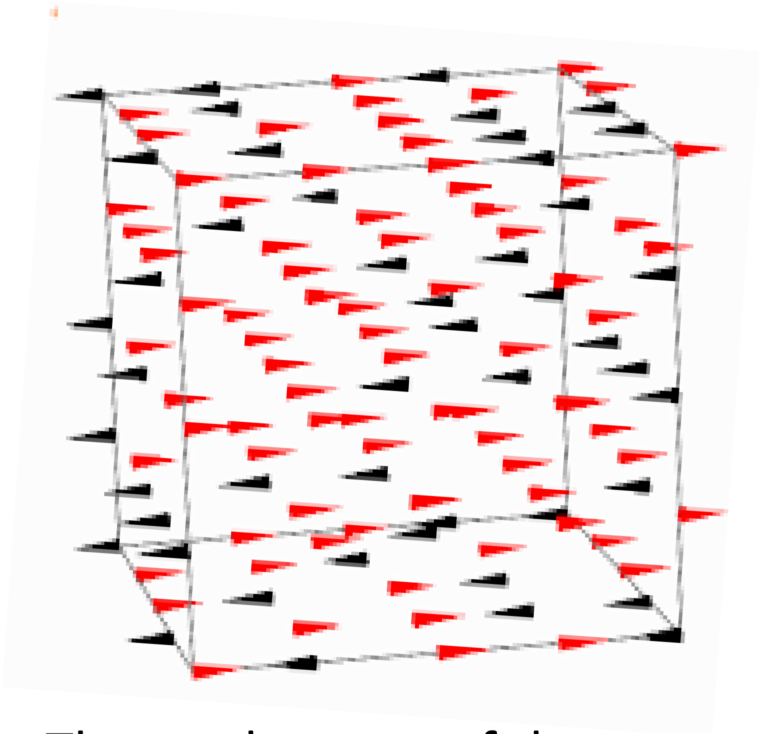
- Each of the minima is a "stored" pattern
  - If the network is initialized close to a stored pattern, it will inevitably evolve to the pattern
- **This is a *content addressable memory***
  - Recall memory content from partial or corrupt values
- Also called ***associative memory***

Recap – Analogy: Spin Glasses



- Magnetic diploes
- 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…

Recap – Analogy: Spin Glasses



Total field at current dipole:

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

Response of current diplose

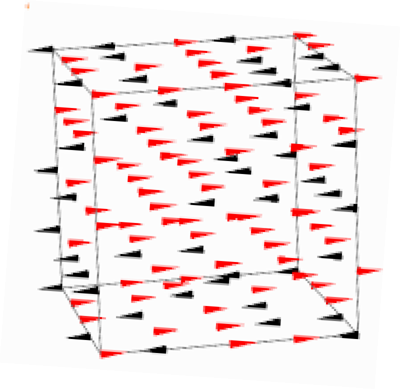
$$x_i = \begin{cases} x_i \ if \ sign\left(x_i \ f(p_i)\right) = 1 \\ -x_i \ otherwise \end{cases}$$

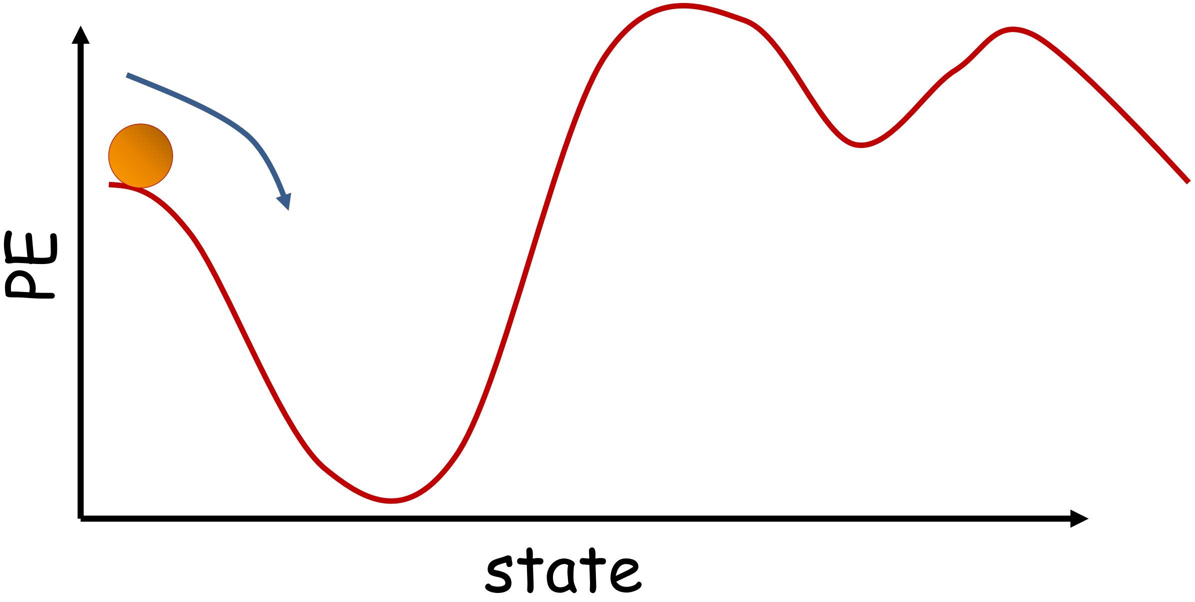
- The total energy of the system

$$E(s) = C - \frac{1}{2} \sum_i x_i f(p_i) = -\sum_i \sum_{j>i} J_{ij} x_i x_j - \sum_i b_i x_j$$

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

Recap : Spin Glasses





- 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 and returns to it

Recap: Hopfield net computation

1. Initialize network with initial pattern

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

2. Iterate until convergence

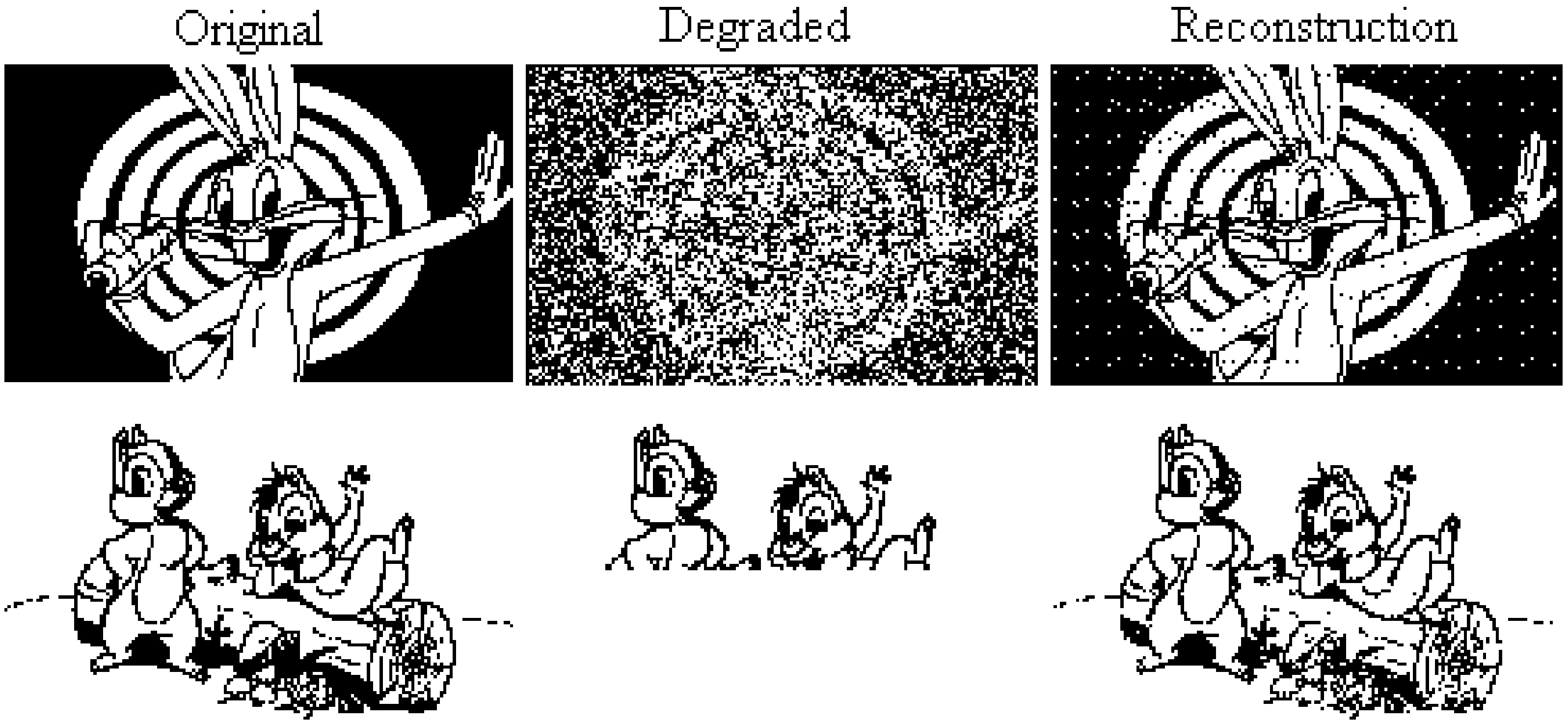
$$y_i(t+1) = \Theta\left(\sum_{j \neq i} w_{ji} y_j\right), \qquad 0 \leq i \leq N-1$$

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

$$E = -\sum_i \sum_{j>i} w_{ji} y_j y_i$$

does not change significantly any more

Examples: Content addressable memory



Hopfield network reconstructing degraded images from noisy (top) or partial (bottom) cues.

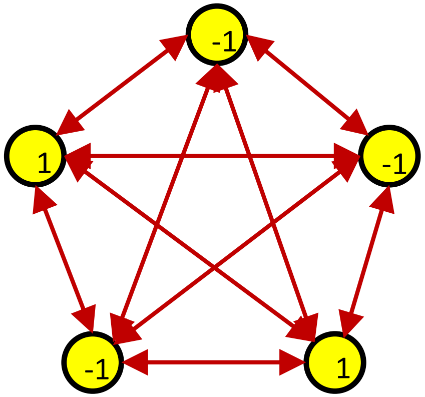
- http://staff.itee.uq.edu.au/janetw/cmc/chapters/Hopfield/

“Training” the network

- How do we make the network store *a specific* pattern or set of patterns?
  - Hebbian learning
  - Geometric approach
  - Optimization

- Secondary question
  - How many patterns can we store?

Recap: Hebbian Learning to Store a Specific Pattern



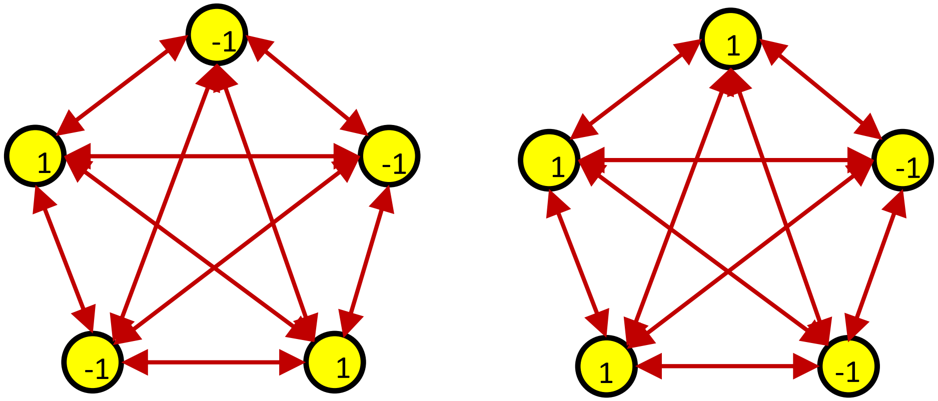
HEBBIAN LEARNING:

$$w_{ji} = y_j y_i$$

$$\mathbf{W} = \mathbf{y}_p \mathbf{y}_p^T - \mathrm{I}$$

- For a single stored pattern, Hebbian learning results in a network for which the target pattern is a global minimum

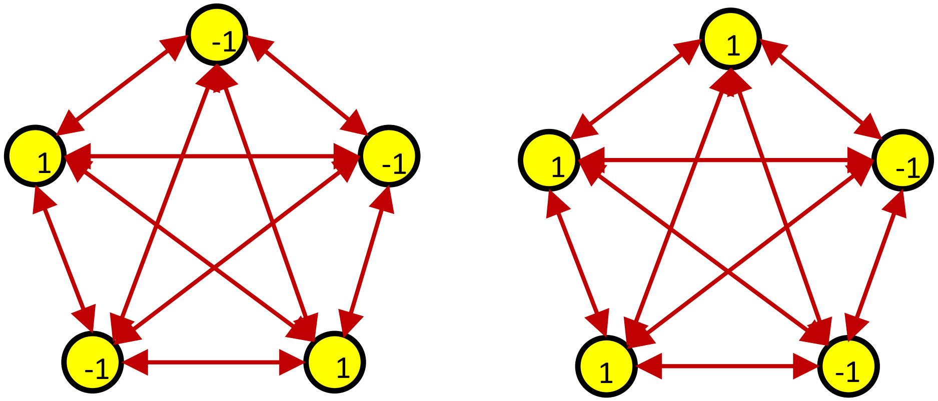
Storing multiple patterns



$$w_{ji} = \sum_{p\in\{y_p\}} y_i^p y_j^p$$

- $\{y_p\}$ is the set of patterns to store
- Superscript p represents the specific pattern

Storing multiple patterns

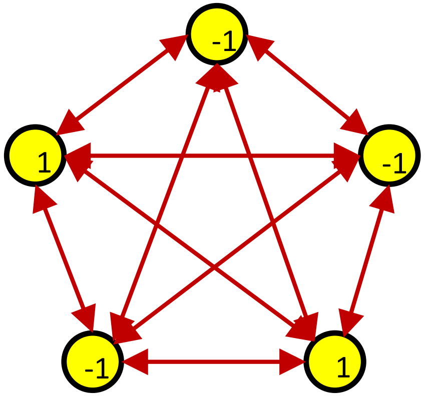


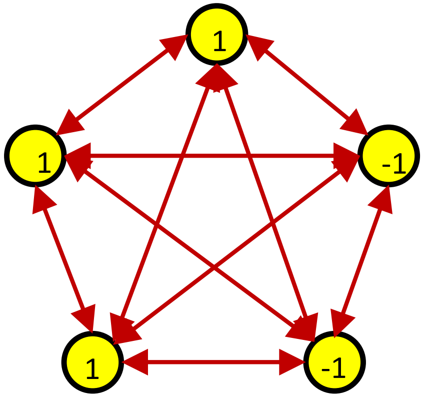
- Let $\mathbf{y}_p$ be the vector representing p-th pattern
- Let $\mathbf{Y} = [\mathbf{y}_1\ \mathbf{y}_2\ \ \dots]$ be a matrix with all the stored pattern
- Then..

$$\mathbf{W} = \sum_{\boldsymbol{p}} (\mathbf{y}_p \mathbf{y}_p^T - \mathrm{I}) = \mathbf{Y}\mathbf{Y}^T - N_p \mathbf{I}$$

Number of patterns

Recap: Hebbian Learning to Store Multiple Patterns



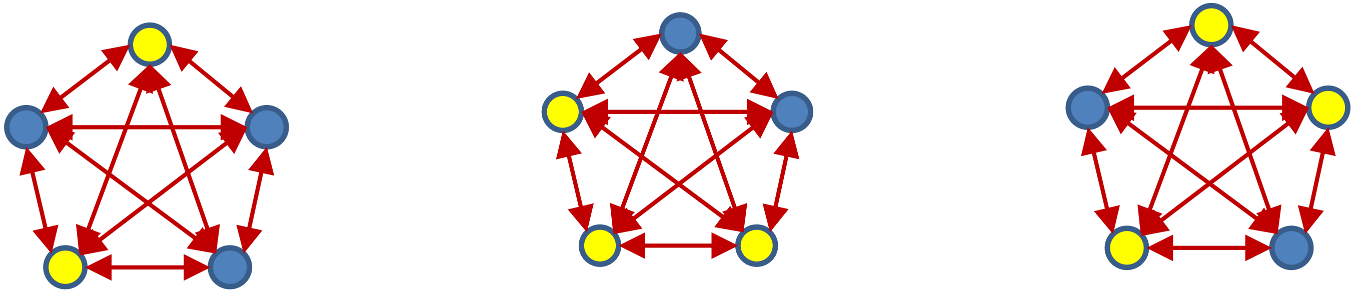


$$\boxed{w_{ji} = \sum_{p \in \{p\}} y_i^p y_j^p}$$

$$\boxed{\mathbf{W} = \sum_p \left(\mathbf{y}_p \mathbf{y}_p^T - \mathbf{I}\right) = \mathbf{Y}\mathbf{Y}^T - N_p \mathbf{I}}$$

- {p} is the set of patterns to store
  - – Superscript p represents the specific pattern
- N_p is the number of patterns to store

How many patterns can we store?



- Hopfield: For a network of N neurons can store up to $0.14N$ patterns
- In reality, seems possible to store K > 0.14N patterns
  – i.e. obtain a weight matrix W such that K > 0.14N patterns are stationary

Bold Claim

- I can *always* store (upto) N orthogonal patterns such that they are stationary!
  - Although not necessarily stable

- Why?

“Training” the network

- How do we make the network store *a specific* pattern or set of patterns?
  - Hebbian learning
  - Geometric approach
  - Optimization

- Secondary question
  - How many patterns can we store?

A minor adjustment

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

$$\mathbf{W} = \mathbf{Y}\mathbf{Y}^T - N_p\mathbf{I}$$

- Is identical to behavior with

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

- Since

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

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

Energy landscape
only differs by
an additive constant

Gradients and location
of minima remain same

A minor adjustment

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

$$\mathbf{W} = \mathbf{Y}\mathbf{Y}^T - N_p\mathbf{I}$$

behavior with

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

Both have the same Eigen vectors

Energy landscape only differs by an additive constant

Gradients and location of minima remain same

- Since

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

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

A minor adjustment

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

$$\mathbf{W} = \mathbf{Y}\mathbf{Y}^T - N_p\mathbf{I}$$

Energy landscape only differs by an additive constant

Gradients and location of minima remain same

Both have the same Eigen vectors

behavior with

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

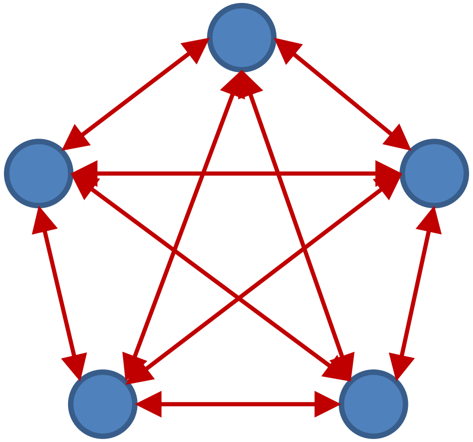
NOTE: This is a positive semidefinite matrix

- S

$$\mathbf{y}^T(\quad N_p\mathbf{I})\mathbf{y} = \mathbf{y}^T\mathbf{Y}\mathbf{Y}^T\mathbf{y} - NN_p$$

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

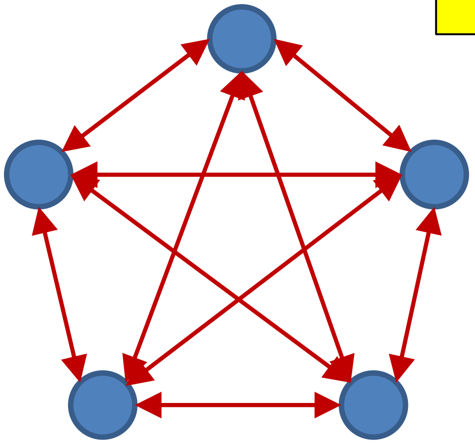
Consider the energy function



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

- Reinstating the bias term for completeness sake

Consider the energy function



This is a quadratic!

For Hebbian learning
W is positive semidefinite

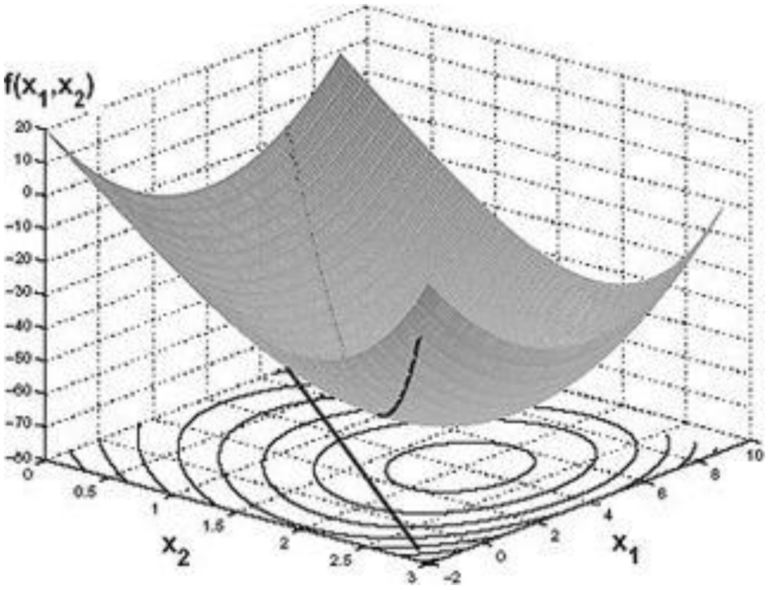
E is convex

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

- Reinstating the bias term for completeness sake

The energy function

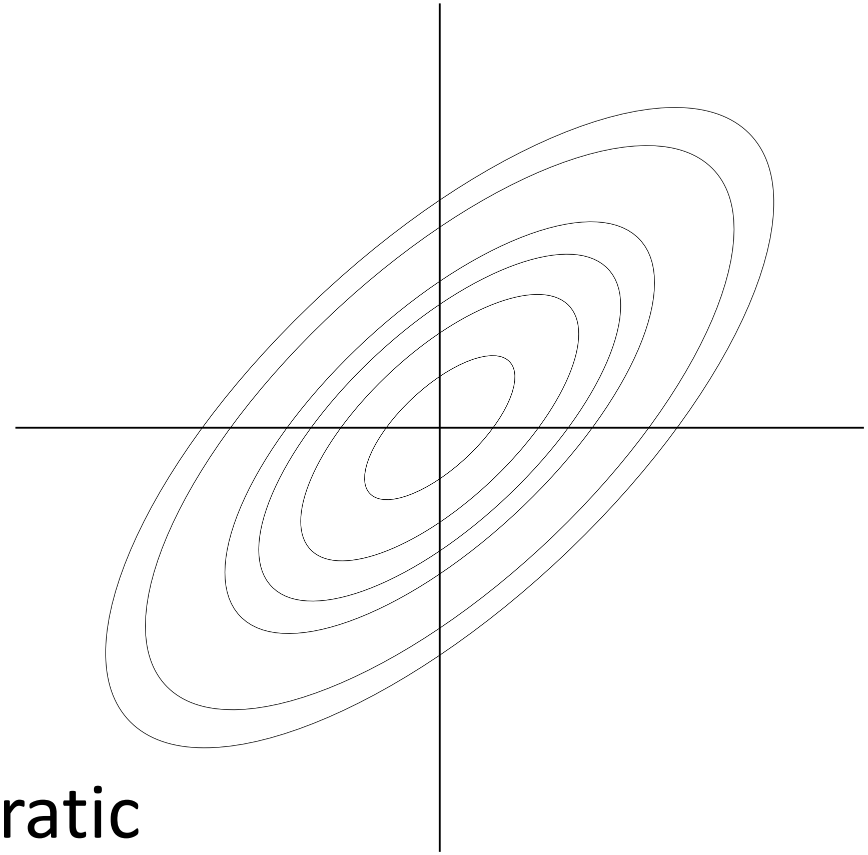
$$E = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$



- E is a convex quadratic

The energy function

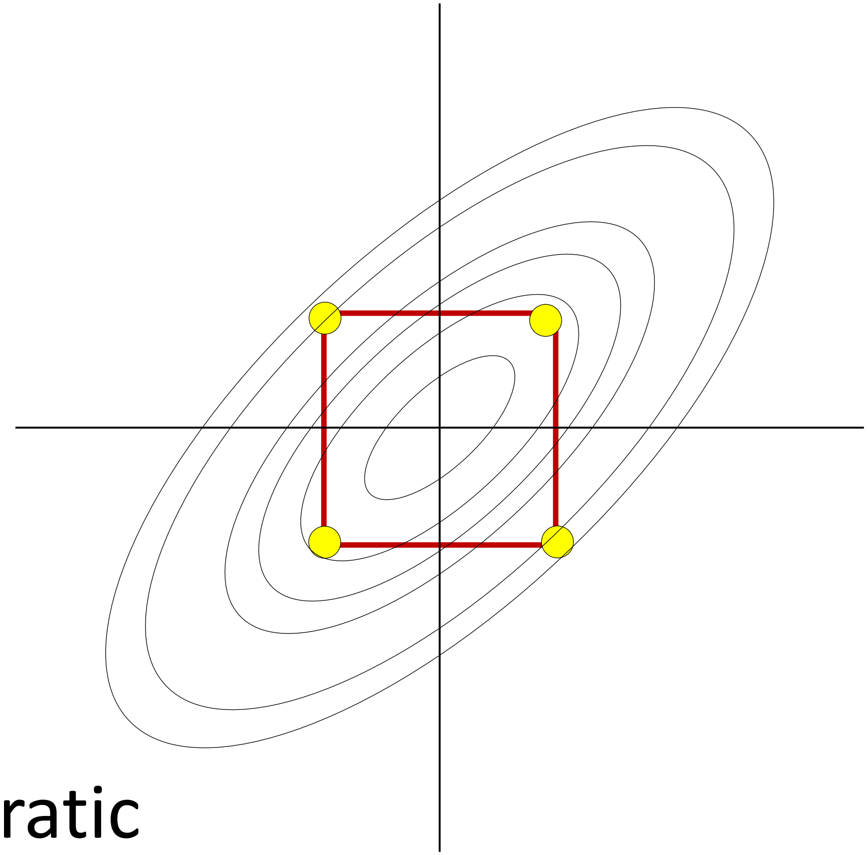
$$E = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$



- E is a convex quadratic
  - Shown from above (assuming 0 bias)

The energy function

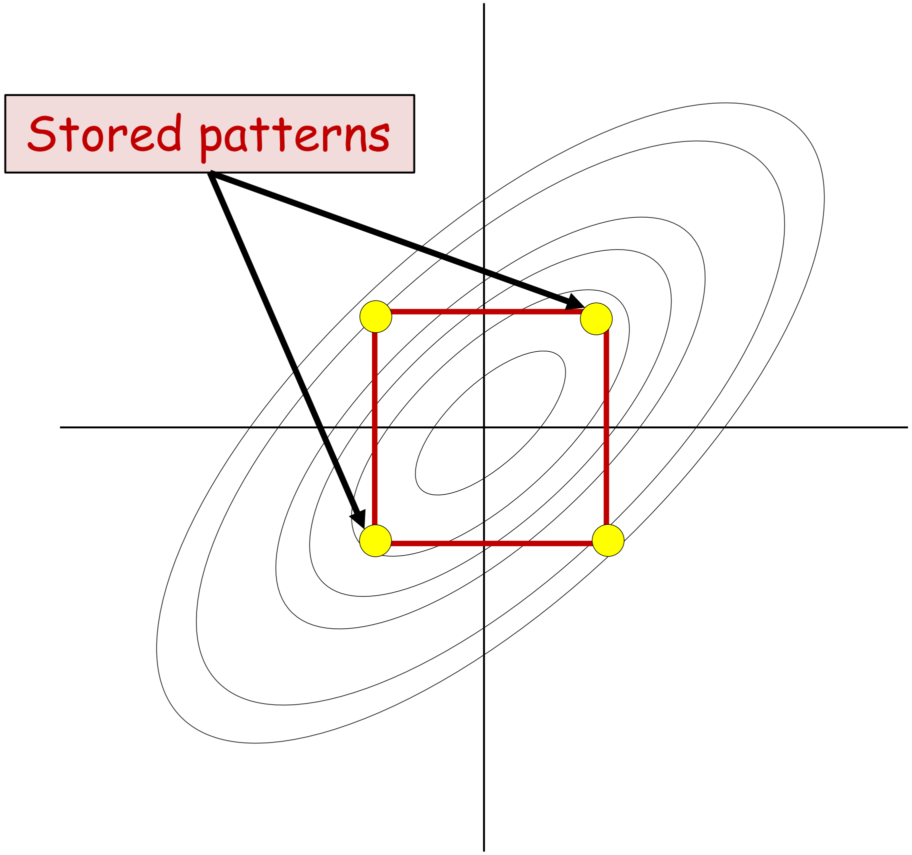
$$E = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$



- E is a convex quadratic
  - Shown from above (assuming 0 bias)
- But components of y can only take values ± 1
  - I.e y lies on the corners of the unit hypercube

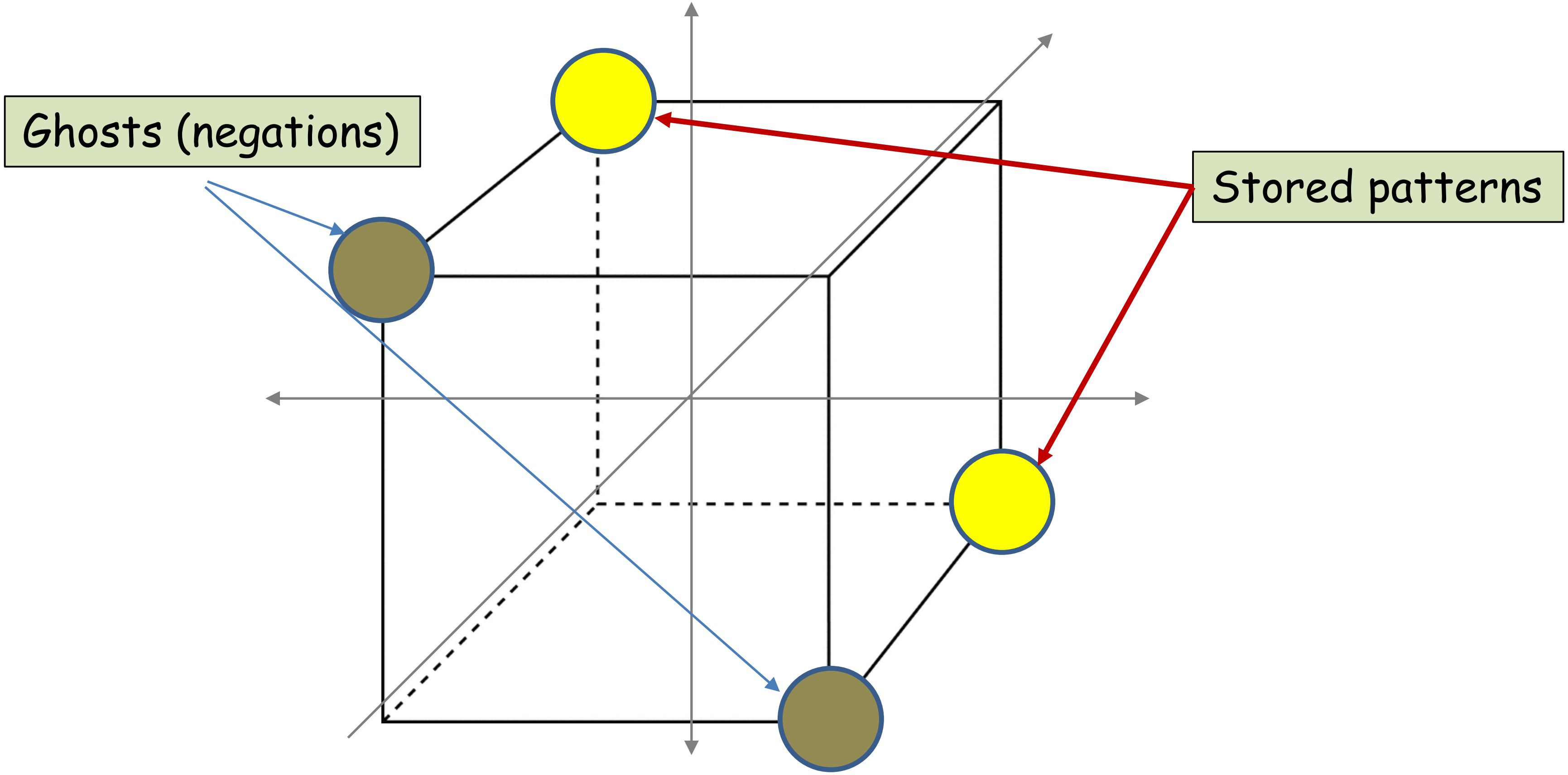
The energy function

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



- The stored values of $\mathbf{y}$ are the ones where all adjacent corners are lower on the quadratic

Patterns you can store

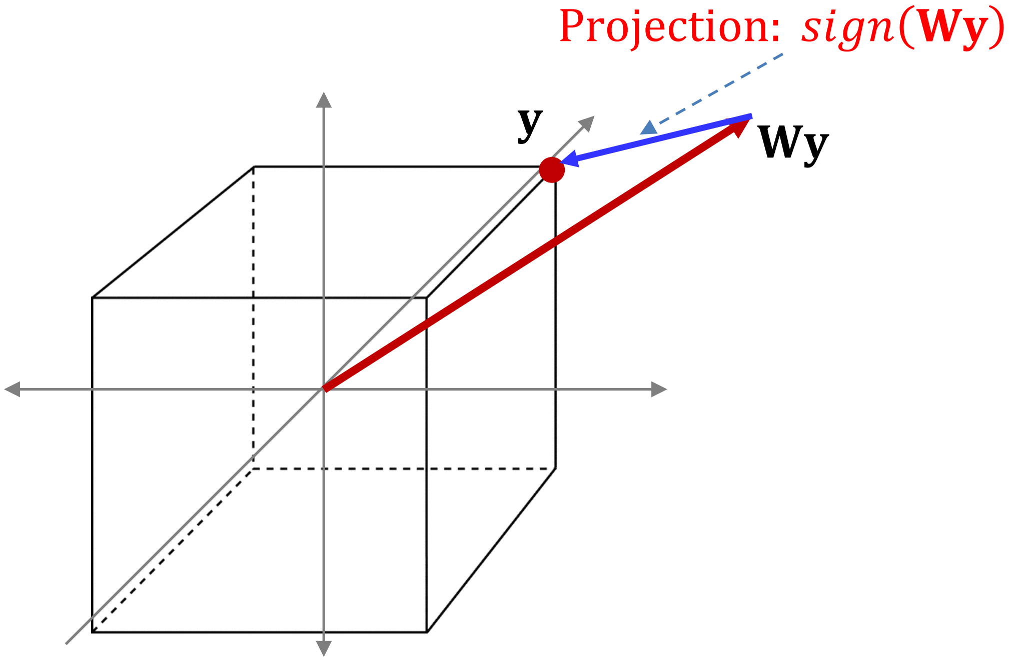


- All patterns are on the corners of a hypercube
  - If a pattern is stored, it's "ghost" is stored as well
  - Intuitively, patterns must ideally be maximally far apart
    - Though this doesn't seem to hold for Hebbian learning

Evolution of the network

- Note: for binary vectors $sign(\mathbf{y})$is a projection
  - Projects $\mathbf{y}$ onto the nearest corner of the hypercube
  - It "quantizes" the space into orthants

- Response to field: $\mathbf{y} \leftarrow sign(\mathbf{Wy})$
  - 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:
  - $sign(\mathbf{W}\mathbf{y}_p) = \mathbf{y}_p$ for all target patterns

- Training: Design $\mathbf{W}$ such that this holds

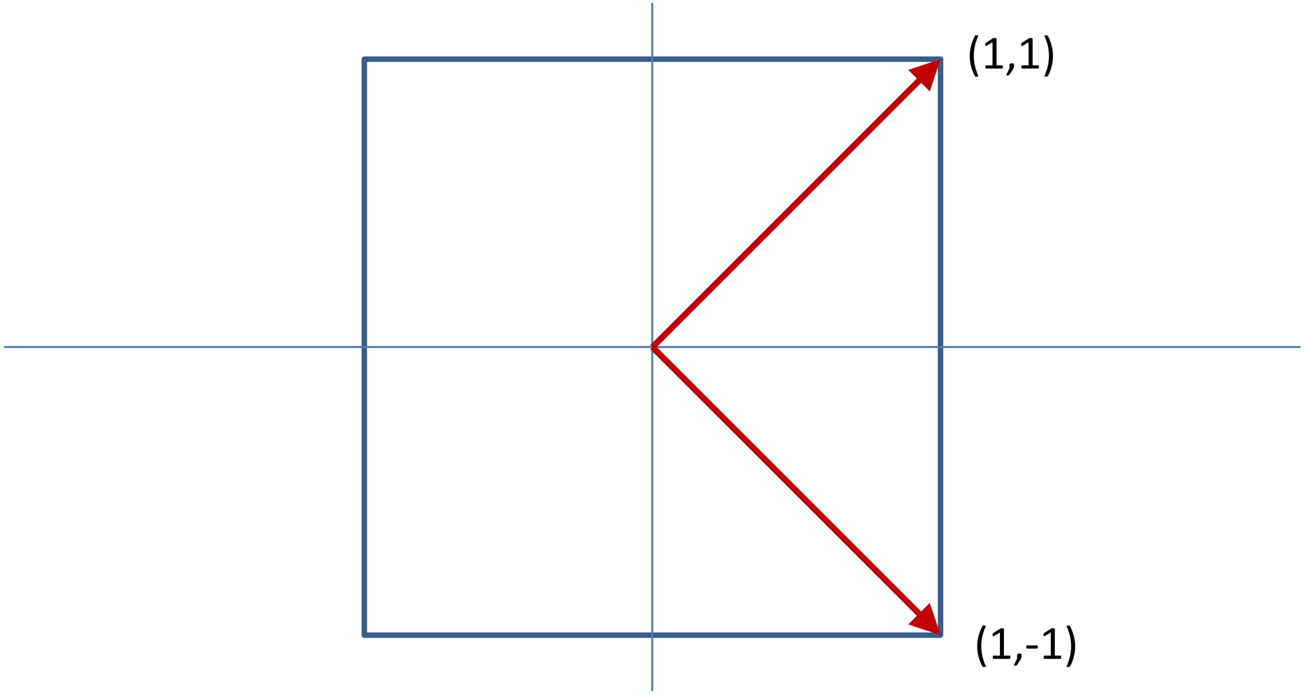
- Simple solution: $\mathbf{y}_p$ is an Eigenvector of $\mathbf{W}$
  - And the corresponding Eigenvalue is positive

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

  - More generally orthant($\mathbf{W}\mathbf{y}_p$) = orthant($\mathbf{y}_p$)

- How many such $\mathbf{y}_p$ can we have?

Only N patterns?



- Patterns that differ in $N/2$ bits are orthogonal
- You can have max N orthogonal vectors in an N-dimensional space

Another random fact that should interest you

- The Eigenvectors of any symmetric matrix $\mathbf{W}$ are orthogonal

- The Eigen*values* may be positive or negative

Storing more than one pattern

- Requirement: Given $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_P$
  - Design $\mathbf{W}$ such that
    - $sign(\mathbf{W}\mathbf{y}_p) = \mathbf{y}_p$ 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 $\mathbf{y}_1$, $\mathbf{y}_2, \ldots, \mathbf{y}_K$ are the Eigen vectors of $\mathbf{W}$
  - Let $\mathbf{Y} = [\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K]$

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

  - $\lambda_1, \ldots, \lambda_K$ are positive
  - For $\lambda_1 = \lambda_2 = \lambda_K = 1$ this is exactly the Hebbian rule
- The patterns are provably stationary

Hebbian rule

- In reality
  - Let $\mathbf{Y} = [\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K\ \mathbf{r}_{K+1}\ \mathbf{r}_{K+2} \ldots \mathbf{r}_N]$

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

  - $\mathbf{r}_{K+1}\ \mathbf{r}_{K+2} \ldots \mathbf{r}_N$ are orthogonal to $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$
  - $\lambda_1 = \lambda_2 = \lambda_K = 1$
  - $\lambda_{K+1}, \ldots, \lambda_N = 0$
- All patterns orthogonal to $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$ are also stationary
  - Although not stable

Storing N orthogonal patterns

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

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

  - $\lambda_1 = \lambda_2 = \lambda_N = 1$
- The Eigen vectors of $\mathbf{W}$ span the space
- Also, for any $\mathbf{y}_k$

$$\mathbf{W}\mathbf{y}_k = \mathbf{y}_k$$

Storing N orthogonal patterns

- The N orthogonal patterns $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N$ *span the space*
- 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 N patterns
  - Let $Y = [\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K\ \mathbf{r}_{\boldsymbol{K}+1}\ \mathbf{r}_{\boldsymbol{K}+2} \ldots \mathbf{r}_N]$

$$W = Y\Lambda Y^T$$

  - $\mathbf{r}_{\boldsymbol{K}+1}\ \mathbf{r}_{\boldsymbol{K}+2} \ldots \mathbf{r}_N$ are orthogonal to $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$
  - $\lambda_1 = \lambda_2 = \lambda_K = 1$
  - $\lambda_{K+1}, \ldots, \lambda_N = 0$
- All patterns orthogonal to $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$ are stationary
- Any pattern that is *entirely* in the subspace spanned by $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$is also stable (same logic as earlier)
- Only patterns that are *partially* in the subspace spanned by $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$ are unstable
  - Get projected onto subspace spanned by $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$

Problem with Hebbian Rule

- Even if we store fewer than N patterns
  - Let $Y = [\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K\ \mathbf{r}_{K+1}\ \mathbf{r}_{K+2} \ldots \mathbf{r}_N]$

$$W = Y \Lambda Y^T$$

  - $\mathbf{r}_{K+1}\ \mathbf{r}_{K+2} \ldots \mathbf{r}_N$ are orthogonal to $\mathbf{y}_1\ \mathbf{y}_2 \ldots \mathbf{y}_K$
  - $\lambda_1 = \lambda_2 = \lambda_K = 1$
- Problems arise because Eigen values are all 1.0
  - Ensures stationarity of vectors in the subspace
  - What if we get rid of this requirement?

Hebbian rule and general (non-orthogonal) vectors

$$w_{ji} = \sum_{p \in \{p\}} y_i^p y_j^p$$

- What happens when the patterns are *not* orthogonal
- What happens when the patterns are presented *more* than once
  - Different patterns presented different numbers of times
  - Equivalent to having unequal Eigen values..
- Can we predict the evolution of any vector $\mathbf{y}$
  - Hint: Lanczos iterations
    - Can write $\mathbf{Y}_P = \mathbf{Y}_{ortho}\mathbf{B}, \rightarrow \mathbf{W} = \mathbf{Y}_{ortho}\mathbf{B}\Lambda\mathbf{B}^T\mathbf{Y}_{ortho}^T$

The bottom line

- 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, 84’
  - 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 $K \leq N$
  - Mostafa and St. Jacques 85’
    - For large N, the upper bound on K is actually N/4logN
      - McElice et. Al. 87’
  - **But this may come with many “parasitic” memories**

The bottom line

- 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, 84’
  - E.g. when we had the H[illegible]use patterns, *all* patterns are stable

How do we find this network?

- For a *specific* set of K patterns, we can *always* build a network for which all K patterns are stable provided $K \leq N$
  - Mostafa and St. Jacques 85’
    - For large N, the upper bound on K is actually N/4logN
      - McElice et. Al. 87’
  - **But this may come with many “parasitic” memories**

The bottom line

- 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, 84’
  - E.g. when we had the H use patterns, *all* patterns are stable

How do we find this network?

- For a *specific* set of K patterns, we can *always* build a network for which all K patterns are stable provided $K \leq N$
  - Mostafa and St. Jacques 85’
    - For large N, the upper bound on K is actually N
      - McElice et. Al. 87’
  - **But this may come with many “parasitic” memories**

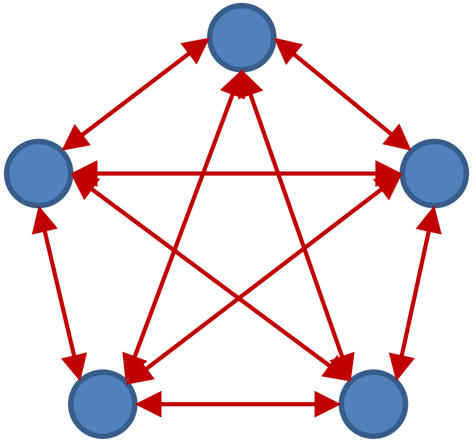
Can we do something about this?

A different tack

- How do we make the network store *a specific* pattern or set of patterns?
  - Hebbian learning
  - Geometric approach
  - Optimization

- Secondary question
  - How many patterns can we store?

Consider the energy function

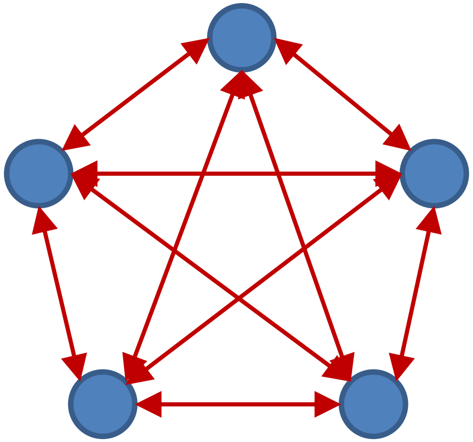


$$E = -\frac{1}{2}\mathbf{y}^T\mathbf{W}\mathbf{y} - \mathbf{b}^T\mathbf{y}$$

- 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

Alternate Approach to Estimating the Network

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



- 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 $\mathbf{y}$
- Caveat: Unrealistic to expect to store more than N patterns, but can we make those N patterns *memorable*

Optimizing W (and b)

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

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{\mathbf{y}\in\mathbf{Y}_P} E(\mathbf{y})$$

The bias can be captured by another fixed-value component

- Minimize total energy of target patterns
  - Problem with this?

Optimizing W

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

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{\mathbf{y}\in\mathbf{Y}_P} E(\mathbf{y}) - \sum_{\mathbf{y}\notin\mathbf{Y}_P} E(\mathbf{y})$$

- Minimize total energy of target patterns
- Maximize the total energy of all *non-target* patterns

Optimizing W

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

- Simple gradient descent:

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

Optimizing W

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

- Can “emphasize” the importance of a pattern by repeating
  - – More repetitions → greater emphasis

Optimizing W

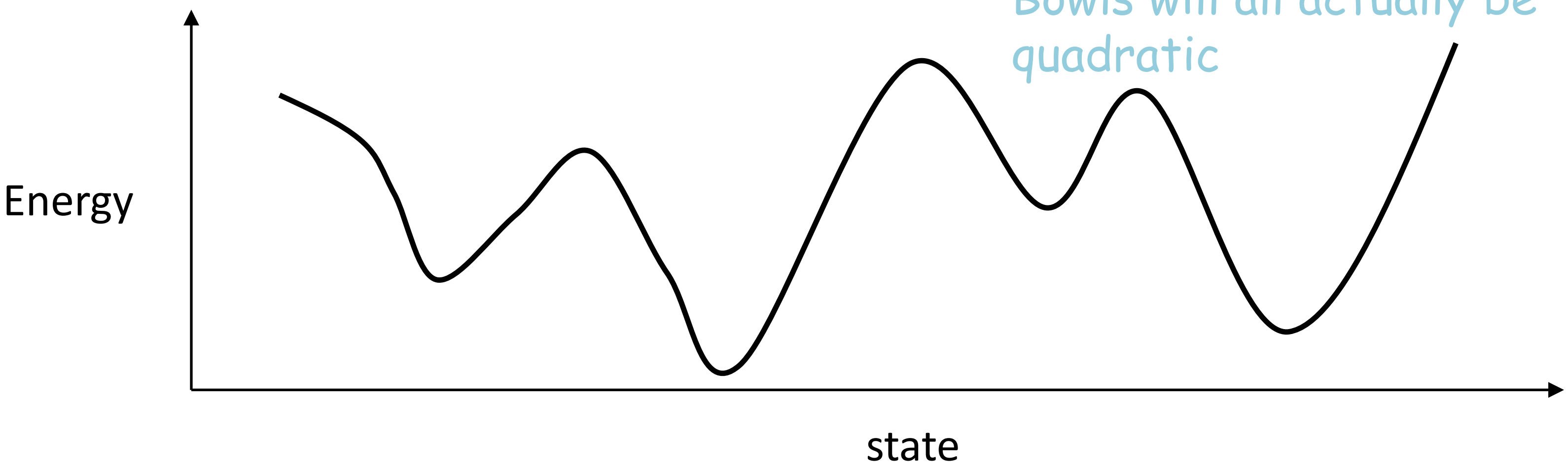
$$\mathbf{W} = \mathbf{W} + \eta\left(\sum_{\mathbf{y}\in\mathbf{Y}_P} \mathbf{y}\mathbf{y}^T - \sum_{\mathbf{y}\notin\mathbf{Y}_P} \mathbf{y}\mathbf{y}^T\right)$$

- Can “emphasize” the importance of a pattern by repeating
  - – More repetitions → greater emphasis
- How many of these?
  - – Do we need to include *all* of them?
  - – Are all equally important?

The training again..

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

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

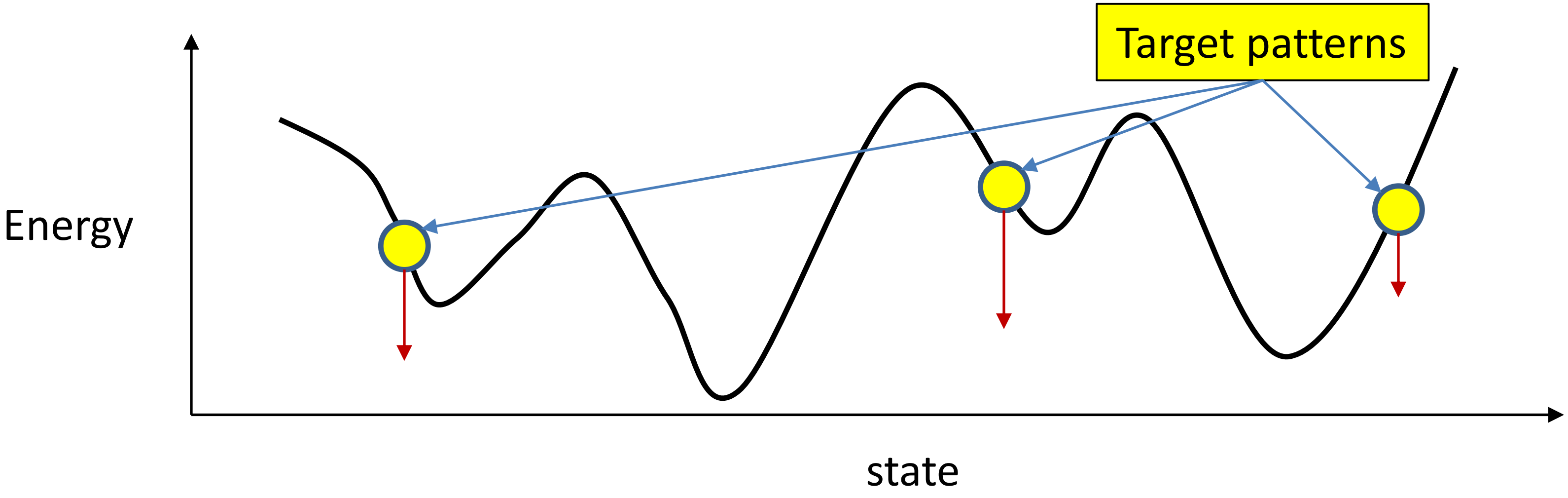


Bowls will all actually be quadratic

The training again

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

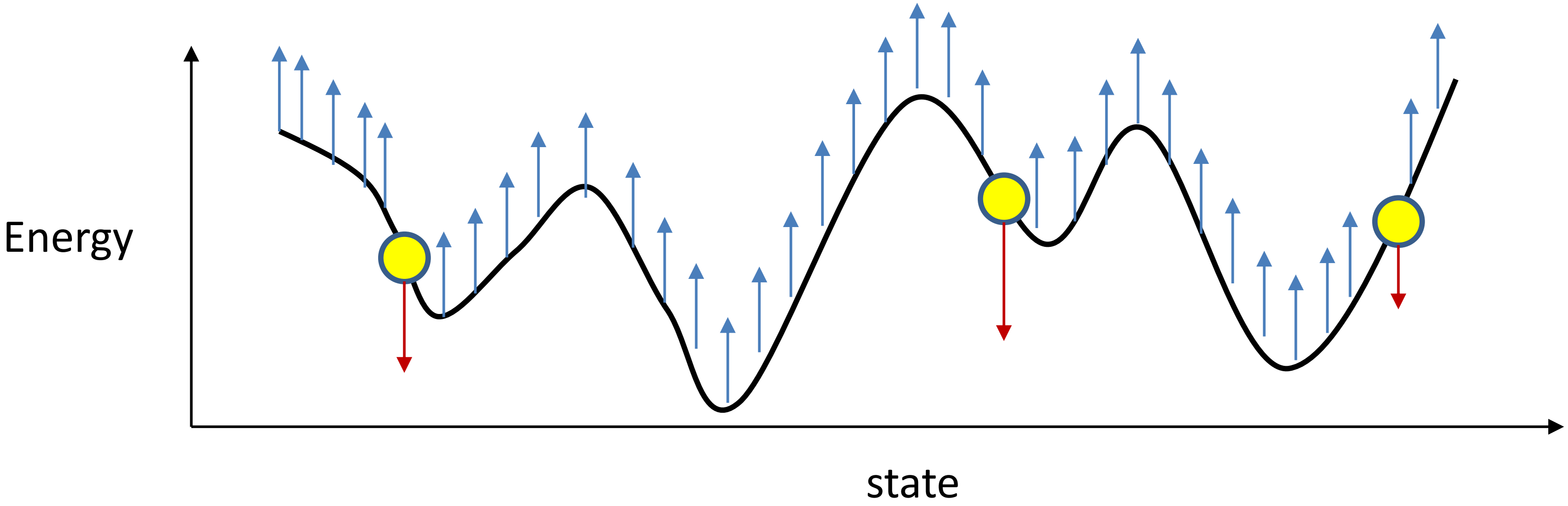
- The first term tries to *minimize* the energy at target patterns
  - Make them local minima
  - Emphasize more "important" memories by repeating them more frequently



The negative class

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

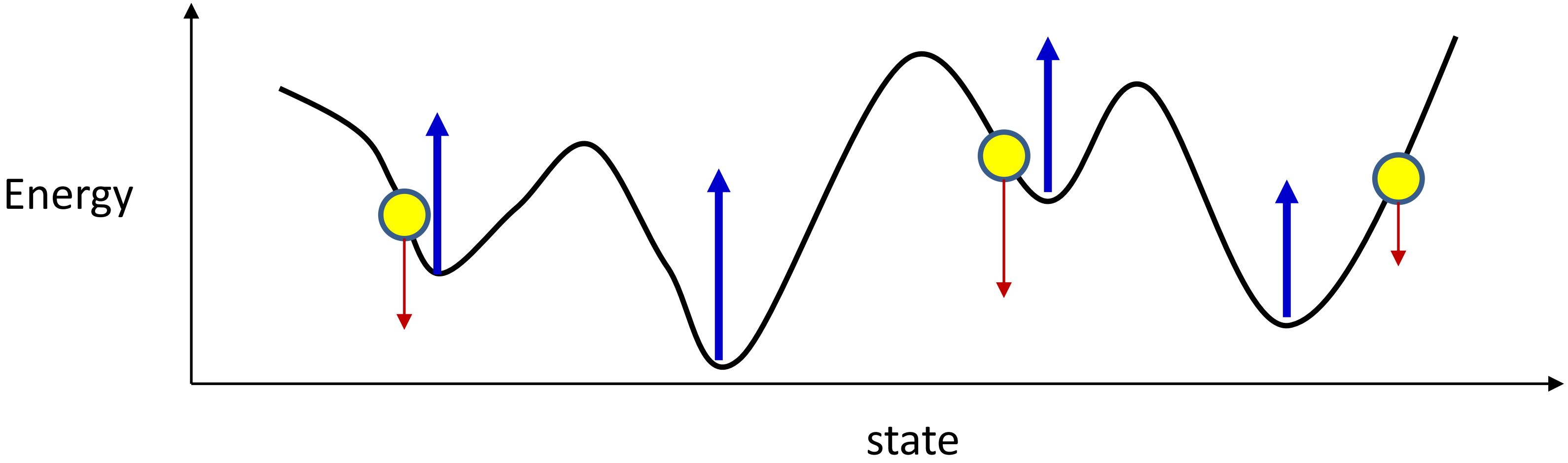
- The second term tries to "raise" all non-target patterns
  - Do we need to raise *everything*?



Option 1: Focus on the valleys

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

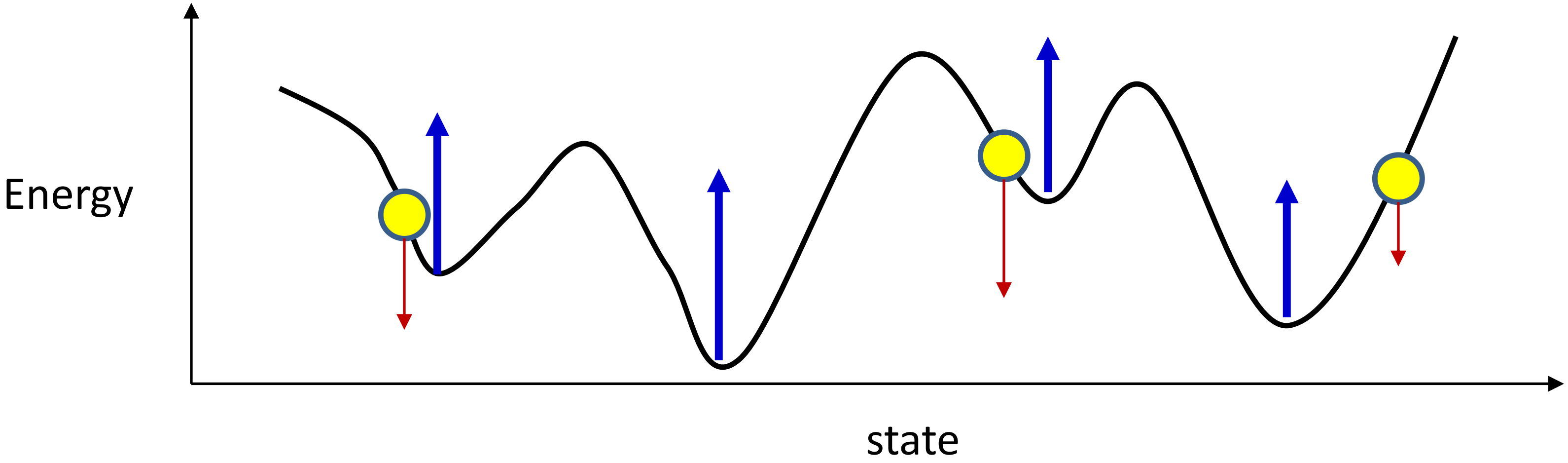
- Focus on raising the valleys
  - If you raise *every* valley, eventually they'll all move up above the target patterns, and many will even vanish



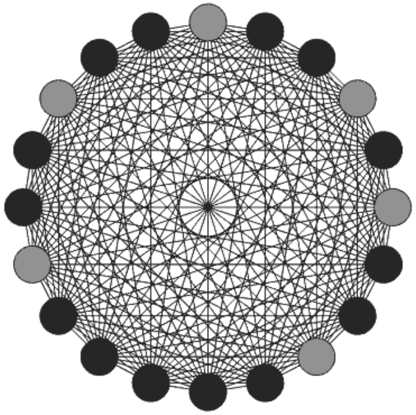
Identifying the valleys..

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

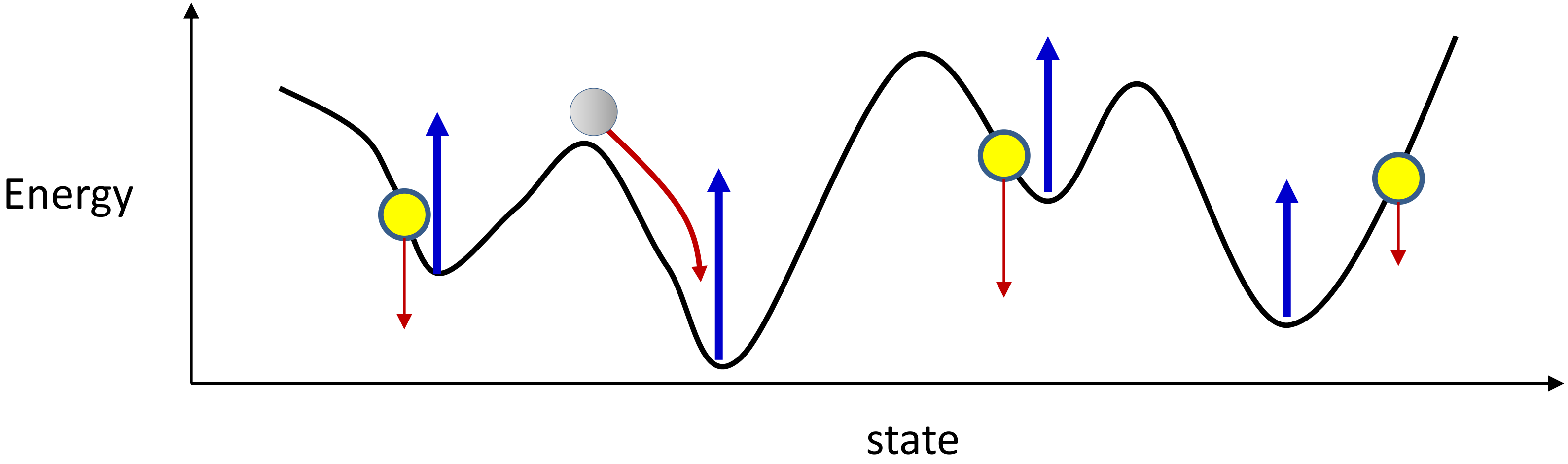
- Problem: How do you identify the valleys for the current $\mathbf{W}$?



Identifying the valleys..



- Initialize the network randomly and let it evolve
  - It will settle in a valley



Training the Hopfield network

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

- Initialize $\mathbf{W}$
- Compute the total outer product of all target patterns
  - More important patterns presented more frequently
- Randomly initialize the network several times and let it evolve
  - And settle at a valley
- Compute the total outer product of valley patterns
- Update weights

Training the Hopfield network: SGD version

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

- 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
  - Randomly initialize the network and let it evolve
    - And settle at a valley $\mathbf{y}_v$
  - Update weights
    - $\mathbf{W} = \mathbf{W} + \eta\left(\mathbf{y}_p\mathbf{y}_p^T - \mathbf{y}_v\mathbf{y}_v^T\right)$

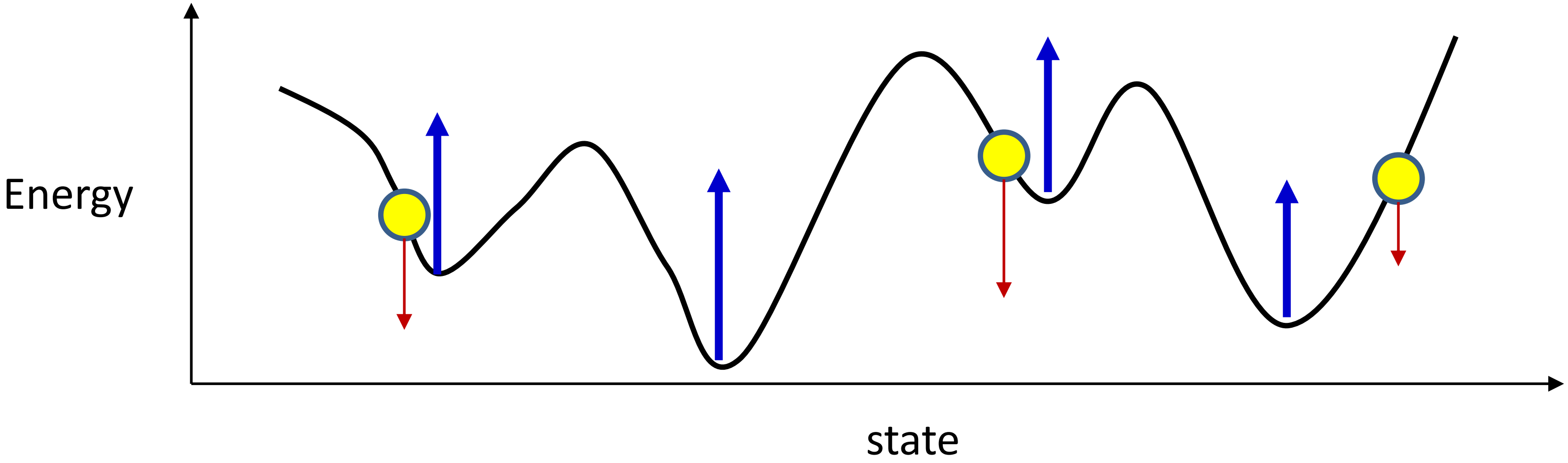
Training the Hopfield network

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

- 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
  - Randomly initialize the network and let it evolve
    - And settle at a valley $\mathbf{y}_v$
  - Update weights
    - $\mathbf{W} = \mathbf{W} + \eta\left(\mathbf{y}_p\mathbf{y}_p^T - \mathbf{y}_v\mathbf{y}_v^T\right)$

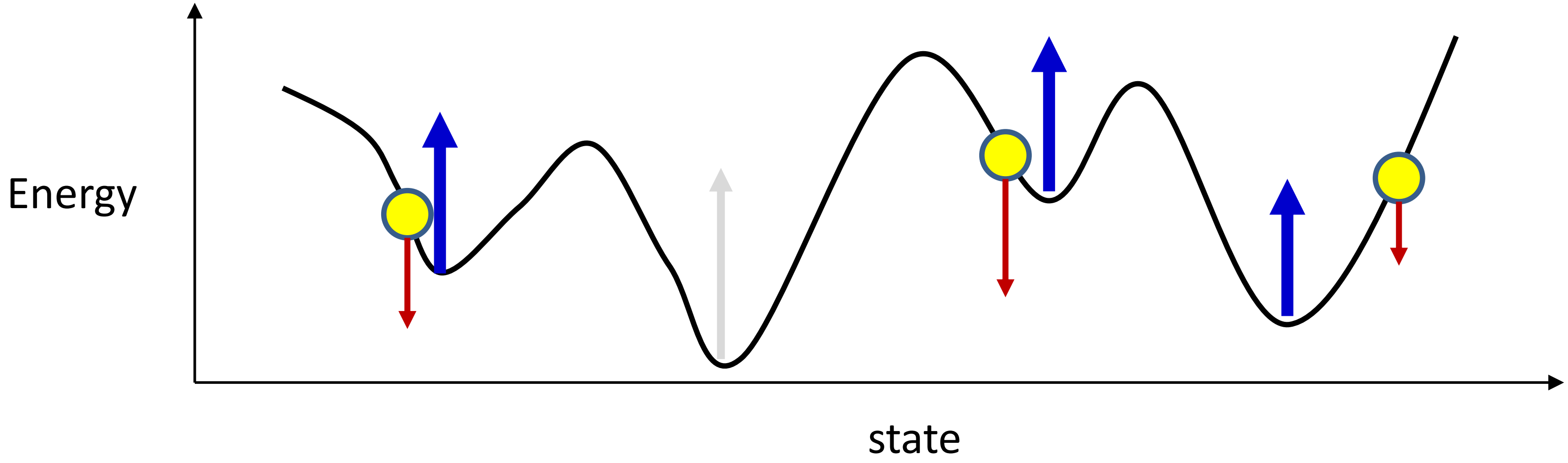
Which valleys?

- Should we *randomly* sample valleys?
  - Are all valleys equally important?

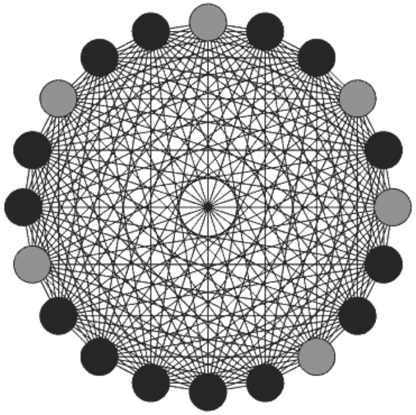


Which valleys?

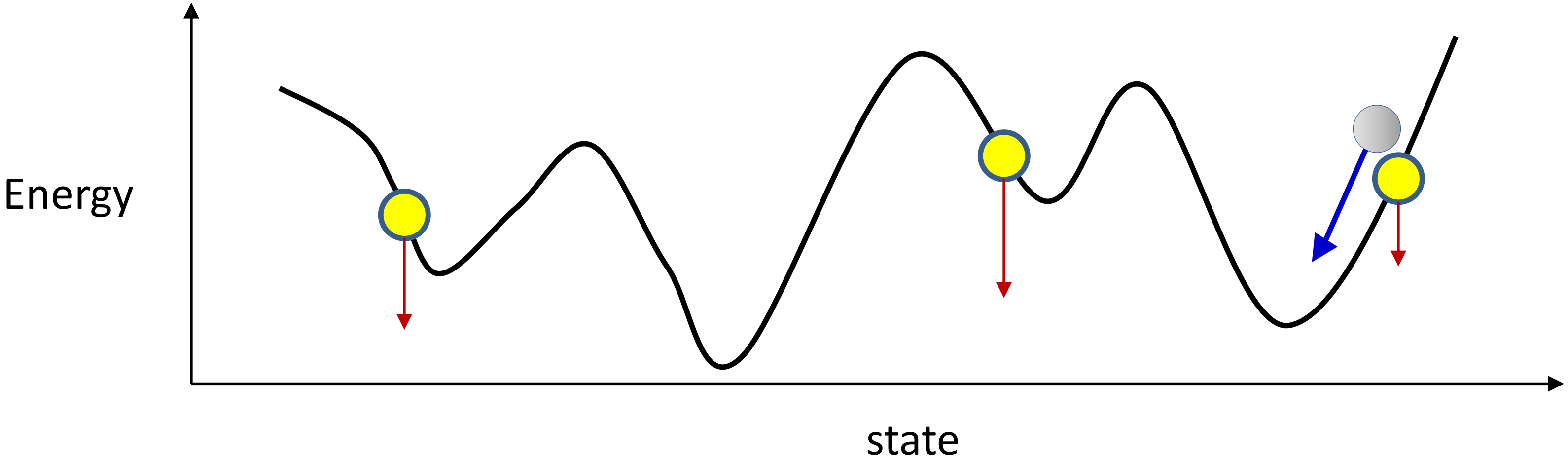
- Should we *randomly* sample valleys?
  - Are all valleys equally important?
- Major requirement: memories must be stable
  - They *must* be broad valleys
- Spurious valleys in the neighborhood of memories are more important to eliminate



Identifying the valleys..



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



Training the Hopfield network

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

- Initialize $\mathbf{W}$
- Compute the total outer product of all target patterns
  - More important patterns presented more frequently
- Initialize the network with each target pattern and let it evolve
  - And settle at a valley
- Compute the total outer product of valley patterns
- Update weights

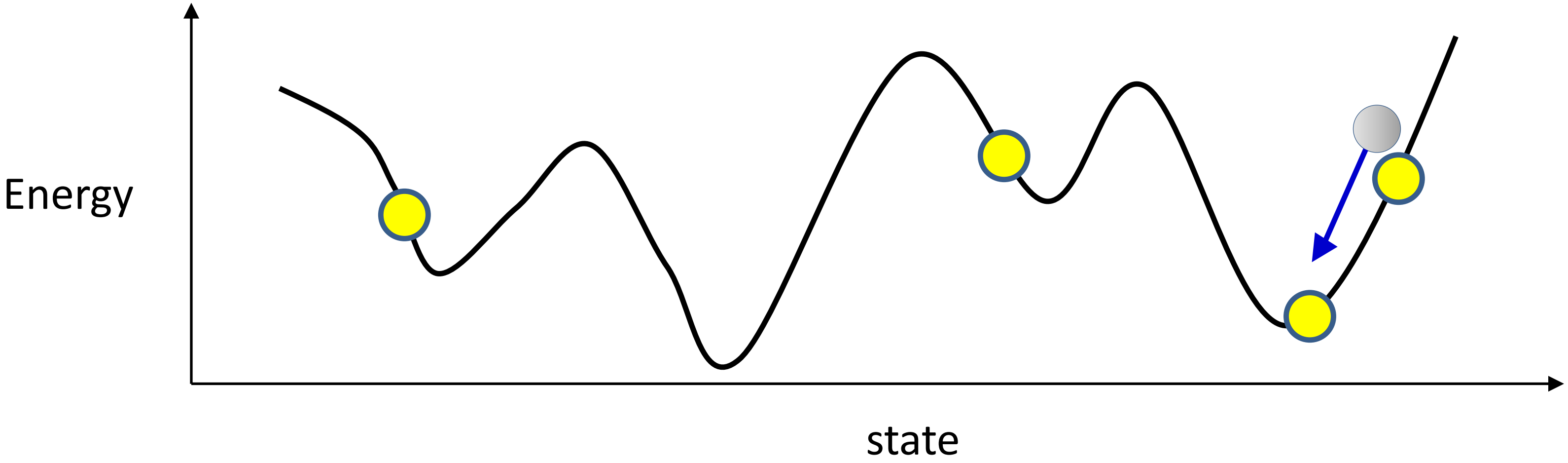
Training the Hopfield network: SGD version

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

- 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
    - And settle at a valley $\mathbf{y}_v$
  - Update weights
    - $\mathbf{W} = \mathbf{W} + \eta\left(\mathbf{y}_p\mathbf{y}_p^T - \mathbf{y}_v\mathbf{y}_v^T\right)$

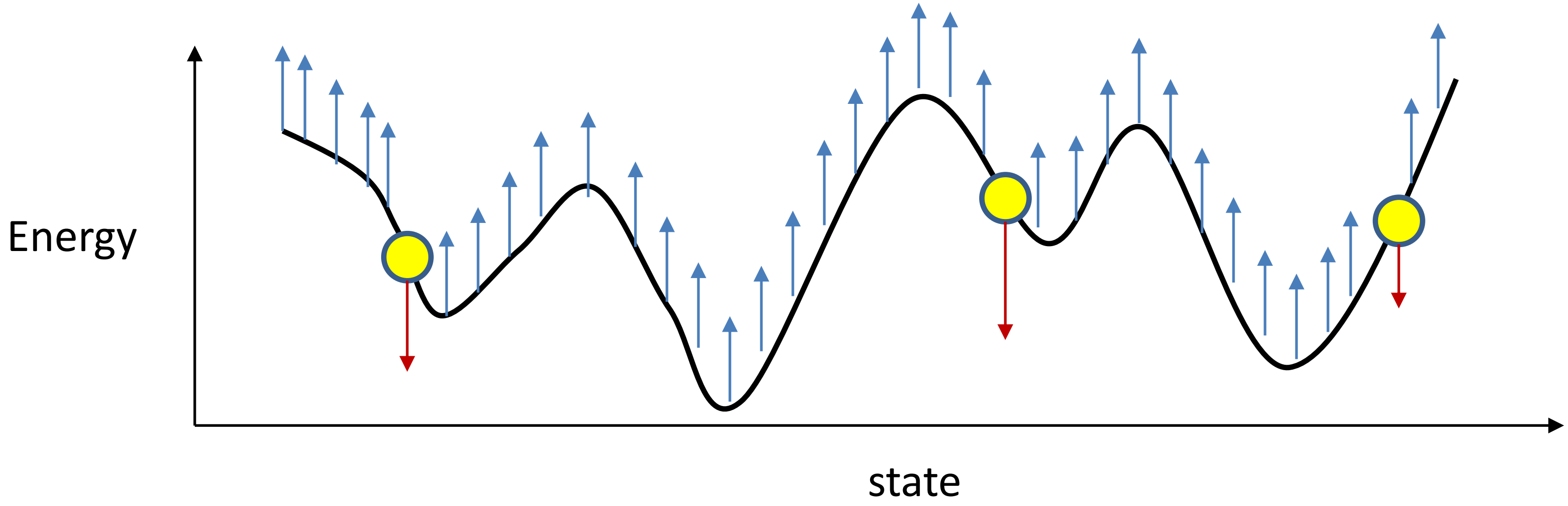
A possible problem

- What if there’s another target pattern downvalley
  - Raising it will destroy a better-represented or stored pattern!



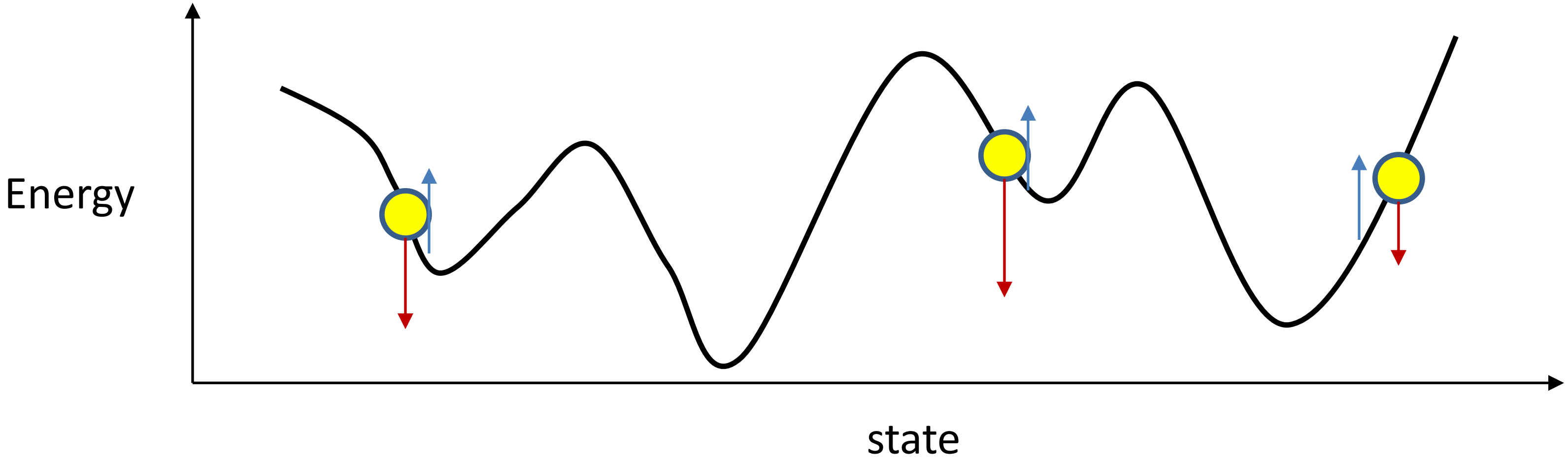
A related issue

- Really no need to raise the entire surface, or even every valley



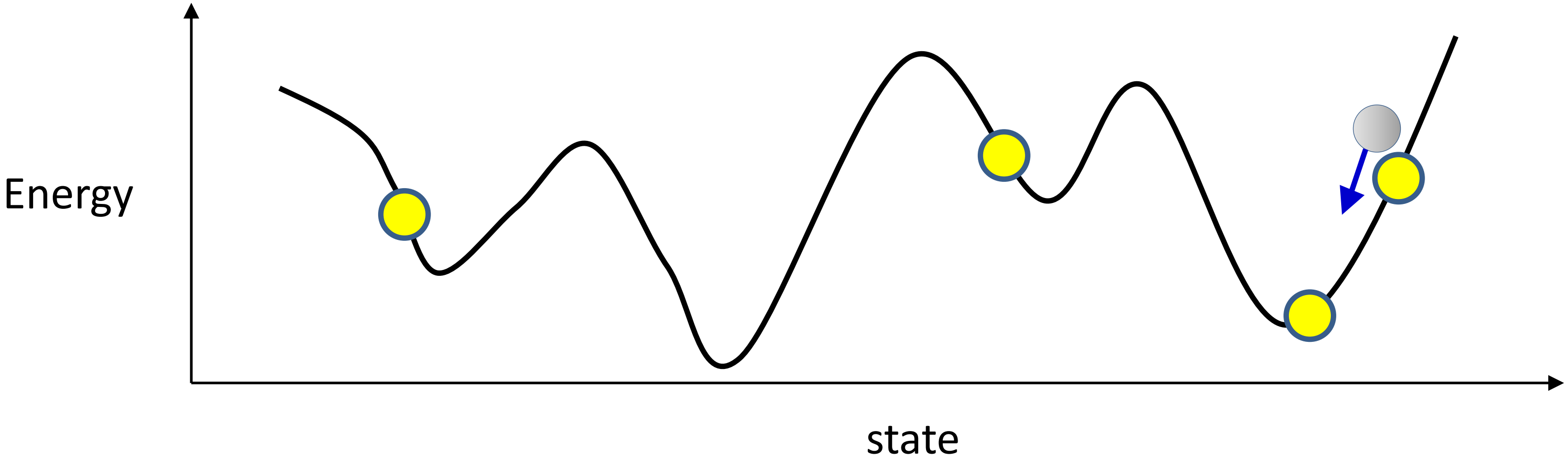
A related issue

- 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
  - The broader the neighborhood considered, the broader the valley



Raising the neighborhood

- Starting from a target pattern, let the network evolve only a few steps
  - Try to raise the resultant location
- Will raise the neighborhood of targets
- Will avoid problem of down-valley targets



Training the Hopfield network: SGD version

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

- 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)***
    - And arrive at a down-valley position $\mathbf{y}_d$
  - Update weights
    - $\mathbf{W} = \mathbf{W} + \eta\left(\mathbf{y}_p\mathbf{y}_p^T - \mathbf{y}_d\mathbf{y}_d^T\right)$

A probabilistic interpretation

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

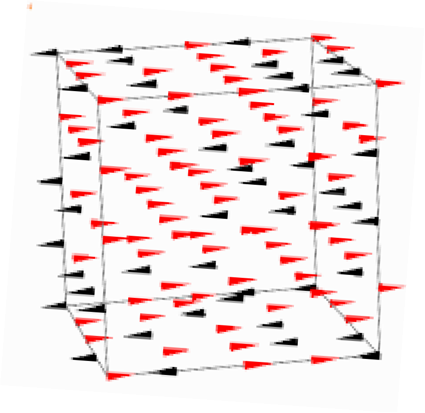
- For continuous $\mathbf{y}$, the *energy* of a pattern is a perfect analog to the *negative log likelihood* of a Gaussian density
- For *binary* y it is the analog of the negative log likelihood of a *Boltzmann distribution*
  - **Minimizing energy maximizes log likelihood**

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

The Boltzmann Distribution

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

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



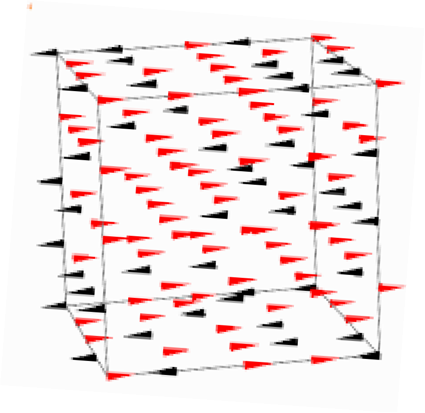
$$C = \frac{1}{\sum_{\mathbf{y}} P(\mathbf{y})}$$

- k is the Boltzmann constant
- T is the temperature of the system
- The energy terms are like the loglikelihood of a Boltzmann distribution at $T = 1$
  - Derivation of this probability is in fact quite trivial..

Continuing the Boltzmann analogy

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

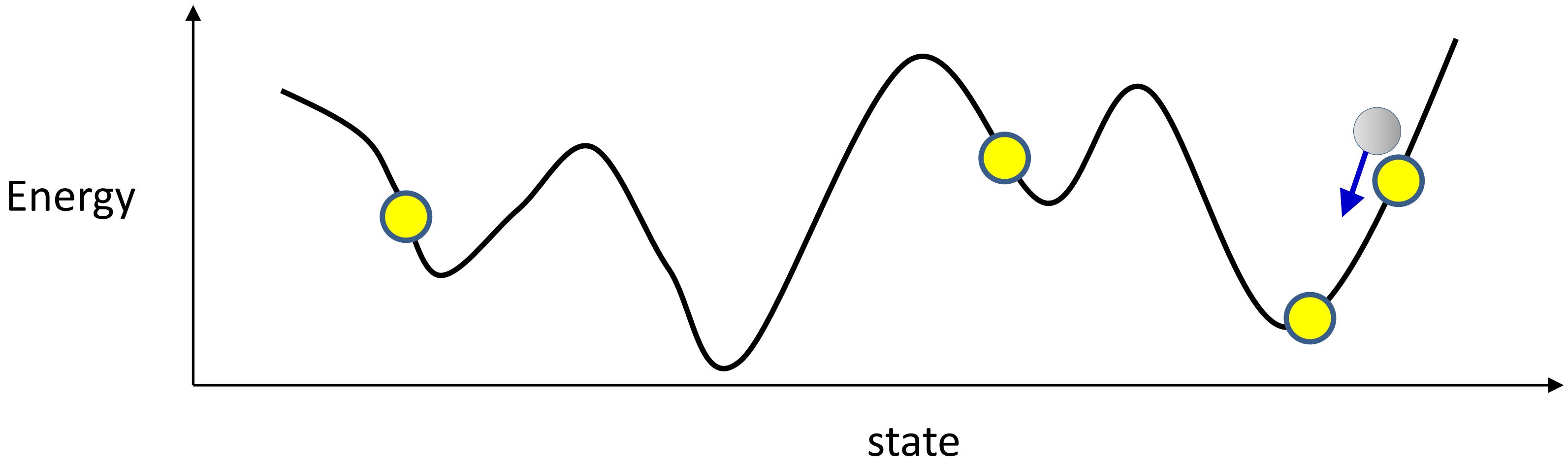
$$P(\mathbf{y}) = Cexp\left(\frac{-E(\mathbf{y})}{kT}\right)$$



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

- The system *probabilistically* selects states with lower energy
  - With infinitesimally slow cooling, at $T = 0$, it arrives at the global minimal state

Spin glasses and Hopfield nets



- Selecting a next state is akin to drawing a sample from the Boltzmann distribution at $T = 1$, in a universe where $k = 1$

Optimizing W

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

- 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

Optimizing W

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

- 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

THIS LOOKS LIKE AN EXPECTATION!

Optimizing W

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

- Update rule

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

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

Natural distribution for variables: The Boltzmann Distribution

Continuing on..

- The Hopfield net as a Boltzmann distribution
- Adding capacity to a Hopfield network
  - The Boltzmann machine

Continuing on..

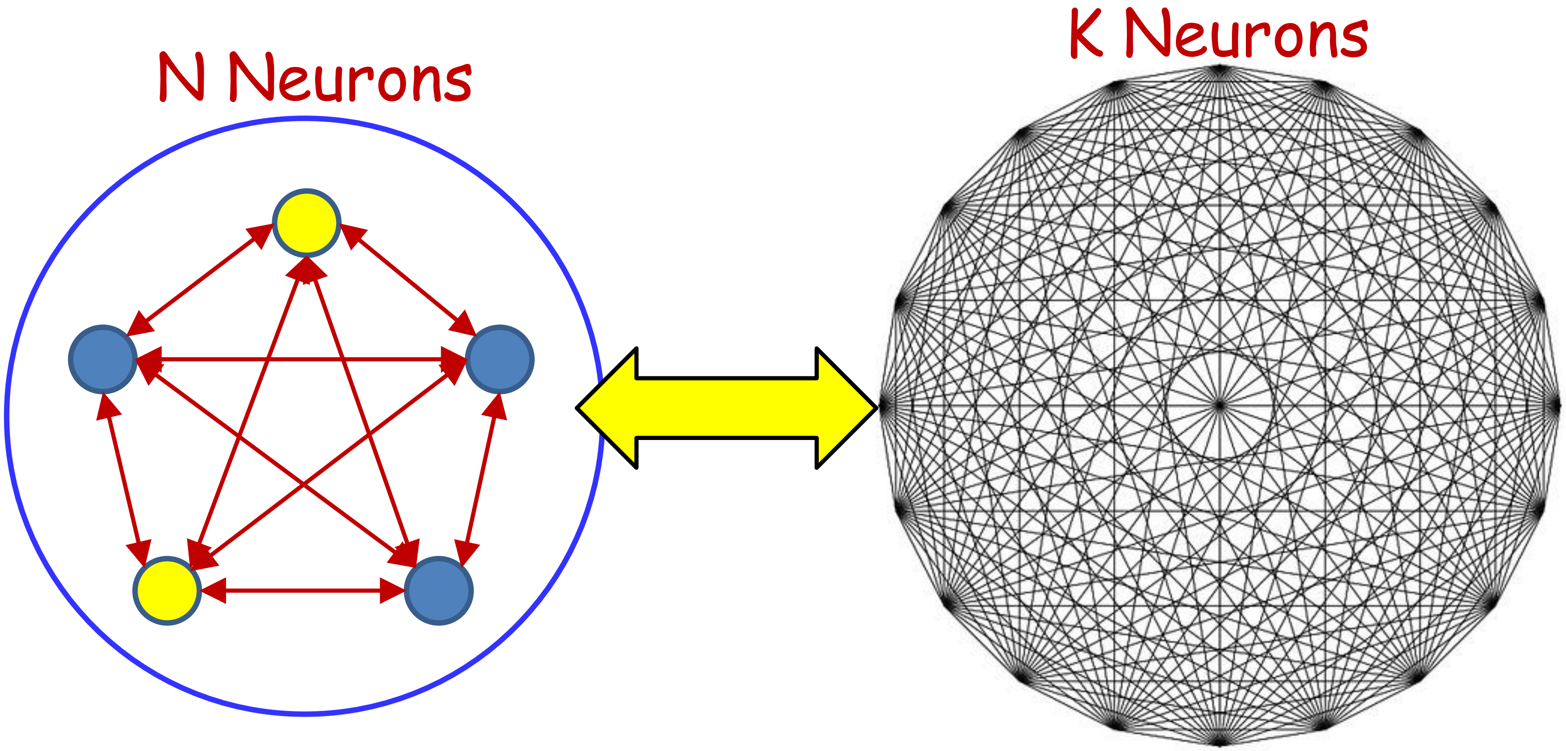
- The Hopfield net as a Boltzmann distribution

- Adding capacity to a Hopfield network
  - The Boltzmann machine

Storing more than N patterns

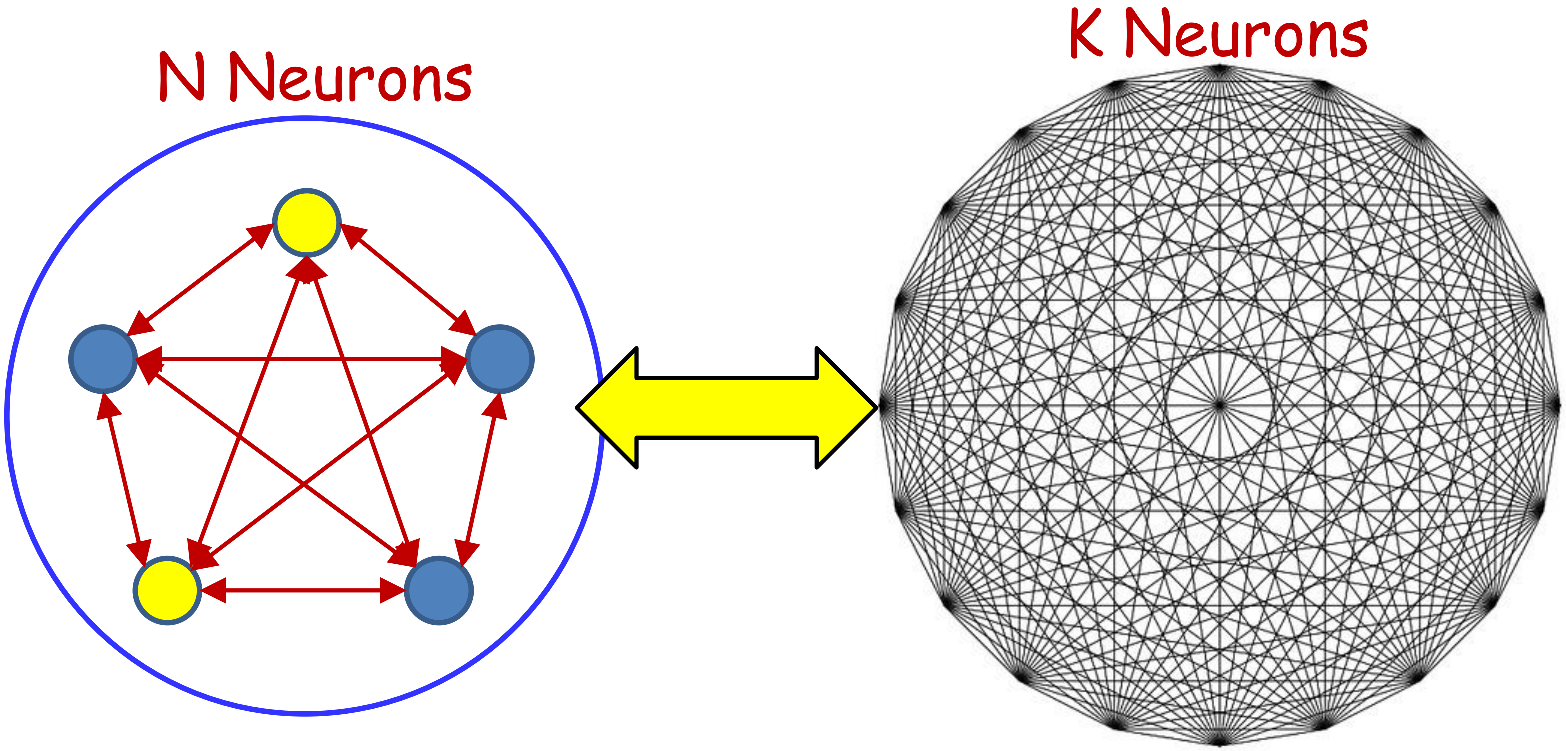
- The memory capacity of an N-bit network is at most N
  - Stable patterns (not necessarily even stationary)
    - Abu Mustafa and St. Jacques, 1985
    - Although "information capacity" is $\mathcal{O}(N^3)$
- How do we increase the capacity of the network
  - Store more patterns

Expanding the network



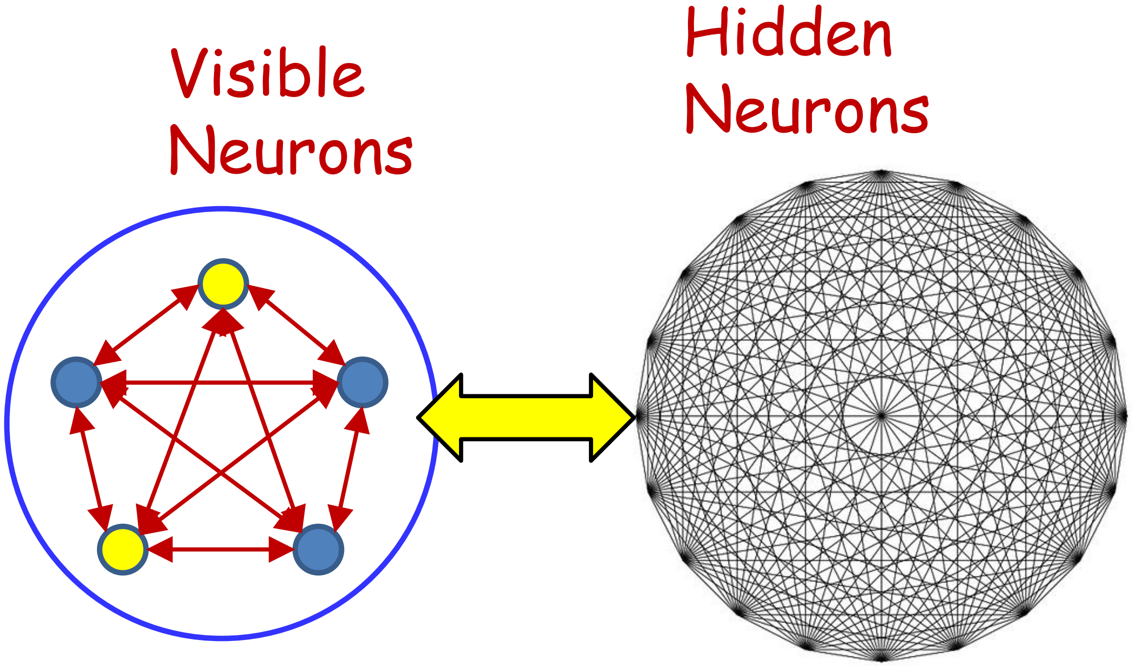
- Add a large number of neurons whose actual values you don’t care about!

Expanded Network



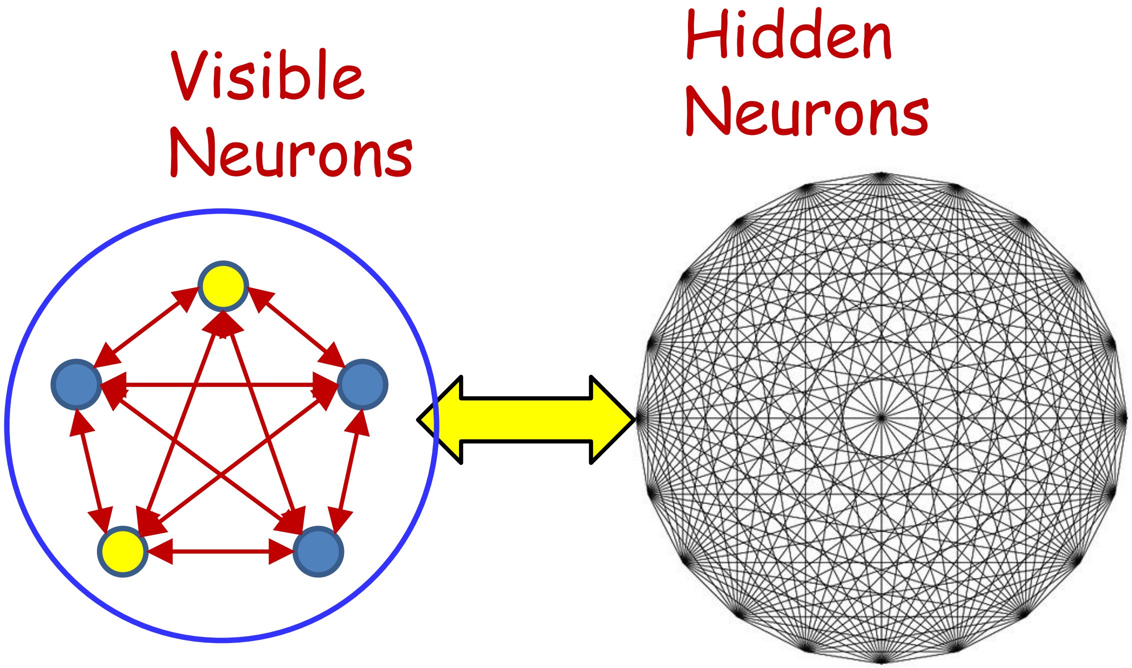
- New capacity: $\sim(N + K)$ patterns
  - Although we only care about the pattern of the first N neurons
  - We're interested in *N-bit* patterns

Terminology



- Terminology:
  - The neurons that store the actual patterns of interest: *Visible neurons*
  - The neurons that only serve to increase the capacity but whose actual values are not important: *Hidden neurons*
  - These can be set to anything in order to store a visible pattern

Training the network



Visible Neurons

Hidden Neurons

- For a given pattern of *visible* neurons, there are any number of *hidden* patterns (2^K)
- Which of these do we choose?
  - Ideally choose the one that results in the lowest energy
  - But that’s an exponential search space!
    - Solution: Combinatorial optimization
      - Simulated annealing

The patterns

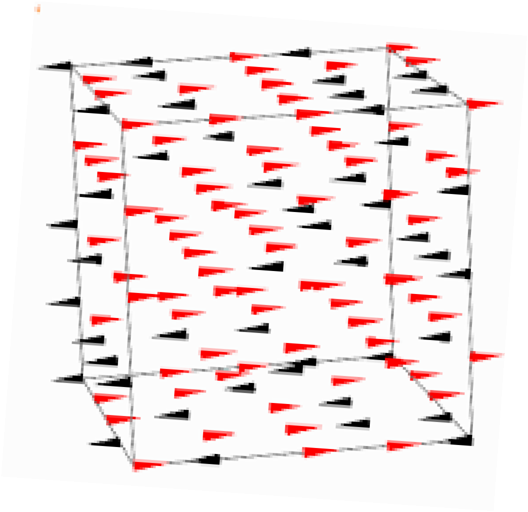
- In fact we could have *multiple* hidden patterns coupled with any visible pattern
  - These would be multiple stored patterns that all give the same visible output
  - How many do we permit

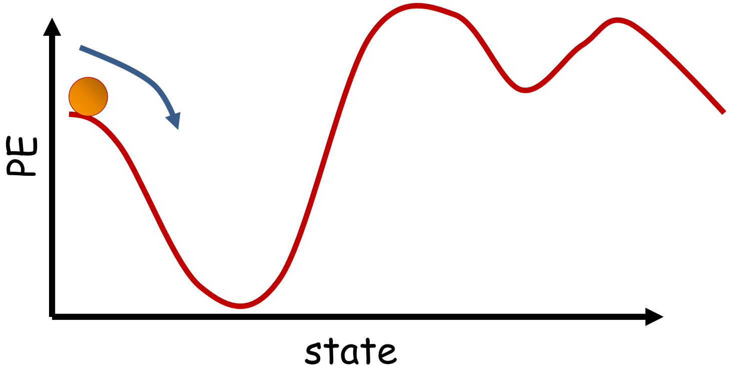
- Do we need to specify one or more particular hidden patterns?
  - How about *all* of them
  - What do I mean by this bizarre statement?

But first..

- The Hopfield net as a distribution..

Revisiting 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 *probability* of the state
  - And the *expected* value of the state

The Helmholtz Free Energy of a System

- A thermodynamic system at temperature T can exist in one of many states
  - Potentially infinite states
  - At any time, 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 Helmholtz Free Energy of a System

- 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 of the system measures the *useful* work derivable from it and combines the two terms

$$F_T = U_T + kTH_T$$

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

The Helmholtz Free Energy of a System

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

- A system held at a specific temperature *anneals* by varying the rate at which it visits the various states, to reduce the free energy in the system, until a minimum free-energy state is achieved

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

The Helmholtz Free Energy of a System

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

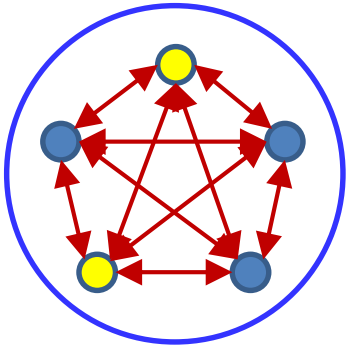
- Minimizing this w.r.t $P_T(s)$, we get

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

  - Also known as the *Gibbs* distribution
  - Z is a normalizing constant
  - Note the dependence on T
  - A T = 0, the system will always remain at the lowest-energy configuration with prob = 1.

The Energy of the Network

Visible Neurons



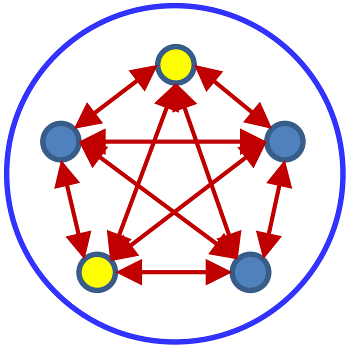
$$E(S) = -\sum_{i<j} w_{ij} s_i s_j - b_i s_i$$

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

- We can define the energy of the system as before
- Since neurons are stochastic, there is disorder or entropy (with T = 1)
- The *equilibribum* probability distribution over states is the Boltzmann distribution at T=1
  - This is the probability of different states that the network will wander over *at equilibrium*

The Hopfield net is a distribution

Visible Neurons



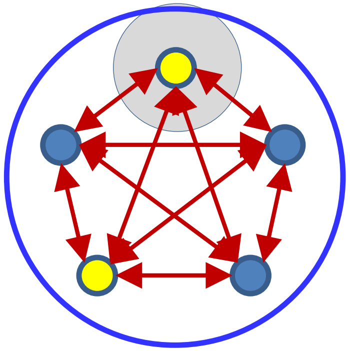
$$E(S) = -\sum_{i<j} w_{ij}s_i s_j - b_i s_i$$

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

- The stochastic Hopfield network models a ***probability distribution*** over states
  - Where a state is a binary string
  - Specifically, it models a *Boltzmann distribution*
  - **The parameters of the model are the weights of the network**
- The probability that (at equilibrium) the network will be in any state is $P(S)$
  - It is a *generative* model: generates states according to $P(S)$

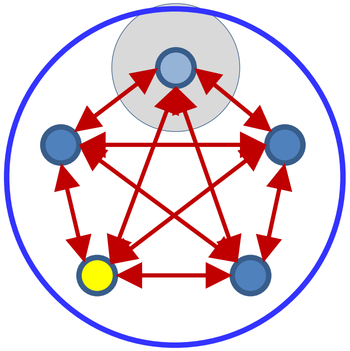
The field at a single node

- Let S and S' be otherwise identical states that only differ in the i-th bit
  - S has i-th bit = $+1$ and S’ has i-th bit = -1



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

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

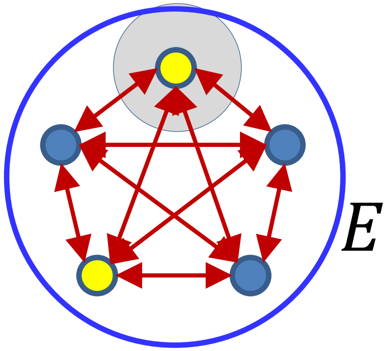


$$logP(S) - logP(S') = logP\left(s_i = 1 \middle| s_{j\neq i}\right) - logP\left(s_i = 0 \middle| s_{j\neq i}\right)$$

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

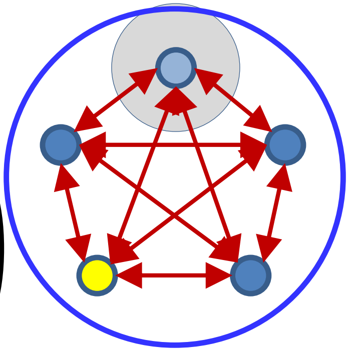
The field at a single node

- Let S and $S\,'$ be the states with the ith bit in the $+1$ and -1 states



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

$$E(S) = -\frac{1}{2}\left(E_{not\ i} + \sum_{j\neq i} w_j s_j + b_i\right)$$



$$E(S') = -\frac{1}{2}\left(E_{not\ i} - \sum_{j\neq i} w_j s_j - b_i\right)$$

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

The field at a single node

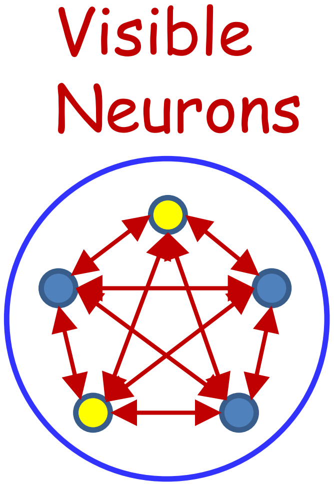
$$log\left(\frac{P\left(s_i = 1 \middle| s_{j\neq i}\right)}{1 - P\left(s_i = 1 \middle| s_{j\neq i}\right)}\right) = \sum_{j\neq i} w_j s_j + b_i$$

- Giving us

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

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

Redefining the network



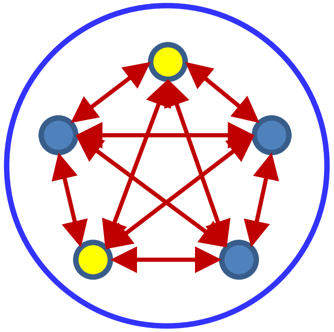
$$z_i = \sum_j w_{ji} s_j + b_i$$

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

- 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; only a matter of convenience

The Hopfield net is a distribution

Visible Neurons

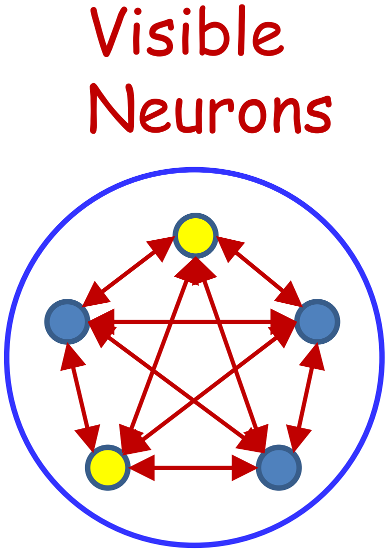


$$z_i = \sum_j w_{ji} s_j + b_i$$

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

- The Hopfield net is a probability distribution over binary sequences
  - The Boltzmann distribution
- The *conditional* distribution of individual bits in the sequence is a logistic

Running the network



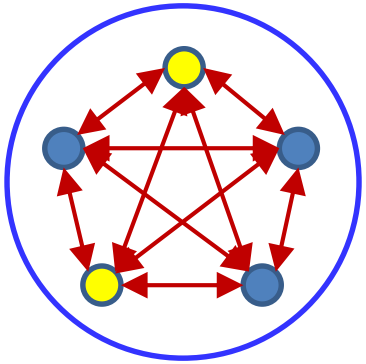
$$z_i = \sum_j w_{ji} s_j + b_i$$

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

- Initialize the neurons
- Cycle through the neurons and randomly set the neuron to 1 or -1 according to the probability given above
  - Gibbs sampling: Fix N-1 variables and sample the remaining variable
  - As opposed to energy-based update (mean field approximation): run the test $z_i > 0$?
- After many many iterations (until “convergence”), *sample* the individual neurons

Training the network

Visible Neurons



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

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

$$\boxed{P(S) = \frac{exp\left(\sum_{i<j} w_{ij} s_i s_j + b_i s_i\right)}{\sum_{S'} exp\left(\sum_{i<j} w_{ij} s_i' s_j' + b_i s_i'\right)}}$$

- 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

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

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

$$= \frac{1}{N}\sum_{S}\left(\sum_{i<j} w_{ij}s_is_j + b_is_i(S)\right) - \log\left(\sum_{S'} exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)\right)$$

- Maximize the average log likelihood of all "training" vectors $\mathbf{S} = \{S_1, S_2, \ldots, SN\}$
  - In the first summation, s_i and s_j are bits of S
  - In the second, s_i' and s_j' are bits of S'

Maximum Likelihood Training

$$\boxed{\langle \log(P(\mathbf{S}))\rangle = \frac{1}{N}\sum_{S}\left(\sum_{i<j} w_{ij}s_is_j + b_is_i(S)\right) - \log\left(\sum_{S\prime} exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)\right)}$$

$$\boxed{\frac{d\langle \log(P(\mathbf{S}))\rangle}{dw_{ij}} = \frac{1}{N}\sum_{S} s_is_j - ???}$$

- We will use gradient descent, but we run into a problem..
- The first term is just the average s_is_j over all training patterns
- But the second term is summed over *all* states
  - Of which there can be an exponential number!

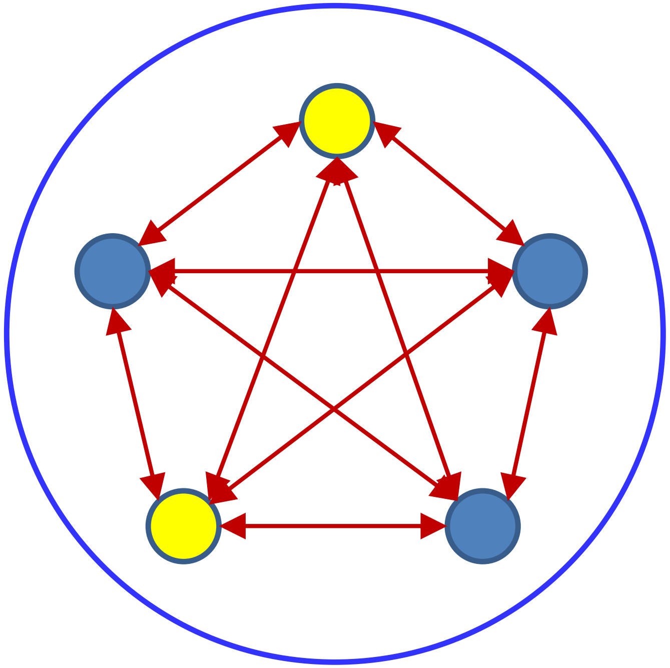
The second term

$$\frac{d\log\left(\sum_{S'} exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)\right)}{dw_{ij}} = \sum_{S'} \frac{exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)}{\sum_{S'} exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)} s_i's_j'$$

$$\frac{d\log\left(\sum_{S'} exp\left(\sum_{i<j} w_{ij}s_i's_j' + b_is_i'\right)\right)}{dw_{ij}} = \sum_{S'} P(S')s_i's_j'$$

- 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 we can compute it by sampling!

The simulation solution



- Initialize the network randomly and let it "evolve"
  - By probabilistically selecting state values according to our model
- After many many epochs, take a snapshot of the state
- Repeat this many many times
- Let the collection of states be

$$\mathbf{S}_{simul} = \{S_{simul,1}, S_{simul,1=2}, \ldots, S_{simul,M}\}$$

The simulation solution for the second term

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

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

- The second term in the derivative is computed as the average of sampled states when the network is running “freely”

Maximum Likelihood Training

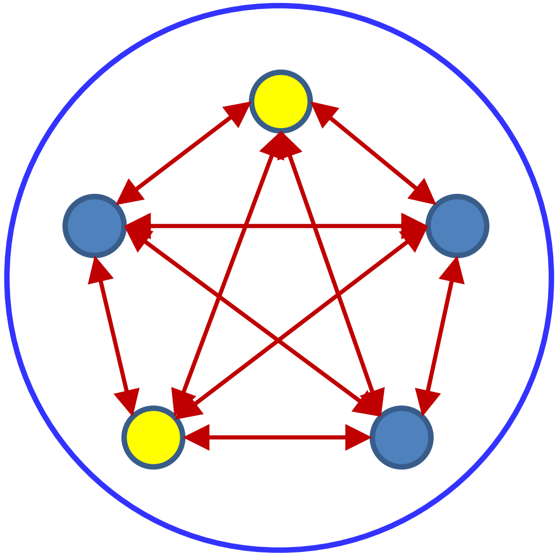
$$\langle \log(P(\mathbf{S}))\rangle = \frac{1}{N}\sum_{S}\left(\sum_{i<j} w_{ij}s_is_j + b_is_i(S)\right) - \log\left(\sum_{S'} exp\left(\sum_{i<j} w_{ij}s'_is'_j + b_is'_i\right)\right)$$

$$\frac{d\langle \log(P(\mathbf{S}))\rangle}{dw_{ij}} = \frac{1}{N}\sum_{S} s_is_j - \frac{1}{M}\sum_{S'\in\mathbf{S}_{simul}} s'_is'_j$$

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

- The overall gradient ascent rule

Overall Training



$$\frac{d\langle \log(P(\mathbf{S}))\rangle}{dw_{ij}} = \frac{1}{N}\sum_{S} s_i s_j - \frac{1}{M}\sum_{S\prime \in \mathbf{S}_{simul}} s_i' s_j'$$

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

- Initialize weights
- Let the network run to obtain simulated state samples
- Compute gradient and update weights
- Iterate

Lookahead..

- Boltzmann Machines
- Training strategies
- RBMs
- DBMs..