Replication of Hopfield Model

Qiyu Chen

School of Physics, Peking University

October 18, 2024

1. Background
2. Discrete Hopfield Model
3. Continuous Hopfield Model
4. Conclusion

McCulloch–Pitts Neuron

- n inputs$(x_1, x_2, \cdots, x_n \in \{0, 1\})$, one output $y \in \{0, 1\}$, a threshold $b \in N$
- an input is either excitatory(1) or inhibitory(0), an output is either quiet(0) or firing(1)
- all the neurons operate in synchronous discrete time-steps of t

$$y(t+1) = \begin{cases} 1, & \sum_i x_i \geq b \text{ and no inhibitory inputs are firing} \\ 0, & \text{otherwise} \end{cases}$$

- each output can be the input to an arbitrary number of neurons, including itself (no more than once)

Hebbian Theory

- introduced by Donald Hebb in his 1949 book The Organization of Behavior
- Neurons that fire together, wire together.
- A formulaic description is

$$w_{ij} = \frac{1}{n}\sum_{s=1}^{n} x_i^s x_j^s, \quad x_{i,j}^s = \{+1, -1\}$$

where w_{ij} is the weight of the connection from neuron j to neuron i and x_i the input for neuron i.

Ising Model

- a mathematical model of ferromagnetism in statistical mechanics
- Hamiltonian function is

$$\mathcal{H} = -\sum_{\langle ij \rangle} J_{ij} s_i s_j - \mu \sum_{i=1}^{N} h_i s_i$$

where s_i is the spin value to each lattice site, μ the magnetic moment, J_{ij} the interaction and h_j the external field

- mean field approximation

$$\mathcal{H} = -\sum_{i} \mu s_i \left(h_i + \frac{1}{\mu} \sum_{j} J_{ij} s_j \right) = -\sum_{i} \mu s_i (h_i + h_i')$$

$$h_i' = \frac{1}{\mu} \sum_{j} J_{ij} s_j \approx \frac{1}{\mu} \sum_{j} \overline{J_{ij} s_j}$$

Ising Model

- Hamiltonian function

$$\mathcal{H} = -\sum_{\langle ij \rangle} J_{ij} s_i s_j - \mu \sum_{i=1}^{N} h_i s_i$$

- configuration probability is given by the Boltzmann distribution

$$P(s) \propto \exp\left(-\frac{\mathcal{H}(s)}{k_B T}\right)$$

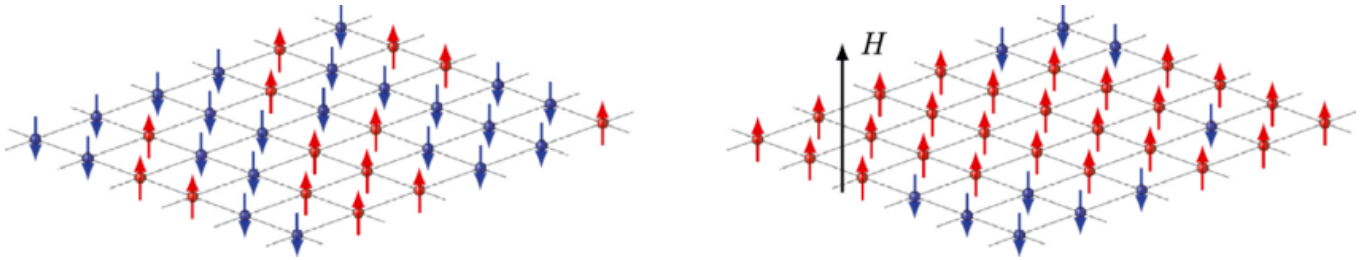


Figure 1: Ising Model

1 Background

2 Discrete Hopfield Model

3 Continuous Hopfield Model

4 Conclusion

Core Issue: Explain the Memory Mechanism

- Content-addressable memory retrieves entire memory item on the basis of sufficient partial information
- A physical system has stable limit points $X_a, X_b, \cdots$. $X = X_a + \Delta X$ will proceed in time until $X \approx X_a$(ΔX is sufficiently small)
- Stable limit points $X_a, X_b, \cdots$ can be seen as memories.

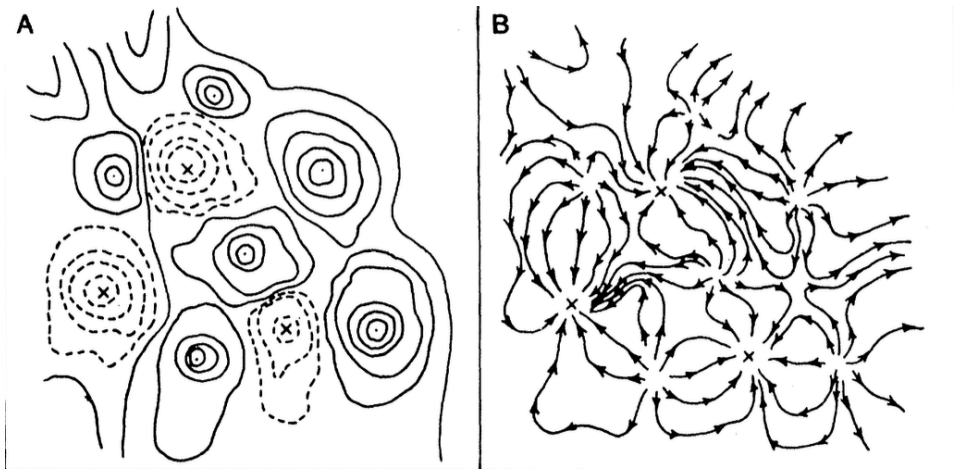


Figure 2: Stable Limit Points as Memories

Model: Combine Physics with Biology

- neuron i has two states like MCP neurons: $V_i \in \{0, 1\}$
- strength of connection from j to i is $T_{ij}(T_{ii} = 0)$
- threshold of neuron i is U_i(default to 0 in this part)
- input signal to neuron i is $\sum_j T_{ij} V_j$
- Analogous to the Ising model, energy function is defined by

$$E = -\frac{1}{2} \sum_i \sum_j T_{ij} V_i V_j = -\frac{1}{2} V^T T V$$

Model: Biological Interpretations

- The biological information sent to other neurons often lies in a short-time average of the firing rate.
- focus: nonlinearity of the input-output relationship
- replace the input-output relationship by dot-dash step

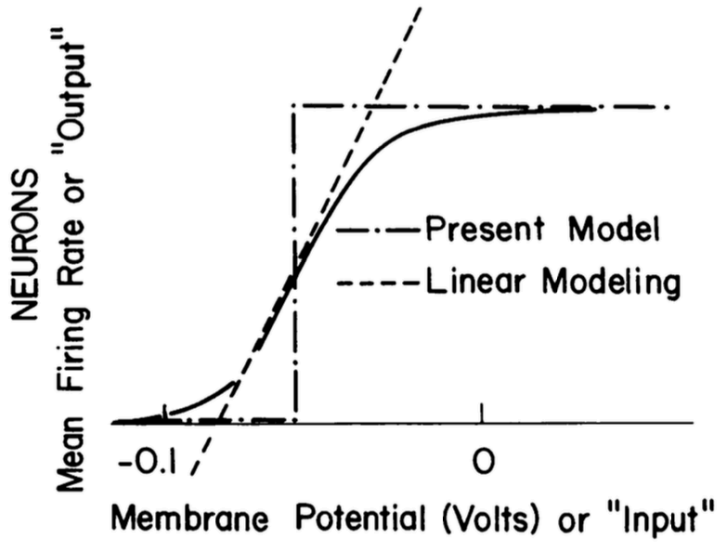


Figure 3: Firing Rate Versus Membrane Voltage

Model: Information Storage Algorithm

- Suppose N neurons are used to store n states. According to Hebbian Theory, T_{ij} is defined by

$$T_{ij} = \sum_{s=1}^{n} (2V_i^s - 1)(2V_j^s - 1)$$

with $T_{ii} = 0$

- From this definition, using mean field approximation

$$\sum_j T_{ij} V_j^{s'} = \sum_s (2V_i^s - 1) \left[\sum_j V_j^{s'} (2V_j^s - 1) \right] \approx (2V_i^{s'} - 1)N/2$$

- term $-V_i^s T_{ij} V_j^s$ always remains minimal
- $\sum_j T_{ij} V_j^{s'}$ is only related to $V^{s'}$

Model: Update Algorithm

- given an initial state V with known T
- neuron i readjusts its state by setting

$$V_i = \begin{cases} 1, & \text{if } \sum_j T_{ij} V_j > U_i \\ 0, & \text{if } \sum_j T_{ij} V_j < U_i \end{cases}$$

- In this way

$$\Delta E = -\Delta V_i \sum_j T_{ij} V_j < 0$$

Thus E is monotonically decreasing function

Model: Code Implementation

```python
#define a Hopfield Model
class Hopfield_Model:
    def __init__(self, num_of_neurons):
        self.num_of_neurons = num_of_neurons
        self.weights = np.zeros((num_of_neurons, num_of_neurons))

    #Hebbian Theory
    def hebbian(self, states):
        for state in states:
            adjusted_state = state*2 - 1
            self.weights += np.outer(adjusted_state, adjusted_state)
        np.fill_diagonal(self.weights, 0)
        self.weights /= self.num_of_neurons

    #update Algorithm
    def update(self, state):
        for i in range(self.num_of_neurons):
            judge = np.dot(self.weights, state)
            if judge[i] >= 0:
                state[i] = 1
            else: state[i] = 0
        return state

    #processing
    def process(self, state, time):
        for _ in range(time):
            state = self.update(state)
        return state
```

Figure 4: Code Implementation of Hopfield Model

Model: An Example

- suppose two images are stored

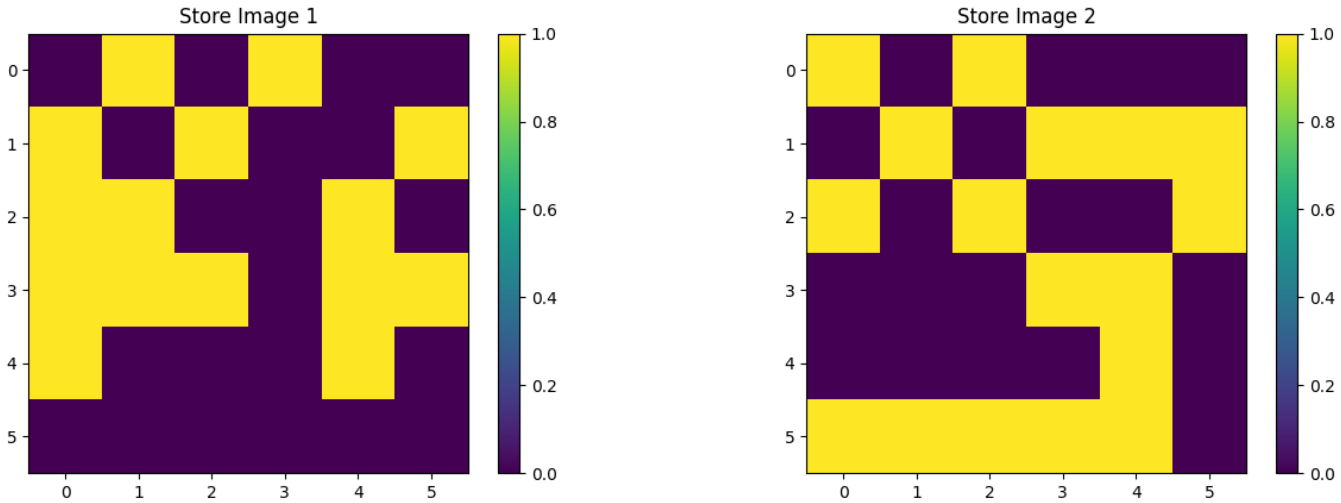
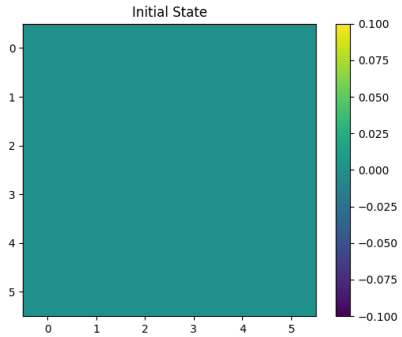


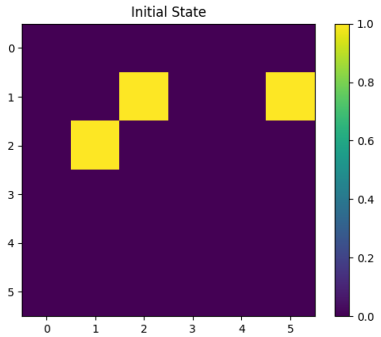
Figure 5: Two Stored Images

Model: An Example

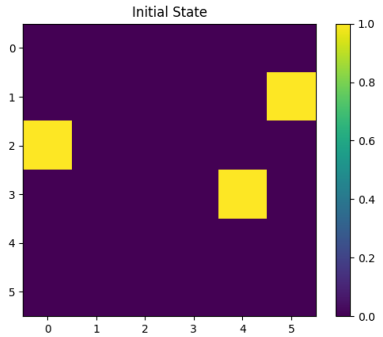
- except for (a) and (c), the other initial states all reached the correct recall



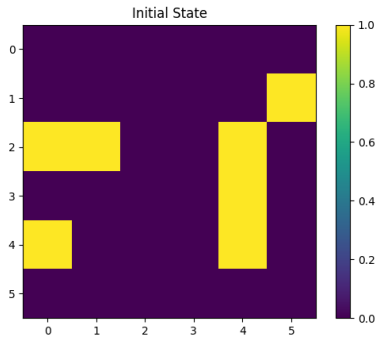
(a) 0%Completeness



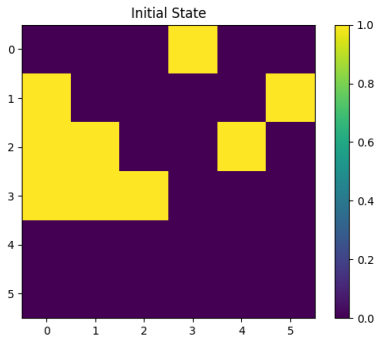
(b) 20%Completeness



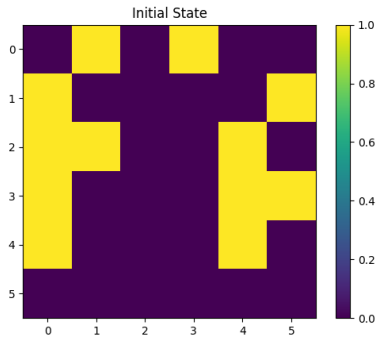
(c) 20%Completeness



(d) 40%Completeness



(e) 60%Completeness

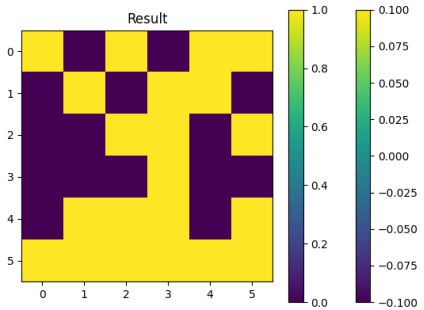


(f) 80%Completeness

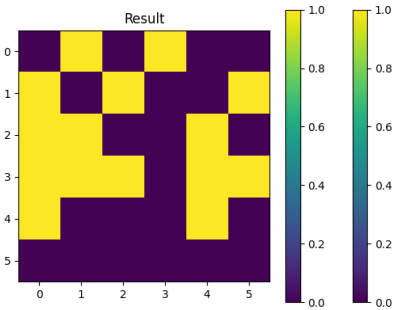
Figure 6: Initial States of Different Completeness

Model: An Example

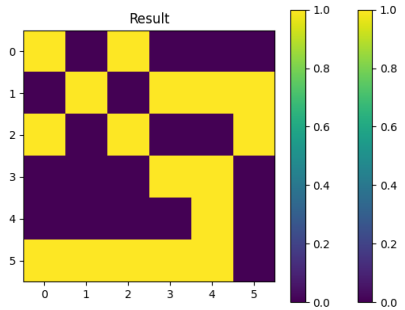
- except for (a) and (c), the other initial states all reached the correct recall



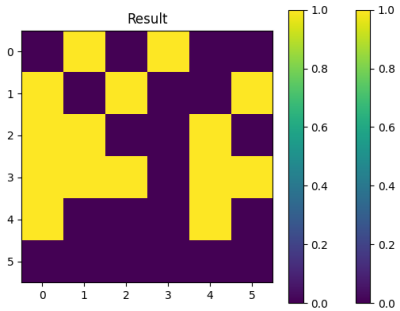
(a) 0%Completeness



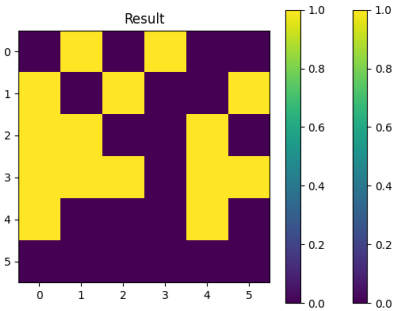
(b) 20%Completeness



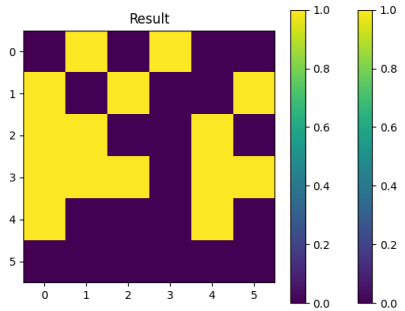
(c) 20%Completeness



(d) 40%Completeness



(e) 60%Completeness



(f) 80%Completeness

Figure 7: Final States of Different Completeness

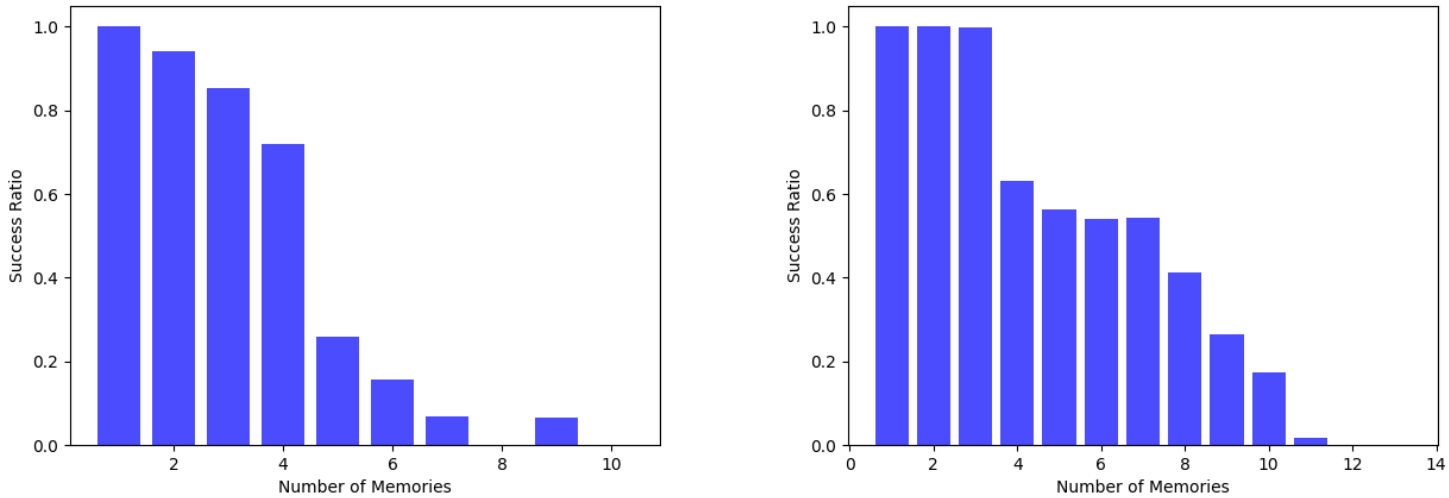
An Interesting Question: the Maximum Capacity of Model

```python
from memory import Hopfield_Model,np
def recall_test(memory, noise_level, times):
    flip_mask = np.random.rand(N) < noise_level
    noisy = np.where(flip_mask, 1 - memory, memory)
    hopfield.process(noisy,time=times)
    if np.array_equal(noisy, memory):
        return 1
    else:
        return 0

#initialization
N, num_of_memories = 50, 1
noise_level = 0.1
tries_per_memory = 1000
times = 100
ans = []
for num_of_memories in range(1,14):
    np.random.seed(42)
    memories = np.random.choice([0, 1],size=(num_of_memories, N))

    hopfield = Hopfield_Model(num_of_neurons=N)
    hopfield.hebbian(memories)

    total_success = 0
    total_tries = num_of_memories * tries_per_memory
    for memory in memories:
        for _ in range(tries_per_memory):
            total_success += recall_test(memory, noise_level, times)
    ans.append(total_success/total_tries)
    print('num_of_memories:',num_of_memories,', success_ratio:',total_success/total_tries)
```

Figure 8: Simulation of the Relationship Between the Number of Memories and Success Ratio

An Interesting Question: the Maximum Capacity of Model

- about 0.15N states can be simultaneously remembered before error in recall is severe(success ratio < 0.5)
- n = 0.138N derived from statistical physics



(a) 30 Neurons (b) 50 Neurons

Figure 9: Success Ratio Versus the Number of Memories with (a)30 Neurons and (b)50 Neurons

1 Background

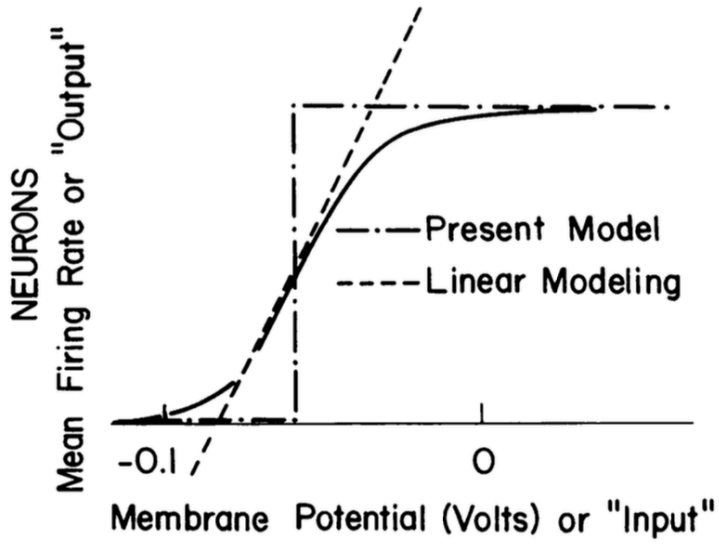
2 Discrete Hopfield Model

3 Continuous Hopfield Model

4 Conclusion

Core Issue: Model Neurons Simply and Appropriately

- limitations of MCP neuron: weakness in analog processing and high interconnectivity
- limitations of previous discrete model: asynchronous processing and "on or off" neurons
- goal: describe a neuron's effective output, input, internal state, input-ouput relation



Model: Simplifications and Assumptions

- neglect electrical effects attributable to the shape of dendrites and axons
  - integrated capacitance C and resistance R
  - input currents are simply additive
- deal only with "fast" synaptic events
  - delay in signal transmission of chemical synapses is much shorter than RC

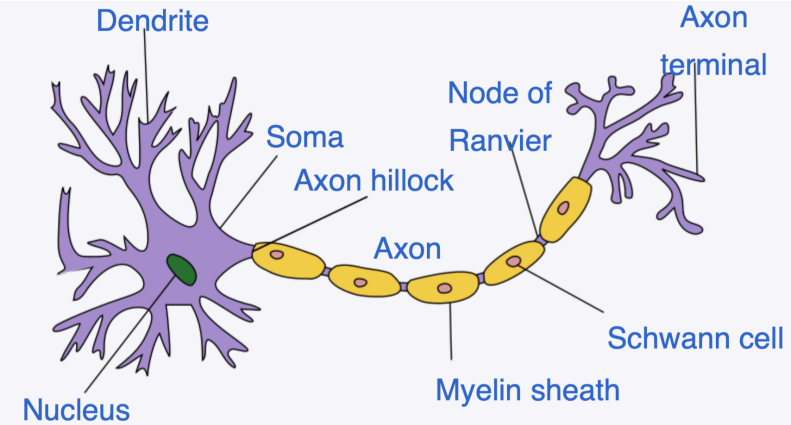


Figure 10: Typical Structure of Neurons

Model: Simplifications and Assumptions

- resistance from neuron j to i is related to input u_i
- input-output relation is sigmoid and monotonic

$$V_i = f_i(u_i)$$

f is firing rate

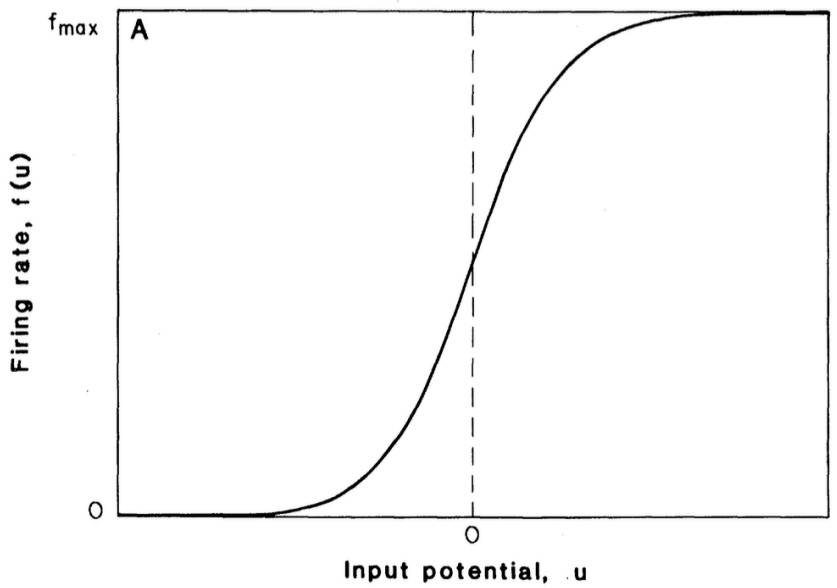


Figure 11: Input-Output Relation

Model: Equations

- postsynaptic current is given by

$$\sum_j T_{ij} V_j = \sum_j T_{ij} f_j(u_j)$$

  where T_{ij} is strength of synapse from neuron j to i

- external currents to neuron i is I_i
- coupled nonlinear differential equations

$$C_i \frac{du_i}{dt} = \sum_{j=1}^{N} T_{ij} f_j(u_j) - \frac{u_i}{R_i} + I_i \quad (i = 1, \cdots, N)$$

- retains two important aspects of computation: dynamics and nonlinearity

Model: An Electrical Circuit Interpretation

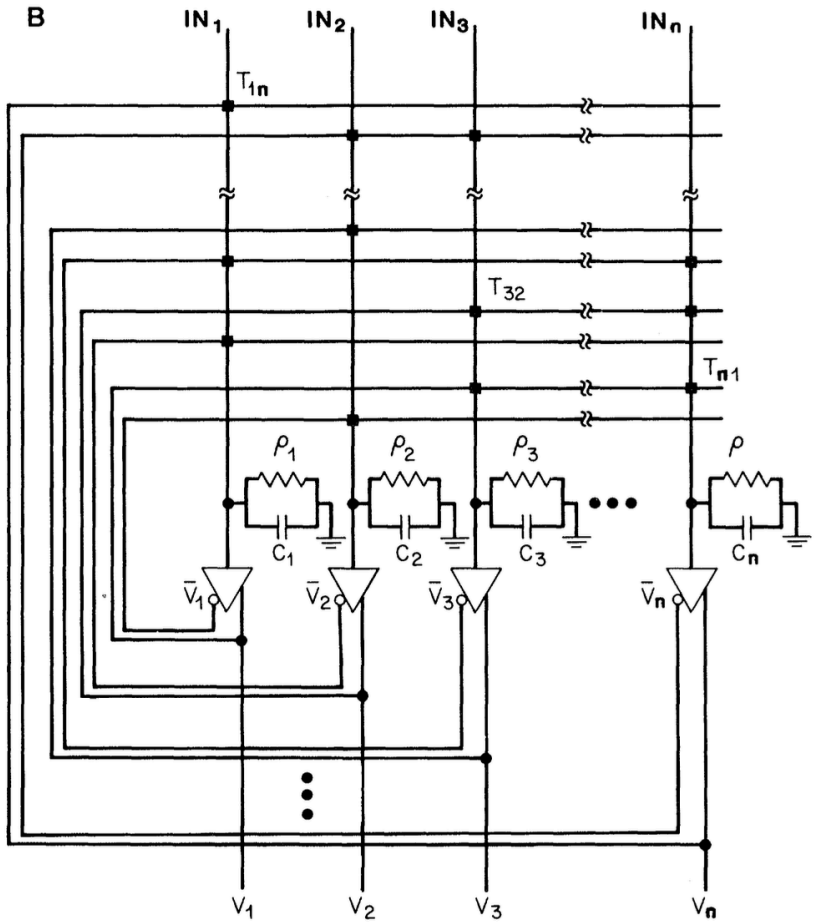
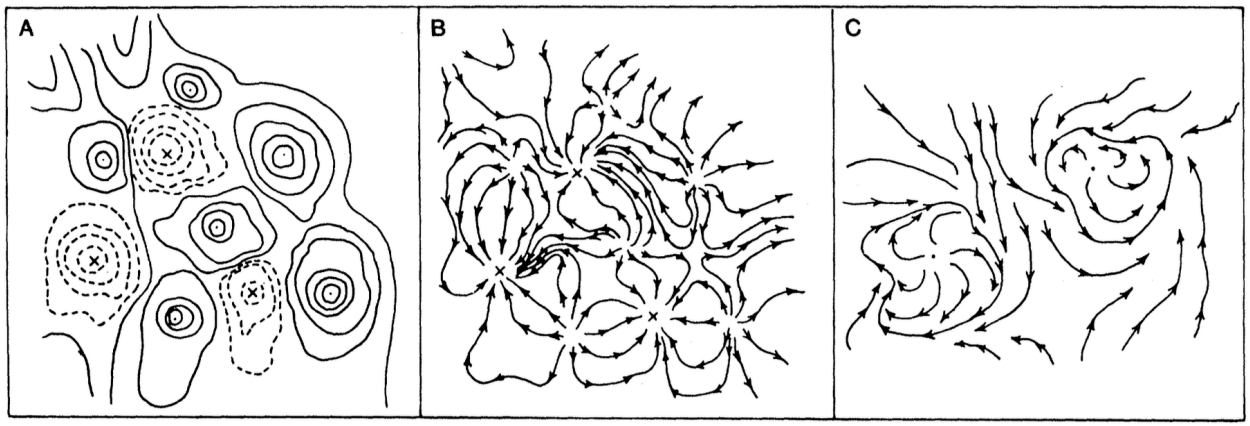


Figure 12: An Equivalent Electrical Circuit of the Equation

Model: Understanding the Dynamics of Neural Circuitry

- symmetric circuit("natural"): $T_{ij} = T_{ji}$
- symmetry of the connections results in a powerful theorem about the behavior of the system
- "computational energy" E always converges to a local minimum
- systems having organized asymmetry can exhibit oscillation and chaos



Model: Energy Function and Its Convergence

- dynamical equations

$$\frac{du_i}{dt} = \sum_{j=1}^{N} T_{ij} f_j(u_j) - \frac{u_i}{\tau_i} + I_i$$

- energy function (Lyapunov function) defined as

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} T_{ij} V_i V_j - \sum_{i=1}^{N} V_i I_i + \sum_{i=1}^{N} \frac{1}{\tau_i} \int_0^{V_i} f^{-1}(V) dV$$

- time derivative of u_i and E(with $T_{ij} = T_{ji}$)

$$\frac{du_i}{dt} = -\frac{\partial E}{\partial V_i}, \quad \frac{dE}{dt} = \sum_i \frac{\partial E}{\partial V_i}\frac{dV_i}{dt} = -\sum_i \left(\frac{dV_i}{dt}\right)^2 (f^{-1})'(V_i) \leq 0$$

Applications

- memory
- more difficult computations like optimization
  - map onto a neural network whose configurations correspond to possible solutions
  - construct energy function E proportional to the cost function
  - a stable-state configuration is reached
- two examples: A/D converter and TSP problem

Application I: A/D Converter

- Goal: given an analog input X and return the binary number $\overline{V_3V_2V_1V_0}$(4-bits)
- energy function

$$E = \frac{1}{2}(X - \sum_{j=0}^{3} 2^j V_j)^2 + \sum_{j=0}^{3}(2^{2j-1})[V_j(1 - V_j)]$$

- compared with original energy function

$$T_{ij} = -2^{i+j}(i \neq j), \quad I_i = -2^{2i-1} + 2^i X$$

- dynamical equations ($\tau_i = 1$ without loss of generality)

$$\frac{du_i}{dt} = \sum_{j=0}^{3} T_{ij} f_j(u_j) - u_i + I_i, \quad f(u_j) = \frac{1}{2}(1 + \tanh u_j/u_0)$$

Application I: A/D Converter-Code Implementation

```
class Hopfield_Model:
    # initialization
    def __init__(self, num_of_neurons):
        self.num_of_neurons = num_of_neurons
        self.u0 = 0.5
        self.delta_t = 0.001
        self.energy = []
        # set T
        self.T = np.zeros((self.num_of_neurons,self.num_of_neurons))
        for i in range(self.num_of_neurons):
            for j in range(self.num_of_neurons):
                if j != i:
                    self.T[i][j] = - 2**(i+j)
                else: self.T[i][j] = 0
        # set u, V
        self.u = np.zeros(self.num_of_neurons)
        self.u1 = self.u.copy()
        self.V = np.zeros(self.num_of_neurons)
    # set I
    def set_I(self, X=int):
        self.X = X
        self.I = np.zeros(self.num_of_neurons)
        self.I = [- 2**(2*i - 1) + 2**i * X for i in range(self.num_of_neurons)]
```

Application I: A/D Converter-Code Implementation

```
# update function
def update(self):
    self.V = [1/2 * (1 + tanh(self.u[i]/self.u0)) for i in range(self.num_of_neurons)]
    for i in range(self.num_of_neurons):
        dot = 0
        for j in range(self.num_of_neurons):
            dot += self.T[i][j]*self.V[j]
        delta_u = self.delta_t * (self.I[i] + dot)
        self.u1[i] += delta_u
    self.u = self.u1.copy()
# processing
def process(self, times):
    for j in range(times):
        self.update()
        self.energy.append(self.energy_cal())
    return self.V, self.energy

def energy_cal(self):
    X1 = 0
    for i in range(self.num_of_neurons):
        X1 += 2**i * self.V[i]
    energy =  1/2 * (self.X - X1)**2
    for i in range(self.num_of_neurons):
        energy = energy + 2**(2*i-1) * self.V[i] * (1-self.V[i])
    return energy
```

Figure 13: Code Implementation of A/D Converter

Application I: A/D Converter-Energy Convergence Curve

- 4-bits A/D converter
- X = 13

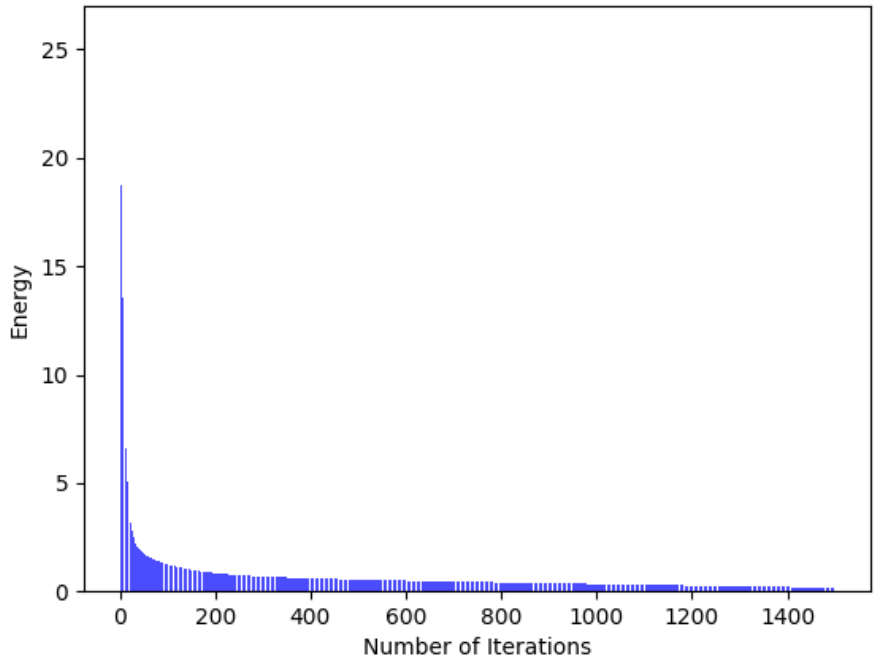


Figure 14: Energy Convergence Curve

Application II: Traveling Salesman Problem (TSP)

- TSP: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- V_{Xi} represents whether city X is in i^{th} position
- energy function of TSP

$$E = \frac{A}{2} \sum_X \sum_i \sum_{j \neq i} V_{Xi} V_{Xj} + \frac{B}{2} \sum_i \sum_X \sum_{Y \neq X} V_{Xi} V_{Yi}$$
$$+ \frac{C}{2} \left(\sum_X \sum_i V_{Xi} - n \right)^2 + \frac{D}{2} \sum_X \sum_{Y \neq X} \sum_i d_{XY} V_{Xi} (V_{Y,i+1} + V_{Y,i-1})$$

Application II: Traveling Salesman Problem (TSP)

- an improved energy function and corresponding dynamical equations

$$E = \frac{A}{2}\sum_X(\sum_i V_{Xi} - 1)^2 + \frac{A}{2}\sum_i(\sum_X V_{Xi} - 1)^2 + \frac{D}{2}\sum_X\sum_Y\sum_i d_{XY}V_{Xi}V_{Y,i+1}$$

$$\frac{du_{Xi}}{dt} = -\frac{\partial E}{\partial V_{Xi}} = -A(\sum_j V_{Xj} - 1) - A(\sum_Y V_{Yi} - 1) - D\sum_Y d_{XY}V_{Y,i+1}\left(-\frac{u_{Xi}}{\tau_i}\right)$$

- term u_{Xi}/τ_i is 0 in high-gain limit ($f(u_i)$ is steep)

Application II: Traveling Salesman Problem (TSP)-Code Implementation

```
class Hopfield:
    def __init__(self, num_of_cities, A = 100, D = 100, study_rate = 0.01, u0 = 0.2):
        self.num_of_cities = num_of_cities
        self.distance = np.zeros((self.num_of_cities, self.num_of_cities))
        self.u = np.zeros((self.num_of_cities, self.num_of_cities))
        self.V = np.zeros((self.num_of_cities, self.num_of_cities))
        self.A = A
        self.D = D
        self.u0 = u0
        self.study_rate = study_rate
        self.energy = []
        for X in range(self.num_of_cities):
            for i in range(self.num_of_cities):
                self.u[X][i] = np.arctanh(2/self.num_of_cities - 1)*(1 + np.random.uniform(-0.1,0.1))

    def cal_energy(self):
        energy = 0
        for X in range(self.num_of_cities):
            t = 0
            for i in range(self.num_of_cities):
                t += self.V[X][i]
            energy += 0.5 * self.A * (t-1)**2
        for i in range(self.num_of_cities):
            t = 0
            for X in range(self.num_of_cities):
                t += self.V[X][i]
            energy += 0.5 * self.A * (t-1)**2
        for X in range(self.num_of_cities):
            for Y in range(self.num_of_cities):
                for i in range(self.num_of_cities):
                    energy += 0.5 * self.D * self.distance[X][Y] * self.V[X][i%self.num_of_cities] * \
                        self.V[Y][(i+1)%self.num_of_cities]
        return energy
```

Application II: Traveling Salesman Problem (TSP)-Code Implementation

```
def get_distance(self, distance):
    self.distance = distance

def update(self):
    for X in range(self.num_of_cities):
        for i in range(self.num_of_cities):
            self.V[X][i] = 0.5 * (1 + np.tanh(self.u[X][i]/self.u0))

    for X in range(self.num_of_cities):
        for i in range(self.num_of_cities):
            delta = 0
            term1, term2 = 0,  0
            for j in range(self.num_of_cities):
                term1 += self.V[X][j]
            for Y in range(self.num_of_cities):
                term2 += self.V[Y][i]
            delta = - self.A * (term1 - 1) - self.A * (term2 - 1)
            for Y in range(self.num_of_cities):
                delta += -self.D * self.distance[X][Y] * self.V[Y][(i+1)%self.num_of_cities]
            self.u[X][i] += delta * self.study_rate

def process(self, times):
    for i in range(1, times+1):
        self.update()
        self.energy.append(self.cal_energy())
    return self.V, self.energy
```

Figure 15: Code Implementation of TSP

Application II: Traveling Salesman Problem (TSP)-Results

- final path is sensitive to initial state, parameters of energy function, and study rate

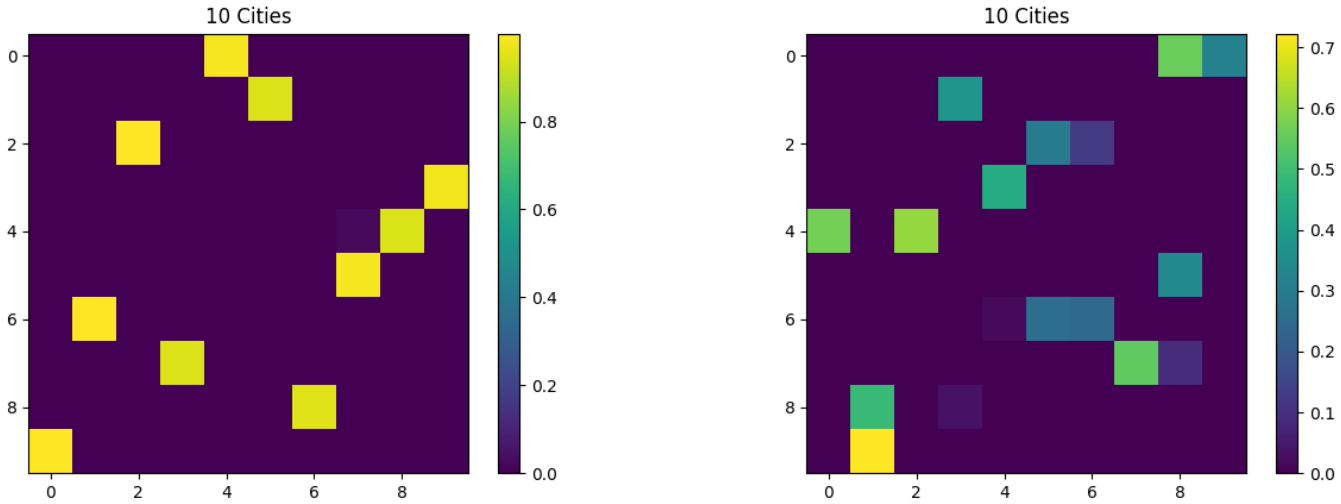


Figure 16: Two Final Paths with Different Initial States

1 Background

2 Discrete Hopfield Model

3 Continuous Hopfield Model

4 Conclusion

Conclusion

- a model of nonlinear neurons organized into networks with symmetric connections has a "natural" capacity for solving optimization problems
- "foward engineering" and "reverse engineering" (to understand the operation of a complex biological circuit with unknown principals)
- effectiveness based on large connectivity, analog response, and reciprocal or reentrant connections

References

Abe. Theories on the Hopfield neural networks[C]//International 1989 Joint Conference on Neural Networks. IEEE, 1989: 557-564 vol. 1.

Cooper L N, Liberman F, Oja E. A theory for the acquisition and loss of neuron specificity in visual cortex[J]. Biological cybernetics, 1979, 33(1): 9-28.

Hopfield J J. Neural networks and physical systems with emergent collective computational abilities[J]. Proceedings of the national academy of sciences, 1982, 79(8): 2554-2558.

Hopfield J J, Tank D W. Computing with neural circuits: A model[J]. Science, 1986, 233(4764): 625-633.

Hopfield J J, Tank D W. "Neural" computation of decisions in optimization problems[J]. Biological cybernetics, 1985, 52(3): 141-152.

Hopfield J J. Neurons with graded response have collective computational properties like those of two-state neurons[J]. Proceedings of the national academy of sciences, 1984, 81(10): 3088-3092.

Tank D, Hopfield J. Simple'neural'optimization networks: An A/D converter, signal decision circuit, and a linear programming circuit[J]. IEEE transactions on circuits and systems, 1986, 33(5): 533-541.

Thanks!