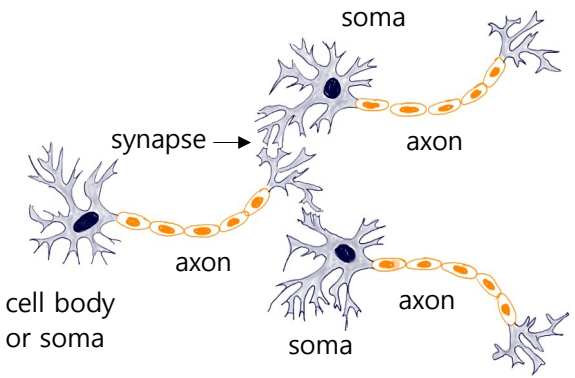
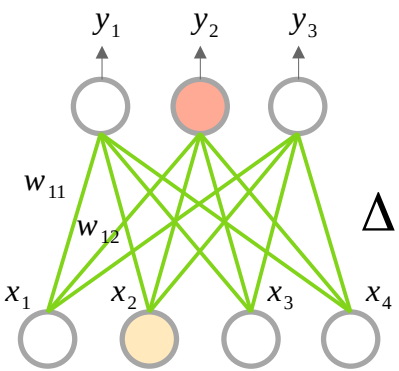




$$\Delta w_{ji} = \alpha x_j y_i$$

15-1. Hebbian learning (1)

Simple Hebbian learning rule

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

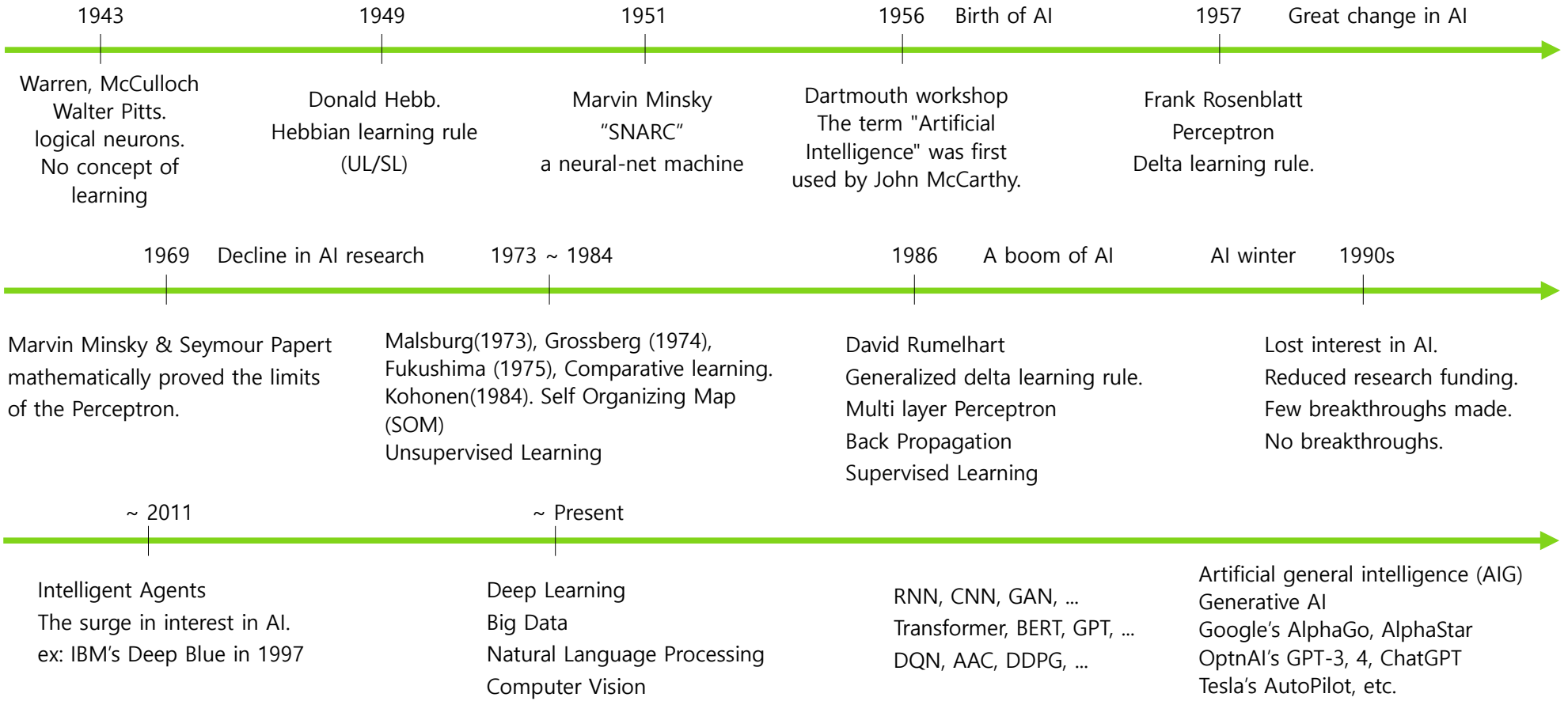
[MXDL-15-12] Deep Learning / Non-backpropagation learning - Contents

MX-AI

MX-AI

Brief history of Artificial Intelligence (AI)

- AI has a long history and has gone through several booms and winters to reach what it is today.



MX-AI

■ Hebbian learning rule

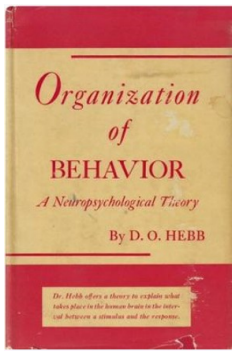


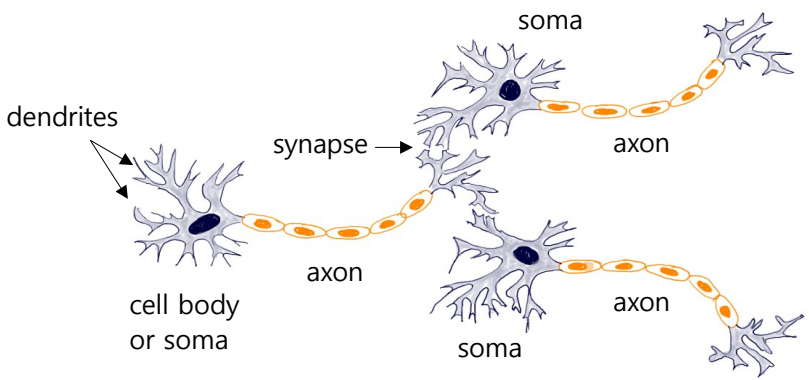
Image source:
https://en.wikipedia.org/wiki/Organization_of_Behavior

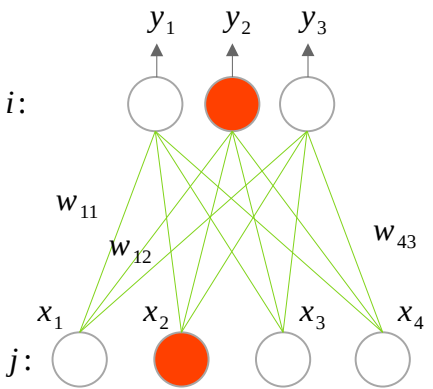
The Hebbian learning rule, or simply Hebb's rule, was introduced by Donald Hebb in his 1949 book "The Organization of Behavior".

This rule is often summarized as "Neurons that fire together, wire together".

Donald Hebb states it as follows (on page 63):
"When one cell repeatedly assists in firing another, the axon of the first cell develops synaptic knobs (or enlarges them if they already exist) in contact with the soma of the second cell."

This means that the synaptic strength between two neurons tends to get strengthened whenever the two neurons both fire.





$$y_i = f\left(\sum_j w_{ji} x_j\right)$$

ex: (i = 1) $y_1 = f\left(w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4\right)$

$w_{ji} = w_{ji} + \alpha x_j y_i$ ← simple Hebbian learning rule

$$\Delta w_{ji} = \alpha x_j y_i$$

MX-AI

A simple example of pattern recognition using Hebb's rule (1)

- **Binary classification**

1	1	1
-1	1	-1
-1	1	-1

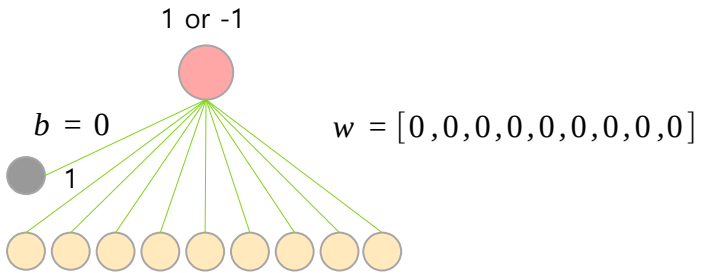
$x^{(1)} = [1,1,1,-1,1,-1,-1,1,-1]$

$y^{(1)} = +1 \rightarrow$ **' T '**

1	-1	1
1	-1	1
1	1	1

$x^{(2)} = [1,-1,1,1,-1,1,1,1,1]$

$y^{(2)} = -1 \rightarrow$ **' U '**



$w = w + \alpha x y \qquad b = b + \alpha y \quad (\alpha = 1.0)$

- **Training stage: 1) First iteration**

$w = [0,0,0,0,0,0,0,0,0] + [1,1,1,-1,1,-1,-1,1,-1] \times (+1)$

$w = [1,1,1,-1,1,-1,-1,1,-1]$

$b = 0 + 1 \times (+1) = 1$

- **Training stage: 2) Second iteration**

$w = [1,1,1,-1,1,-1,-1,1,-1] + [1,-1,1,1,-1,1,1,1,1] \times (-1)$

$w = [0,2,0,-2,2,-2,-2,0,-2]$

$b = 1 + 1 \times (-1) = 0$

- **Recall stage:**

$$y_i = \tanh\left(\sum_j w_{ji} x_j + b\right)$$

1	1	1
-1	1	-1
-1	1	-1

$x^{(t)} = [1,1,1,-1,1,-1,-1,1,-1]$

$y^{(t)} = [0,2,0,-2,2,-2,-2,0,-2] \cdot [1,1,1,-1,1,-1,-1,1,-1]^{\mathrm{T}} + 0$

$y^{(t)} = \tanh(12) = 1.0$

1	-1	1
1	-1	1
1	1	1

$x^{(t)} = [1,-1,1,1,-1,1,1,1,1]$

$y^{(t)} = [0,2,0,-2,2,-2,-2,0,-2] \cdot [1,-1,1,1,-1,1,1,1,1]^{\mathrm{T}} + 0$

$y^{(t)} = \tanh(-12) = -1.0$

1	-1	1
-1	1	-1
-1	1	-1

$x^{(t)} = [1,-1,1,-1,1,-1,-1,1,-1]$

$y^{(t)} = [0,2,0,-2,2,-2,-2,0,-2] \cdot [1,-1,1,-1,1,-1,-1,1,-1]^{\mathrm{T}} + 0$

$y^{(t)} = \tanh(8) = 1.0$

1	-1	1
-1	-1	1
1	1	-1

$x^{(t)} = [1,-1,1,-1,-1,1,1,1,-1]$

$y^{(t)} = [0,2,0,-2,2,-2,-2,0,-2] \cdot [1,-1,1,-1,-1,1,1,1,-1]^{\mathrm{T}} + 0$

$y^{(t)} = \tanh(-4) = -1.0$

MX-AI

A simple example of pattern recognition using Hebb's rule (2) – Training stage

$$w = w + \alpha\, x\, y$$

- **Multiclass classification**

1	1	1
0	1	0
0	1	0

$x^{(1)}$ = [1, 1, 1, 0, 1, 0, 0, 1, 0]

$y^{(1)}$ = 0 → [1, 0, 0] → **' T '**

1	0	1
1	0	1
1	1	1

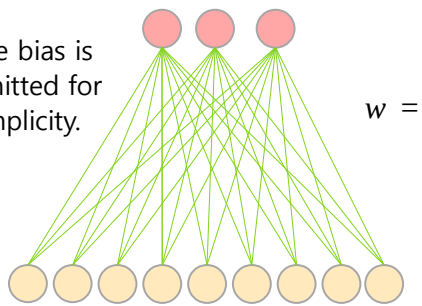
$x^{(2)}$ = [1, 0, 1, 1, 0, 1, 1, 1, 1]

$y^{(2)}$ = 1 → [0, 1, 0] → **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)}$ = [1, 1, 1, 1, 0, 0, 1, 1, 1]

$y^{(3)}$ = 2 → [0, 0, 1] → **' C '**



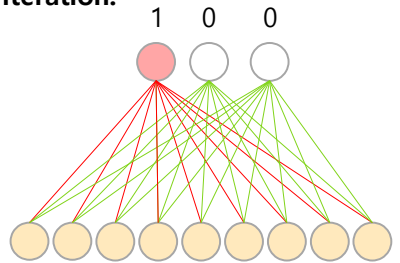
The bias is omitted for simplicity.

$$w = \begin{pmatrix} 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0\\ 0&0&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&0&0\\ 1&0&0\\ 1&0&0\\ 0&0&0\\ 1&0&0\\ 0&0&0\\ 0&0&0\\ 1&0&0\\ 0&0&0 \end{pmatrix}$$

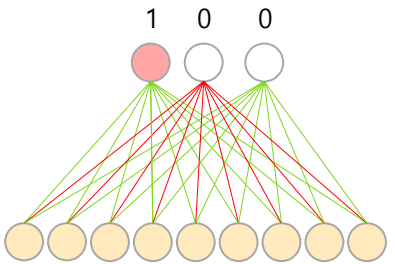
$w = w + \alpha\, x\, y \quad (\alpha = 1.0)$

As the iterations progress, the weights tend to increase. To prevent this, we can normalize the weights at each step.

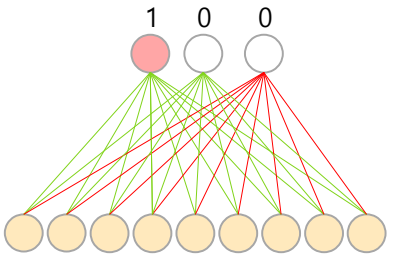
1) First iteration:



$$w[:,0] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} \times y^{(1)}[0] \times (1) = \begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix}$$



$$w[:,1] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} \times y^{(1)}[1] \times (0) = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}$$



$$w[:,2] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} \times y^{(1)}[2] \times (0) = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule (2) – Training stage

$$w = w + \alpha x y$$

- **Multiclass classification**

1	1	1
0	1	0
0	1	0

$x^{(1)} = [\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y^{(1)} = [1,\ 0,\ 0] \rightarrow$ **' T '**

1	0	1
1	0	1
1	1	1

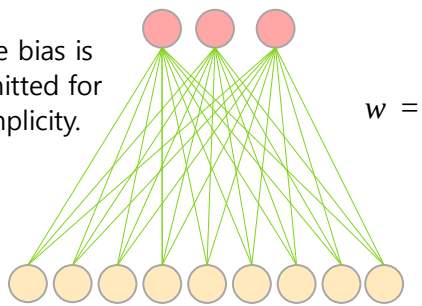
$x^{(2)} = [\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1]$

$y^{(2)} = [0,\ 1,\ 0] \rightarrow$ **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)} = [\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1]$

$y^{(3)} = [0,\ 0,\ 1] \rightarrow$ **' C '**



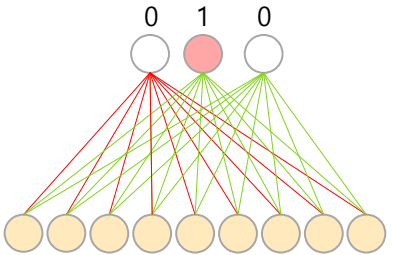
The bias is omitted for simplicity.

$$w = \begin{pmatrix} 1&0&0\\1&0&0\\1&0&0\\0&0&0\\1&0&0\\0&0&0\\0&0&0\\1&0&0\\0&0&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&1&0\\1&0&0\\1&1&0\\0&1&0\\1&0&0\\0&1&0\\0&1&0\\1&1&0\\0&1&0 \end{pmatrix}$$

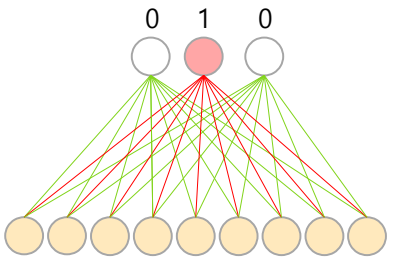
As the iterations progress, the weights tend to increase. To prevent this, we can normalize the weights at each step.

$w = w + \alpha x y \quad (\alpha = 1.0)$

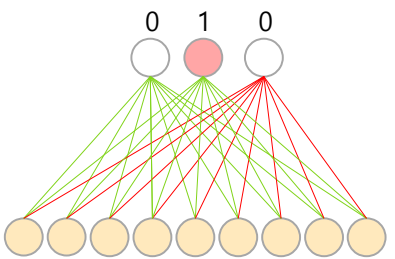
2) Second iteration:



$$w[:,0] = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix} + \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} \times y^{(2)}[0] \times (0) = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix}$$



$$w[:,1] = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} \times y^{(2)}[1] \times (1) = \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix}$$



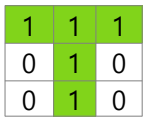
$$w[:,2] = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} \times y^{(2)}[2] \times (0) = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule (2) – Training stage

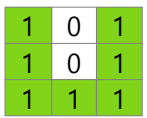
$$w = w + \alpha x y$$

- **Multiclass classification**



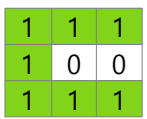
$x^{(1)}$ = [1, 1, 1, 0, 1, 0, 0, 1, 0]

$y^{(1)}$ = [1, 0, 0] → **' T '**



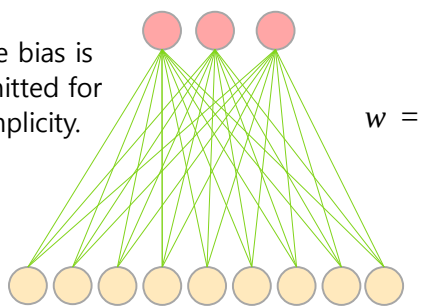
$x^{(2)}$ = [1, 0, 1, 1, 0, 1, 1, 1, 1]

$y^{(2)}$ = [0, 1, 0] → **' U '**



$x^{(3)}$ = [1, 1, 1, 1, 0, 0, 1, 1, 1]

$y^{(3)}$ = [0, 0, 1] → **' C '**



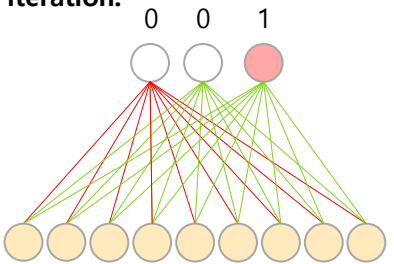
The bias is omitted for simplicity.

$$w = \begin{bmatrix} 1&1&0\\ 1&0&0\\ 1&1&0\\ 0&1&0\\ 1&0&0\\ 0&1&0\\ 0&1&0\\ 1&1&0\\ 0&1&0 \end{bmatrix} \rightarrow \begin{bmatrix} 1&1&1\\ 1&0&1\\ 1&1&1\\ 0&1&1\\ 1&0&0\\ 0&1&0\\ 0&1&1\\ 1&1&1\\ 0&1&1 \end{bmatrix}$$

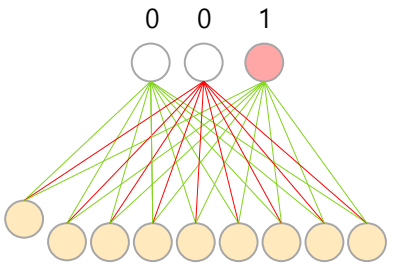
As the iterations progress, the weights tend to increase. To prevent this, we can normalize the weights at each step.

$$w = w + \alpha x y \quad (\alpha = 1.0)$$

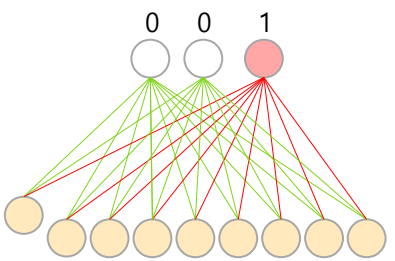
3) Third iteration:



$$w[:,0] = \begin{bmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{bmatrix} + \begin{bmatrix} 1\\1\\1\\1\\0\\0\\1\\1\\1 \end{bmatrix} \times y^{(3)}[0] (0) = \begin{bmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{bmatrix}$$



$$w[:,1] = \begin{bmatrix} 1\\0\\1\\1\\0\\1\\1\\1\\1 \end{bmatrix} + \begin{bmatrix} 1\\1\\1\\1\\0\\0\\1\\1\\1 \end{bmatrix} \times y^{(3)}[1] (0) = \begin{bmatrix} 1\\0\\1\\1\\0\\1\\1\\1\\1 \end{bmatrix}$$



$$w[:,2] = \begin{bmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{bmatrix} + \begin{bmatrix} 1\\1\\1\\1\\0\\0\\1\\1\\1 \end{bmatrix} \times y^{(3)}[2] (1) = \begin{bmatrix} 1\\1\\1\\1\\0\\0\\1\\1\\1 \end{bmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule (2) – Prediction stage

Multiclass classification

1	1	1
0	1	0
0	1	0

$x^{(1)} = [\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y^{(1)} = [1,\ 0,\ 0] \rightarrow$ **' T '**

1	0	1
1	0	1
1	1	1

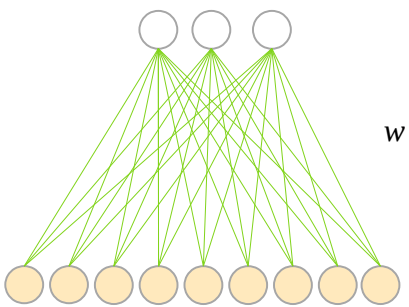
$x^{(2)} = [\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1]$

$y^{(2)} = [0,\ 1,\ 0] \rightarrow$ **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)} = [\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1]$

$y^{(3)} = [0,\ 0,\ 1] \rightarrow$ **' C '**



$$w = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$w = w + \alpha\, x\, y$ (assume: $\alpha = 1.0$)

Prediction

1	0	1
0	1	0
0	1	0

$x_t^{(1)} = [\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y_t^{(1)} = softmax(x_t^{(1)} \cdot w)$

$$x_t^{(1)} \quad w$$

$$[1\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0] \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = [4\ 3\ 3] \xrightarrow{\text{softmax}} [\mathbf{0.58}\ 0.21\ 0.21]$$

The first neuron is most likely to be activated.

$x_t^{(1)}$ looks like the letter 'T'.

0	1	1
1	0	0
0	1	1

$x_t^{(2)} = [\ 0,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 1,\ 1]$

$y_t^{(2)} = softmax(x_t^{(2)} \cdot w)$

$$[1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1] \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = [3\ 4\ 5] \xrightarrow{\text{softmax}} [0.09\ 0.24\ \mathbf{0.67}]$$

The third neuron is most likely to be activated.

$x_t^{(2)}$ looks like the letter 'C'.

1	0	1
0	0	1
1	1	0

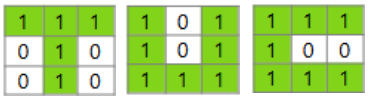
$x_t^{(3)} = [\ 0,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 1,\ 1]$

$y_t^{(3)} = softmax(x_t^{(3)} \cdot w) = [0.09\ \mathbf{0.67}\ 0.24]$ → $x_t^{(3)}$ looks like the letter 'U'.

MX-AI

■ A simple example of pattern recognition using Hebb's rule

```
# [MXDL-15-01] 1.Hebb_rule.py
# A simple example of pattern recognition using Hebb's rule
import numpy as np

# Input data: shape = (3, 9)
x = np.array([[1, 1, 1, 0, 1, 0, 0, 1, 0],  # the letter 'T'
              [1, 0, 1, 1, 0, 1, 1, 1, 1],  # the letter 'U'
              [1, 1, 1, 1, 0, 0, 1, 1, 1]]) # the letter 'C'

# Output data: shape = (3, 3)
y = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
labels = np.array(['T', 'U', 'C'])

# Initialize the connection weights
w = np.zeros((x.shape[1],y.shape[0]))  # (9, 3)

# Update w with the Hebbian rule
alpha = 0.1
for i in range(100):
    for i in range(x.shape[0]):
        w += alpha * np.outer(x[i], y[i])
        w = (w - w.mean()) / np.std(w)    # Normalize

print("\nWeights (w):\n\n{}".format(w.round(3)))

# Test data: corrupted images, shape = (3, 9)
xt = np.array([[ 1, 0, 1, 0, 1, 0, 0, 1, 0], # corrupted 'T'
               [ 0, 1, 1, 1, 0, 0, 0, 1, 1], # corrupted 'C'
               [ 1, 0, 1, 0, 0, 1, 1, 1, 0]])# corrupted 'U'
```

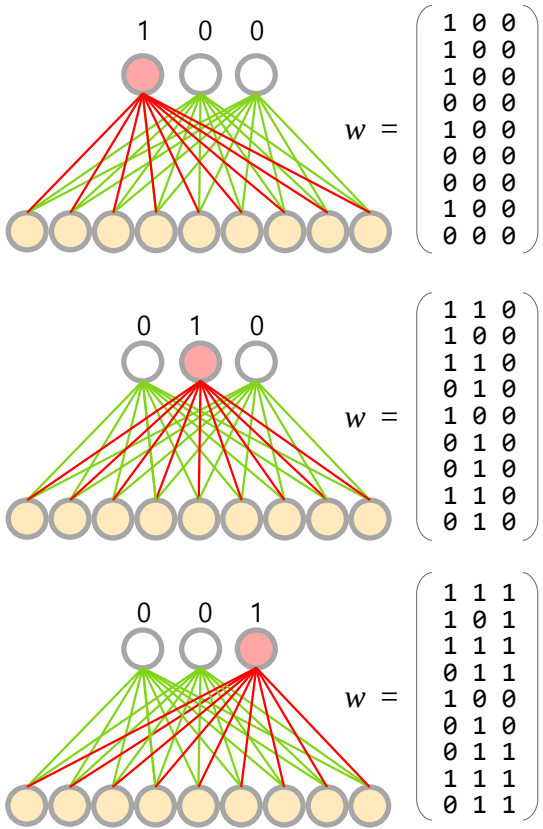


```
def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

# For each corrupted image, we determine which neuron
# in the output layer is activated.
yt = softmax(xt @ w)
print("\ny = x @ w: \n{}".format(yt.round(2)))

y_hat = np.argmax(yt, axis=1)

# Predictions:
print("\nPrediction results:")
for i in range(xt.shape[0]):
    print("\n{} --> It looks like the letter '{}'."\
          .format(xt[i].reshape(3,3), labels[y_hat[i]]))
```

```
yt = xt @ w:

[[0.81 0.09 0.1 ]
 [0.01 0.09 0.9 ]
 [0.01 0.87 0.11]]
```

```
Weights (w):

[[ 0.64  0.63  0.67]
 [ 0.64 -1.54  0.67]
 [ 0.64  0.63  0.67]
 [-1.54  0.63  0.67]
 [ 0.64 -1.54 -1.54]
 [-1.54  0.63 -1.54]
 [-1.54  0.63  0.67]
 [ 0.64  0.63  0.67]
 [-1.54  0.63  0.67]]
```

```
Prediction results:
[[1 0 1]
 [0 1 0]
 [0 1 0]] --> It looks like the letter 'T'.

[[0 1 1]
 [1 0 0]
 [0 1 1]] --> It looks like the letter 'C'.

[[1 0 1]
 [0 0 1]
 [1 1 0]] --> It looks like the letter 'U'.
```



15-2. Hebbian learning (2)

Weight decay in Hebbian learning

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ Hebbian learning rule with weight decay

▪ Simple Hebbian learning

1	1	1
0	1	0
0	1	0

$x^{(1)} = [\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y^{(1)} = [1,\ 0,\ 0] \rightarrow$ **'T'**

1	0	1
1	0	1
1	1	1

$x^{(2)} = [\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1]$

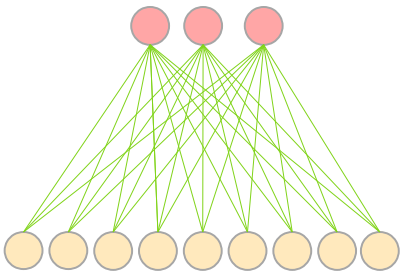
$y^{(2)} = [0,\ 1,\ 0] \rightarrow$ **'U'**

1	1	1
1	0	0
1	1	1

$x^{(3)} = [\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1]$

$y^{(3)} = [0,\ 0,\ 1] \rightarrow$ **'C'**

$$y_i = f\left(\sum_j w_{ji} x_j\right)$$

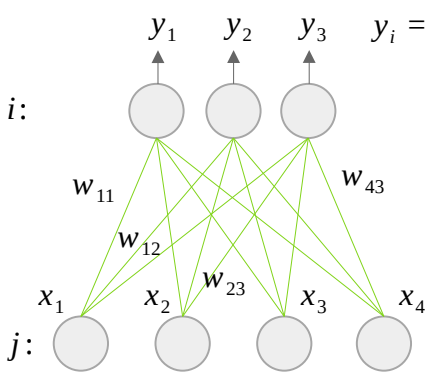


$$w = w + \alpha\, x\, y \quad (\alpha = 1.0)$$

$$w = \begin{pmatrix} 1&0&0\\1&0&0\\1&0&0\\0&0&0\\1&0&0\\0&0&0\\0&0&0\\1&0&0\\0&0&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&1&1\\1&0&1\\1&1&1\\0&1&1\\1&0&0\\0&1&0\\0&1&1\\1&1&1\\0&1&1 \end{pmatrix} \rightarrow \begin{pmatrix} 2&2&2\\2&0&2\\2&2&2\\0&2&2\\2&0&0\\0&2&0\\0&2&2\\2&2&2\\0&2&2 \end{pmatrix} \rightarrow \begin{pmatrix} 3&3&3\\3&0&3\\3&3&3\\0&3&3\\3&0&0\\0&3&0\\0&3&3\\3&3&3\\0&3&3 \end{pmatrix}$$

As the iterations progress, the weights tend to increase. To prevent this, we can normalize the weights at each step.

▪ Hebbian learning with weight decay proposed by Stephen Grossberg (1960s)



$$y_i = f\left(\sum_i w_{ji} x_j\right)$$

$$w = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

▪ Instar learning rule

$$\Delta w_{ji} = \alpha\left(x_j - w_{ji}\right) y_i$$

$$w_{ji} \leftarrow w_{ji} + \alpha\left(x_j - w_{ji}\right) y_i$$

ex) $w_{23} \leftarrow w_{23} + \alpha\left(x_2 - w_{23}\right) y_3$

▪ Outstar learning rule

$$\Delta w_{ji} = \alpha\left(y_i - w_{ji}\right) x_j$$

$$w_{ji} \leftarrow w_{ji} + \alpha\left(y_i - w_{ji}\right) x_j$$

ex) $w_{23} \leftarrow w_{23} + \alpha\left(y_3 - w_{23}\right) x_2$

MX-AI

Interpretation of Hebbian learning rule with weight decay

$$w_t = w_{t-1} + \alpha(x_t - w_{t-1})y_t$$

if y is either 0 or 1:

$$w_t = \begin{cases} w_{t-1} + \alpha(x_t - w_{t-1}) & \leftarrow y = 1 \\ w_{t-1} & \leftarrow y = 0 \end{cases}$$

$$w_t = \frac{1}{t}\sum_{k=1}^{t} x_k \quad \leftarrow \text{simple average}$$

$$= \frac{x_1 + x_2 + \ldots + x_{t-1} + x_t}{t-1} \cdot \frac{t-1}{t}$$

$$= \left(w_{t-1} + \frac{x_t}{t-1}\right) \cdot \frac{t-1}{t}$$

$$= w_{t-1} \cdot \frac{t-1}{t} + \frac{x_t}{t}$$

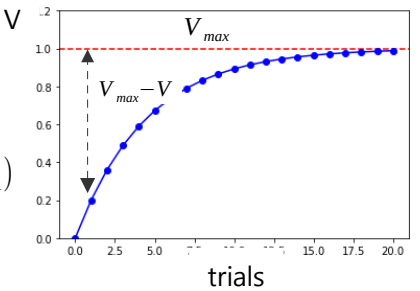
$$= w_{t-1} + \frac{1}{t}(x_t - w_{t-1}) \quad \leftarrow \text{simple average}$$

$$w_t = w_{t-1} + \alpha(x_t - w_{t-1}) \quad \leftarrow \text{moving average}$$

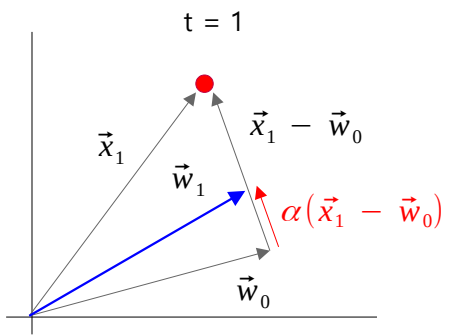
The Rescorla-Wagner Model of classical conditioning

$$\Delta V = \alpha(V_{max} - V)$$

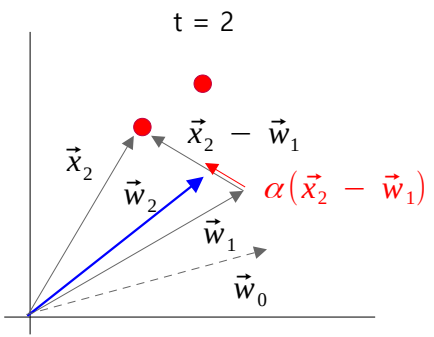
$$V_t = V_{t-1} + \alpha(V_{max} - V_{t-1})$$



- **Geometric interpretation of x and w**

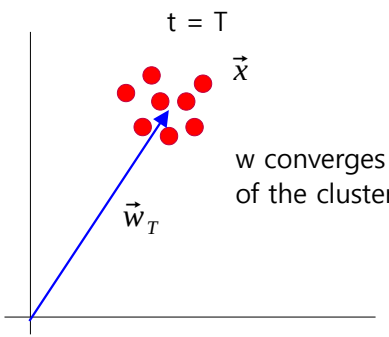


$$\vec{w}_1 = \vec{w}_0 + \alpha(\vec{x}_1 - \vec{w}_0)$$



$$\vec{w}_2 = \vec{w}_1 + \alpha(\vec{x}_2 - \vec{w}_1)$$

...



w converges to the center of the cluster of x.

MX-AI

A simple example of pattern recognition using Hebb's rule with weight decay – Training stage

- **Multiclass classification**

1	1	1
0	1	0
0	1	0

$x^{(1)}$ = [1, 1, 1, 0, 1, 0, 0, 1, 0]

$y^{(1)}$ = 0 → [1, 0, 0] → **' T '**

1	0	1
1	0	1
1	1	1

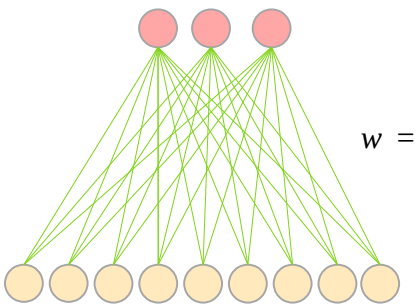
$x^{(2)}$ = [1, 0, 1, 1, 0, 1, 1, 1, 1]

$y^{(2)}$ = 1 → [0, 1, 0] → **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)}$ = [1, 1, 1, 1, 0, 0, 1, 1, 1]

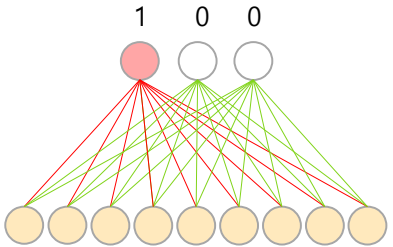
$y^{(3)}$ = 2 → [0, 0, 1] → **' C '**



$$w = \begin{pmatrix} 0&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0\\0&0&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&0&0\\1&0&0\\1&0&0\\0&0&0\\1&0&0\\0&0&0\\0&0&0\\1&0&0\\0&0&0 \end{pmatrix}$$

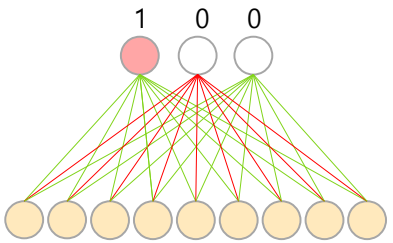
$$w_{ji} = w_{ji} + \alpha(x_j - w_{ji})y_i \quad (\alpha = 1.0)$$

- **First iteration:**



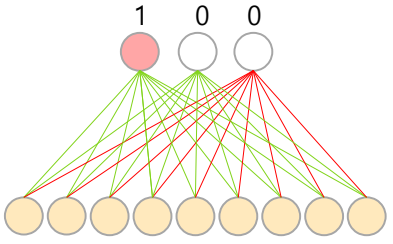
$w[:, 0] + (x^{(1)} - w[:, 0]) \times y^{(1)}[0]$

$$w[:,0] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \left[\begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} - \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} \right] \times 1 = \begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix}$$



$w[:, 1] + (x^{(1)} - w[:, 1]) \times y^{(1)}[1]$

$$w[:,1] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \left[\begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} - \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} \right] \times 0 = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}$$



$w[:, 2] + (x^{(1)} - w[:, 2]) \times y^{(1)}[2]$

$$w[:,2] = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} + \left[\begin{pmatrix} 1\\1\\1\\0\\1\\0\\0\\1\\0 \end{pmatrix} - \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix} \right] \times 0 = \begin{pmatrix} 0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule with weight decay – Training stage

- **Multiclass classification**

1	1	1
0	1	0
0	1	0

$x^{(1)}$ = [1, 1, 1, 0, 1, 0, 0, 1, 0]

$y^{(1)}$ = 0 → [1, 0, 0] → **'T'**

1	0	1
1	0	1
1	1	1

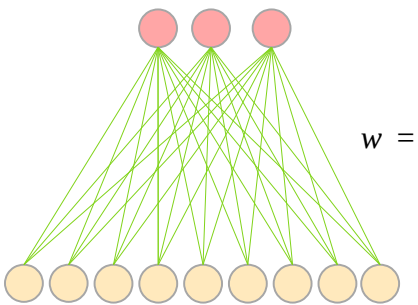
$x^{(2)}$ = [1, 0, 1, 1, 0, 1, 1, 1, 1]

$y^{(2)}$ = 1 → [0, 1, 0] → **'U'**

1	1	1
1	0	0
1	1	1

$x^{(3)}$ = [1, 1, 1, 1, 0, 0, 1, 1, 1]

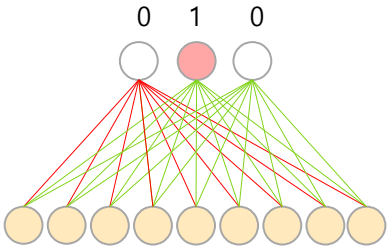
$y^{(3)}$ = 2 → [0, 0, 1] → **'C'**



$$w = \begin{pmatrix} 1&0&0\\1&0&0\\1&0&0\\0&0&0\\1&0&0\\0&0&0\\0&0&0\\1&0&0\\0&0&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&1&0\\1&0&0\\1&1&0\\0&1&0\\1&0&0\\0&1&0\\0&1&0\\1&1&0\\0&1&0 \end{pmatrix}$$

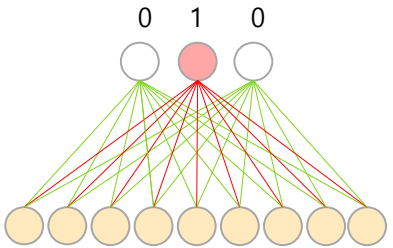
$$w_{ji} = w_{ji} + \alpha(x_j - w_{ji})y_i \quad (\alpha = 1.0)$$

- **Second iteration:**



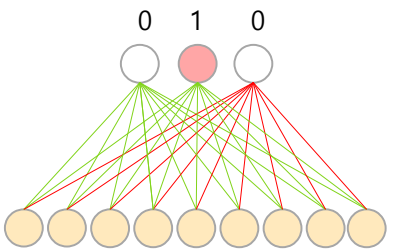
$w[:, 0] + (x^{(2)} - w[:, 0]) \times y^{(2)}[0]$

$$w[:,0] = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix} + \left[\begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} - \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix}\right] \times 0 = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix}$$



$w[:, 1] + (x^{(2)} - w[:, 1]) \times y^{(2)}[1]$

$$w[:,1] = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \left[\begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} - \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix}\right] \times 1 = \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix}$$



$w[:, 2] + (x^{(2)} - w[:, 2]) \times y^{(2)}[2]$

$$w[:,2] = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \left[\begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} - \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix}\right] \times 0 = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule with weight decay – Training stage

- **Multiclass classification**

1	1	1
0	1	0
0	1	0

$x^{(1)}$ = [1, 1, 1, 0, 1, 0, 0, 1, 0]

$y^{(1)}$ = 0 → [1, 0, 0] → **' T '**

1	0	1
1	0	1
1	1	1

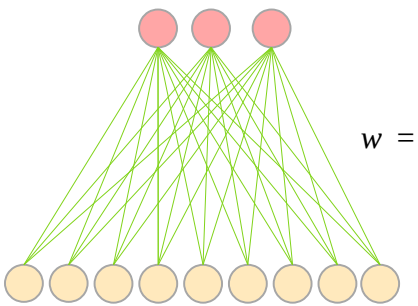
$x^{(2)}$ = [1, 0, 1, 1, 0, 1, 1, 1, 1]

$y^{(2)}$ = 1 → [0, 1, 0] → **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)}$ = [1, 1, 1, 1, 0, 0, 1, 1, 1]

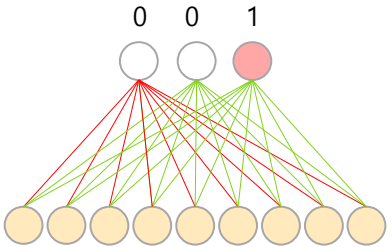
$y^{(3)}$ = 2 → [0, 0, 1] → **' C '**



$$w = \begin{pmatrix} 1&1&0\\1&0&0\\1&1&0\\0&1&0\\1&0&0\\0&1&0\\0&1&0\\1&1&0\\0&1&0 \end{pmatrix} \rightarrow \begin{pmatrix} 1&1&1\\1&0&1\\1&1&1\\0&1&1\\1&0&0\\0&1&0\\0&1&1\\1&1&1\\0&1&1 \end{pmatrix}$$

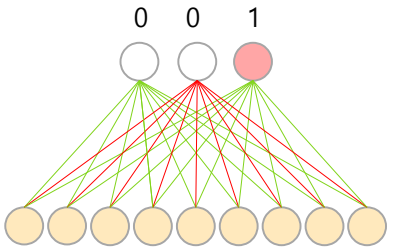
$$w_{ji} = w_{ji} + \alpha(x_j - w_{ji})y_i \quad (\alpha = 1.0)$$

- **Third iteration:**



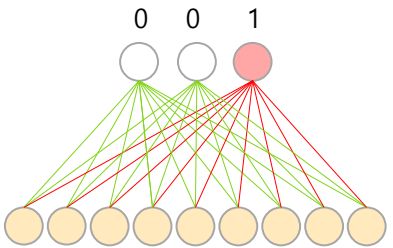
$w[:, 0] + (x^{(3)} - w[:, 0]) \times y^{(3)}[0]$

$$w[:,0] = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix} + \left[\begin{pmatrix}1\\1\\1\\1\\0\\0\\1\\1\\1\end{pmatrix} - \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix}\right] \times 0 = \begin{pmatrix}1\\1\\1\\0\\1\\0\\0\\1\\0\end{pmatrix}$$



$w[:, 1] + (x^{(3)} - w[:, 1]) \times y^{(3)}[1]$

$$w[:,1] = \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix} + \left[\begin{pmatrix}1\\1\\1\\1\\0\\0\\1\\1\\1\end{pmatrix} - \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix}\right] \times 0 = \begin{pmatrix}1\\0\\1\\1\\0\\1\\1\\1\\1\end{pmatrix}$$



$w[:, 2] + (x^{(2)} - w[:, 2]) \times y^{(3)}[2]$

$$w[:,2] = \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} + \left[\begin{pmatrix}1\\1\\1\\1\\0\\0\\1\\1\\1\end{pmatrix} - \begin{pmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\end{pmatrix}\right] \times 1 = \begin{pmatrix}1\\1\\1\\1\\0\\0\\1\\1\\1\end{pmatrix}$$

MX-AI

A simple example of pattern recognition using Hebb's rule with weight decay – Prediction stage

Multiclass classification

1	1	1
0	1	0
0	1	0

$x^{(1)} = [\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y^{(1)} = [1,\ 0,\ 0] \rightarrow$ **' T '**

1	0	1
1	0	1
1	1	1

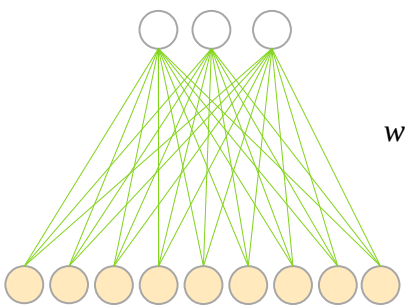
$x^{(2)} = [\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1]$

$y^{(2)} = [0,\ 1,\ 0] \rightarrow$ **' U '**

1	1	1
1	0	0
1	1	1

$x^{(3)} = [\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1]$

$y^{(3)} = [0,\ 0,\ 1] \rightarrow$ **' C '**



$$w = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$w = w + \alpha\, x\, y$ (assume: $\alpha = 1.0$)

Prediction

1	0	1
0	1	0
0	1	0

$x_t^{(1)} = [\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$y_t^{(1)} = softmax(x_t^{(1)} \cdot w)$

$$[1\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0] \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = [4\ 3\ 3] \xrightarrow{\text{softmax}} [\mathbf{0.58}\ 0.21\ 0.21]$$

The first neuron is most likely to be activated.

$x_t^{(1)}$ looks like the letter 'T'.

0	1	1
1	0	0
0	1	1

$x_t^{(2)} = [\ 0,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 1,\ 1]$

$y_t^{(2)} = softmax(x_t^{(2)} \cdot w)$

$$[1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1] \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} = [3\ 4\ 5] \xrightarrow{\text{softmax}} [0.09\ 0.24\ \mathbf{0.67}]$$

The third neuron is most likely to be activated.

$x_t^{(2)}$ looks like the letter 'C'.

1	0	1
0	0	1
1	1	0

$x_t^{(3)} = [\ 0,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 1,\ 1]$

$y_t^{(3)} = softmax(x_t^{(3)} \cdot w) = [0.09\ \mathbf{0.67}\ 0.24]$ → $x_t^{(3)}$ looks like the letter 'U'.

MX-AI

■ A simple example of pattern recognition using Hebb's rule with weight decay

```
# [MXDL-15-02] 2.Hebb_decay.py
# A simple example of pattern recognition using Hebb's rule
# with weight decay
import numpy as np

# Training stage: Input data: shape = (3, 9)
x = np.array([[1, 1, 1, 0, 1, 0, 0, 1, 0],  # the letter 'T'
              [1, 0, 1, 1, 0, 1, 1, 1, 1],  # the letter 'U'
              [1, 1, 1, 1, 0, 0, 1, 1, 1]]) # the letter 'C'

# Output data: shape = (3, 3)
y = np.array([0, 1, 2])
labels = np.array(['T', 'U', 'C'])

# Initialize the connection weights
w = np.zeros((y.shape[0], x.shape[1]))  # (3, 9)

# Update w using Hebb's rule with weight decay
alpha = 0.1  # learning rate
for k in range(100):
    for xi, yi in zip(x, y):
        w[yi, :] += alpha * (xi - w[yi, :])
print("\nWeights (w):\n\n{}". format(w.T.round(2)))

# Test stage: Corrupted images: shape = (3, 9)
xt = np.array([[ 1, 0, 1, 0, 1, 0, 0, 1, 0],  # corrupted 'T'
               [ 0, 1, 1, 1, 0, 0, 0, 1, 1],  # corrupted 'C'
               [ 1, 0, 1, 0, 0, 1, 1, 1, 0]]) # corrupted 'U'
```

$$w = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$w_t = \begin{cases} w_{t-1} + \alpha(x_t - w_{t-1}) & (y = 1) \\ w_{t-1} & (y = 0) \end{cases}$$

```
def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

# For each corrupted image, we determine which neuron
# in the output layer is activated.
yt = softmax(xt @ w.T)
print("\nyt = xt @ w: \n{}".format(yt.round(2)))

y_hat = np.argmax(yt, axis=1)

# Predictions:
print("\nPrediction results:")
for i in range(xt.shape[0]):
    print("\n{} --> It looks like the letter '{}'."\
          .format(xt[i].reshape(3,3), labels[y_hat[i]]))
```

```
yt = xt @ w:

[[0.58 0.21 0.21]
 [0.09 0.24 0.67]
 [0.09 0.67 0.24]]
```

```
Weights (w):

[[1. 1. 1.]
 [1. 0. 1.]
 [1. 1. 1.]
 [0. 1. 1.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 1. 1.]
 [1. 1. 1.]
 [0. 1. 1.]]
```

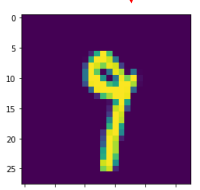
```
Prediction results:
[[1 0 1]
 [0 1 0]
 [0 1 0]] --> It looks like the letter 'T'.

[[0 1 1]
 [1 0 0]
 [0 1 1]] --> It looks like the letter 'C'.

[[1 0 1]
 [0 0 1]
 [1 1 0]] --> It looks like the letter 'U'.
```

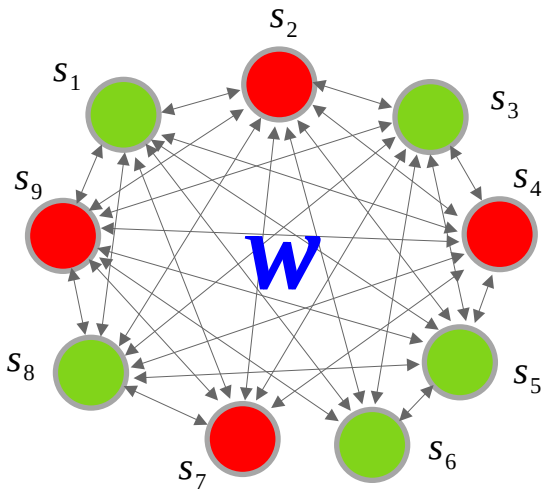
MX-AI

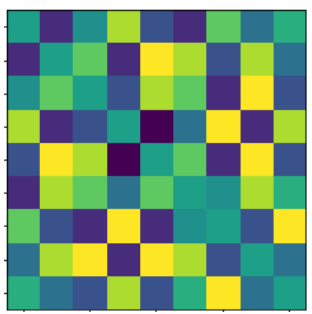
■ MNIST image classification using Hebb's rule with weight decay

```
# [MXDL-15-02] 3.Hebb(mnist).py
# MNIST image classification using Hebb's rule with weight
# decay
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

# Load MNIST data set
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)    # x: (70000, 784), y: (70000, 1)

y = y.reshape(-1,)
x_train, x_test, y_train, y_test = train_test_split(x, y)

# Initialize the connection weights # (10, 784)
w = np.zeros((10, x_train.shape[1]))

# Update w using Hebb's rule with weight decay
alpha = 0.01     # learning rate
n_iters = 50     # number of iterations
for k in range(n_iters):
    for xi, yi in zip(x_train, y_train):
        w[yi, :] += alpha * (xi - w[yi, :])

    print("Epoch:", k+1)
```

```
def softmax(x):
    e = np.exp(x - np.max(x, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

# For each corrupted image, we determine which neuron
# in the output layer is activated.
y_prob = softmax(x_test @ w.T)
y_pred = np.argmax(y_prob, axis=1).reshape(-1,)
acc = (y_test == y_pred).mean()
print('Accuracy on the test data ={:.2f}'.format(acc))

y_prob = softmax(x_test @ w.T):

  0     1     2     3     4     5     6     7     8     9  ← labels
[[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.99, 0.  , 0.  , 0.  ],
 [0.  , 0.  , 0.  , 0.  , 0.21, 0.  , 0.77, 0.  , 0.  , 0.01],
 [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  ],
 ...
 [0.  , 0.  , 0.15, 0.  , 0.  , 0.  , 0.  , 0.  , 0.85, 0.  ],
 [0.  , 0.  , 0.  , 0.05, 0.  , 0.  , 0.  , 0.29, 0.34, 0.32],
 [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.98, 0.  , 0.02]])

Epoch: 1
Epoch: 2
Epoch: 3
...
Epoch: 48
Epoch: 49
Epoch: 50
Accuracy on the test data = 0.64
```





$w =$ 

15-3. Hopfield Netowork (1)

Storing and recalling information

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

The original paper of Hopfield network

- The Hopfield network was first introduced by John Hopfield in 1982.

Proc. NatL Acad. Sci. USA
Vol. 79, pp. 2554-2558, April 1982
Biophysics

Neural networks and physical systems with emergent collective computational abilities

(associative memory/parallel processing/categorization/content-addressable memory/fail-soft devices)

J. J. HOPFIELD
Division of Chemistry and Biology, California Institute of Technology, Pasadena, California 91125; and Bell Laboratories, Murray Hill, New Jersey 07974
Contributed by John J. Hopfield, January 15, 1982

ABSTRACT Computational properties of use to biological organisms or to the construction of computers can emerge as collective properties of systems-having a large number of simple equivalent components (or neurons). The physical meaning of content-addressable memory is described by an appropriate phase space flow of the state of a system. A model of such a system is given, based on aspects of neurobiology but readily adapted to integrated circuits. The collective properties of this model produce a content-addressable memory which correctly yields an entire memory from any subpart of sufficient size. The algorithm for the time evolution of the state of the system is based on asynchronous parallel processing. Additional emergent collective properties include some capacity for generalization, familiarity recognition, categorization, error correction, and time sequence retention. The collective properties are only weakly sensitive to details of the modeling or the failure of individual devices.

Given the dynamical electrochemical properties of neurons and their interconnections (synapses), we readily understand schemes that use a few neurons to obtain elementary useful biological behavior (1-3). Our understanding of such simple circuits in electronics allows us to plan larger and more complex circuits which are essential to large computers. Because evolution has no such plan, it becomes relevant to ask whether the ability of large collections of neurons to perform "computational" tasks may in part be a spontaneous collective consequence of having a large number of interacting simple neurons.

MX-AI

■ The structure of the Hopfield network

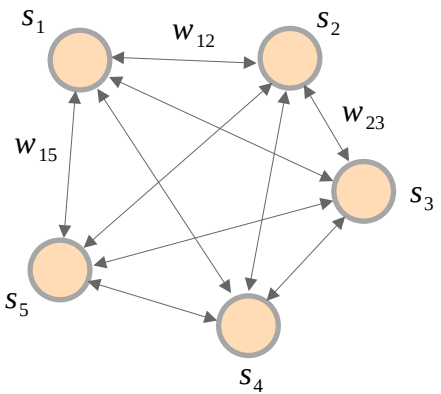
$$s^{(1)} = [s_1^{(1)},\ s_2^{(1)},\ s_3^{(1)},\ s_4^{(1)},\ s_5^{(1)}]$$

$$s^{(2)} = [s_1^{(2)},\ s_2^{(2)},\ s_3^{(2)},\ s_4^{(2)},\ s_5^{(2)}]$$

$$\vdots$$

binary state bipolar state

$$s_i = \{0,\ 1\} \rightarrow \{-1, +1\}$$



▪ **Storing information to a network by Hebb's rule**

bipolar state: -1 or +1

$$w_{ij} = \sum_{p=1}^{n} (2s_i^{(p)} - 1)(2s_j^{(p)} - 1) \qquad [2]$$

$$w_{ii} = 0 \qquad w_{ij} = w_{ji}$$

$$w = \begin{pmatrix} 0 & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & 0 & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & 0 & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & 0 & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & 0 \end{pmatrix}$$

▪ **Recalling information from the trained network**

$s^{(t)} = [s_1^{(t)},\ s_2^{(t)},\ s_3^{(t)},\ s_4^{(t)},\ s_5^{(t)}]$ ← partial information that we want to recall from the memory

Each state s_i is determined by the following rules:

$$s_i \rightarrow 1 \quad if \quad \sum_{j \neq i} w_{ij} s_j \quad > U_i$$
$$s_i \rightarrow 0 \quad if \quad \sum_{j \neq i} w_{ij} s_j \quad < U_i \qquad [1]$$

the activation value of neuron i.

ex: activation (i=1) = $w_{12}s_2 + w_{13}s_3 + w_{14}s_4 + w_{15}s_5$

$\hat{s}^{(t)} = [\hat{s}_1^{(t)},\ \hat{s}_2^{(t)},\ \hat{s}_3^{(t)},\ \hat{s}_4^{(t)},\ \hat{s}_5^{(t)}]$ ← Information retrieved from memory based on partial information.

- Each neuron i has two states: s_i = 0 ("not firing") and s_i = 1 ("firing at maximum rate"). When neuron i has a connection made to it from neuron j, the strength of connection is defined as w_{ij}. (Nonconnected neurons have w_{ij} 0.) The instantaneous state of the system is specified by listing the N values of s_i, so it is represented by a binary word of N bits.
- The state changes in time according to the following algorithm. For each neuron i there is a fixed threshold U_i. Each neuron i readjusts its state randomly in time. Thus, each neuron randomly and asynchronously evaluates whether it is above or below threshold and readjusts accordingly. (Unless otherwise stated, we choose U_i = 0.)

MX-AI

A simple example for storing and recalling image patterns.

- The process of storing image patterns in a Hopfield network.

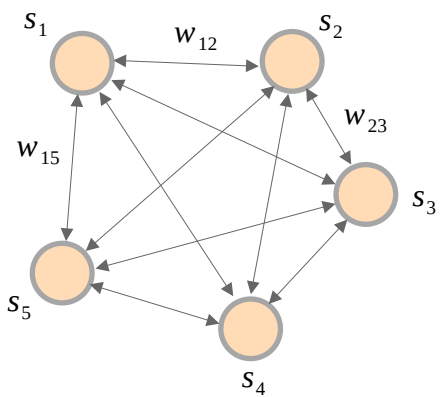
2 input patterns → bipolar state

$$s^{(1)} = [0,1,0,1,1] \longrightarrow 2s^{(1)} - 1 = [-1,+1,-1,+1,+1]$$

$$s^{(2)} = [1,0,1,1,0] \longrightarrow 2s^{(2)} - 1 = [+1,-1,+1,+1,-1]$$

$$s = [s_1, s_2, s_3, s_4, s_5]$$

$$s_i = \{0,\ 1\}$$



- **Storing the set of states s by Hebb's rule**

$$w_{ij} = \sum_{p=1}^{n} (2s_i^{(p)} - 1)(2s_j^{(p)} - 1)$$

$w_{ij} = w_{ji}$ ← The connection weights between neurons are symmetrical.

$w_{ii} = 0$ ← Each neuron has no self feedback (no connection to itself)

$$w = \begin{bmatrix} -1, & 1 \\ 1, & -1 \\ -1, & 1 \\ 1, & 1 \\ 1, & -1 \end{bmatrix} \cdot \begin{bmatrix} -1, & 1, & -1, & 1, & 1 \\ 1, & -1, & 1, & 1, & -1 \end{bmatrix} \xrightarrow{w_{ii} = 0}$$

$(2s - 1)^{\mathrm{T}}$ $(2s - 1)$

This is a symmetric matrix with a zero diagonal.

	1	2	3	4	5
1	0	-2	2	0	-2
2	-2	0	-2	0	2
3	2	-2	0	0	-2
4	0	0	0	0	0
5	-2	2	-2	0	0

Two input patterns are stored in the connection weights w.

MX-AI

A simple example for storing and recalling image patterns.

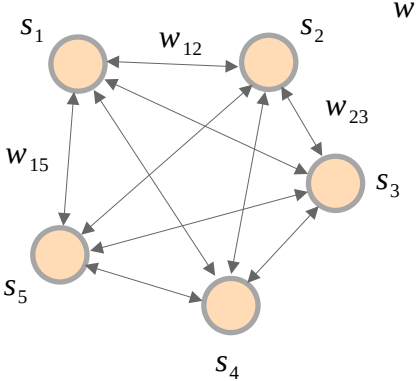
- The process of recalling an image pattern from a Hopfield network.

2 input patterns bipolar state

$$s^{(1)} = [0,1,0,1,1] \longrightarrow 2s^{(1)} - 1 = [-1,+1,-1,+1,+1]$$

$$s^{(2)} = [1,0,1,1,0] \longrightarrow 2s^{(2)} - 1 = [+1,-1,+1,+1,-1]$$

$s_i = \{0,\ 1\}$



$$w = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & -2 & 2 & 0 & -2 \\ -2 & 0 & -2 & 0 & 2 \\ 2 & -2 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & 2 & -2 & 0 & 0 \end{pmatrix} \end{array}$$

- **Determine the states**

$$s_i \rightarrow 1 \quad if \sum_{j \neq i} w_{ij} s_j > U_i$$

$$s_i \rightarrow 0 \quad if \sum_{j \neq i} w_{ij} s_j < U_i$$

$$s^{(t)} = [0,1,0,1,0] \longrightarrow \hat{s}^{(t)} = [0,1,0,1,1] = s^{(1)}$$

i = [1, 3, 5, 2, 4] ← random order

i=1, j=[2,3,4,5] $\rightarrow \sum_{j \neq i} w_{ij} s_j^{(t)} = w_{12} s_2^{(t)} + w_{13} s_3^{(t)} + w_{14} s_4^{(t)} + w_{15} s_5^{(t)}$

= -2*1 + 2*0 + 0*1 + (-2)*0 = -2 → $s_1 = 0$

i=3, j=[1,2,4,5] $\rightarrow \sum_{j \neq i} w_{ij} s_j^{(t)} = w_{31} s_1^{(t)} + w_{32} s_2^{(t)} + w_{34} s_4^{(t)} + w_{35} s_5^{(t)}$

= 2*0 + (-2)*1 + 0*1 + (-2)*0 = -2 → $s_3 = 0$

i=5, j=[1,2,3,4] $\rightarrow \sum_{j \neq i} w_{ij} s_j^{(t)} = w_{51} s_1^{(t)} + w_{52} s_2^{(t)} + w_{53} s_3^{(t)} + w_{54} s_4^{(t)}$

= -2*0 + 2*1 + (-2)*0 + 0*1 = +2 → $s_5 = 1$

i=2, j=[1,3,4,5] $\rightarrow \sum_{j \neq i} w_{ij} s_j^{(t)} = w_{21} s_1^{(t)} + w_{23} s_3^{(t)} + w_{24} s_4^{(t)} + w_{25} s_5^{(t)}$

= -2*0 + (-2)*0 + 0*1 + 2*1= +2 → $s_2 = 1$

i=4, j=[1,2,3,5] $\rightarrow \sum_{j \neq i} w_{ij} s_j^{(t)} = w_{41} s_1^{(t)} + w_{42} s_2^{(t)} + w_{43} s_3^{(t)} + w_{45} s_5^{(t)}$

= 0*0 + 0*1 + 0*0 + 0*1= 0 → $s_4 = 1$

MX-AI

- A simple example code for storing and recalling image patterns.

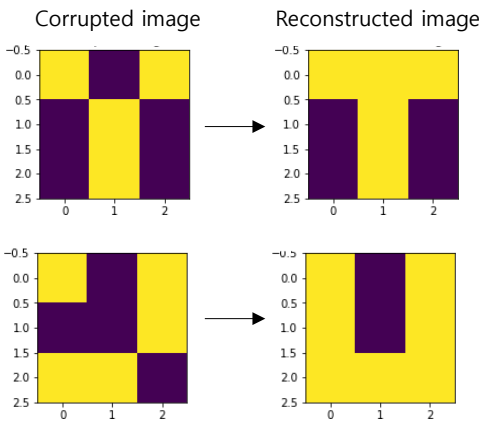
```
# [MXDL-15-03] 4.Hopfield(1).py
import numpy as np
import matplotlib.pyplot as plt

# Input data: shape = (2, 9)
s = np.array([[1, 1, 1, 0, 1, 0, 0, 1, 0],    # letter 'T'
              [1, 0, 1, 1, 0, 1, 1, 1, 1]])   # letter 'U'
n, m = s.shape

w = (2*s-1).T @ (2*s-1)
np.fill_diagonal(w, 0)
print(w.round(2))

sp = np.array([[ 1, 0, 1, 0, 1, 0, 0, 1, 0],  # corrupted 'T'
               [ 1, 0, 1, 0, 0, 1, 1, 1, 0]]) # corrupted 'U'
for p in sp:
    fig, ax = plt.subplots(1, 2, figsize=(8,2))
    ax[0].imshow(p.reshape(3,3))
    ax[0].set_title('corrupted image')

    for k in range(10):
        i_rnd = np.random.choice(m, (m,), replace=False)
        for i in i_rnd:
            activation = w[i, :] @ p
            if activation > 0:
                p[i] = 1
            elif activation == 0:
                ;
            else:
                p[i] = 0
    ax[1].imshow(p.reshape(3,3))
    ax[1].set_title('reconstructed image')
    plt.show()
```

$$w_{ij} = \sum_{p=1}^{n} (2s_i^{(p)} - 1)(2s_j^{(p)} - 1)$$

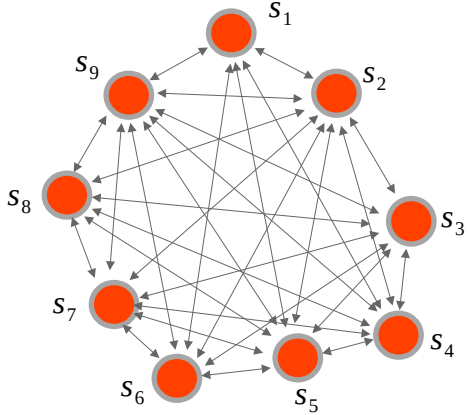
$$w_{ii} = 0$$

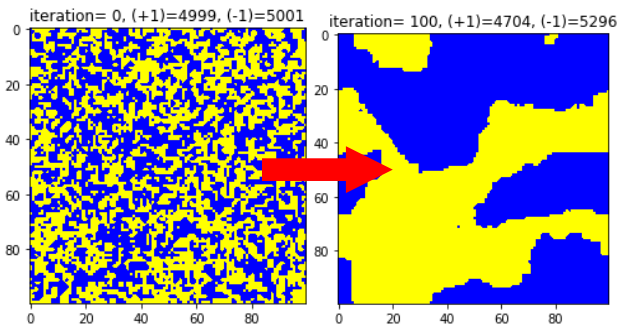
$$\sum_{j \neq i} w_{ij} s_j$$

```
Weights:
[[ 0  0  2  0  0  0  0  2  0]
 [ 0  0  0 -2  2 -2 -2  0 -2]
 [ 2  0  0  0  0  0  0  2  0]
 [ 0 -2  0  0 -2  2  2  0  2]
 [ 0  2  0 -2  0 -2 -2  0 -2]
 [ 0 -2  0  2 -2  0  2  0  2]
 [ 0 -2  0  2 -2  2  0  0  2]
 [ 2  0  2  0  0  0  0  0  0]
 [ 0 -2  0  2 -2  2  2  0  0]]
```



The corrupted images are successfully reconstructed.





15-4. Hopfield Netowork (2)

Ising model and energy function

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

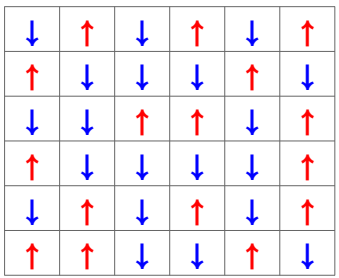
www.youtube.com/@meanxai

MX-AI

■ Ising model

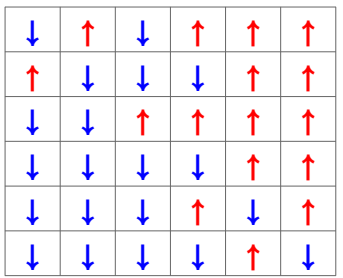
- The Ising model (or Lenz–Ising model), named after the physicists Ernst Ising and Wilhelm Lenz, is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic "spins" that can be in one of two states (+1 or −1).
- A system of magnetic particles can be modeled as a linear chain in one dimension or a lattice in two dimension, with one molecule or atom at each lattice site i.

- A magnet. A square lattice



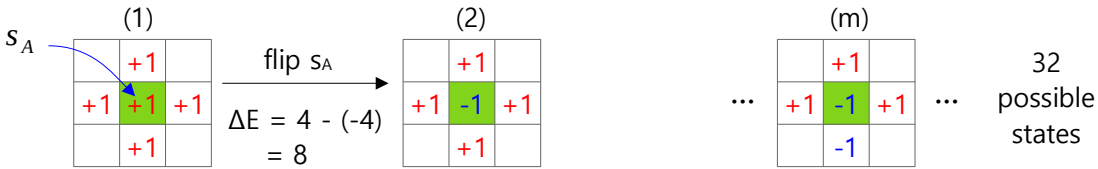
[High energy]

[↑: +1, ↓: -1]



[Low energy]

- considering 4 neighbors around s_A (assume J=1, h=0)



E_1 = -(1+1+1+1)(+1) = -4 ~ most likely p_1

E_2 = -(1+1+1+1)(-1) = 4 p_2

E_m = -(1+1+1-1)(-1) = 2 p_3

E_{32} p_{32}

- Energy of a configuration s given by the Hamiltonian function.

$$E(s) = -\sum_{<i,j>} J_{ij} s_i s_j - \mu \sum_j h_j s_j$$

- s_i : individual spin on each of a lattice site
- The notation ⟨i, j⟩ indicates that sites i and j are nearest neighbors.
- J_{ij} : interaction strength between two spins, s_i and s_j
- μ : magnetic moment
- h_j : external magnetic field interacting with s_j

- Boltzmann distribution

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

- Simulation (Metropolis algorithm)

$\Delta E_A < 0 \longrightarrow$ flip the spin s_A

$\Delta E_A > 0 \longrightarrow$ flip the spin s_A with probability $\exp\left(\frac{-\Delta E_A}{kT}\right)$

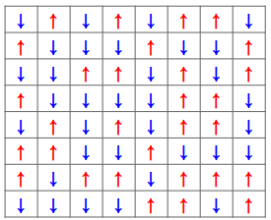
Reference: Ashkan Shekaari, MahmoudJafari, 2021, Theory and simulation of the Ising model. function Metropolis in 3.4.1 code1.py

MX-AI

- A simple simulation of the Ising model.

```
# [MXDL-15-04] 5.Ising_model.py
# Ising Model simulation by metropolis algorithm
# reference: Ashkan Shekaari, MahmoudJafari, 2021,
#            Theory and simulation of the Ising model.
import numpy as np
import numba as nb
import matplotlib.pyplot as plt
from matplotlib import colors
cmap = colors.ListedColormap(['blue', 'yellow'])

epochs = 1000
k = 100        # size of the square lattice, k * k
kT = 2.2       # critical temperature
lattice = np.random.choice([1,-1], [k, k])  # (-1 or +1)

@nb.jit(nopython=True)
def calculate_energy(sites):
    energy = 0
    for n in range(sites.shape[0]):
        for m in range(sites.shape[1]):
            ss = sites[(n + 1) % k, m] + sites[n, (m+1) % k] + \
                 sites[(n-1) % k, m] + sites[n,(m-1) % k]
            energy -= ss * sites[n, m]
    return energy

@nb.jit(nopython=True)
def change_state(sites, n_iters):
    for i in range(n_iters):
        n = np.random.randint(k)
        m = np.random.randint(k)
```

lattice sites

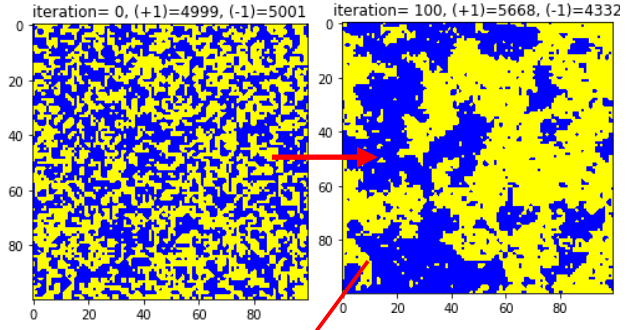


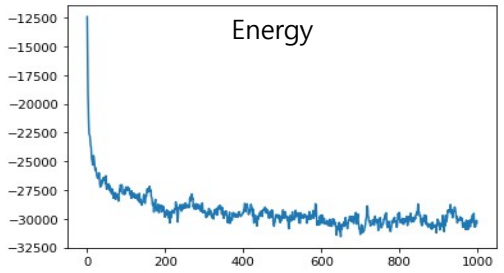
$$E(s) = -\sum_{<i,j>} J_{ij} s_i s_j - \mu \sum_j h_j s_j$$

```
        # Sum of the spins of the nearest neighbors around s.
        ss = sites[(n + 1) % k, m] + sites[n, (m+1) % k] + \
             sites[(n-1) % k, m] + sites[n,(m-1) % k]

        # dE = the energy difference before and after flipping
               the spin at a randomly chosen lattice site.
        dE = -ss * (-sites[n, m]) + ss * sites[n, m]

        if dE < 0:
            sites[n, m] *= -1            # flip the spin
        else:
            prob = np.exp(-dE / kT)
            if np.random.random() <= prob:
                sites[n, m] *= -1        # flip the spin
    return sites

energy = []
for t in range(epochs):
    lattice = change_state(lattice, k * k)
    energy.append(calculate_energy(lattice))

    # plot the changed lattice sites
    if t % 100 == 0:
        pos = str((lattice > 0).sum())
        neg = str((lattice < 0).sum())
        plt.figure(figsize=(4,4))
        plt.imshow(lattice, cmap=cmap)
        plt.title("iteration= "+str(t)+", (+1)="+pos+", (-1)="+neg)
        plt.show()
plt.plot(energy); plt.show()
```

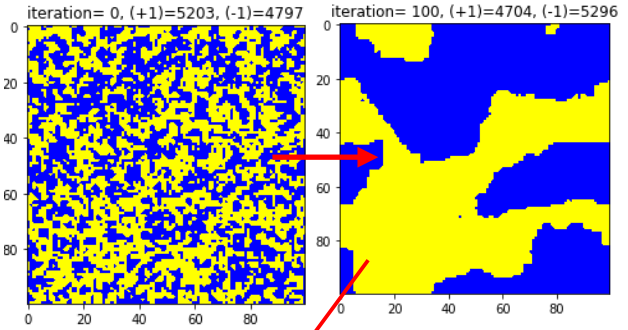
MX-AI

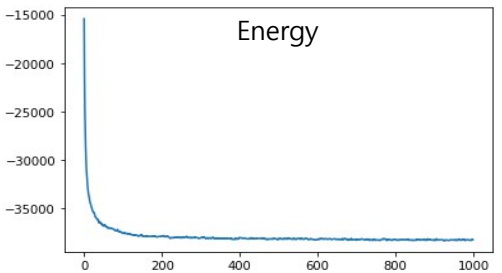
- A simple simulation of the Ising model.
  - **J = 1.0, kT = 2.2 (critical temperature)**



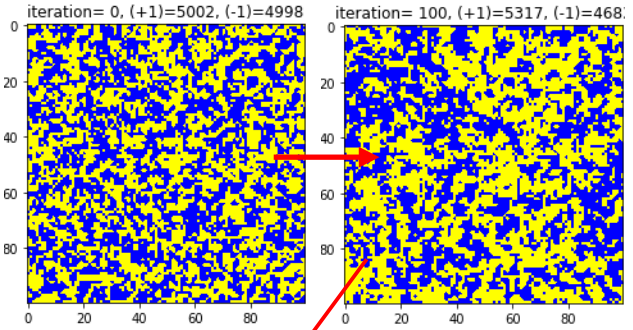


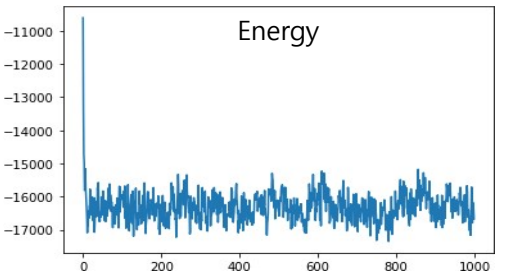
  - **J = 1.0, kT = 1.0 (below the critical temperature)**





  - **J = 1.0, kT = 3.0 (above the critical temperature)**





MX-AI

A simple example for storing and recalling image patterns using the energy function

- The process of storing image patterns in a Hopfield network.

T

1	1	1
0	1	0
0	1	0

$s^{(1)} = [\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0]$

$2s^{(1)}-1 = [+1,+1,+1,-1,+1,-1,-1,+1,-1]$

U

1	0	1
1	0	1
1	1	1

$s^{(2)} = [\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1,\ 1]$

$2s^{(2)}-1 = [+1,-1,+1,+1,-1,+1,+1,+1,+1]$

C

1	1	1
1	0	0
1	1	1

$s^{(3)} = [\ 1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1]$

$2s^{(3)}-1 = [+1,+1,+1,+1,-1,-1,+1,+1,+1]$

- **Storing a set of states s in the network by Hebb's rule.**

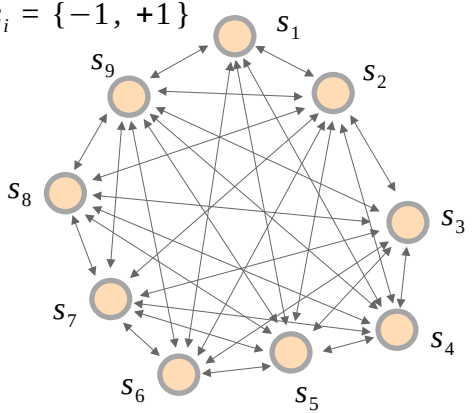
$$w_{ij} = \sum_{p=1}^{n} (2s_i^{(p)}-1)(2s_j^{(p)}-1)$$

$w_{ij} = w_{ji}$ ← The connection weights between neurons are symmetrical.

$w_{ii} = 0$ ← Each neuron has no self feedback (no connection to itself)

$s = [s_1, s_2, s_3, \ldots, s_9]$

$s_i = \{-1,\ +1\}$



$$w = (2s-1)^{\mathrm{T}} \bullet (2s-1)$$

$$(2s-1)^{\mathrm{T}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & 1 & 1 \end{pmatrix}$$

$$(2s-1) = \begin{pmatrix} 1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 \\ 1 & -1 & 1 & 1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 \end{pmatrix}$$

$w_{ii} = 0$ →

Three input patterns are stored in this matrix.

	1	2	3	4	5	6	7	8	9
1	0	1	3	1	-1	-1	1	3	1
2	1	0	1	-1	1	-3	-1	1	-1
3	3	1	0	1	-1	-1	1	3	1
4	1	-1	1	0	-3	1	3	1	3
5	-1	1	-1	-3	0	-1	-3	-1	-3
6	-1	-3	-1	1	-1	0	1	-1	1
7	1	-1	1	3	-3	1	0	1	3
8	3	1	3	1	-1	-1	1	0	1
9	1	-1	1	3	-3	1	3	1	0

This is a symmetric matrix with a zero diagonal.

MX-AI

A simple example for storing and recalling image patterns using the energy function

- The process of recalling an image pattern from a Hopfield network.

T

1	1	1
0	1	0
0	1	0

U

1	0	1
1	0	1
1	1	1

C

1	1	1
1	0	0
1	1	1

* To simplify notation, we will denote the bipolar state as s.

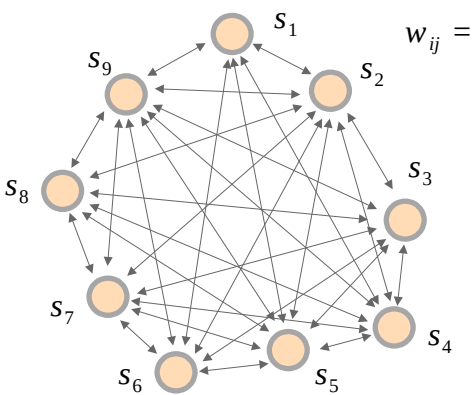
$s^{(1)}$ = [+1,+1,+1,-1,+1,-1,-1,+1,-1]

$s^{(2)}$ = [+1,-1,+1,+1,-1,+1,+1,+1,+1]

$s^{(3)}$ = [+1,+1,+1,+1,-1,-1,+1,+1,+1]

$$E(s) = -\sum_{<i,j>} J_{ij} s_i s_j - \mu \sum_j h_j s_j$$ - Energy function of the Ising model

$$E(s) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j$$ - Energy function of the Hopfield Network



$$w_{ij} = \sum_{p=1}^{n} s_i^{(p)} s_j^{(p)}$$

$$w = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 0 & 1 & 3 & 1 & -1 & -1 & 1 & 3 & 1 \\ 1 & 0 & 1 & -1 & 1 & -3 & -1 & 1 & -1 \\ 3 & 1 & 0 & 1 & -1 & -1 & 1 & 3 & 1 \\ 1 & -1 & 1 & 0 & -3 & 1 & 3 & 1 & 3 \\ -1 & 1 & -1 & -3 & 0 & -1 & -3 & -1 & -3 \\ -1 & -3 & -1 & 1 & -1 & 0 & 1 & -1 & 1 \\ 1 & -1 & 1 & 3 & -3 & 1 & 0 & 1 & 3 \\ 3 & 1 & 3 & 1 & -1 & -1 & 1 & 0 & 1 \\ 1 & -1 & 1 & 3 & -3 & 1 & 3 & 1 & 0 \end{pmatrix} \end{array}$$

- Reconstructing the corrupted pattern.

1	0	1
0	1	0
0	1	0

s = [+1,-1,+1,-1,+1,-1,-1,+1,-1]

i = [2, 3, 8, 5, 1, 6, 4, 7, 9] ← random order

i = 2: s = [+1,-1,+1,-1,+1,-1,-1,+1,-1]

s' = [+1,+1,+1,-1,+1,-1,-1,+1,-1] ← flip the spin s_2

$E_{before} = -0.5\, s^T \cdot w \cdot s = -12$

$E_{after} = -0.5\, s'^T \cdot w \cdot s' = -32$

$\Delta E = E_{after} - E_{before} = -20 < 0$ ← flip the spin s_2

i = 3: s = [+1,+1,+1,-1,+1,-1,-1,+1,-1]

s' = [+1,+1,-1,-1,+1,-1,-1,+1,-1] ← flip the spin s_3

$E_{before} = -0.5\, s^T \cdot w \cdot s = -32$

$E_{after} = -0.5\, s'^T \cdot w \cdot s' = -24$

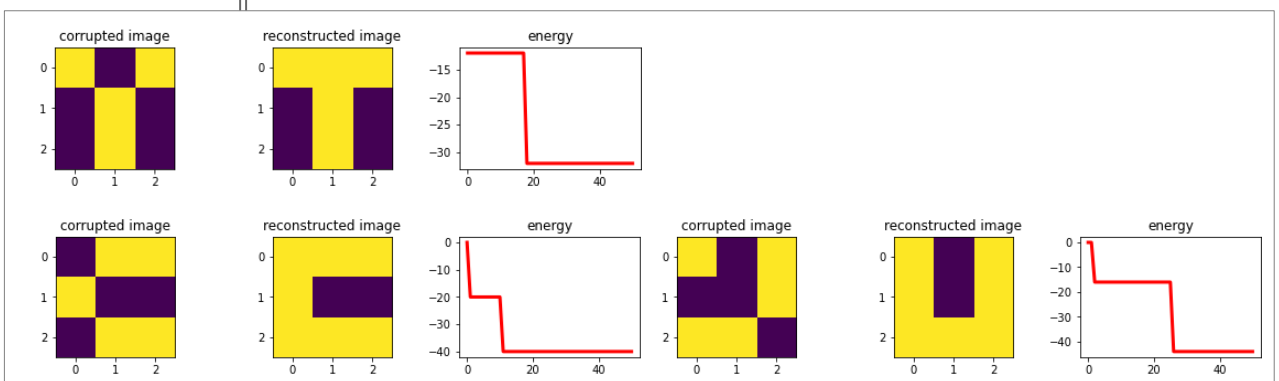
$\Delta E = E_{after} - E_{before} = 8 > 0$ ← don't flip the spin s_3

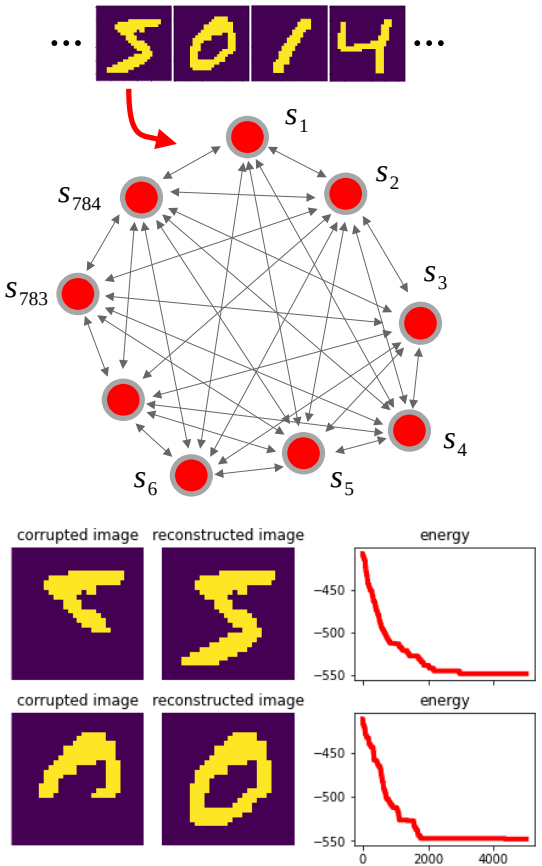
⋮ Update all other states in this way.

MX-AI

- A simple example code for storing and recalling image patterns using the energy function

```
# [MXDL-15-04] 6.Hopfield(2).py
import numpy as np
import matplotlib.pyplot as plt

# Input data: shape = (3, 9)
s = np.array([[1, 1, 1, 0, 1, 0, 0, 1, 0],      # the letter 'T'
              [1, 0, 1, 1, 0, 1, 1, 1, 1],      # the letter 'U'
              [1, 1, 1, 1, 0, 0, 1, 1, 1]])     # the letter 'C'
s = 2 * s -1    # bipolar state
n, m = s.shape

w = s.T @ s     # weight matrix   w_ij = Σ_{p=1}^{n} (2 s_i^(p) − 1)(2 s_j^(p) − 1)
np.fill_diagonal(w, 0)

sp = np.array([[ 1, 0, 1, 0, 1, 0, 0, 1, 0],    # corrupted 'T'
               [ 0, 1, 1, 1, 0, 0, 0, 1, 1],    # corrupted 'C'
               [ 1, 0, 1, 0, 0, 1, 1, 1, 0]])   # corrupted 'U'
sp = 2 * sp -1  # bipolar state

def energy(s, w):                                # E(s) = −(1/2) Σ_{i,j} w_ij s_i s_j
    return -0.5 * np.dot(np.dot(s.T, w), s)

for p in sp:
    fig, ax = plt.subplots(1, 3, figsize=(10,2))
    ax[0].imshow(p.reshape(3,3))
    ax[0].set_title('corrupted image')

    energies = [energy(p, w)]
    for k in range(50):
        E1 = energy(p, w)   # Energy before flip
        i = np.random.choice(m, 1)
        p[i] *= -1          # flip the spin
        E2 = energy(p, w) # Energy after flip

        # delta E = difference before and after flipping the spin.
        dE = E2 - E1

        if dE >= 0:
            p[i] *= -1   # flip the spin back to its original state

        E = energy(p, w)
        energies.append(E)

    ax[1].imshow(p.reshape(3,3))
    ax[1].set_title('reconstructed image')
    ax[2].plot(energies, color='red')
    ax[2].set_title('energy')
    plt.show()
```





15-5. Hopfield Netowork (3)

Pseudo-inverse learning rule

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

Storage capacity of Hopfield network

- The storage capacity of a Hopfield network refers to the maximum number of patterns that can be stored and successfully recalled for a given number of nodes, N. When trained with Hebbian rule, the storage capacity of a Hopfield network is limited to approximately 0.14N.
- The critical capacity of Hopfield model has been extensively studied.
- α becomes approximately 0.14 when some error is allowed in the recalled patterns.

$$\alpha = \frac{p}{N} \simeq 0.14 \qquad p = 0.14\,N$$

p: The number of patterns that can be successfully stored.
N: the number of neurons in a Hopfield network

Daniel J. Amit and Hanoch Gutfreund, 1986, Statistical Mechanics of Neural Networks near Saturation.

"More careful arguments, based on probability theory, lead to the conclusion that already for

$$p > \frac{N}{2\ln(N)} \qquad (1.7)$$

the original patterns become unstable."

This implies that retrieval of patterns free of errors occurs when

$$p < \frac{N}{2\ln(N)}$$

Retrieval Rate vs. α

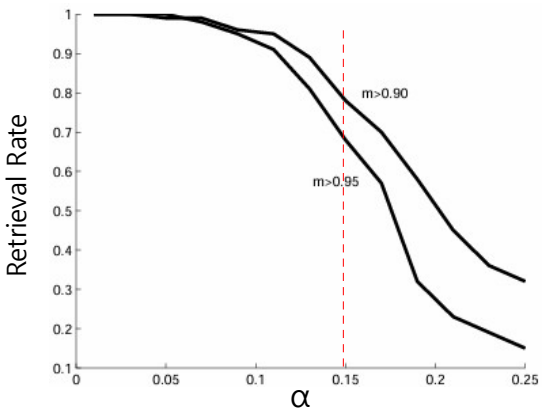
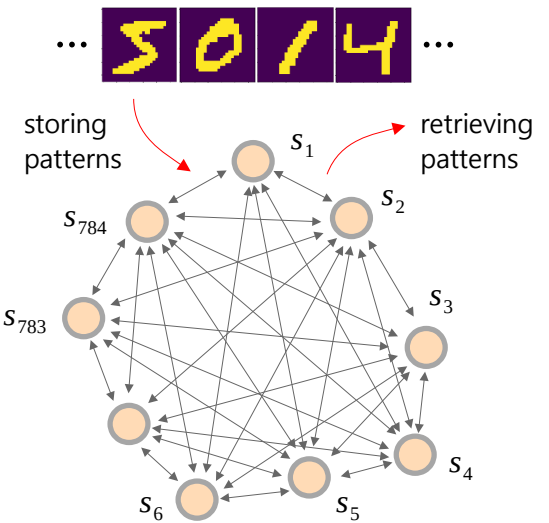


FIG. 2: Retrieval rate plotted against ratio of stored patterns to size of network. Initial states are constructed as stored patterns with an error percentage of 20%. Overlap threshold set to 0.95 and 0.90.

image source:
Kevin Takasaki, 2007, Critical Capacity of Hopfield Networks

How many patterns can a Hopfield network store?



Hebb's rule: $w_{ij} = \frac{1}{N}\sum_{p=1}^{n} s_i^{(p)} s_j^{(p)}$

MX-AI

- Storing and recalling MNIST image patterns in a Hopfield network using the Hebbian learning rule

```
# [MXDL-15-05] 7.Hopfield(hebbian).py
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Load MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)    # x: (70000, 784), y: (70000, 1)

x = x[:2]
x = np.where(x > 0, 1, -1)   # bipolar states
n, m = x.shape

# Hebbian learning rule
w = (1/m) * np.dot(x.T, x)   # weight matrix (784, 784)
np.fill_diagonal(w, 0)
```

$$w_{ij} = \frac{1}{m}\sum_{p=1}^{n} x_i^{(p)} x_j^{(p)}$$

```
def energy(s, w):
    E = -0.5 * np.dot(np.dot(s.T, w), s)
    return E
```

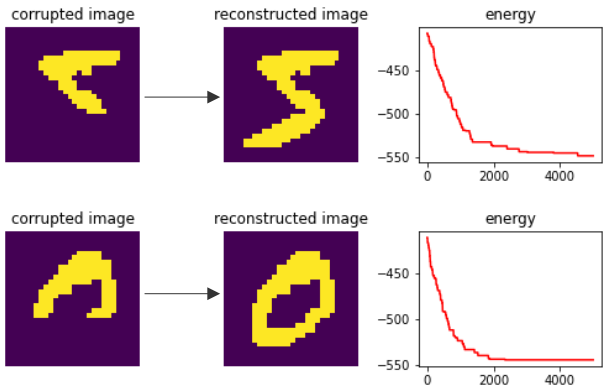
$$E = -\frac{1}{2}\sum_{\mathrm{i,j}} w_{ij} x_i x_j$$

```
def corrupted_img(p):
    p.reshape(28,28)[18:, :] = -1
    return p.reshape(-1,)

for t in range(n):
    xh = corrupted_img(x[t])
    fig, ax = plt.subplots(1, 3, figsize=(9, 2))
    ax[0].imshow(xh.reshape(28, 28))
    ax[0].axis('off')
    ax[0].set_title('corrupted image')

    energies = [energy(xh, w)]
    for k in range(100):
        i_rnd = np.random.choice(m, (50,), replace=False)
        for i in i_rnd:
            E1 = energy(xh, w)  # Energy before flip

            xh[i] *= -1         # Flip the spin of the i-th neuron
            E2 = energy(xh, w)  # Energy after flip

            # dE = difference before and after flipping the spin.
            dE = E2 - E1

            if dE < 0:
                ;               # xh[i] is already flipped above
            else:
                xh[i] *= -1     # flip back to the original spin

            E = energy(xh, w)
            energies.append(E)

    ax[1].imshow(xh.reshape(28,28))
    ax[1].axis('off')
    ax[1].set_title('reconstructed image')
    ax[2].plot(energies, color='red')
    ax[2].set_title('energy')
    plt.show()
```

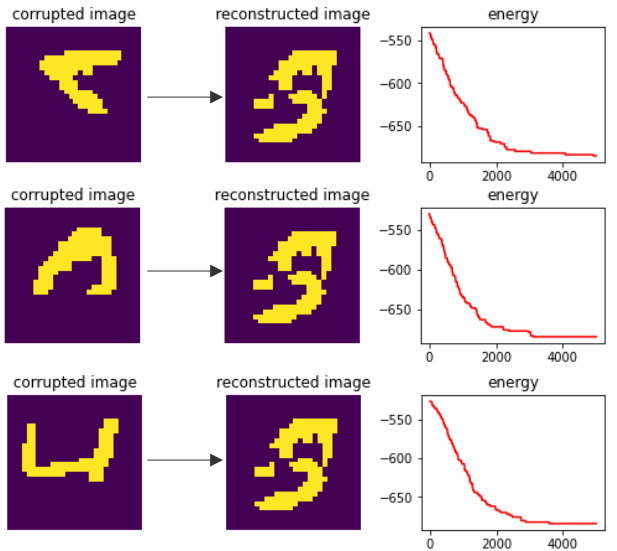
MX-AI

- Storing and recalling MNIST image patterns in a Hopfield network using the Hebbian learning rule

```
# Load MNIST data set
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)      # x: (70000, 784)

x = x[:2]
x = np.where(x > 0, 1, -1)
```




```
# Load MNIST data set
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)      # x: (70000, 784)

x = x[:3]
x = np.where(x > 0, 1, -1)
```



- To increase the storage capacity of the Hopfield network, various methods have been proposed, such as the **pseudo-inverse learning rule** and the **Storkey learning rule**.

MX-AI

■ Pseudo-inverse learning rule

(IJACSA) International Journal of Advanced Computer Science and Applications,
Vol. 11, No. 1, 2020

Behavior of Learning Rules in Hopfield Neural Network for Odia Script

Ramesh Chandra Sahoo[1]
Research Scholar, Dept. of Computer Science and Applications, Utkal University, Bhubaneswar, India

Sateesh Kumar Pradhan[2]
Dept. of Computer Science & Applications
Utkal University, Bhubaneswar, India

IV. HOPFIELD NEURAL NETWORK B. Pseudo-Inverse Learning Rule [23]

Pseudo-inverse learning rule uses the pseudoinverse of the pattern matrix while Hebbian learning rule uses the pattern correlation pattern matrix. When pattern vectors are not orthogonal this learning method is more efficient and it is neither local nor incremental which means a new pattern cannot incrementally added to the network and update does not depend on either side of the connection as it calculate the inverse of the weight matrix. The procedure to apply pseudoinverse learning rule to obtain the weight matrix of Hopfield network is explained as follows.

Let input patterns X of vectors $\{x_1, x_2, \ldots, x_n\}$ and target pattern Y of vector $\{y_1, y_2, \ldots, y_n\}$ then we can write:

$$WX = Y \quad (8)$$ where W is the weight matrix.

If the matrix X has an inverse, then we can rewrite:

$$W = YX^{-1} \quad (9)$$

Again if X is not a square matrix then no exact inverse will exist, but it has been shown that it will minimize to $F(W) = \sum_{i=1}^{n} \| y_i - W x_i \|^2$ by using the pseudoinverse matrix and is as follows:

$$W = YX^{+} \quad (10) \qquad \left(X X^{+} = I, \quad X^{+} = X^{T}(X X^{T})^{-1}, \quad W = YX^{+} = Y X^{T}(X X^{T})^{-1}\right)$$

Where X^{+} is the Moore Penrose pseudoinverse, then the pseudoinverse of real matrix X is the unique matrix that satisfies:

$$XX^{+}X = X, \quad X^{+}X X^{+} = X^{+}, \quad X^{+}X = (X^{+}X)^{T}, \quad X X^{+} = (X X^{+})^{T}$$

So the pseudo-inverse weight matrix can be calculated as :

$$W_{pinv} = W^{T}(W W^{T})^{-1}$$

MX-AI

- Storing and recalling MNIST image patterns in a Hopfield network using the pseudo-inverse learning rule

```
# [MXDL-15-05] 8.Hopfield(pseudo_inverse).py
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Load MNIST data set
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)    # x: (70000, 784), y: (70000, 1)

x = x[:100].T                # (784, 100)
x = np.where(x > 0, 1, -1)
m, n = x.shape

# Compute the weights by pseudo-inverse learning rule.
# Reference: R.C. Sahoo and S.K. Pradhan, 2020, Behavior of
# Learning Rules in Hopfield Neural Network for Odia Script.
# (IV-B. Pseudo-Inverse Learning Rule)
x_pinv = np.linalg.pinv(x)  # Moore-Penrose pseudo-inverse of x
w = (1/m) * np.dot(x, x_pinv)
np.fill_diagonal(w, 0)
w_pinv = np.linalg.pinv(w)
np.fill_diagonal(w_pinv, 0)

def energy(s, w):
    return -0.5 * np.dot(np.dot(s.T, w), s)

def corrupted_img(p):
    p.reshape(28,28)[18:, :] = -1
    return p.reshape(-1,)
```

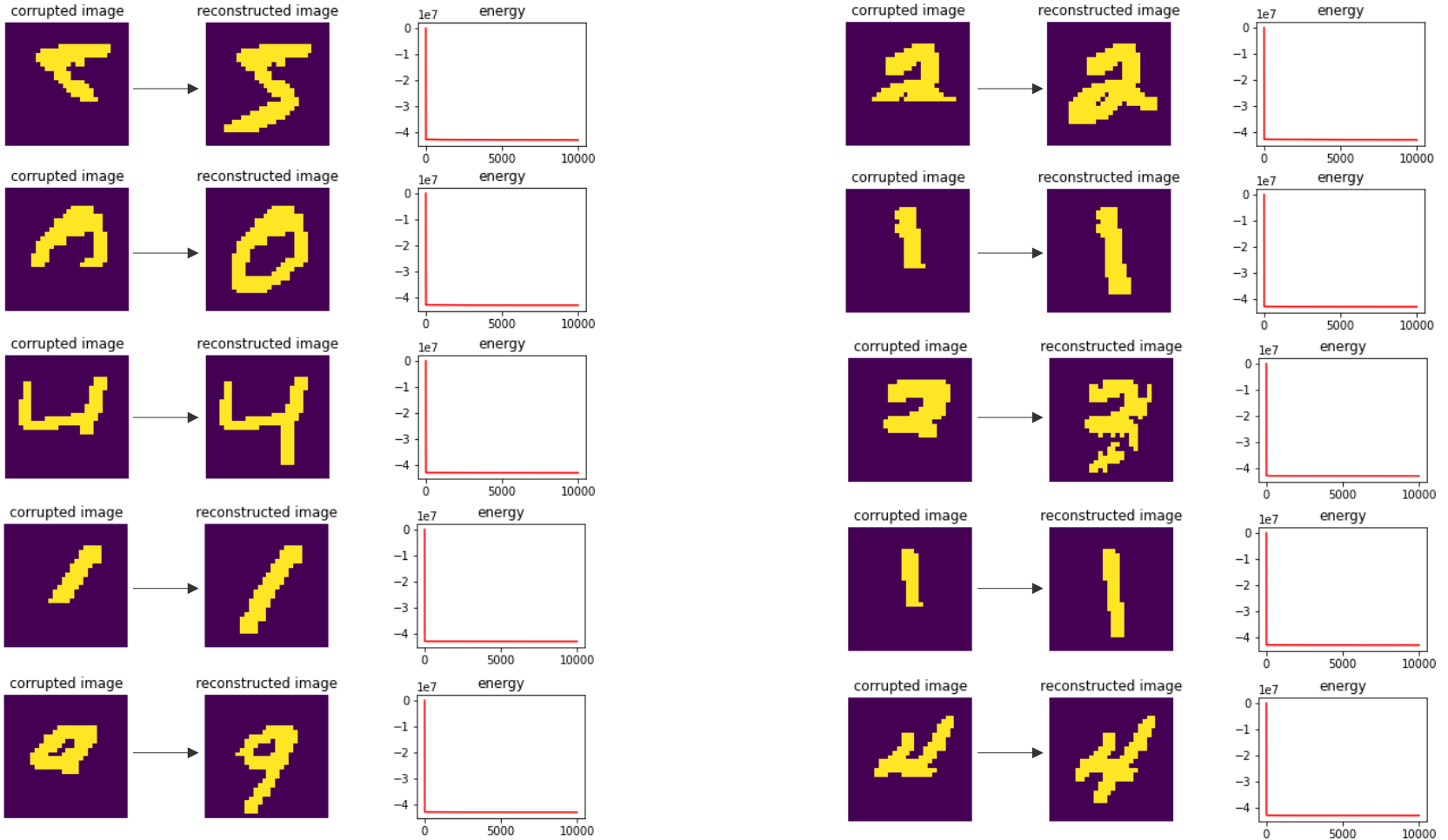
$$W = \frac{1}{m} XX^{+}$$

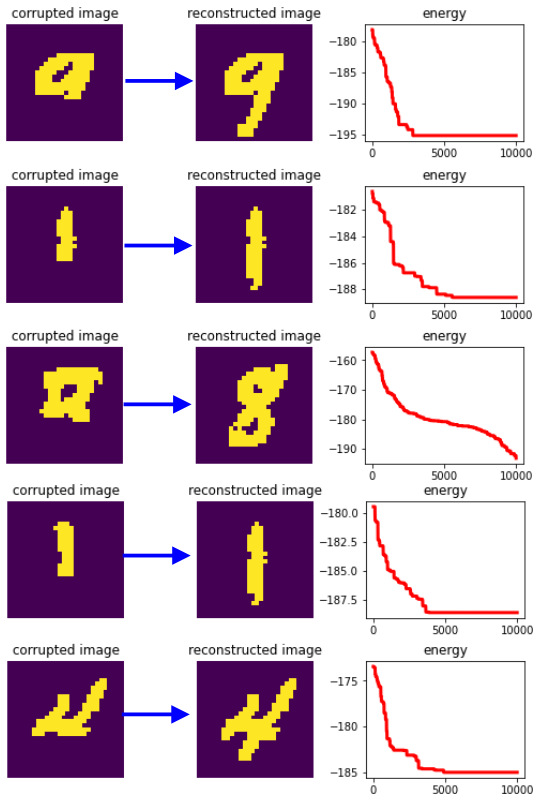
$$W_{pinv} = W^{T} \cdot (W \cdot W^{T})^{-1}$$

```
for t in range(10):
    xh = corrupted_img(x[:, t])

    fig, ax = plt.subplots(1, 3, figsize=(9, 2))
    ax[0].imshow(xh.reshape(28, 28))
    ax[0].axis('off'); ax[0].set_title('corrupted image')

    energies = [energy(xh, w)]
    for k in range(100):
        i_rnd = np.random.choice(m, (100,), replace=False)
        for i in i_rnd:
            E1 = energy(xh, w_pinv)  # Energy before flip

            xh[i] *= -1      # Flip the spin of the i-th neuron
            E2 = energy(xh, w_pinv)  # Energy after flip
            dE = E2 - E1
            if dE < 0:
                ;              # xh[i] is already flipped above
            else:
                xh[i] *= -1    # flip back to the original spin

            E = energy(xh, w_pinv)
            energies.append(E)

    ax[1].imshow(xh.reshape(28,28))
    ax[1].axis('off')
    ax[1].set_title('reconstructed image')
    ax[2].plot(energies, color='red')
    ax[2].set_title('energy'); plt.show()
```

MX-AI

- Storing and recalling MNIST image patterns in a Hopfield network using the pseudo-inverse learning rule





15-6. Hopfield Netowork (4)

Storkey learning rule

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ Storkey learning rule

Increasing the capacity of a Hopfield network without sacrificing functionality

Amos Storkey, Imperial College

Abstract. Hopfield networks are commonly trained by one of two algorithms. The simplest of these is the Hebb rule, which has a low absolute capacity of n/(2ln n), where n is the total number of neurons. This capacity can be increased to n by using the pseudo-inverse rule. However, capacity is not the only consideration. It is important for rules to be local (the weight of a synapse depends only on information available to the two neurons it connects), incremental (learning a new pattern can be done knowing only the old weight matrix and not the actual patterns stored) and immediate (the learning process is not a limit process). The Hebbian rule is all of these, but the pseudo-inverse is never incremental, and local only if not immediate. The question addressed by this paper is, 'Can the capacity of the Hebbian rule be increased without losing locality, incrementality or immediacy?'

Here a new algorithm is proposed. This algorithm is local, immediate and incremental. In addition it has an absolute capacity significantly higher than that of the Hebbian method: n/√2ln n.

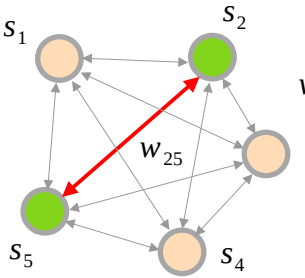
2.1 Characteristics of a learning rule

Definition 3 The new learning rule. The weight matrix of an attractor neural network is said to follow the new learning rule if it obeys

$$w_{ij}^0 = 0 \ \ \forall i,j \quad \text{and} \quad w_{ij}^p = w_{ij}^{p-1} + \frac{1}{n}\left(s_i^p s_j^p - s_i^p h_{ji}^p - h_{ij}^p s_j^p\right)$$

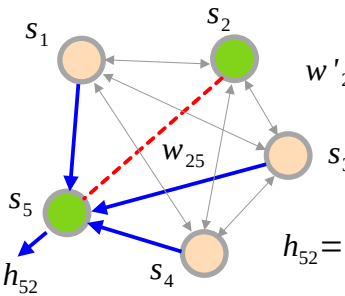
where $h_{ij}^p = \sum_{k=1,k\neq i,j}^{n} w_{ik}^{p-1} s_k^p$ is a form of local field at neuron i, and s^p is the new pattern to be learnt.

Example: i=2, j=5, p=1



$$w_{25} = w_{25}^{old} + \frac{1}{n} s_2 s_5$$

Hebb's rule.

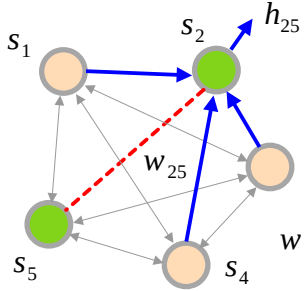


$$w'_{25} = w_{25} - \frac{1}{n} s_2 h_{52}$$

$$h_{52} = w_{51}s_1 + w_{53}s_3 + w_{54}s_4$$

i=2, k=[1, 3, 4]

$$h_{25} = w_{21}s_1 + w_{23}s_3 + w_{24}s_4$$



$$w''_{25} = w'_{25} - \frac{1}{n} h_{25} s_5$$

MX-AI

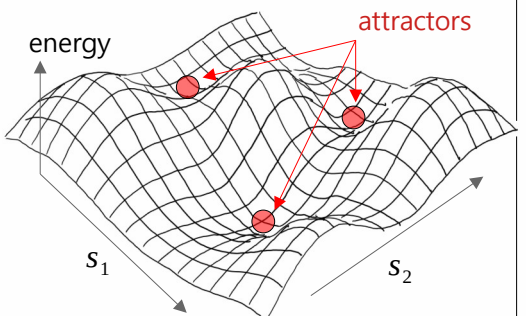
■ Storkey learning rule

Increasing the capacity of a Hopfield network without sacrificing functionality

Amos Storkey, Imperial College

Abstract. Hopfield networks are commonly trained by one of two algorithms. The simplest of these is the Hebb rule, which has a low absolute capacity of n/(2ln n), where n is the total number of neurons. This capacity can be increased to n by using the pseudo-inverse rule. However, capacity is not the only consideration. It is important for rules to be local (the weight of a synapse depends only on information available to the two neurons it connects), incremental (learning a new pattern can be done knowing only the old weight matrix and not the actual patterns stored) and immediate (the learning process is not a limit process). The Hebbian rule is all of these, but the pseudo-inverse is never incremental, and local only if not immediate. The question addressed by this paper is, 'Can the capacity of the Hebbian rule be in incrementality or immediacy?'

Here a new algorithm is proposed. This algorithm is local, immediate a absolute capacity significantly higher than that of the Hebbian method



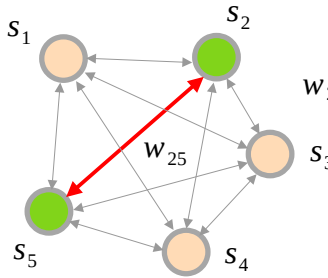
2.1 Characteristics of a learning rule

Definition 3 The new learning rule. The weight matrix of an attractor learning rule if it obeys

$$w_{ij}^0 = 0 \;\; \forall i,j \quad \text{and} \quad w_{ij}^p = w_{ij}^{p-1} + \frac{1}{n}\left(s_i^p s_j^p - s_i^p h_{ji}^p - h_{ij}^p s_j^p\right)$$

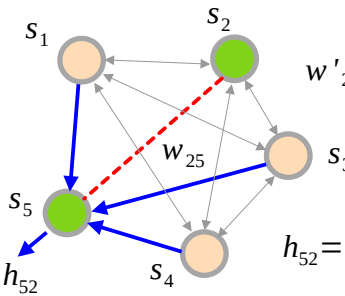
where $h_{ij}^p = \sum_{k=1,k\neq i,j}^{n} w_{ik}^{p-1} s_k^p$ is a form of local field at neuron i, and s^p is the new pattern to be learnt.

Example: i=2, j=5, p=1



$$w_{25} = w_{25}^{old} + \frac{1}{n} s_2 s_5$$

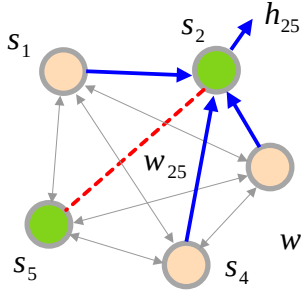
Hebb's rule.



$$w'_{25} = w_{25} - \frac{1}{n} s_2 h_{52}$$

$$h_{52} = w_{51}s_1 + w_{53}s_3 + w_{54}s_4$$

i=2, k=[1, 3, 4]



$$h_{25} = w_{21}s_1 + w_{23}s_3 + w_{24}s_4$$

$$w''_{25} = w'_{25} - \frac{1}{n} h_{25} s_5$$

MX-AI

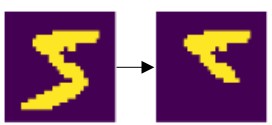
- Storing and recalling MNIST image patterns in a Hopfield network using Storkey learning rule

$$h_{ij}^{p} = \sum_{k=1, k\neq i,j}^{n} w_{ik}^{p-1} s_{k}^{p}$$

$$w_{ij}^{p} = w_{ij}^{p-1} + \frac{1}{n}\left(s_{i}^{p} s_{j}^{p} - s_{i}^{p} h_{ji}^{p} - h_{ij}^{p} s_{j}^{p}\right)$$

```
# [MXDL-15-06] 9.Hopfield(storkey).py
import numpy as np
import numba as nb
from tqdm import tqdm
import matplotlib.pyplot as plt
import pickle

# Load the MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
    x, y = pickle.load(f)      # x: (70000, 784), y: (70000, 1)

x = x[:50]
x = np.where(x > 0, 1, -1)
n, m = x.shape  # n=50, m=784

# Compute the weights by Storkey learning rule.
@nb.jit(nopython=True)
def update_weights(s, w, nw):
    for k in range(nw):
        i = np.random.randint(m)
        j = np.random.randint(m)
        if i != j:      # if not diagonal
            # Calculate the local field at neurons i,j
            mask = np.ones(m)
            mask[i] = mask[j] = 0   # constraints: k != i, j
            hij = np.sum(w[i] * s * mask, axis=1)
            hji = np.sum(w[j] * s * mask, axis=1)

            # Increment weight for the pattern
            dw = s[:, i]*s[:, j] - s[:, i]*hji - hij*s[:, j]
            w[i][j] += np.sum(dw / m)
    return w

w = np.zeros([m, m])
n_iter = 5      # number of iterations
n_size = 10     # batch size of the input patterns
n_batch = int(x.shape[0] / n_size)
for e in range(n_iter):
    for batch in tqdm(range(n_batch), desc="epochs-" + str(e)):
        i = np.random.choice(n, (n_size,), replace=False)
        w = update_weights(x[i], w, int(m * m / 2))

def energy(s, w):
    E = -0.5 * np.dot(np.dot(s, w), s)
    return E

def corrupted_img(p):
    p.reshape(28,28)[18:, :] = -1
    return p.reshape(-1,)
```

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} x_i x_j$$

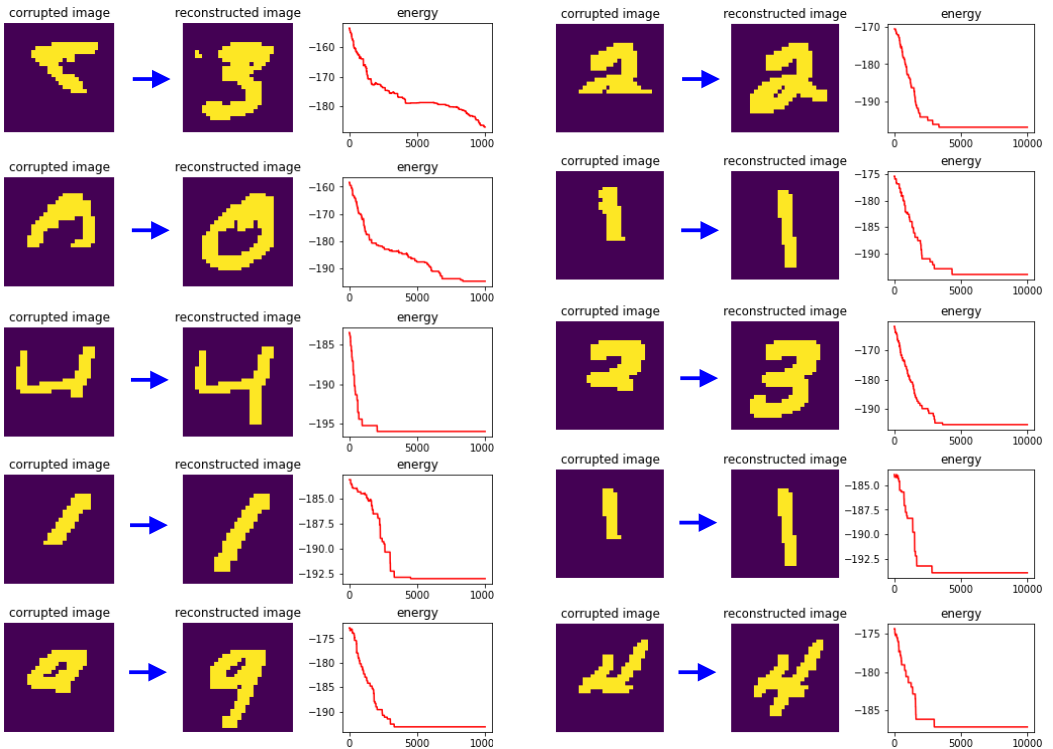


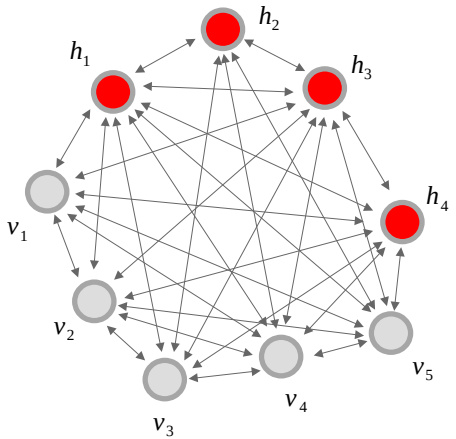
MX-AI

- Storing and recalling MNIST image patterns in a Hopfield network using Storkey learning rule

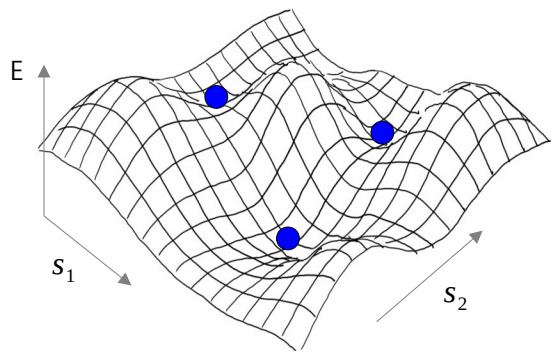
```
for t in range(10):
    p = corrupted_img(x[t])

    fig, ax = plt.subplots(1, 3, figsize=(9, 2))
    ax[0].imshow(p.reshape(28, 28))
    ax[0].axis('off')
    ax[0].set_title('corrupted image')

    energies = [energy(p, w)]
    for k in range(200):
        i_rnd = np.random.choice(m, (50,), replace=False)
        for i in i_rnd:
            activation = np.dot(w[i, :], p)
            if activation > 0:
                p[i] = 1
            elif activation == 0:
                ;
            else:
                p[i] = -1

            E = energy(p, w)
            energies.append(E)

    ax[1].imshow(p.reshape(28,28))
    ax[1].axis('off')
    ax[1].set_title('reconstructed image')

    ax[2].plot(energies, color='red')
    ax[2].set_title('energy'); plt.show()
```

```
epochs-0: 100%|██████████| 5/5 [00:21<00:00,  4.21s/it]
epochs-1: 100%|██████████| 5/5 [00:19<00:00,  3.95s/it]
epochs-2: 100%|██████████| 5/5 [00:19<00:00,  3.94s/it]
epochs-3: 100%|██████████| 5/5 [00:19<00:00,  3.96s/it]
epochs-4: 100%|██████████| 5/5 [00:19<00:00,  3.96s/it]
```





$$w_{ij} = w_{ij} + \alpha[E_{data}(s_i s_j) - E_{model}(s_i s_j)]$$



15-7. Boltzmann machine

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

The history of Boltzmann machines

Donald Hebb (1949) → Hebbian learning rule

John Hopfield (1982) → Hopfield Network

David Ackley, Geoffrey Hinton, Terrence Sejnowski (1985) → **Boltzmann machine**

Paul Smolensky (1986), Geoffrey Hinton (2002) → Restricted Boltzmann machine

Ising model

$$E(s) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j$$

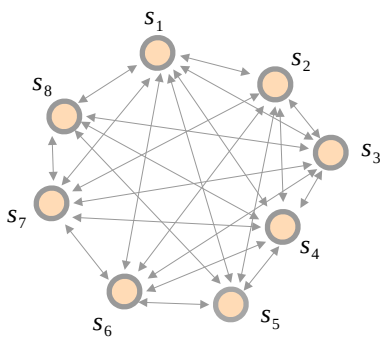
$$\begin{cases} s_i \rightarrow 1 \\ s_i \rightarrow 0 \end{cases} \quad if \quad \sum_{j \neq i} w_{ij} s_j \quad \begin{matrix} > U_i \\ < U_i \end{matrix}$$

- Energy function

$$E(s) = -\sum_{<i,j>} J_{ij} s_i s_j - \mu \sum_j h_j s_j$$

- Boltzmann distribution

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



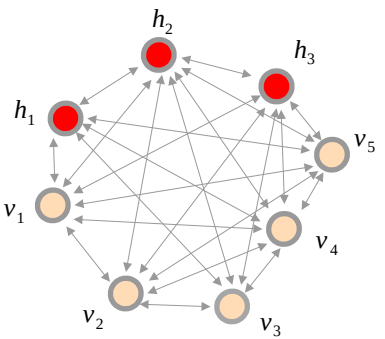
$$E(s^{(k)}) = -\sum_{i<j} w_{ij} s_i^{(k)} s_j^{(k)} + \sum_i b_i s_i^{(k)}$$

$$\Delta E(s_i^{(k)}) = \sum_{j \neq i} w_{ij} s_j - b_i$$

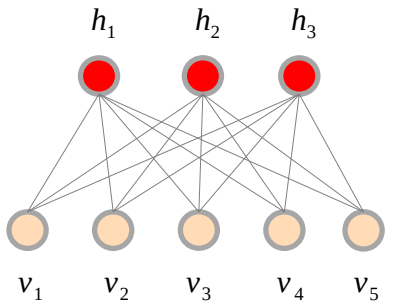
$$p(s_i^{(k)}) = \frac{1}{1 + \exp\left(\frac{-\Delta E(s_i^{(k)})}{T}\right)}$$

- Gibb's sampling

$$w_{ij} = w_{ij} + \alpha[E_{data}(s_i s_j) - E_{model}(s_i s_j)]$$



- contrastive divergence



hidden neurons

visible neurons

COGNITIVE SCIENCE 9, 147-169 (1985)

A Learning Algorithm for Boltzmann Machines*

David H. Ackley
Geoffrey E. Hinton
Computer Science Department, Carnegie-Mellon University
Terrence J. Sejnowski
Biophysics Department, The Johns Hopkins University

The computational power of massively parallel networks of simple processing elements resides in the communication bandwidth provided by the hardware connections between elements. These connections con allow a significant fraction of the knowledge of the system to be applied to an instance of a problem in o very short time. One kind of computation for which massively parallel networks appear to be well suited is large constraint satisfaction searches, but to use the connections efficiently two conditions must be met: First, a search technique that is suitable for parallel networks must be found. Second, there must be some way of choosing internal representations which allow the preexisting hardware connections to be used efficiently for encoding the constraints in the domain being searched. We describe a general parallel search method, based on statistical mechanics, and we show how it leads to a general learning rule for modifying the connection strengths so as to incorporate knowledge about o task domain in on efficient way. We describe some simple examples in which the learning algorithm creates internal representations that ore demonstrably the most efficient way of using the preexisting connectivity structure.

1. INTRODUCTION

<...>

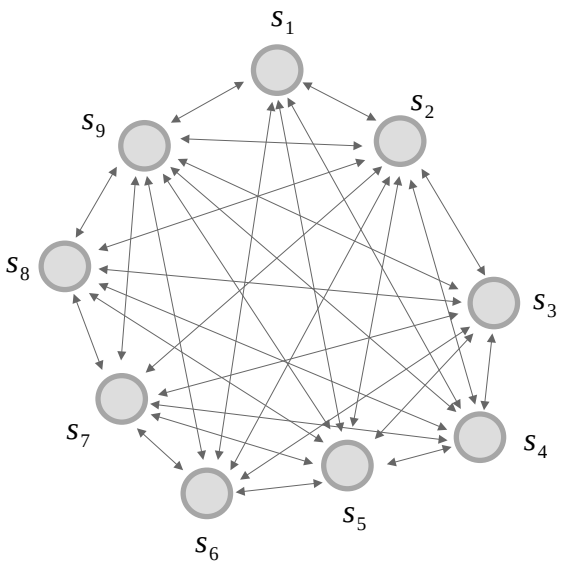
This paper presents a type of parallel constraint satisfaction network which we call a "**Boltzmann Machine**" that is capable of learning the underlying constraints that characterize a domain simply by being shown examples from the domain. The network modifies the strengths of its connections so as to construct an internal **generative model** that produces examples with the same probability distribution as the examples it is shown. Then, when shown any particular example, the network can "interpret" it by finding values of the variables in the internal model that would generate the example. When shown a partial example, the network can complete it by finding internal variable values that generate the partial example and using them to generate the remainder. At present, we have an interesting mathematical result that guarantees that a certain learning procedure will build internal representations which allow the connection strengths to capture the underlying constraints that are implicit in a large ensemble of examples taken from a domain. We also have simulations which show that the theory works for some simple cases, but the current version of the learning algorithm is very slow.

MX-AI

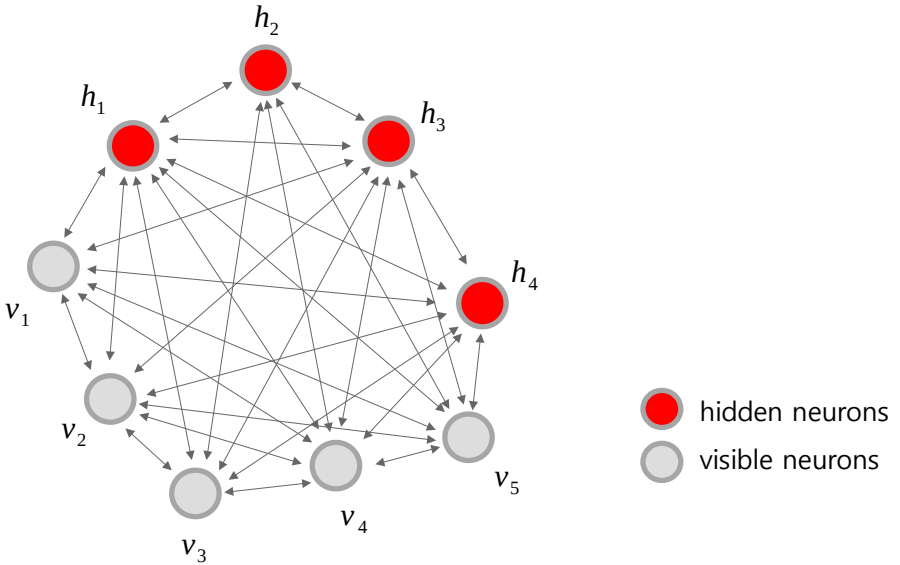
The structure of a Boltzmann machine

- A Boltzmann machine has a similar network architecture to the Hopfield network, but differs in stochastic update and learning properties.
- Additionally, we can add hidden units to the Boltzmann machine. Then, the Boltzmann machine consists of visible units and hidden units, and the input data is fed into the visible units.
- Hidden units are used to model higher-order correlations between visible units and to capture complex patterns and dependencies that are not possible with direct connections between visible units alone. They essentially act as mediators in the relationships between the visible units, allowing the model to represent a richer probability distribution over the input data.
- Even with hidden units, the learning rule for Boltzmann machines remains the same.

[A Boltzmann machine without hidden neurons]



[A Boltzmann machine with hidden neurons]

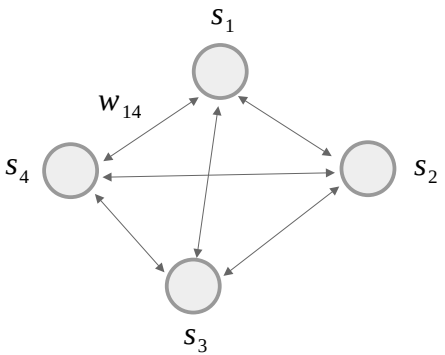


MX-AI

■ Energy function and the probability that a neuron will "turn on"

▪ A simple Boltzmann machine

$$s^{(k)} = \{ s_1^{(k)}, s_2^{(k)}, s_3^{(k)}, s_4^{(k)} \}$$



▪ Energy function

$$E(s^{(k)}) = -\sum_{i<j} w_{ij} s_i^{(k)} s_j^{(k)} + \sum_i b_i s_i^{(k)} \quad \text{---- (1)}$$

i < j			
1	2	$-w_{12}s_1s_2$	b_1s_1
1	3	$-w_{13}s_1s_3$	b_2s_2
1	4	$-w_{14}s_1s_4$	b_3s_3
2	3	$-w_{23}s_2s_3$	b_4s_4
2	4	$-w_{24}s_2s_4$	
3	4	$-w_{34}s_3s_4$	

▪ Energy gap

$$\Delta E(s_i^{(k)}) = \sum_{j \neq i} w_{ij} s_j^{(k)} - b_i \quad \text{---- (2)}$$

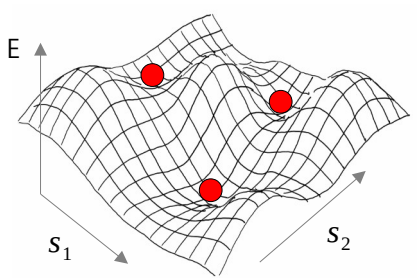
ex) i = 0

$E(s_1^{(k)}=0)$		$E(s_1^{(k)}=1)$	
$-w_{23}s_2s_3$	b_2s_2	$-w_{12}s_2$	b_1
$-w_{24}s_2s_4$	b_3s_3	$-w_{13}s_3$	b_2s_2
$-w_{34}s_3s_4$	b_4s_4	$-w_{14}s_4$	b_3s_3
		$-w_{23}s_2s_3$	b_4s_4
		$-w_{24}s_2s_4$	
		$-w_{34}s_3s_4$	

$$\Delta E(s_1^{(k)}) = E(s_1^{(k)}=0) - E(s_1^{(k)}=1)$$

$$= w_{12}s_2^{(k)} + w_{13}s_3^{(k)} + w_{14}s_4^{(k)} - b_1$$

▪ Total Energy landscape for s1 and s2

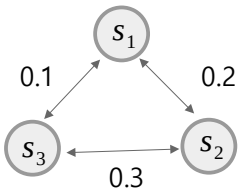


▪ The probability that the neuron i turns "on"

$$p(s_i^{(k)}) = \frac{1}{1 + \exp\left(\frac{-\Delta E(s_i^{(k)})}{T}\right)} \quad \text{---- (5)}$$

MX-AI

■ Learning rule



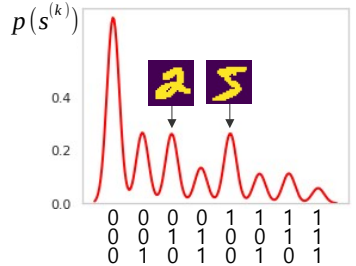
$$E(s^{(k)}) = -\sum_{i<j} w_{ij} s_i^{(k)} s_j^{(k)} + \sum_i b_i s_i^{(k)}$$

$\exp(-E(s^{(k)}))$

$$p(s^{(k)}) = \frac{1}{Z}\exp(-E(s^{(k)}))$$

s_1	s_2	s_3	(b = 1)		
0	0	0	(-0.2*0*0 - 0.1*0*0 - 0.3*0*0) + (1*0 + 1*0 + 1*0) = 0.0	1.00	0.37
0	0	1	(-0.2*0*0 - 0.1*0*1 - 0.3*0*1) + (1*0 + 1*0 + 1*1) = 1.0	0.37	0.14
0	1	0	(-0.2*0*1 - 0.1*0*0 - 0.3*1*0) + (1*0 + 1*1 + 1*0) = 1.0	0.37	0.14
0	1	1	(-0.2*0*1 - 0.1*0*1 - 0.3*1*1) + (1*0 + 1*1 + 1*1) = 1.7	0.18	0.07
1	0	0	(-0.2*1*0 - 0.1*1*0 - 0.3*0*0) + (1*1 + 1*0 + 1*0) = 1.0	0.37	0.14
1	0	1	(-0.2*1*0 - 0.1*1*1 - 0.3*0*1) + (1*1 + 1*0 + 1*1) = 1.9	0.15	0.06
1	1	0	(-0.2*1*1 - 0.1*1*0 - 0.3*1*0) + (1*1 + 1*1 + 1*0) = 1.8	0.17	0.06
1	1	1	(-0.2*1*1 - 0.1*1*1 - 0.3*1*1) + (1*1 + 1*1 + 1*1) = 2.4	0.09	0.03

$Z = 2.70$



$$X = \{s^{(1)}, s^{(2)}, \ldots, s^{(N)}\}$$

$$s^{(k)} = \{s_1^{(k)}, s_2^{(k)}, \ldots, s_m^{(k)}\}$$

$$p(s^{(k)}) = \frac{1}{Z}\exp(-E(s^{(k)}))$$

$$Z = \sum_s \exp(-E(s))$$

$$L = \frac{1}{N}\sum_{k=1}^{N} \log p(s^{(k)})$$

$$w_{ij} = w_{ij} + \alpha \frac{\partial L}{\partial w_{ij}}$$

$$L = \frac{1}{N}\sum_{k=1}^{N}[-E(s^{(k)} - \log(Z))]$$

$$= [-\frac{1}{N}\sum_{k=1}^{N} E(s^{(k)})] - \log(Z)$$

$$\frac{\partial L}{\partial w_{ij}} = -\frac{1}{N}\sum_{k=1}^{N}\frac{\partial E(s^{(k)})}{\partial w_{ij}} - \frac{\partial \log(Z)}{\partial w_{ij}}$$

$$= \frac{1}{N}\sum_{k=1}^{N} s_i^{(k)} s_j^{(k)} - \frac{1}{Z}\frac{\partial Z}{\partial w_{ij}}$$

$$= \frac{1}{N}\sum_{k=1}^{N} s_i^{(k)} s_j^{(k)} - \frac{1}{Z}\sum_{k=1}^{N}\exp(-E(s^{(k)}))\frac{\partial[-E(s^{(k)})]}{\partial w_{ij}}$$

$$\frac{\partial L}{\partial w_{ij}} = \frac{1}{N}\sum_{k=1}^{N} s_i^{(k)} s_j^{(k)} - \frac{1}{Z}\sum_{k=1}^{N}\exp(-E(s^{(k)}))s_i^{(k)} s_j^{(k)}$$

$$= \frac{1}{N}\sum_{k=1}^{N} s_i^{(k)} s_j^{(k)} - \sum_{k=1}^{N} p(s^{(k)}) s_i^{(k)} s_j^{(k)}$$

$$= E_{data}(s_i s_j) - E_{model}(s_i s_j)$$

$$w_{ij} = w_{ij} + \alpha[E_{data}(s_i s_j) - E_{model}(s_i s_j)]$$

MX-AI

■ Computing $E_{data}(s_i s_j)$ and $E_{model}(s_i s_j)$

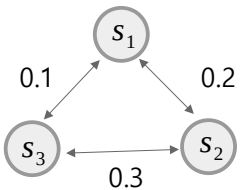
$$\Delta E(s_i^{(k)}) = \sum_{j \neq i} w_{ij} s_j^{(k)} - b_i \qquad p(s_i^{(k)}) = \frac{1}{1 + \exp(\frac{-\Delta E(s_i^{(k)})}{T})}$$

(b = 0, T = 1)

▪ **Training data**

N=10

	$s_1^{(k)}$	$s_2^{(k)}$	$s_3^{(k)}$
$s^{(1)}$	0	1	0
$s^{(2)}$	1	0	0
$s^{(3)}$	0	1	0
$s^{(4)}$	1	1	0
$s^{(5)}$	1	0	0
$s^{(6)}$	1	1	1
$s^{(7)}$	1	0	1
$s^{(8)}$	1	1	1
$s^{(9)}$	0	1	0
$s^{(10)}$	0	0	1



N=10

	$s_1^{(k)}$	$s_2^{(k)}$	$s_3^{(k)}$	$\Delta E(s_1^{(k)})$	$\Delta E(s_2^{(k)})$	$\Delta E(s_3^{(k)})$	$p(s_1^{(k)})$	$p(s_2^{(k)})$	$p(s_3^{(k)})$	$\hat{s}_1^{(k)}$	$\hat{s}_2^{(k)}$	$\hat{s}_3^{(k)}$
$s^{(1)}$	1	1	0	0.2	0.2	0.4	0.55	0.55	0.60	1	1	1
$s^{(2)}$	0	0	1	0.1	0.3	0.0	0.52	0.57	0.50	0	1	0
$s^{(3)}$	0	1	1	0.3	0.3	0.3	0.57	0.57	0.57	0	0	1
$s^{(4)}$	1	0	0	0.0	0.2	0.1	0.50	0.55	0.52	1	0	1
$s^{(5)}$	1	1	0	0.2	0.2	0.4	0.55	0.55	0.60	1	1	0
$s^{(6)}$	1	0	1	0.1	0.5	0.1	0.52	0.62	0.52	1	1	1
$s^{(7)}$	0	0	1	0.1	0.3	0.0	0.52	0.57	0.50	1	1	1
$s^{(8)}$	1	0	1	0.1	0.5	0.1	0.52	0.62	0.52	1	0	0
$s^{(9)}$	0	1	0	0.2	0.0	0.3	0.55	0.50	0.57	1	0	0
$s^{(10)}$	1	0	0	0.0	0.2	0.1	0.50	0.55	0.52	1	0	1

energy gap → probability → sampling

$$E_{data}(s_i s_j) = \frac{\text{The number of data points where both } s_i \text{ and } s_j \text{ are turned on}}{\text{The number of data points}}$$

$$E_{model}(s_i s_j) = \frac{\text{The number of sampled instances where both } s_i \text{ and } s_j \text{ are turned on}}{\text{The number of sampled instances}}$$

ex) $E_{data}(s_1 s_2) = \frac{3}{10} = 0.3$

ex) $E_{model}(s_1 s_2) = \frac{4}{10} = 0.4$

▪ **weight update**

$$w_{12} = w_{12} + \alpha[E_{data}(s_1 s_2) - E_{model}(s_1 s_2)]$$

$$= 0.2 + 0.1 \times [0.3 - 0.4] = 0.19$$

MX-AI

■ Implementing a simple Boltzmann machine

```
# [MXDL-15-07] 10.Boltzmann.py
import numpy as np
import matplotlib.pyplot as plt

# Input data
s = np.array([[1, 1, 1, 0, 1, 0, 0, 1, 0],  # 'T'
              [1, 0, 1, 1, 0, 1, 1, 1, 1],  # 'U'
              [1, 0, 0, 1, 0, 0, 1, 1, 1],  # 'L'
              [1, 0, 1, 1, 1, 1, 1, 0, 1]]) # 'H'
n,m = s.shape

# Compute the energy
def energy(x, w):
    return -0.5 * np.dot(np.dot(x, w), x)

# Initialize weights randomly
w = np.random.normal(size=(m, m))
np.fill_diagonal(w, 0)

# Update the weights
E = []
for epoch in range(1000):
    for k in range(100):
        i, j = np.random.randint(m, size=2)
        if i != j:
            # Compute the mean of si * sj from the data
            E_data = np.mean(s[:, i] * s[:, j])
            # Expectation of si * sj from the model
            y = np.random.randint(0, 2, size=(1000, m))
            dE = np.dot(y, w)                    # energy gap
            pk = 1 / (1 + np.exp(-dE))           # probability
            y = np.random.binomial(n=1, p=pk)    # sampling
            E_model = np.mean(y[:, i] * y[:, j])

            # Update the weights by gradient ascent
            w[i, j] += 0.1 * (E_data - E_model)

    if epoch % 50 == 0:
        # Compute the total energy
        E.append(np.sum([energy(s[e], w) for e in range(n)]))
        print("Epochs:{}, Energy = {:.2f}".format(epoch, E[-1]))

# Plot the energy changes
plt.plot(E)
plt.show()

# Test data. Corrupted images
cx = np.array([[1, 1, 1, 0, 1, 0, 0, 0, 0],  # corrupted 'T'
               [1, 0, 1, 0, 0, 1, 1, 1, 1],  # corrupted 'U'
               [1, 0, 0, 1, 0, 0, 0, 1, 1],  # corrupted 'L'
               [1, 0, 1, 0, 1, 1, 1, 0, 1]]) # corrupted 'H'
```

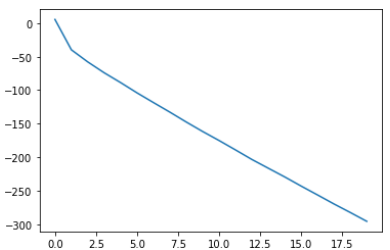
MX-AI

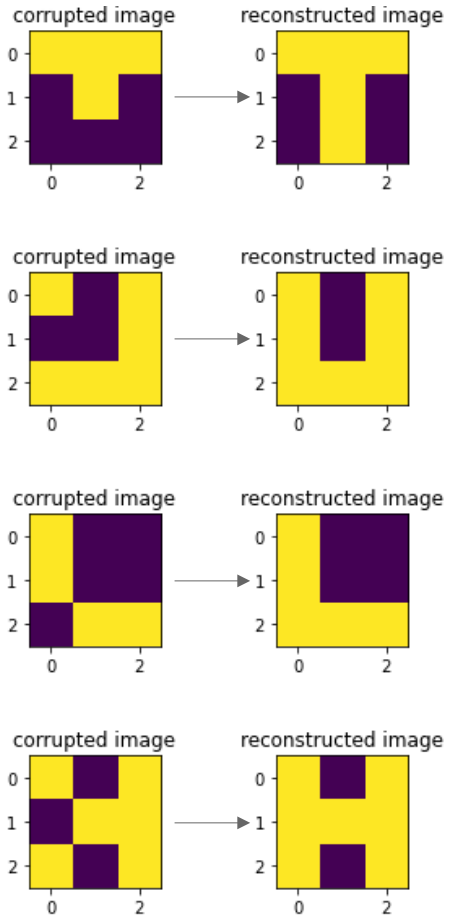
■ Implementing a simple Boltzmann machine

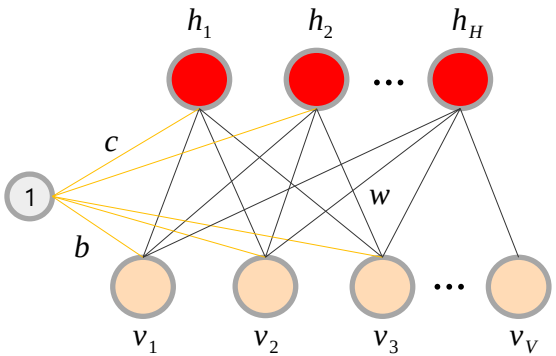
```
# Reconstructing the corrupted images
for p in cx:
    fig, ax = plt.subplots(1, 2, figsize=(5,1.5))
    ax[0].imshow(p.reshape(3,3))
    ax[0].set_title('corrupted image')

    dE = np.dot(p, w)
    pk = 1 / (1 + np.exp(-dE))
    y = np.where(pk >= 0.5, 1, 0)

    ax[1].imshow(y.reshape(3,3))
    ax[1].set_title('reconstructed image')
plt.show()
```

```
Epochs:0, Energy = 5.51
Epochs:50, Energy = -39.79
Epochs:100, Energy = -57.88
Epochs:150, Energy = -74.06
Epochs:200, Energy = -88.71
Epochs:250, Energy = -104.02
Epochs:300, Energy = -118.49
Epochs:350, Energy = -132.78
Epochs:400, Energy = -147.51
...
Epochs:800, Energy = -256.36
Epochs:850, Energy = -269.79
Epochs:900, Energy = -282.58
Epochs:950, Energy = -295.73
```

Energy changes







15-8. Restricted Boltzmann Machine (RBM) - (1)

$$\frac{\partial L}{\partial w_{ij}} = \frac{1}{N}\sum_{n=1}^{N} p\left(h_j^{(n)}=1 \mid v^{(n)}\right)\cdot v_i^{(n)}$$

$$- \sum_{v} p(v)\cdot p\left(h_j=1 \mid v\right)\cdot v_i$$

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

The history of Boltzmann machines

Donald Hebb (1949) → Hebbian learning rule

John Hopfield (1982) → Hopfield Network

David Ackley, Geoffrey Hinton, Terrence Sejnowski (1985) → Boltzmann machine

Paul Smolensky (1986), Geoffrey Hinton (2002) → **Restricted Boltzmann machine**

Ising model

$$E(s) = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j$$

$$\begin{cases} s_i \rightarrow 1 \\ s_i \rightarrow 0 \end{cases} \quad if \quad \sum_{j \neq i} w_{ij} s_j \quad \begin{matrix} > U_i \\ < U_i \end{matrix}$$

- Energy function

$$E(s) = -\sum_{<i,j>} J_{ij} s_i s_j - \mu \sum_j h_j s_j$$

- Boltzmann distribution

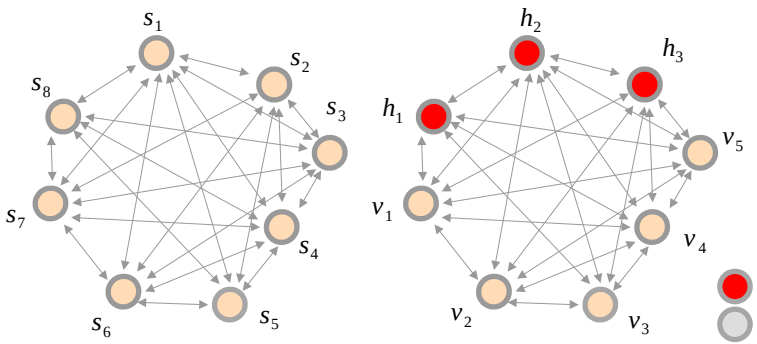
$$p(s) = \frac{1}{Z}\exp\left(\frac{-E(s)}{kT}\right)$$

- contrastive divergence

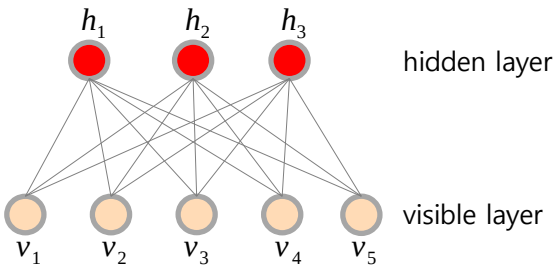
$$\begin{cases} \frac{\partial L}{\partial w_{ij}} = \frac{1}{N}\sum_{n=1}^{N} p\left(h_j^{(n)}=1 \mid v^{(n)}\right)\cdot v_i^{(n)} - \sum_v p(v)\cdot p(h_j=1 \mid v)\cdot v_i \\ \frac{\partial L}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N} v_i^{(n)} - \sum_v p(v) v_i \\ \frac{\partial L}{\partial c_i} = \frac{1}{N}\sum_{n=1}^{N} p\left(h_i^{(n)}=1 \mid v^{(n)}\right) - \sum_v p(v)\, p(h_i=1 \mid v) \end{cases}$$

$$w_{ij} = w_{ij} + \alpha \frac{\partial L}{\partial w_{ij}} \qquad b_i = b_i + \alpha \frac{\partial L}{\partial b_i} \qquad c_i = c_i + \alpha \frac{\partial L}{\partial c_i}$$

[Boltzmann machine]

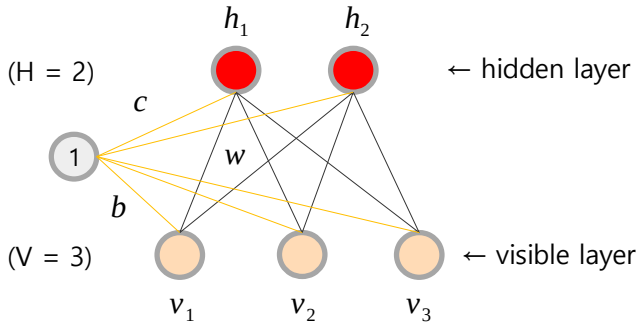


[Restricted Boltzmann machine]



MX-AI

The joint probability distribution of RBM




```
b = [ 0.1   0.2   0.1 ]ᵀ  – biases for the visible units
c = [ 0.3   0.2 ]ᵀ        – biases for the hidden units
w = [ -0.1   0.2 ]        – weights
    [  0.3   0.1 ]
    [  0.1  -0.3 ]
```

$$E(v,h) = -\sum_{j=1}^{V}\sum_{k=1}^{H} w_{jk} h_j v_k - \sum_{k=1}^{V} b_k v_k - \sum_{j=1}^{H} c_j h_j$$

$$E(v,h) = -v^T \cdot w \cdot h - b^T \cdot v - c^T \cdot h$$

$$p(v,h) = \frac{1}{Z}\exp(-E(v,h))$$

$$Z = \sum_i \exp(-E_i(v,h))$$

- **Computing the energy for the given state vectors v and h**

```
v = [ 1   0   1 ]ᵀ
h = [ 0,  1 ]ᵀ
```

$$E(v,h) = -\begin{bmatrix}1 & 0 & 1\end{bmatrix}\begin{bmatrix}-0.1 & 0.2\\ 0.3 & 0.1\\ 0.1 & -0.3\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

$$-\begin{bmatrix}0.1 & 0.2 & 0.1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix}$$

$$-\begin{bmatrix}0.3 & 0.2\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

```
E(v,h) = -0.3
```

- **Computing the joint probability for v and h**

```
p(v=[1,0,1], h=[0,1])=exp(0.3)/Z = 0.023
```

Computing Z is intractable because it requires computing the sum over all possible states.

- **Computing the marginal probability of v**

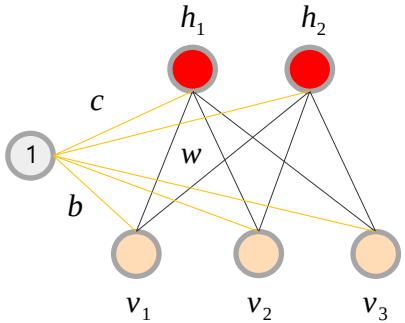
$$p(v) = \sum_h p(v,h)$$

```
p(v=[1,1,1]) = 0.03 + 0.03 + 0.05 + 0.06
             = 0.17
```

v	h	$E(v,h)$	$e^{-E(v,h)}$	$p(v,h)$
0 0 0	0 0	-0.0	1.00	0.02
0 0 0	0 1	-0.2	1.22	0.02
0 0 0	1 0	-0.3	1.35	0.02
0 0 0	1 1	-0.5	1.65	0.03
0 0 1	0 0	-0.1	1.11	0.02
0 0 1	0 1	-0.0	1.00	0.02
0 0 1	1 0	-0.5	1.65	0.03
0 0 1	1 1	-0.4	1.49	0.03
0 1 0	0 0	-0.2	1.22	0.02
0 1 0	0 1	-0.5	1.65	0.03
0 1 0	1 0	-0.8	2.23	0.04
0 1 0	1 1	-1.1	3.00	0.05
0 1 1	0 0	-0.3	1.35	0.02
0 1 1	0 1	-0.3	1.35	0.02
0 1 1	1 0	-1.0	2.72	0.05
0 1 1	1 1	-1.0	2.72	0.05
1 0 0	0 0	-0.1	1.11	0.02
1 0 0	0 1	-0.5	1.65	0.03
1 0 0	1 0	-0.3	1.35	0.02
1 0 0	1 1	-0.7	2.01	0.03
1 0 1	0 0	-0.2	1.22	0.02
1 0 1	**0 1**	**-0.3**	**1.35**	**0.02**
1 0 1	1 0	-0.5	1.65	0.03
1 0 1	1 1	-0.6	1.82	0.03
1 1 0	0 0	-0.3	1.35	0.02
1 1 0	0 1	-0.8	2.23	0.04
1 1 0	1 0	-0.8	2.23	0.04
1 1 0	1 1	-1.3	3.67	0.06
1 1 1	0 0	-0.4	1.49	0.03
1 1 1	0 1	-0.6	1.82	0.03
1 1 1	1 0	-1.0	2.72	0.05
1 1 1	1 1	-1.2	3.32	0.06
			Z = 57.68	

MX-AI

- The conditional probability distribution of RBM



$b = [\ 0.1 \quad 0.2 \quad 0.1\ ]^T$

$c = [\ 0.3 \quad 0.2\ ]^T$

$$w = \begin{bmatrix} -0.1 & 0.2 \\ 0.3 & 0.1 \\ 0.1 & -0.3 \end{bmatrix}$$

$p(h_1=1|v=[0,0,1]) = ?$

v	h	$E(v,h)$	$e^{-E(v,h)}$	$p(h\|v)$
0 0 1	0 0	-0.1	1.1052	0.2107
0 0 1	0 1	0.0	1.0	0.1906
0 0 1	1 0	-0.5	1.6487	0.3143
0 0 1	1 1	-0.4	1.4918	0.2844

Z = 5.2457

$p(v_1=1|h=[0,1]) = ?$

v	h	$E(v,h)$	$e^{-E(v,h)}$	$p(h\|v)$
0 0 0	0 1	-0.2	1.2214	0.0996
0 0 1	0 1	0.0	1.0	0.0815
0 1 0	0 1	-0.5	1.6487	0.1344
0 1 1	0 1	-0.3	1.3499	0.11
1 0 0	0 1	-0.5	1.6487	0.1344
1 0 1	0 1	-0.3	1.3499	0.11
1 1 0	0 1	-0.8	2.2255	0.1814
1 1 1	0 1	-0.6	1.8221	0.1485

Z = 12.2662

$$p(h_1=1|v=[0,0,1]) = 0.3143 + 0.2844 = 0.5987$$

$$p(h_k=1|v) = \frac{1}{1 + \exp(-(v^T \cdot w_{.k} + c_k))}$$

$$p(h_1=1|v=[0,0,1]) = sigmoid(v^T \cdot w_{.1} + c_1)$$

$$= \text{sigmoid}([0\ 0\ 1] \begin{bmatrix} -0.1 \\ 0.3 \\ 0.1 \end{bmatrix} + 0.3) = \text{sigmoid}(0.4)$$

$$= 0.5987$$

$$p(v_1=1|h=[0,1]) = 0.1344 + 0.11 + 0.1814 + 0.1485 = 0.5744$$

$$p(v_k=1|h) = \frac{1}{1 + \exp(-(w_{k.} \cdot h + b_k))}$$

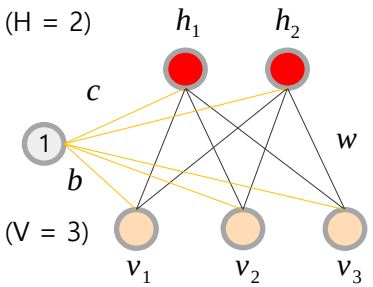
$$p(v_1=1|h=[0,1]) = sigmoid(w_{1.} h + b_1)$$

$$= \text{sigmoid}([-0.1 \quad 0.2] \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.1) = \text{sigmoid}(0.3)$$

$$= 0.5744$$

MX-AI

- The conditional probability distribution of RBM



$$E(v,h) = -v^T \cdot w \cdot h - b^T \cdot v - c^T \cdot h$$

$$p(v,h) = \frac{1}{Z}\exp(-E(v,h))$$

$$p(v) = \frac{1}{Z}\sum_h \exp(-E(v,h))$$

$$p(v,h) = p(h|v)\,p(v)$$

$$p(v) = \sum_h p(v,h)$$

$$p(h|v) = \frac{p(v,h)}{\sum_h p(v,h)}$$

$$p(h|v) = \frac{\exp(v^T \cdot w \cdot h + b^T \cdot v + c^T \cdot h)/Z}{\sum_h \exp(v^T \cdot w \cdot h + b^T \cdot v + c^T \cdot h)/Z}$$

$$= \frac{\exp(v^T \cdot w \cdot h + c^T \cdot h)}{\sum_h \exp(v^T \cdot w \cdot h + c^T \cdot h)}$$

$$= \frac{\exp(v^T \cdot w_{.1} h_1 + c_1 h_1)\cdot\exp(v^T \cdot w_{.2} h_2 + c_2 h_2)\cdots}{(1 + \exp(v^T \cdot w_{.1} + c_1))\cdot(1 + \exp(v^T \cdot w_{.2} + c_2))\cdots}$$

$$= \prod_{k=1}^{H} \frac{\exp(v^T \cdot w_{.k} h_k + c_k h_k)}{1 + \exp(v^T \cdot w_{.k} + c_k)} = \prod_{k=1}^{H} p(h_k|v)$$

$$p(h_k{=}1|v) = \frac{\exp(v^T \cdot w_{.k} + c_k)}{1 + \exp(v^T \cdot w_{.k} + c_k)}$$

$$= sigmoid(v^T \cdot w_{.k} + c_k)$$

$$p(v_k{=}1|h) = \frac{\exp(w_{k.} \cdot h + b_k)}{1 + \exp(w_{k.} \cdot h + b_k)}$$

$$= sigmoid(w_{k.} h + b_k)$$

$$\sum_h \exp(v^T \cdot w \cdot h + c^T \cdot h) = \quad \text{(assume V=3 and H = 2)}$$

$$\sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp\left([v_1 v_2 v_3] \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + [c_1 c_2] \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \right)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp\left([v_1 v_2 v_3] \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \end{bmatrix} h_1 + c_1 h_1 + [v_1 v_2 v_3] \begin{bmatrix} w_{12} \\ w_{22} \\ w_{32} \end{bmatrix} h_2 + c_2 h_2 \right)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{.1} h_1 + c_1 h_1 + v^T \cdot w_{.2} h_2 + c_2 h_2)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{.1} h_1 + c_1 h_1)\cdot\exp(v^T \cdot w_{.2} h_2 + c_2 h_2)$$

$$= \sum_{h_1 \in \{0,1\}} \exp(v^T \cdot w_{.1} h_1 + c_1 h_1)\cdot \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{.2} h_2 + c_2 h_2)$$

$h_1 = 0$ $h_1 = 1$ $h_2 = 0$ $h_2 = 1$

$$= (1 + \exp(v^T \cdot w_{.1} + c_1))\cdot(1 + \exp(v^T \cdot w_{.2} + c_2))$$

$$= \exp(\log(1+\exp(v^T \cdot w_{.1}+c_1)))\cdot\exp(\log(1+\exp(v^T \cdot w_{.2}+c_2)))$$

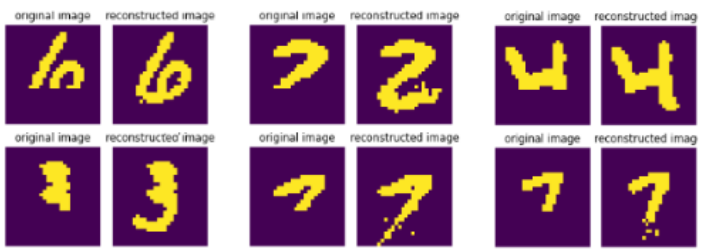
$$= \exp\left(\sum_{k=1}^{H} \log(1 + \exp(v^T \cdot w_{.k} + c_k))\right)$$

$$\frac{\partial L}{\partial w_{ij}} \simeq E_{data}[h_j(v) \cdot v_i] - E_{model}[h_j(v) \cdot v_i]$$

$$\frac{\partial L}{\partial b_i} \simeq E_{data}[v_i] - E_{model}[v_i]$$

$$\frac{\partial L}{\partial c_j} \simeq E_{data}[h_j(v)] - E_{model}[h_j(v)]$$

$$w_{ij} = w_{ij} + \alpha \frac{\partial L}{\partial w_{ij}}$$



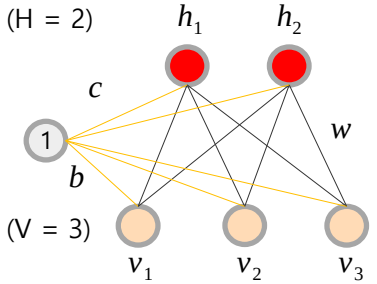
15-9. Restricted Boltzmann Machine (RBM) - (2)

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

Learning rule for RBM



$$E(v,h) = -v^T \cdot w \cdot h - b^T \cdot v - c^T \cdot h$$

$$p(v) = \frac{1}{Z}\sum_h \exp(-E(v,h))$$

$$Z = \sum_v \sum_h \exp(-E(v,h))$$

- **Learning rule**

$$L = \frac{1}{N}\sum_{n=1}^{N} \log p(v^{(n)})$$

$$w_{ij} = w_{ij} + \alpha \frac{\partial L}{\partial w_{ij}}$$

$$p(v) = \frac{1}{Z}\sum_h \exp(v^T \cdot w \cdot h + b^T \cdot v + c^T \cdot h)$$

$$= \frac{1}{Z}\exp(b^T \cdot v)\sum_h \exp(v^T \cdot w \cdot h + c^T \cdot h)$$

$$= \frac{1}{Z}\exp\left(b^T \cdot v + \sum_k^H \log(1 + \exp(v^T \cdot w_{\cdot k} + c_k))\right)$$

-F(v) : free energy

$$\log p(v) = b^T \cdot v + \sum_k^H \log(1 + \exp(v^T \cdot w_{\cdot k} + c_k)) - \log Z$$

$$\frac{\partial L}{\partial w_{21}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial \log p(v^{(n)})}{\partial w_{21}}$$

$$= \frac{1}{N}\sum_{n=1}^{N} \frac{\partial}{\partial w_{21}}\left[b^T \cdot v + \sum_k^H \log(1 + \exp(v^T \cdot w_{\cdot k} + c_k))\right] - \frac{\partial \log Z}{\partial w_{21}}$$

$$\log(1 + \exp(v^T \cdot w_{\cdot 1} + c_1)) + \log(1 + \exp(v^T \cdot w_{\cdot 2} + c_2)) + \ldots$$

$$\frac{\partial L}{\partial w_{21}} = \frac{1}{N}\sum_{n=1}^{N} \frac{\exp(v^T \cdot w_{\cdot 1} + c_1) \cdot v_2^{(n)}}{1 + \exp(v^T \cdot w_{\cdot 1} + c_1)} - \frac{\partial \log Z}{\partial w_{21}}$$

$$= \frac{1}{N}\sum_{n=1}^{N} p(h_1 = 1 \mid v^{(n)}) \cdot v_2^{(n)} - \frac{\partial \log Z}{\partial w_{21}}$$

$$\sum_h \exp(v^T \cdot w \cdot h + c^T \cdot h) =$$

(assume V=3 and H = 2)

$$\sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp\left([v_1 v_2 v_3] \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + [c_1 c_2] \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \right)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp\left([v_1 v_2 v_3] \begin{bmatrix} w_{11} \\ w_{21} \\ w_{31} \end{bmatrix} h_1 + c_1 h_1 + [v_1 v_2 v_3] \begin{bmatrix} w_{12} \\ w_{22} \\ w_{32} \end{bmatrix} h_2 + c_2 h_2 \right)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{\cdot 1} h_1 + c_1 h_1 + v^T \cdot w_{\cdot 2} h_2 + c_2 h_2)$$

$$= \sum_{h_1 \in \{0,1\}} \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{\cdot 1} h_1 + c_1 h_1) \cdot \exp(v^T \cdot w_{\cdot 2} h_2 + c_2 h_2)$$

$$= \sum_{h_1 \in \{0,1\}} \exp(v^T \cdot w_{\cdot 1} h_1 + c_1 h_1) \cdot \sum_{h_2 \in \{0,1\}} \exp(v^T \cdot w_{\cdot 2} h_2 + c_2 h_2)$$

$h_1 = 0$, $h_1 = 1$, $h_2 = 0$, $h_2 = 1$

$$= (1 + \exp(v^T \cdot w_{\cdot 1} + c_1)) \cdot (1 + \exp(v^T \cdot w_{\cdot 2} + c_2))$$

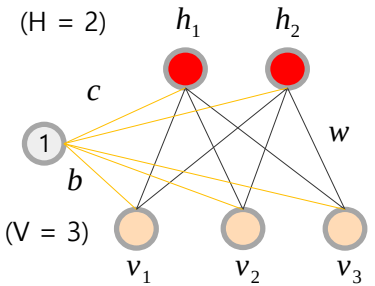
$$= \exp(\log(1 + \exp(v^T \cdot w_{\cdot 1} + c_1))) \cdot \exp(\log(1 + \exp(v^T \cdot w_{\cdot 2} + c_2)))$$

$$= \exp\left(\sum_{k=1}^{H} \log(1 + \exp(v^T \cdot w_{\cdot k} + c_k))\right)$$

MX-AI

■ Learning rule for RBM

$$\frac{\partial L}{\partial w_{21}} = \frac{1}{N}\sum_{n=1}^{N} p(h_1=1|v^{(n)})\cdot v_2^{(n)} - \frac{\partial \log Z}{\partial w_{21}}$$



$$p(v) = \frac{1}{Z}\exp\left(b^T\cdot v + \sum_{k}^{H}\log\left(1 + \exp\left(v^T\cdot w_{.k} + c_k\right)\right)\right)$$

$$p(h_1=1|v) = \frac{1}{1 + \exp\left(-\left(v^T\cdot w_{.1} + c_1\right)\right)}$$

$$\sum_h \exp\left(v^T\cdot w\cdot h + c^T\cdot h\right) = \exp\left(\sum_{k=1}^{H}\log\left(1 + \exp\left(v^T\cdot w_{.k} + c_k\right)\right)\right)$$

$$\begin{aligned} Z &= \sum_v\sum_h \exp\left(-E(v,h)\right) \\ &= \sum_v\sum_h \exp\left(v^T\cdot w\cdot h + b^T\cdot v + c^T\cdot h\right) \\ &= \sum_v \exp\left(b^T\cdot v\right)\sum_h \exp\left(v^T\cdot w\cdot h + c^T\cdot h\right) \\ &= \sum_v \exp\left(b^T\cdot v\right)\exp\left(\sum_{k=1}^{H}\log\left(1 + \exp\left(v^T\cdot w_{.k} + c_k\right)\right)\right) \end{aligned}$$

$$\log\left(1+\exp\left(v^T\cdot w_{.1}+c_1\right)\right)+\log\left(1+\exp\left(v^T\cdot w_{.2}+c_2\right)\right)+\ldots$$

$$\frac{\partial Z}{\partial w_{21}} = \sum_v \exp\left(b^T\cdot v\right)\exp\left(\sum_{k=1}^{H}\log\left(1 + \exp\left(v^T\cdot w_{.k} + c_k\right)\right)\right)\cdot\frac{\exp\left(v^T w_{.1}+c_1\right)v_2}{1+\exp\left(v^T w_{.1}+c_1\right)}$$

$$= \sum_v Z\, p(v)\cdot p(h_1=1|v)\cdot v_2$$

$$\frac{\partial \log Z}{\partial w_{21}} = \frac{1}{Z}\frac{\partial Z}{\partial w_{21}} = \sum_v p(v)\cdot p(h_1=1|v)\cdot v_2$$

$$\frac{\partial L}{\partial w_{21}} = \frac{1}{N}\sum_{n=1}^{N} p(h_1=1|v^{(n)})\cdot v_2^{(n)} - \sum_v p(v)\cdot p(h_1=1|v)\cdot v_2$$

- The gradients of the log-likelihood function with respect to the parameters

$$\frac{\partial L}{\partial w_{ij}} = \frac{1}{N}\sum_{n=1}^{N} p(h_j^{(n)}=1|v^{(n)})\cdot v_i^{(n)} - \sum_v p(v)\cdot p(h_j=1|v)\cdot v_i$$

$$\frac{\partial L}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N} v_i^{(n)} - \sum_v p(v)\, v_i$$

$$\frac{\partial L}{\partial c_i} = \frac{1}{N}\sum_{n=1}^{N} p(h_i^{(n)}=1|v^{(n)}) - \sum_v p(v)\, p(h_i=1|v)$$

- Gradient ascent

$$w_{ij} = w_{ij} + \alpha\frac{\partial L}{\partial w_{ij}} \qquad b_i = b_i + \alpha\frac{\partial L}{\partial b_i} \qquad c_i = c_i + \alpha\frac{\partial L}{\partial c_i}$$

MX-AI

■ Contrastive Divergence (CD)

▪ **Learning rules for RBM**

$$\frac{\partial L}{\partial w_{ij}} = \frac{1}{N}\sum_{n=1}^{N} p\left(h_j^{(n)}=1 \mid v^{(n)}\right)\cdot v_i^{(n)} - \sum_{v} p(v)\cdot p\left(h_j=1 \mid v\right)\cdot v_i$$

$$\frac{\partial L}{\partial b_i} = \frac{1}{N}\sum_{n=1}^{N} v_i^{(n)} - \sum_{v} p(v) v_i$$

$$\frac{\partial L}{\partial c_j} = \frac{1}{N}\sum_{n=1}^{N} p\left(h_j^{(n)}=1 \mid v^{(n)}\right) - \sum_{v} p(v)\, p\left(h_j=1 \mid v\right)$$

$$w_{ij} = w_{ij} + \alpha\frac{\partial L}{\partial w_{ij}} \qquad b_i = b_i + \alpha\frac{\partial L}{\partial b_i} \qquad c_j = c_j + \alpha\frac{\partial L}{\partial c_j}$$

$$h_j(x) \overset{\text{def}}{\equiv} p\left(h_j=1 \mid x\right)$$

approximation by CD

▪ **Contrastive Divergence learning rules**

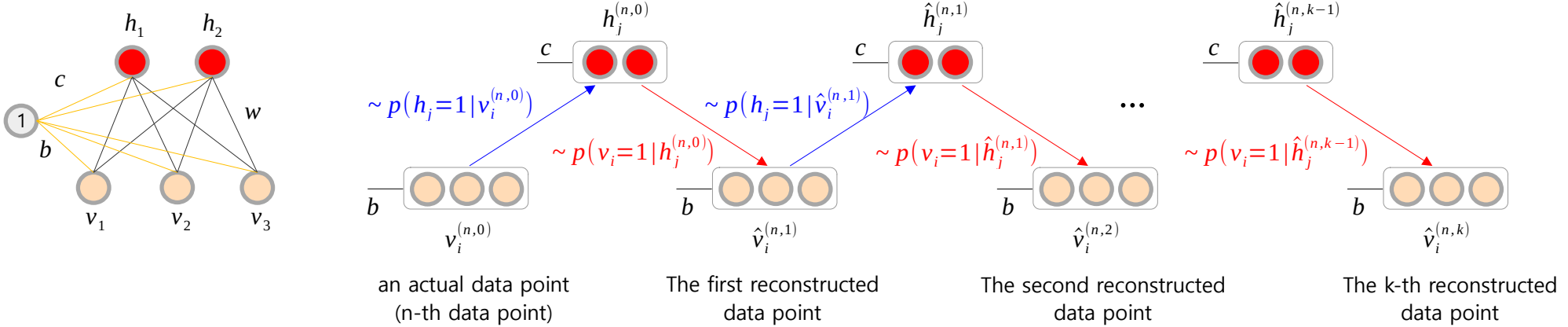
$$\begin{aligned}\frac{\partial L}{\partial w_{ij}} &\simeq \frac{1}{N}\sum_{n=1}^{N} h_j\left(v^{(n)}\right)\cdot v_i^{(n)} - \frac{1}{N}\sum_{n=1}^{N} h_j\left(\hat{v}^{(n)}\right)\cdot \hat{v}_i^{(n)} \\ &= E\left[h_j(v)\cdot v_i\right] - E\left[h_j(\hat{v})\cdot \hat{v}_i\right] \\ &= E_{data}\left[h_j(v)\cdot v_i\right] - E_{model}\left[h_j(v)\cdot v_i\right]\end{aligned}$$

$$\frac{\partial L}{\partial b_i} \simeq E_{data}\left[v_i\right] - E_{model}\left[v_i\right]$$

$$\frac{\partial L}{\partial c_j} \simeq E_{data}\left[h_j(v)\right] - E_{model}\left[h_j(v)\right]$$

Boltzmann machine: $\frac{\partial L}{\partial w_{ij}} = E_{data}\left[s_i s_j\right] - E_{model}\left[s_i s_j\right]$

▪ **k-steps Contrastive Divergence algorithm (CD-k)**

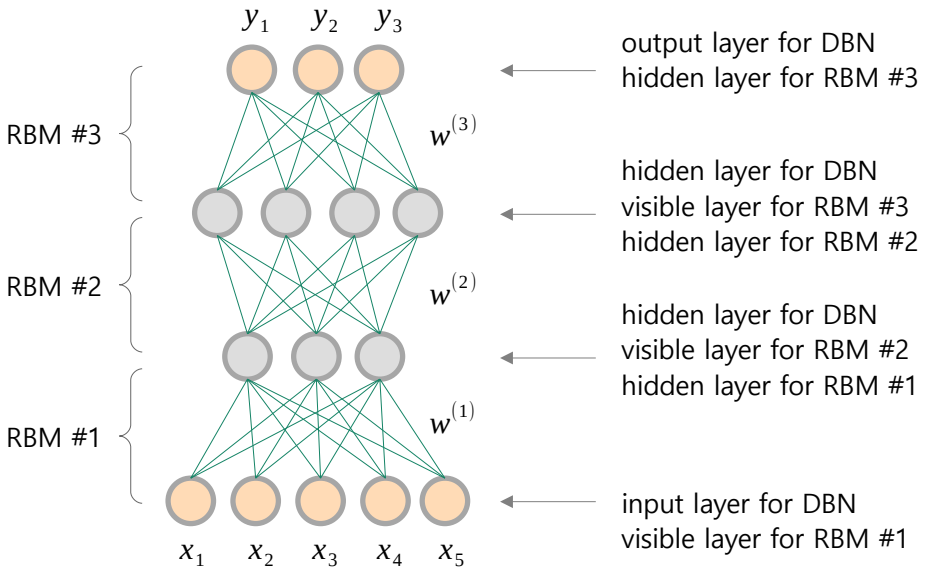


MX-AI

■ Deep Belief Network (DBN)

- A Deep Belief Network is a type of deep learning network formed by stacking multiple Restricted Boltzmann machines.
- Like RBM, it was also introduced by Geoffrey Hinton in 2006.
- Typically, a DBN is trained as follows:
  - The first RBM is trained to reconstruct the input data.
  - The hidden layer of the first RBM is treated as the visible layer of the second RBM, and the second RBM is trained using the output of the first RBM.
  - This process is repeated until all layers of the network are trained.
- DBN can be used to pre-train networks for supervised learning that perform tasks such as classification.
- The purpose of the pre-training is to initialize the weights of the DBN so that they directly represent the input data. In this stage, each RBM module is independently trained via unsupervised learning.
- Once pre-training is complete and the model's weights are properly initialized, they are further fine-tuned for downstream tasks such as classification. Fine-tuning is performed using labeled data and supervised learning algorithms such as backpropagation. This allows you to train models and improve their performance for a variety of tasks, such as classification and regression.
- The figure on the right is an example of a DBN architecture that performs multi-class classification.

[DBN architecture for multiclass classification]



MX-AI

- Implementing a simple Restricted Boltzmann machine

```
# [MXDL-15-09] 11.RBM.py
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Load the MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
    x_data, _ = pickle.load(f)     # x_data: (70000, 784)

x_data = x_data[:100]
x_data = np.where(x_data > 0.2, 1, 0)
n, m = x_data.shape      # n=100, m=784

NV = m     # the number of visible neurons
NH = 500   # the number of hidden neurons
RL = 0.01  # Learning rate
CD_K = 1   # one step Contrastive Divergence

def sampling(p):
    return np.random.binomial(n=1, p=p)

def sigmoid(z):
    return 1. / (1. + np.exp(-z))

# Initialize the model parameters
w = np.random.normal(0, 0.01, size=(NV, NH))  # weights (784, 500)
b = np.zeros([NV])   # biases for visible neurons (784,)
c = np.zeros([NH])   # biases for hidden neurons  (500,)
```

```
n_iter = 2000     # the number of iterations
n_size = 20       # a mini-batch size
n_batch = int(x_data.shape[0] / n_size) # the number of mini-batches
loss = []
for epoch in range(n_iter):
    for k in range(n_batch):
        # Yield a mini-batch
        start = k * n_size
        end = start + n_size
        bx = x_data[start:end]

        # K-steps Contrastive Divergence
        v = bx.copy()        # the starting point
        for i in range(CD_K):
            h = sigmoid(np.dot(v, w) + c)    # p(h=1 | v)
            h = sampling(h)                  # h ∈ {0, 1}, (20,500)

            v = sigmoid(np.dot(h, w.T) + b)  # p(v=1 | h)
            v = sampling(v)                  # v ∈ {0, 1}, (20,784)

        # E_data
        h_data = sigmoid(np.dot(bx, w) + c)
        E_data = np.dot(bx.T, h_data) / n_size

        # E_model
        h_model = sigmoid(np.dot(v, w) + c)
        E_model = np.dot(v.T, h_model) / n_size
```

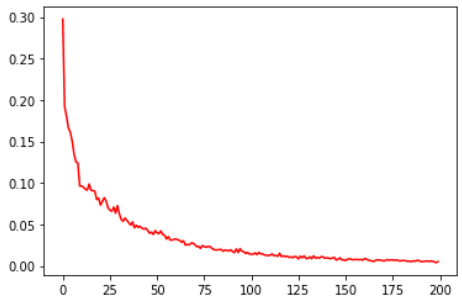
$$\frac{\partial L}{\partial w_{ij}} \simeq E_{data}[h_j(v)\cdot v_i] - E_{model}[h_j(v)\cdot v_i]$$

MX-AI

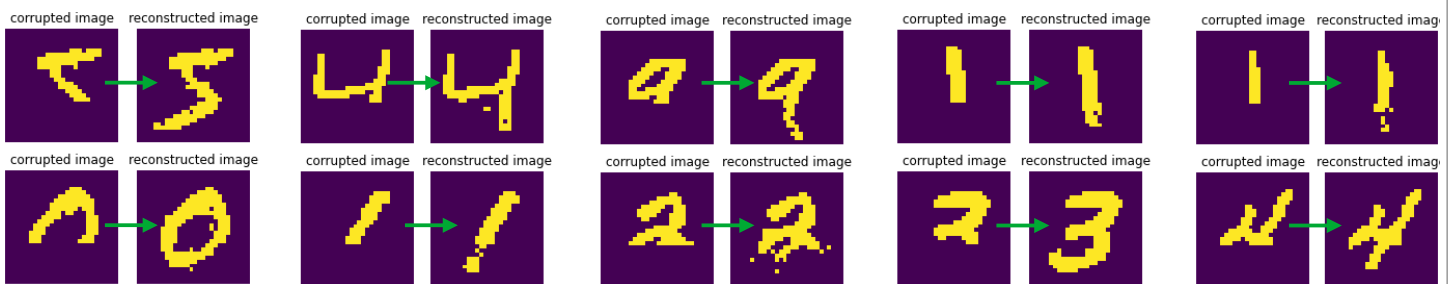
- Implementing a simple Restricted Boltzmann machine

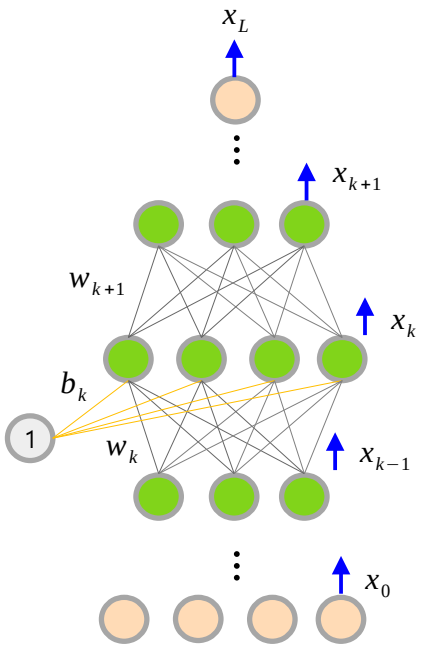
```
        # update parameters
        w += RL * (E_data - E_model)
        b += RL * np.mean(bx - v, axis=0)
        c += RL * np.mean(h_data - h_model, axis=0)

    if epoch % 10 == 0:
        loss.append(np.mean(np.square(bx - v)))
        print("epochs: {}, loss: {:.4f}".format(epoch, loss[-1]))

# Plot the loss change
plt.plot(loss)
plt.show()

# Reconstruct an image vector x
def reconstruct(x):
    h = sigmoid(np.dot(x, w) + c)
    h = sampling(h)

    v = sigmoid(np.dot(h, w.T) + b)
    v = np.where(v > 0.2, 1, 0)

    fig = plt.figure(figsize=(5,2))
    ax1 = fig.add_subplot(1,2,1)
    ax2 = fig.add_subplot(1,2,2)

    x_org = x.reshape(28,28)
    ax1.imshow(x_org)
    ax1.set_title("corrupted image")
    ax1.axis('off')
    x_hat = v.reshape(28,28)
    ax2.imshow(x_hat)
    ax2.set_title("reconstructed image")
    ax2.axis('off')
    plt.show()

def corrupted_img(p):
    p.reshape(28,28)[18:, :] = 0
    return p.reshape(-1,)

# Reconstruct 10 image patterns
for i in range(10):
    x = x_data[i].reshape(1, -1)
    reconstruct(corrupted_img(x))
```

▪ **Change in loss**



▪ **The corrupted and reconstructed images**





$$W_k \leftarrow W_k + \alpha\, y^{k-L}\left(\hat{x}_k \hat{x}_{k-1}^T - \check{x}_k \check{x}_{k-1}^T\right)$$

$$b_k \leftarrow b_k + \alpha\, y^{k-L}\left(\hat{x}_k - \check{x}_k\right)$$

15-10. Contrastive Hebbian Learning (CHL)

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

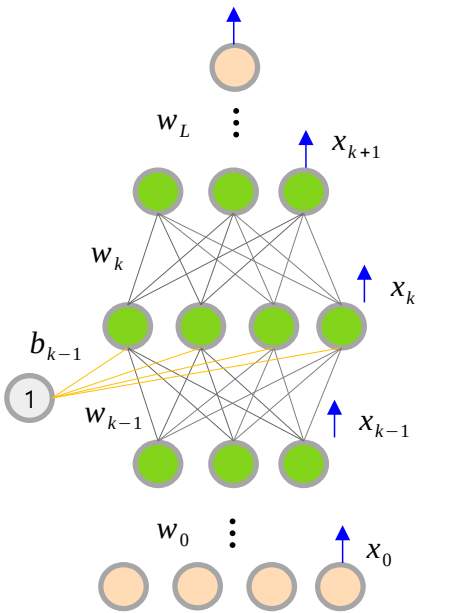
www.youtube.com/@meanxai

MX-AI

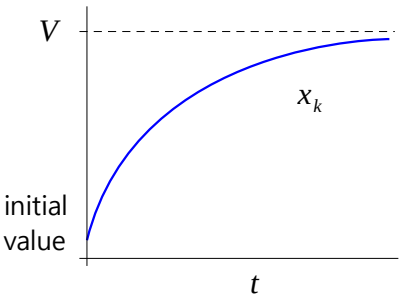
The dynamics of the neurons at k-th layer.

- Contrastive Hebbian learning is based on the contrastive divergence algorithm.
- This means that learning takes place in two phases. In the forward (free) phase, the input is presented and the output is built by forward propagation of neural activities. In the backward (clamped) phase, the outputs are clamped, and the activity is propagated backwards towards the input layer.

target data: x_L = [4, 8, 3, 1, 9, 2, ...]



x_0 : input data set



Example:

$$x_k = -C e^{-t} + V$$

$$\frac{dx_k}{dt} = C e^{-t}$$

$$\frac{dx_k}{dt} = -x_k + V$$

$$\frac{dx_k}{dt} + x_k = V$$

- **General form**

$$\frac{dx_k}{dt} + x_k = f_k(x)$$

$$\frac{dx_k}{dt} + x_k = f_k(x_{k-1} \cdot w_{k-1} + \gamma x_{k+1} \cdot w_k^T + b_{k-1})$$

$$\frac{dx_k}{dt} = -x_k + f_k(x_{k-1} \cdot w_{k-1} + \gamma x_{k+1} \cdot w_k^T + b_{k-1})$$

- **Forward Euler method**

$$x_k^{(t)} - x_k^{(t-1)} = \Delta t\,(-x_k^{(t-1)} + f_k(x_{k-1}^{(t-1)} \cdot w_{k-1} + \gamma x_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}))$$

$$x_k^{(t)} = x_k^{(t-1)} + \Delta t\,(-x_k^{(t-1)} + f_k(x_{k-1}^{(t-1)} \cdot w_{k-1} + \gamma x_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}))$$

Δt : Forward Euler method's time step

γ : Feedback gain

f_k : Activation function

MX-AI

■ Learning algorithm for CHL

(Reference: G Detorakis, et, al., 2018, Contrastive Hebbian Learning with Random Feedback Weights)

Algorithm: L is the number of layers of the network, T is the simulation time and dt the Forward Euler method's time-step.

$\check{x}_0 = \hat{x}_0 =$ input data $\quad \check{x}_L^{(0)} = 0 \quad \check{x}_L^{(t)}$: predicted target data $\quad \hat{x}_L$: target data (clamped) $\quad \check{x}_{1:L-1}^{(0)} = \hat{x}_{1:L-1}^{(0)} = 0$

Initialize w_k and b_k randomly for k=0,...,L-1

for e = 0, ..., epochs-1 **do**

  for t = 1, ..., T with step dt **do**

    for k = 1,..., L do

      $\check{x}_k^{(t)} = \check{x}_k^{(t-1)} + dt\left(-\check{x}_k^{(t-1)} + f_k\left(\check{x}_{k-1}^{(t-1)} \cdot w_{k-1} + \gamma\, \check{x}_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}\right)\right)$ ← **Forward (free) phase**

    end for

  end for

$k=1 \rightarrow dt\left(-\check{x}_1^{(t-1)} + f_1\left(\check{x}_0^{(t-1)} \cdot w_0 + \gamma\, \check{x}_2^{(t-1)} \cdot w_1^T + b_0\right)\right)$

$k=4 \rightarrow dt\left(-\check{x}_4^{(t-1)} + f_4\left(\check{x}_3^{(t-1)} \cdot w_3 + \gamma \underbrace{\check{x}_5^{(t-1)}}_{=0} \cdot w_4^T + b_3\right)\right)$

  for t = 1, ..., T with step dt **do**

    for k = 1,..., L-1 **do**

      $\hat{x}_k^{(t)} = \hat{x}_k^{(t-1)} + dt\left(-\hat{x}_k^{(t-1)} + f_k\left(\hat{x}_{k-1}^{(t-1)} \cdot w_k + \gamma\, \hat{x}_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}\right)\right)$ ← **Backward (clamped) phase**

    end for

  end for

$k=1 \rightarrow dt\left(-\hat{x}_1^{(t-1)} + f_1\left(\hat{x}_0^{(t-1)} \cdot w_0 + \gamma\, \hat{x}_2^{(t-1)} \cdot w_1^T + b_0\right)\right)$

$k=3 \rightarrow dt\left(-\hat{x}_3^{(t-1)} + f_3\left(\hat{x}_2^{(t-1)} \cdot w_2 + \gamma\, \hat{x}_4^{(t-1)} \cdot w_3^T + b_2\right)\right)$

  for k = 0,..., L-1 **do**

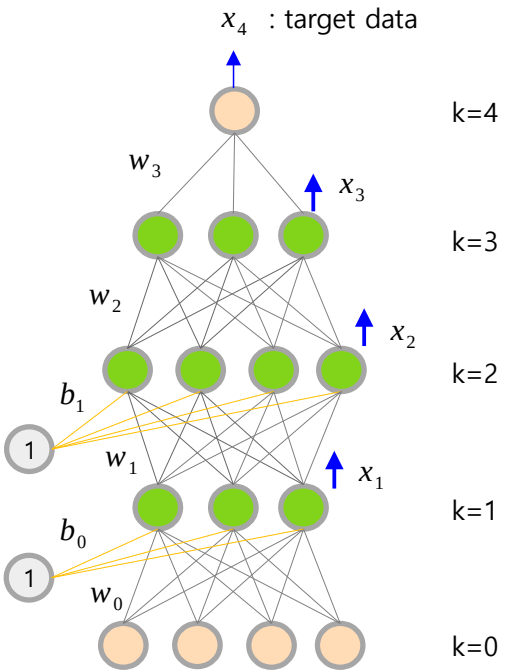
    $w_k \leftarrow w_k + \eta\, \gamma^{k-L}\left(\hat{x}_k \hat{x}_{k+1}^T - \check{x}_k \check{x}_{k+1}^T\right)$ ← **Update parameters**

    $b_k \leftarrow b_k + \eta\, \gamma^{k-L}\left(\hat{x}_{k+1} - \check{x}_{k+1}\right)$

  end for

[Learning rule for RBM]

$$\frac{\partial L}{\partial w_{ij}} \simeq E_{data}[h_j(v) \cdot v_i] - E_{model}[h_j(v) \cdot v_i]$$



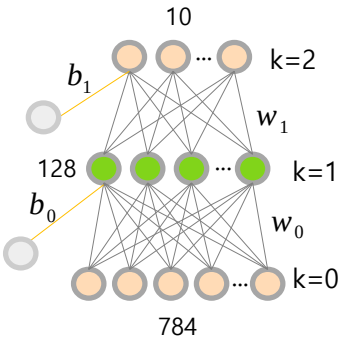
x_0 : input data set

L = 4

MX-AI

- Implementing a simple CHL model for MNIST classification

```
# [MXDL-15-10] 12.ContrastiveHebb.py
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from scipy.special import expit as sigmoid
from sklearn.model_selection \
     import train_test_split
import pickle

# Load the MNIST dataset
# x_data: (70000, 784), x_data: (70000, 1)
with open('data/mnist.pkl', 'rb') as f:
    x_input, y_target = pickle.load(f)

y_target = np.eye(10)[y_target].reshape(-1, 10) # one hot encoding
x_train, x_test, y_train, y_test = train_test_split(x_input,
                                                    y_target)

ETA = 0.01                  # learning rate
GAMMA = 0.5                 # feedback gain
n_neurons = [784, 128, 10]  # the number of neurons in each layer
L = len(n_neurons) - 1      # L = 2

n_n_epochs = 10             # the number of iterations
b_size = 10                 # a mini-batch size
n_batches= int(x_train.shape[0] / b_size) # number of mini-batches
T = 10                      # simulation time
dt = 0.5                    # Forward Euler method's time step
```




```
# Initialize the parameters: w[0], w[1], b[0], b[1]
w = [np.random.normal(0, 0.01, size=(i, o))
     for i, o in zip(n_neurons[:-1], n_neurons[1:])]
b = [np.random.normal(0, 0.01, size=(1, i))
     for i in n_neurons[1:]]
```

$$\check{x}_k^{(t)} = \check{x}_k^{(t-1)} + dt\left(-\check{x}_k^{(t-1)} + f_k\left(\check{x}_{k-1}^{(t-1)} \cdot w_{k-1} + \gamma \check{x}_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}\right)\right)$$

```
def forward_phase(x_input):
    x = [np.zeros((x_input.shape[0], m)) for m in n_neurons]
    x[0] = x_input   # input data

    for t in np.arange(1, T+1, dt):
        for k in range(1, L+1):   # k = 1, 2
            activation = np.dot(x[k-1], w[k-1])
            if k < L:
                activation += GAMMA * np.dot(x[k+1], w[k].T)
            activation += b[k-1]
            x[k] += dt * (-x[k] + sigmoid(activation))
    return x
```

$$\hat{x}_k^{(t)} = \hat{x}_k^{(t-1)} + dt\left(-\hat{x}_k^{(t-1)} + f_k\left(\hat{x}_{k-1}^{(t-1)} \cdot w_k + \gamma \hat{x}_{k+1}^{(t-1)} \cdot w_k^T + b_{k-1}\right)\right)$$

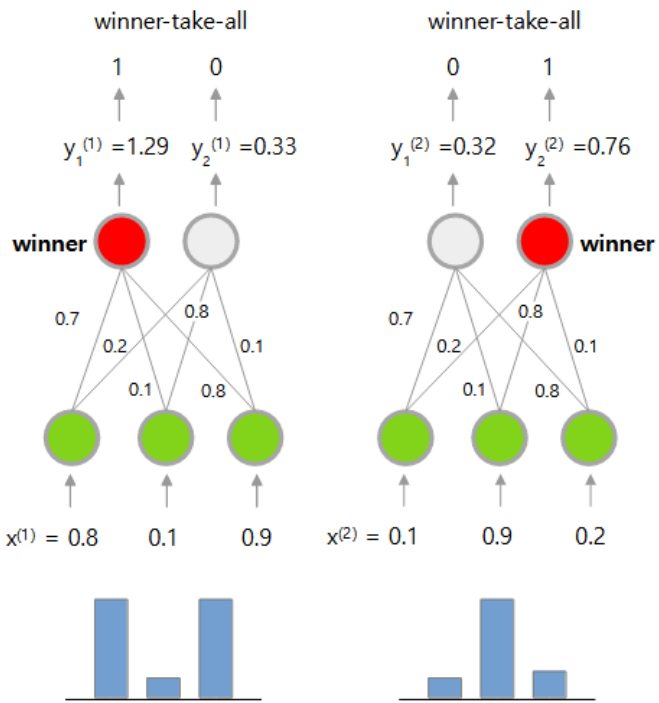
```
def backward_phase(x_input, y_target):
    x = [np.zeros((x_input.shape[0], m)) for m in n_neurons]
    x[0] = x_input     # input data
    x[-1] = y_target   # target data

    for t in np.arange(1, T+1, dt):
        for k in range(1, L):  # k = 1
            activation = np.dot(x[k-1], w[k-1]) +\
                 GAMMA * np.dot(x[k+1], w[k].T) + b[k-1]
            x[k] += dt * (-x[k] + sigmoid(activation))
    return x
```

MX-AI

- Implementing a simple CHL model for MNIST classification

```
for e in range(n_epochs):
    for j in tqdm(range(n_batches), desc=f"Epoch {e+1}/{n_epochs}"):
        b_start = j * b_size
        b_end = b_start + b_size

        x_batch = x_train[b_start:b_end]
        y_batch = y_train[b_start:b_end]

        # Forward and backward phases
        x_fwd = forward_phase(x_batch)
        x_bwd = backward_phase(x_batch, y_batch)

        # Update parameters
        for k in range(L):    # k = 0, 1
            C = ETA * GAMMA ** (k - L)
            w[k] += C * (np.dot(x_bwd[k].T, x_bwd[k+1]) -
                         np.dot(x_fwd[k].T, x_fwd[k+1]))
            b[k] += C * np.mean(x_bwd[k+1] - x_fwd[k+1], axis=0)

# get predictions on test set
x_fwd = forward_phase(x_test)
y_pred = np.argmax(x_fwd[-1], axis=1)
y_true = np.argmax(y_test, axis=1)

acc = (y_true == y_pred).mean()
print('\nAccuracy on test data ={:.4f}'.format(acc))
```

```
Epoch 1/10: 100%|██████████| 5250/5250 [00:16<00:00, 312.62it/s]
Epoch 2/10: 100%|██████████| 5250/5250 [00:17<00:00, 298.47it/s]
Epoch 3/10: 100%|██████████| 5250/5250 [00:15<00:00, 332.79it/s]
Epoch 4/10: 100%|██████████| 5250/5250 [00:15<00:00, 333.31it/s]
Epoch 5/10: 100%|██████████| 5250/5250 [00:17<00:00, 302.21it/s]
Epoch 6/10: 100%|██████████| 5250/5250 [00:16<00:00, 317.78it/s]
Epoch 7/10: 100%|██████████| 5250/5250 [00:15<00:00, 344.14it/s]
Epoch 8/10: 100%|██████████| 5250/5250 [00:16<00:00, 317.22it/s]
Epoch 9/10: 100%|██████████| 5250/5250 [00:16<00:00, 316.76it/s]
Epoch 10/10:100%|██████████| 5250/5250 [00:17<00:00, 308.26it/s]

Accuracy on test data = 0.9675
```



15-11. Competitive Learning Model

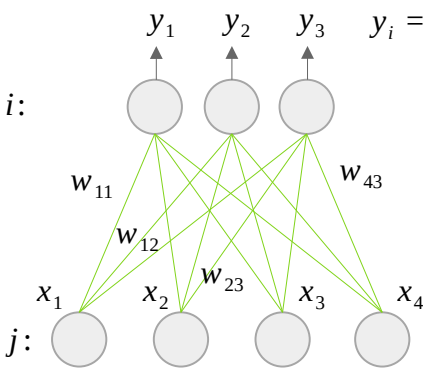
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ Instar learning rule and competitive learning rule

- **Hebbian learning with weight decay proposed by Stephen Grossberg (1960s) ← [MXDL-15-02]**



- **Instar learning rule**

$$\Delta w_{ji} = \alpha(x_j - w_{ji}) y_i$$

$$w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji}) y_i$$

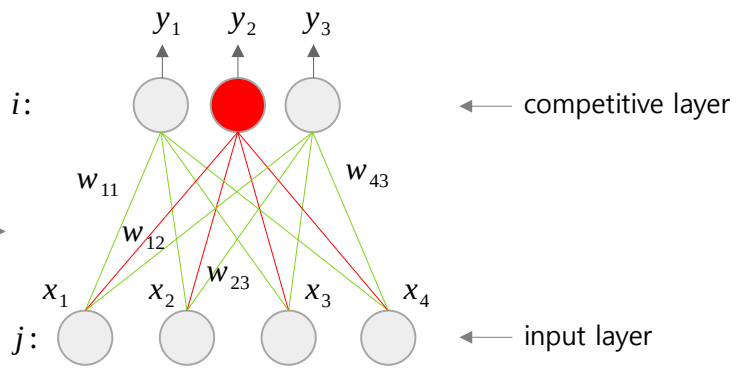
ex) $w_{23} \leftarrow w_{23} + \alpha(x_2 - w_{23}) y_3$

- **Competitive learning rule**

$$\begin{cases} w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji}) & \leftarrow y_i = 1 \\ w_{ji} \leftarrow w_{ji} & \leftarrow y_i = 0 \end{cases}$$

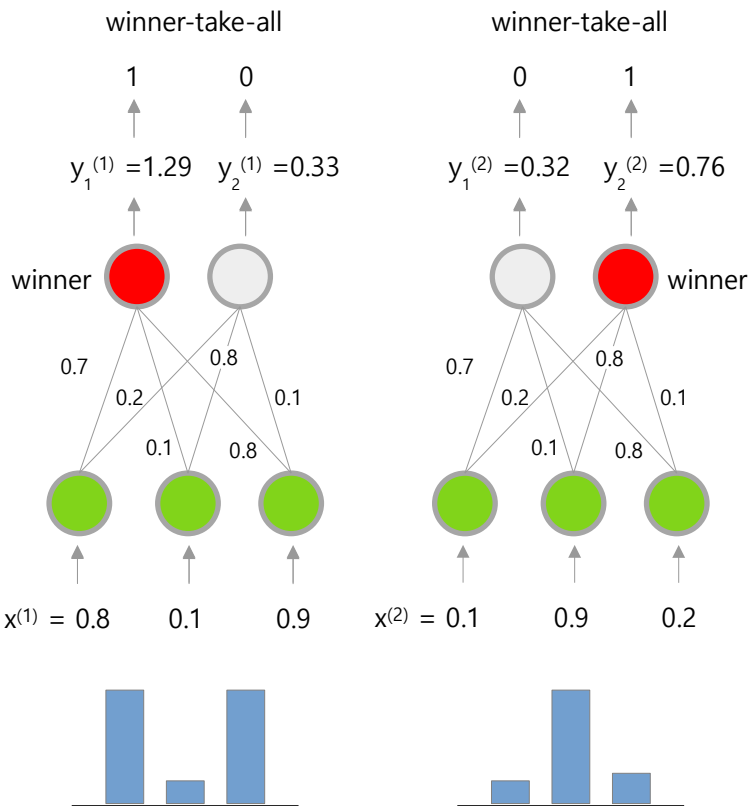
$$w = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

- Find the winner neuron in the competitive layer.
- If y2 is the winner, only y2 is activated, and all other neurons are inactivated. That is, y1 and y3 become 0, and only y2 becomes 1.
- Then, only the weights connected to y2 have the right to be updated for a given input pattern x. Other weights are not updated for that pattern.
- The winner takes it all.



MX-AI

How to find the winner neuron



1. The first way is to consider the neuron with the largest activation value as the winner.

$y_1^{(1)} = 0.8 * 0.7 + 0.1 * 0.1 + 0.9 * 0.8 = 1.29$

$y_2^{(1)} = 0.8 * 0.2 + 0.1 * 0.8 + 0.9 * 0.1 = 0.33$

$y_1^{(2)} = 0.1 * 0.7 + 0.9 * 0.1 + 0.2 * 0.8 = 0.32$

$y_2^{(2)} = 0.1 * 0.2 + 0.9 * 0.8 + 0.2 * 0.1 = 0.76$

2. The second approach is to consider the neuron with the shortest distance between the input pattern and its weights as the winner.

$y_2^{(1)} = \text{sqrt}((0.8 - 0.7)^2 + (0.1 - 0.1)^2 + (0.9 - 0.8)^2) = 0.14$

$y_2^{(1)} = \text{sqrt}((0.8 - 0.2)^2 + (0.1 - 0.8)^2 + (0.9 - 0.1)^2) = 1.22$

$y_1^{(2)} = \text{sqrt}((0.1 - 0.7)^2 + (0.9 - 0.1)^2 + (0.2 - 0.8)^2) = 1.17$

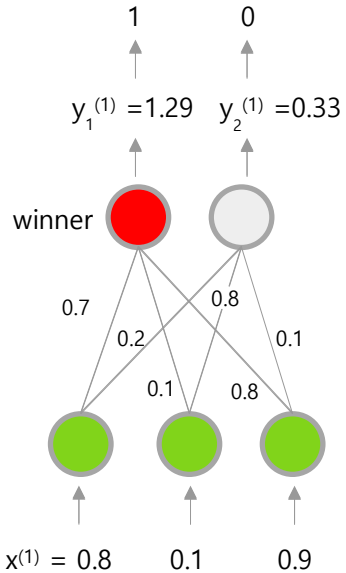
$y_2^{(2)} = \text{sqrt}((0.1 - 0.2)^2 + (0.9 - 0.8)^2 + (0.2 - 0.1)^2) = 0.17$

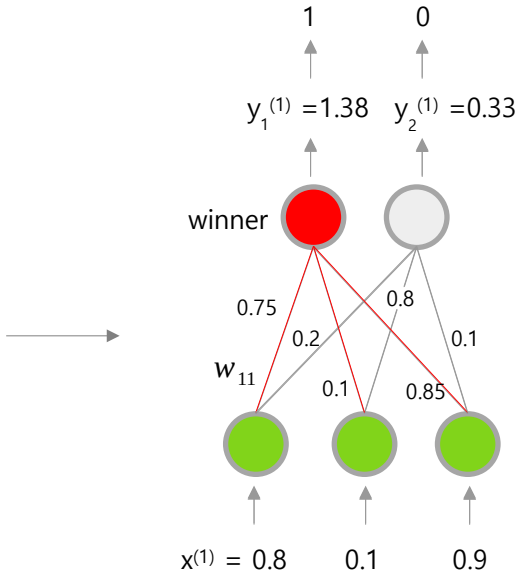
$$w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji})$$

```
w = np.array([[0.7, 0.2],
              [0.1, 0.8],
              [0.8, 0.1]])
x = np.array([[0.8, 0.1, 0.9],
              [0.1, 0.9, 0.2]])

y = np.dot(x, w) = [1.29, 0.33]
                   [0.32, 0.76]




xp = x[:, np.newaxis, :]
wp = w.T[np.newaxis, :, :]
d = np.sqrt(np.sum(np.square(xp-wp),
                   axis=2))
d = [0.14, 1.22],
    [1.17, 0.17]])
```

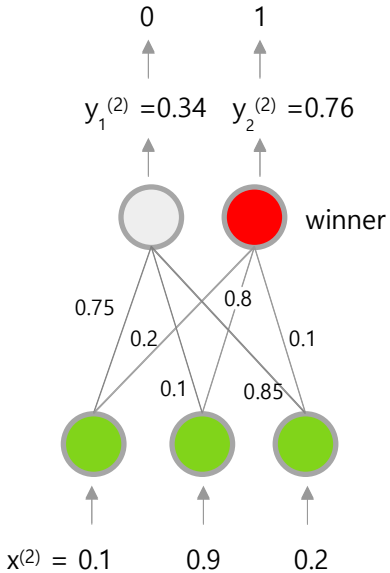
MX-AI

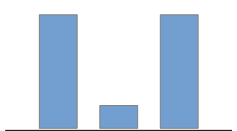
- How to update the weights






```
w = np.array([[0.75, 0.2],
              [0.1,  0.8],
              [0.85, 0.1]])
x = np.array([[0.8, 0.1, 0.9],
              [0.1, 0.9, 0.2]])

y = np.dot(x, w) = [1.38, 0.33]
                   [0.34, 0.76]
```

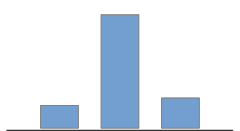




w_{11} = 0.7 + 0.5 * (0.8 - 0.7) = 0.75
w_{21} = 0.1 + 0.5 * (0.1 - 0.1) = 0.10
w_{31} = 0.8 + 0.5 * (0.9 - 0.8) = 0.85

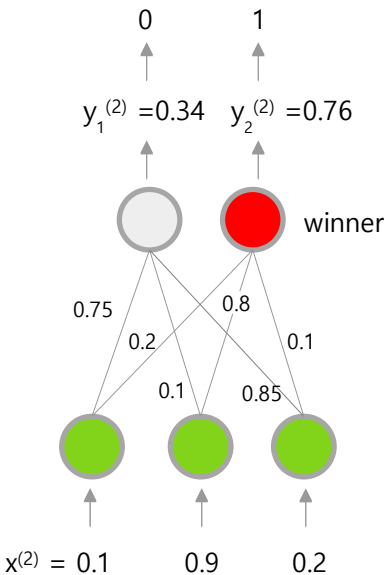
w[:, 0] = w[:, 0] + 0.5 * (x[0] - w[:, 0])

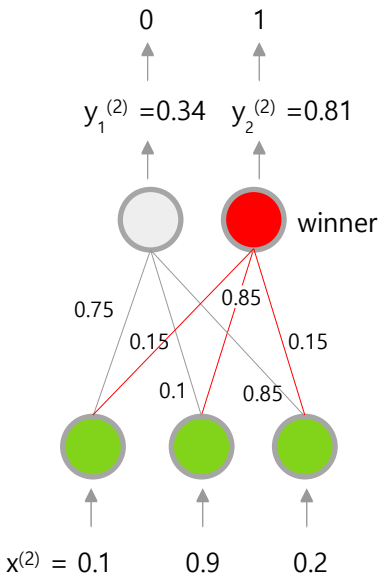
$$w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji})$$



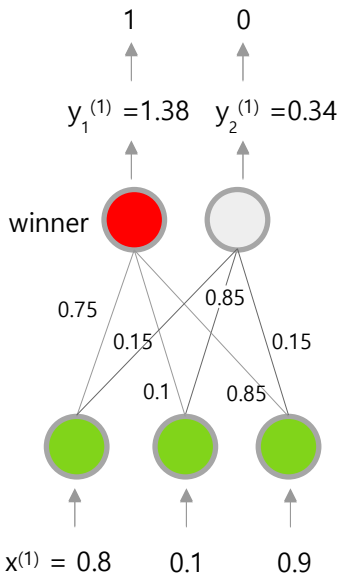
MX-AI

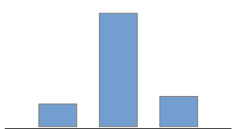
- How to update the weights






```
w = np.array([[0.75, 0.15],
              [0.1,  0.85],
              [0.85, 0.15]])
x = np.array([[0.8, 0.1, 0.9],
              [0.1, 0.9, 0.2]])

y = np.dot(x, w) = [1.38, 0.34]
                   [0.34, 0.81]
```

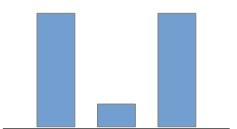




w_{12} = 0.2 + 0.5 * (0.1 - 0.2) = 0.15
w_{22} = 0.8 + 0.5 * (0.9 - 0.8) = 0.85
w_{32} = 0.1 + 0.5 * (0.2 - 0.1) = 0.15

w[:, 1] = w[:, 1] + 0.5 * (x[1] - w[:, 1])

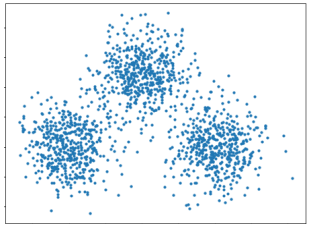
$$w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji})$$



MX-AI

■ Implementing a competitive learning model for clustering

```
# [MXDL-15-10] 13.Competitive_clust(blobs).py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from matplotlib.pyplot import cm

# Generate a dataset with 3 Gaussian clusters
n = 1500
x_input, _ = make_blobs(n_samples = n, n_features = 2,
                        centers=[(0., 0), (0.5, 0.5), (1., 0.)],
                        cluster_std = 0.15)
n_input = 2     # the number of features
n_output = 3    # 3 clusters
ALPHA = 0.01    # learning rate

# Among the neurons in the competitive layer, find the winner
# neuron with the shortest distance between the input vector and
# the weights.
def find_winner(x, w):
    x = x.reshape(1, -1)
    d = np.sqrt(np.sum(np.square(x.T - w), axis=0))  # distance
    winner = np.argmin(d)
    return np.min(d), winner

# Update the weights using the competitive learning rule.
# Only update the weights connected to the winner neuron.
# Learning rule : w = w + alpha * (x - w)
def update_weights(w, winner, x):
    w[:, winner] += ALPHA * (x - w[:, winner])
    return w
```



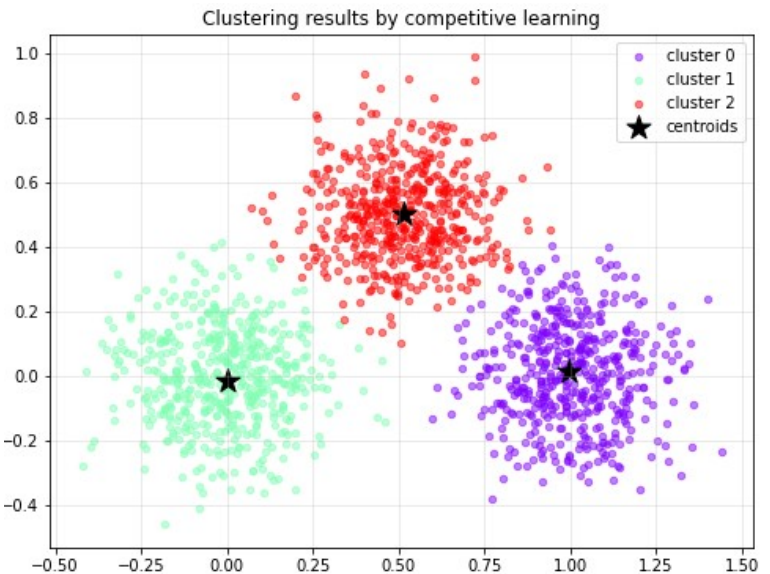
```
# Initialize the weights
w = np.random.rand(n_input, n_output)

# Training
for i in range(10):
    err = 0
    for k in range(n):
        dx = x_input[k]

        # Find the winner neuron
        d, winner = find_winner(dx, w)

        # Update the weights connected to the winner neuron
        w = update_weights(w, winner, dx)

        # The distance between dx and the weights connected
        # to the winner. This can be considered an error in
        # unsupervised learning.
        err += d

    print("{} error = {:.4f}".format(i+1, err / n))

# Predict cluster index for each data point.
clust = []
for k in range(n):
    dx = x_input[k].reshape(1, -1)
    _, winner = find_winner(dx, w)
    clust.append(winner)
```

MX-AI

- Implementing a competitive learning model for clustering

```
# Visualize the clustering results
clust = np.array(clust)
color = cm.rainbow(np.linspace(0, 1, n_output))
plt.figure(figsize=(8, 6))
for i, c in zip(range(n_output),color):
    plt.scatter(x_input[clust == i, 0], x_input[clust == i, 1],
                s=20, color=c, marker='o', alpha=0.5,
                label='cluster ' + str(i))
plt.scatter(w[0, :], w[1, :], s=250, marker='*', color='black',
            label='centroids')
plt.title("Clustering results by competitive learning")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

```
1 error = 0.2240
2 error = 0.1888
3 error = 0.1888
4 error = 0.1888
5 error = 0.1888
6 error = 0.1888
7 error = 0.1888
8 error = 0.1888
9 error = 0.1888
10 error = 0.1888
```



MX-AI

- Implementing a competitive learning model for MNIST clustering

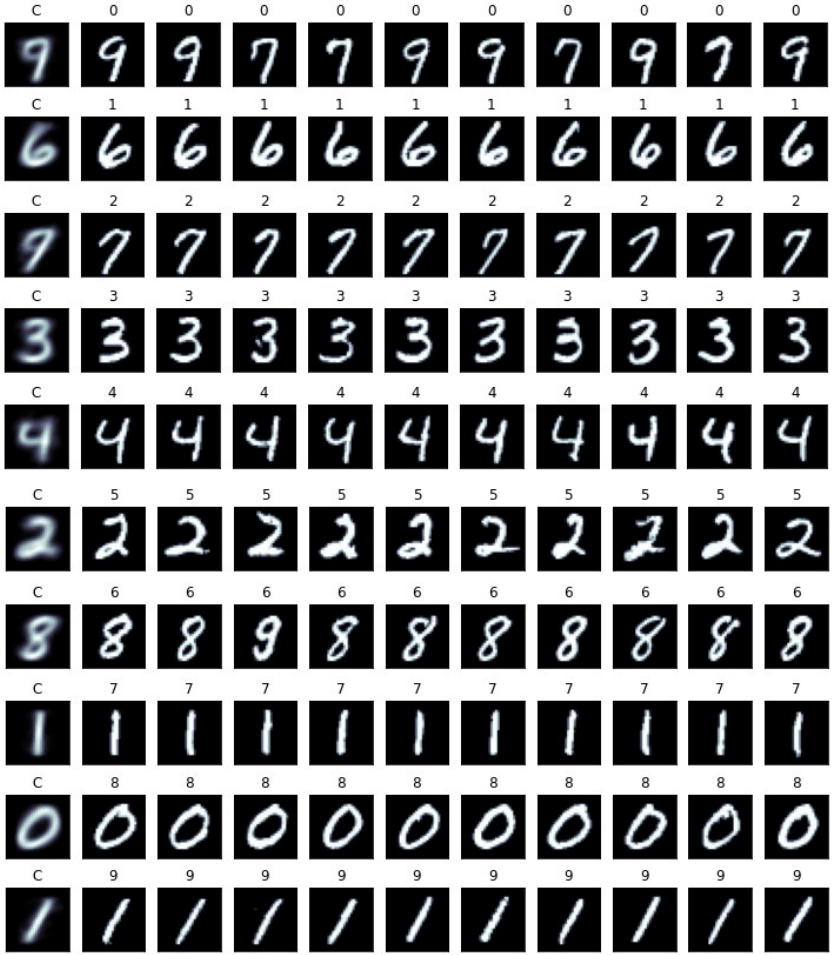
```python
# [MXDL-15-10] 14.Competitive_clust(MNIST).py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import pickle

# Load MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
    x_input, _ = pickle.load(f)     # x_input: (70000, 784)

n_input = 784
n_output = 10
ALPHA = 0.01
n = x_input.shape[0]

# Among the neurons in the competitive layer, find the winner
# neuron with the shortest distance between the input vector and
# the weights.
def find_winner(x, w):
    x = x.reshape(1, -1)
    d = np.sqrt(np.sum(np.square(x.T - w), axis=0))  # distance
    winner = np.argmin(d)
    return np.min(d), winner

# Update the weights using the competitive learning rule.
# Only update the weights connected to the winner neuron.
# Learning rule : w = w + alpha * (x - w)
def update_weights(w, winner, x):
    w[:, winner] += ALPHA * (x - w[:, winner])
    return w
```

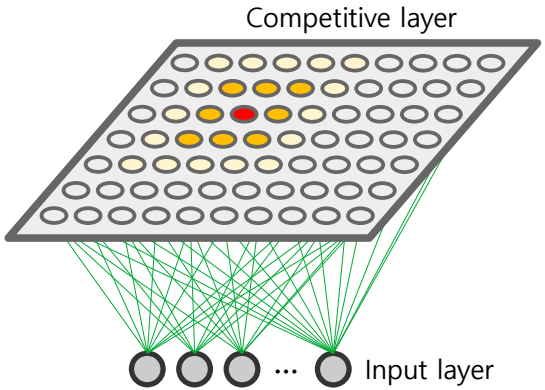
```python
# Initialize the weights
w = np.random.normal(0, 0.1, size=(n_input, n_output))

# Training
for i in range(10):
    err = 0
    for k in range(n):
        dx = x_input[k]

        # Find the winner neuron
        d, winner = find_winner(dx, w)

        # Update the weights connected to the winner neuron
        w = update_weights(w, winner, dx)

        # The distance between dx and the weights connected
        # to the winner. This can be considered an error in
        # unsupervised learning.
        err += d

    print("{} error = {:.4f}".format(i+1, err / n))

# Predict cluster index for each data point.
clust = []
for k in range(n):
    dx = x_input[k].reshape(1, -1)
    _,winner = find_winner(dx, w)
    clust.append(winner)
clust = np.array(clust)
```

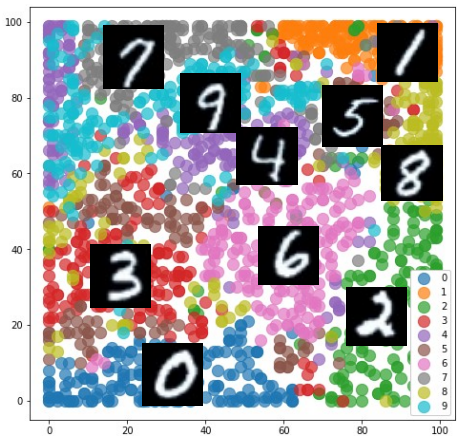
MX-AI

- Implementing a competitive learning model for MNIST clustering

```
# Visualize the clustering results
for k in np.unique(clust):
    # Find the images belonging to cluster k
    idx = np.where(clust == k)[0]
    images = x_input[idx]

    # Centroid image
    centroid = w[:, k]

    # Find 10 images near the centroid image.
    d = np.sqrt(np.sum((images - centroid)**2, axis=1))
    nearest = np.argsort(d)[:10]
    images = x_input[idx[nearest]]

    # plot the centroid image
    f = plt.figure(figsize=(10, 2))
    image = centroid.reshape(28, 28)
    ax = f.add_subplot(1, 11, 1)
    ax.imshow(image, cmap=plt.cm.bone)
    ax.grid(False); ax.set_title("C")
    ax.xaxis.set_ticks([]); ax.yaxis.set_ticks([])
    plt.tight_layout()

    for i in range(10):
        image = images[i].reshape(28,28)
        ax = f.add_subplot(1, 11, i+2)
        ax.imshow(image, cmap=plt.cm.bone)
        ax.grid(False); ax.set_title(k)
        ax.xaxis.set_ticks([]); ax.yaxis.set_ticks([])
        plt.tight_layout()
```







15-12. Self-Organizing Maps (SOMs)

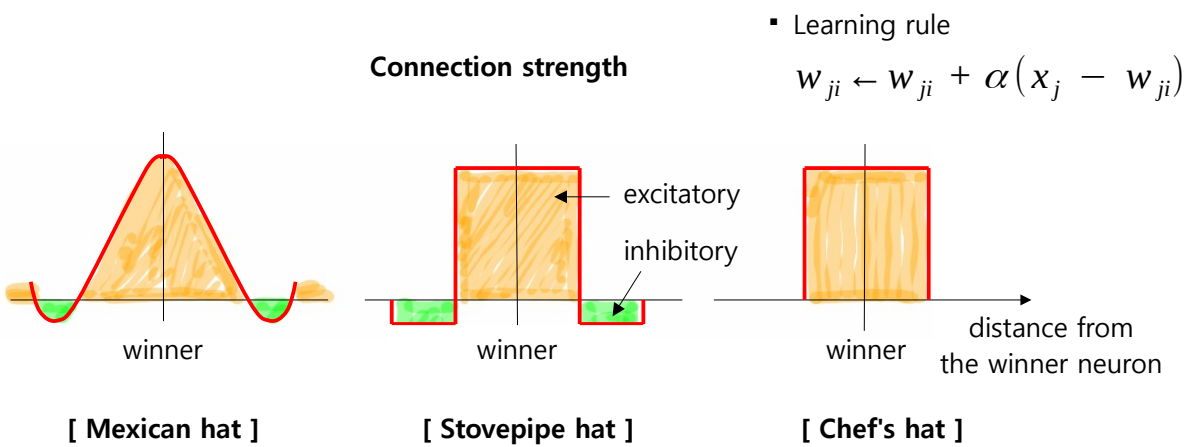
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

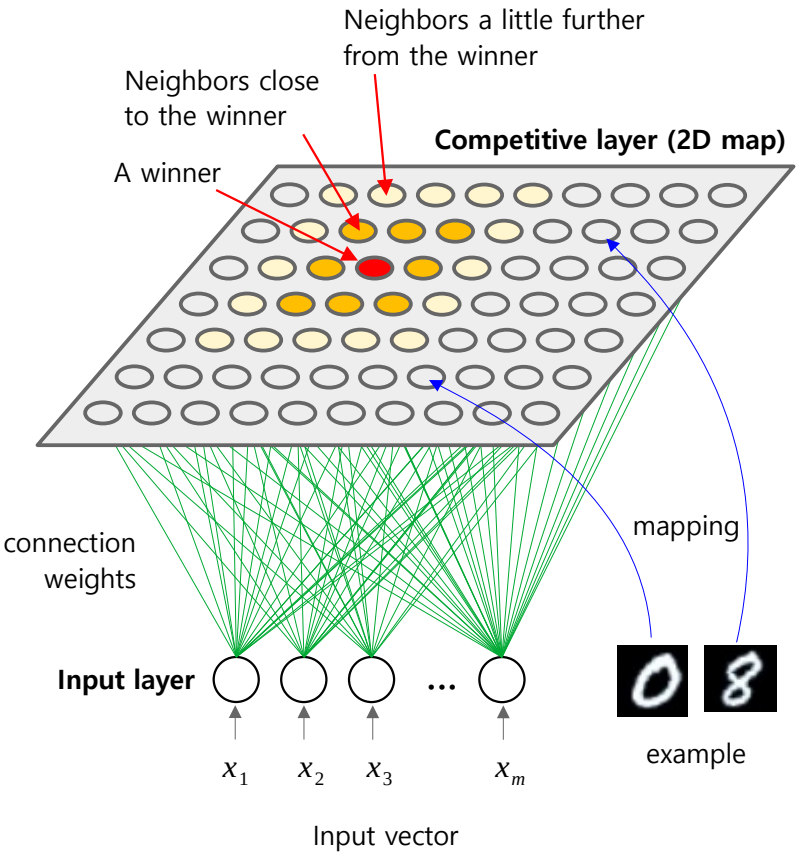
An overview of Self-Organizing Maps

- The Self-Organizing Map (SOM) was introduced by the Finnish professor Teuvo Kohonen in the 1980s and therefore is sometimes called a Kohonen map or Kohonen network.
- This is an unsupervised neural network that uses **competitive learning** to reduce high-dimensional data into a low-dimensional, often 2D, "map".
- The SOM operates in two modes: training and prediction (or mapping).
- During training, input data is presented, and the winner neuron and its neighboring neurons on the map are updated to become more like the input data.
- The strength of the weights for the winner and the neighbors can be adjusted using a Mexican hat, a Stovepipe hat, or a chef's hat function.
- During prediction, each input data point is mapped to the generated map.
- The SOM can be used for clustering, dimensionality reduction, anomaly detection, data visualization, etc.



- Learning rule

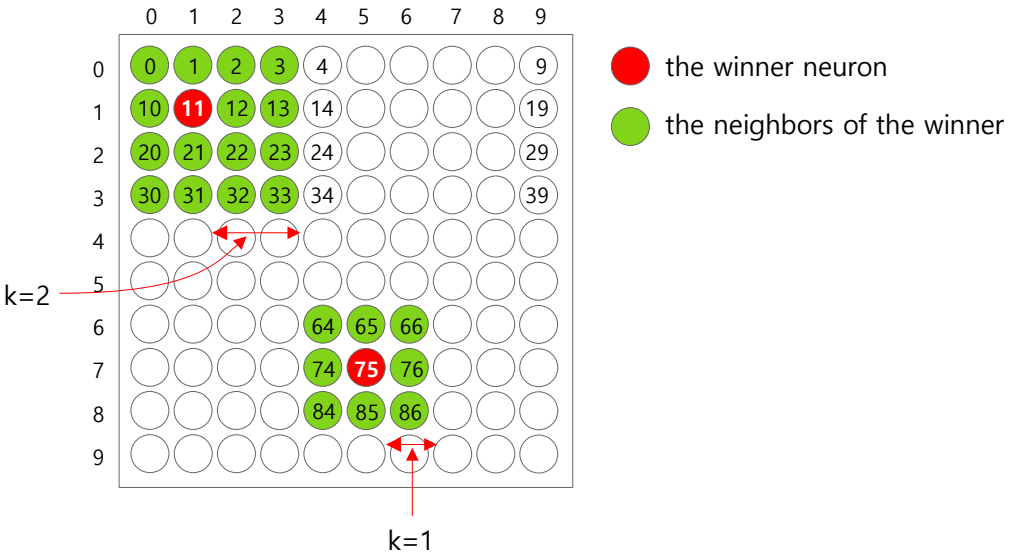
$$w_{ji} \leftarrow w_{ji} + \alpha(x_j - w_{ji})$$



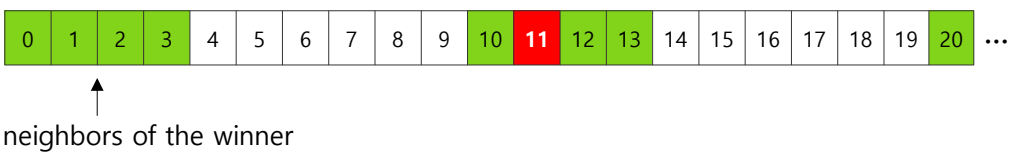
MX-AI

How to find the neighbors of the winner neuron

- A 2D competitive layer with 10 x 10 neurons



- A 1D competitive layer with 100 neurons

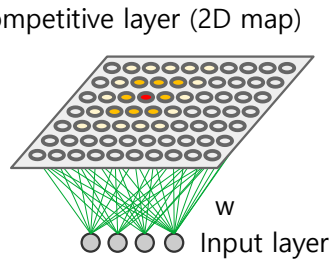



```python
# Find the neighbors around the winner neuron
# winner: 1D index of the winner neuron
# k: the maximum distance between the winner and its neighbors
def find_neighbors(winner, k):
    # convert 1D index to 2D coordinates
    # ex: winner=11 --> (row, col)=(1, 1)
    row, col = (winner // m_rows, winner % m_cols)
    from_i = np.max([0, row - k])
    to_i = row + k + 1
    if to_i > m_rows - 1:
        to_i = m_rows

    from_j = np.max([0, col - k])
    to_j = col + k + 1
    if to_j > m_cols - 1:
        to_j = m_cols

    neighbors = [i * m_cols + j for i in range(from_i, to_i)\
                                for j in range(from_j, to_j)]
    return np.array(neighbors)

m_rows, m_cols = (10, 10)
neighbors = find_neighbors(11, 2)
print(neighbors)
[ 0  1  2  3 10 11 12 13 20 21 22 23 30 31 32 33]

neighbors = find_neighbors(75, 1)
print(neighbors)
[64 65 66 74 75 76 84 85 86]
```

MX-AI

■ Implementing a SOM model to cluster the Iris dataset

```
# [MXDL-15-12] 15.SOM(iris).py
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import numpy as np
import pickle

# Load and normalize the Iris dataset
x, y = load_iris(return_X_y = True)
x_input = (x - x.mean(axis=0)) / x.std(axis=0)
y_target = y

# Initialize the weights between the input layer of size 4
# and the competitive layer of size 50x50. w: (2500, 4)
m_rows, m_cols = (50, 50)
w = np.random.normal(0, 0.1, size=(m_rows * m_cols, x_input.shape[1]))

ALPHA = 0.05              # learning rate
d_neighbor = 20           # the initial maximum distance
                          # between the winner and its neighbors
n_data = x_input.shape[0] # the number of data points

# Among the neurons in the competitive layer, find the winner
# neuron with the shortest distance between the input vector and
# the weights.
def find_winner(w, x):
    dist = np.sqrt(np.sum(np.square(x - w), axis=1))   # distance
    winner = np.argmin(dist)
    return dist[winner], winner
```




```
# Find the neighbors around the winner neuron
# winner: 1D index of the winner neuron
# k: the maximum distance between the winner and its neighbors
def find_neighbors(winner, k):
    # convert 1D index to 2D coordinates
    # ex: winner=751 --> (row, col)=(15, 1)
    row, col = (winner // m_rows, winner % m_cols)
    from_i = np.max([0, row - k])
    to_i = row + k + 1
    if to_i > m_rows - 1:
        to_i = m_rows

    from_j = np.max([0, col - k])
    to_j = col + k + 1
    if to_j > m_cols - 1:
        to_j = m_cols

    neighbors = [i * m_cols + j for i in range(from_i, to_i)\
                                for j in range(from_j, to_j)]
    return np.array(neighbors)

# Update the weights using the competitive learning rule.
# Only update the weights connected to the winner and neighbors.
# Learning rule : w = w + alpha * (x - w)
def update_weights(w, winner, x):
    w[winner, :] += ALPHA * (x.reshape(-1,) - w[winner, :])
    return w
```

MX-AI

- Implementing a SOM model to cluster the Iris dataset

```
# Training
err = []
for i in range(300):
    err.append(0)
    n_sample = 100
    for j in np.random.choice(n_data, n_sample):
        dx = x_input[j, :].reshape(1, -1)

        # Find the winner neuron
        dist, winner = find_winner(w, dx)

        # Find the neighbors of the winner neuron
        neighbors = find_neighbors(winner, d_neighbor)

        # Update the weights connected to the winner neuron
        # and its neighbors via Hebb's rule.
        for m in neighbors:
            w = update_weights(w, m, dx)

        err[-1] += dist / n_sample

    # Reduce the maximum distance between the winner and
    # its neighbors by 1.
    if i % 3 == 0:
        d_neighbor = np.max([0, d_neighbor - 1])

    print("{} error = {:.2f}, d_neighbor = {}"\
          .format(i+1, err[-1], d_neighbor))
```

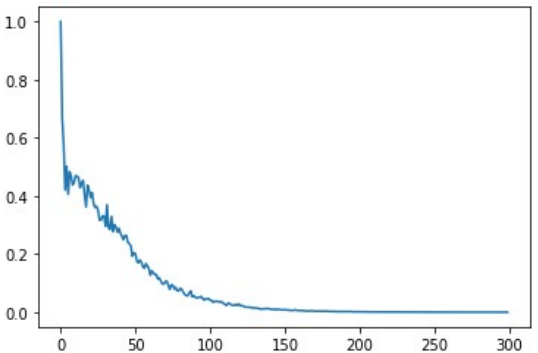
```
# Plot the error history
plt.plot(err)
plt.show()

# Find the 2D coordinates of the winner neurons for given
# data points in the competitive layer.
winners = []
for n in range(x_input.shape[0]):
    x = x_input[n].reshape(1, -1)
    _, winner = find_winner(w, x)

    row = winner // m_rows
    col = winner % m_cols
    winners.append((row, col))
winners = np.array(winners)

# Visualize the self-organizing map.
plt.figure(figsize=(6,6))
mark = ['o', 's', '^']
for i in np.unique(y_target):
    idx = np.where(y_target == i)[0]
    z = winners[idx]
    plt.scatter(z[:, 0], z[:, 1], s=300, marker=mark[i],
                alpha=0.7)
plt.legend(['setosa', 'versicolor', 'virginica'],
           bbox_to_anchor=(1, 1), prop={'size': 15})
plt.title('A self organizing map for Iris dataset')
plt.show()
```

MX-AI

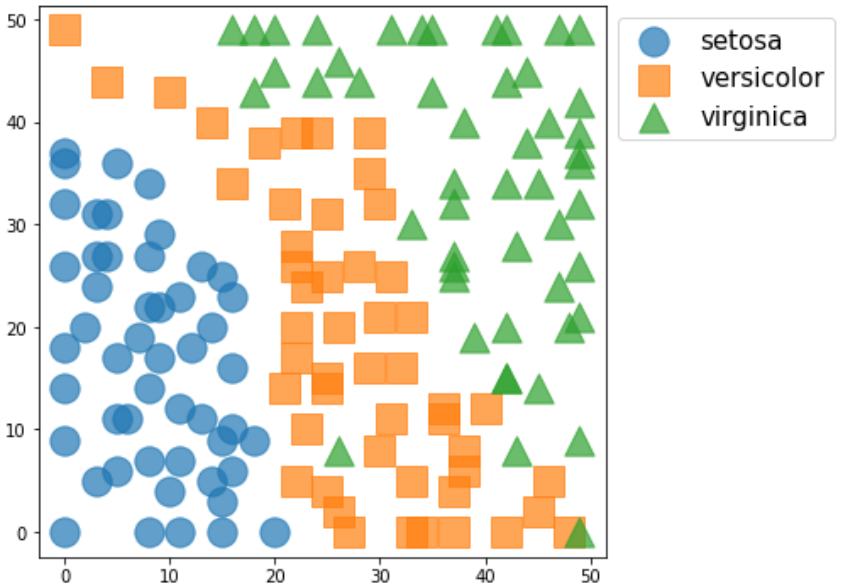
- Implementing a SOM model to cluster the Iris dataset

```
  1 error = 0.9999, d_neighbor = 19
  2 error = 0.6698, d_neighbor = 19
  3 error = 0.5780, d_neighbor = 19
  4 error = 0.4205, d_neighbor = 18
  5 error = 0.5021, d_neighbor = 18
  6 error = 0.4060, d_neighbor = 18
  7 error = 0.4833, d_neighbor = 17
  8 error = 0.4671, d_neighbor = 17
  9 error = 0.4379, d_neighbor = 17
 10 error = 0.4466, d_neighbor = 16
 11 error = 0.4700, d_neighbor = 16
...
292 error = 0.0001, d_neighbor = 0
293 error = 0.0001, d_neighbor = 0
294 error = 0.0001, d_neighbor = 0
295 error = 0.0001, d_neighbor = 0
296 error = 0.0001, d_neighbor = 0
297 error = 0.0001, d_neighbor = 0
298 error = 0.0001, d_neighbor = 0
299 error = 0.0001, d_neighbor = 0
300 error = 0.0000, d_neighbor = 0
```

▪ Error history



▪ A Self-Organizing Map for Iris dataset



MX-AI

■ Implementing a SOM model to cluster the MNIST dataset

```
# [MXDL-15-12] 16.SOM(mnist).py
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Load and normalize the MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
       x, y = pickle.load(f)

x_input = x[:2000] * 2 - 1   # -1 ~ +1
y_target = y.reshape(-1,)[:2000]

# Initialize the weights between the input layer of size 784
# and the competitive layer of size 100x100. w: (10000, 784)
m_rows, m_cols = (100, 100)
w = np.random.normal(0, 0.1, size=(m_rows * m_cols, x_input.shape[1]))

ALPHA = 0.05                # learning rate
d_neighbor = 20             # the initial maximum distance
                            # between the winner and its neighbors
n_data = x_input.shape[0]   # the number of data points

# Among the neurons in the competitive layer, find the winner
# neuron with the shortest distance between the input vector and
# the weights.
def find_winner(w, x):
    dist = np.sqrt(np.sum(np.square(x - w), axis=1))
    winner = np.argmin(dist)
    return dist[winner], winner
```

```
# Find the neighbors around the winner neuron
# winner: 1D index of the winner neuron
# k: the maximum distance between the winner and its neighbors
def find_neighbors(winner, k):
    # convert 1D index to 2D coordinates
    # ex: winner=751 --> (row, col)=(15, 1)
    row, col = (winner // m_rows, winner % m_cols)
    from_i = np.max([0, row - k])
    to_i = row + k + 1
    if to_i > m_rows - 1:
        to_i = m_rows

    from_j = np.max([0, col - k])
    to_j = col + k + 1
    if to_j > m_cols - 1:
        to_j = m_cols

    neighbors = [i * m_cols + j for i in range(from_i, to_i)\
                                for j in range(from_j, to_j)]
    return np.array(neighbors)

# Update the weights using the competitive learning rule.
# Only update the weights connected to the winner and neighbors.
# Learning rule : w = w + alpha * (x - w)
def update_weights(w, winner, x):
    w[winner, :] += ALPHA * (x.reshape(-1,) - w[winner, :])
    return w
```

MX-AI

- Implementing a SOM model to cluster the MNIST dataset

```
# Training
err = []
for i in range(300):
    err.append(0)
    n_sample = 500
    for j in np.random.choice(n_data, n_sample):
        dx = x_input[j, :].reshape(1, -1)

        # Find the winner neuron
        dist, winner = find_winner(w, dx)

        # Find the neighbors of the winner neuron
        if d_neighbor > 0:
            neighbors = find_neighbors(winner, d_neighbor)
        else:
            neighbors = [winner]

        # Update the weights connected to the winner neuron
        # and its neighbors via Hebb's rule.
        for m in neighbors:
            w = update_weights(w, m, dx)

        err[-1] += dist / n_sample

    # Reduce the maximum distance between the winner and
    # its neighbors by 1.
    if i % 2 == 0: d_neighbor = np.max([0, d_neighbor - 1])

    print("{} error = {:.2f}, d_neighbor = {}"\
          .format(i+1, err[-1], d_neighbor))
```

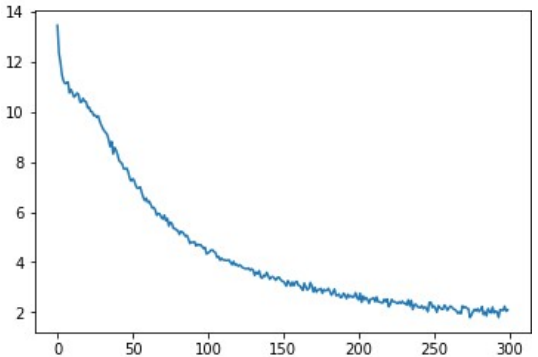
```
# Plot the error history
plt.plot(err)
plt.show()

# Find the 2D coordinates of the winner neurons for given
# data points in the competitive layer.
winners = []
for n in range(n_data):
    x = x_input[n].reshape(1, -1)
    _, winner = find_winner(w, x)

    row = winner // m_rows
    col = winner % m_cols
    winners.append((row, col))
winners = np.array(winners)

# Visualize the self-organizing map.
plt.figure(figsize=(8,8))
for i in np.unique(y_target):
    idx = np.where(y_target == i)[0]
    z = winners[idx]
    plt.scatter(z[:, 0], z[:, 1], s=150, alpha=0.7, label=str(i))
plt.legend(framealpha=1.0)
plt.title('A self organizing map for MNIST digits')
plt.show()
```

MX-AI

- Implementing a SOM model to cluster the MNIST dataset

```
  1 error = 13.44, d_neighbor = 19
  2 error = 12.34, d_neighbor = 19
  3 error = 11.97, d_neighbor = 18
  4 error = 11.47, d_neighbor = 18
  5 error = 11.23, d_neighbor = 17
  6 error = 11.13, d_neighbor = 17
  7 error = 11.15, d_neighbor = 16
  8 error = 11.19, d_neighbor = 16
  9 error = 10.76, d_neighbor = 15
 10 error = 10.88, d_neighbor = 15
 11 error = 10.75, d_neighbor = 14
...
292 error = 2.05, d_neighbor = 0
293 error = 2.01, d_neighbor = 0
294 error = 1.80, d_neighbor = 0
295 error = 2.10, d_neighbor = 0
296 error = 2.08, d_neighbor = 0
297 error = 2.08, d_neighbor = 0
298 error = 2.24, d_neighbor = 0
299 error = 2.04, d_neighbor = 0
300 error = 2.09, d_neighbor = 0
```

- Error history



- A Self-Organizing Map for MNIST digits