Week 4: Hopfield Network

Phong Le, Willem Zuidema

November 20, 2013

Last week we studied multi-layer perceptron, a neural network in which information is only allowed to transmit in one direction (from input units to output units), for supervised learning. This week, we will study another famous net, namely Hopfield net where information is allowed to transmit backward.

Required Library We use the library 'png' for reading and writing png files. Install it by typing `install.packages("png")`.

Required R Code At `http://www.illc.uva.nl/LaCo/clas/fncm13/assignments/computerlab-week4/` you can find the R-files you need for this exercise.

1 Definition

Hopfield network is a recurrent neural network in which any neuron is an input as well as output unit, and

- each neuron i is a perceptron with the binary threshold activation function,
- any pair of neurons (i, j) are connected by two weighted links w_{ij} and w_{ji}.

Formally speaking, a neuron is characterized by

$$v_i(t) = sign(u_i(t)) = \begin{cases} 1 & \text{if } u_i(t) \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

where $u_i(t)$, the total input at time t, is computed by

$$u_i(t) = \sum_{j \neq i} v_j(t-1) w_{ji} + \theta_i$$

A neuron i is called *stable* at time t if $v_i(t) = v_i(t-1) = sign(u_i(t-1))$. A Hopfield net is called stable if all of its neurons are stable. Here after, we drop the time t, and assume that all the biases are 0; i.e., we use u_i instead of $u_i(t)$, and $\theta_i = 0$ for all i.

2 Activation

There are two ways to update a Hopfield net, *asynchronous* – where only one neuron, which is randomly picked up, is updated at a time – and *synchronous* – where all neurons are updated at a time. It is proved that, if the net is symmetric, i.e. $w_{ij} = w_{ji}$ for all i, j, then it will definitely converge to a stable point (see Theorem 2, page 51, Kröse and van der Smagt (1996)), which is a local minimum of the following energy function

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} v_i v_j = -\frac{1}{2} \mathbf{v}^T \mathbf{W} \mathbf{v}$$

Exercise 1: In this exercise, you will study how a Hopfield net updates its state. Let consider the network given in Figure 1.

1. First of all, you need to load the file 'hopfield.R' (typing `source("hopfield.R")`).

2. Then, create a Hopfield net, by create a weight matrix and then put it in a list as follows

```
weights = matrix(c(
                0,  1,  -2,
                1,  0,  1,
                -2, 1,  0),
            3, 3)
hopnet = list(weights = weights)
```

3. Set the initial state `init.y = c(1,-1,1)`.

4. Next, activate the net by typing
`run.hopfield(hopnet, init.y, stepbystep=T, topo=c(3,1))`. You should see the below

```
weights =
     [,1] [,2] [,3]
[1,]    0    1   -2
[2,]    1    0    1
[3,]   -2    1    0
input pattern =  1 -1 1
```

and a black-white plot illustrating the current state of the network (black represents -1, white 1).

5. Press Enter, you should see something like this

```
iter  1
pick neuron  2
input to this neuron  2
ouput of this neuron  1
new state  1 1 1
```

This means that, in the first iteration, we pick the neuron number 2. The total input to this neuron is 2, and hence its output is 1.

6. Press Enter three or four times to see what happens next.

7. Now, do all the steps above again with your own weight matrix and initial state. Report what you get (just copy what you see in your R console and paste it into your report).

Exercise 2: To see why symmetry is important for the convergence, let's consider a 2-neuron Hopfield net with the weight matrix

$$\mathbf{W} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Your task is to examine whether the net converges or not if the initial state is $\mathbf{v} = (1, -1)^T$. To do so,

- load the R file 'hopfield.R' (`source("hopfield.R")`),
- create the corresponding Hopfield net by typing

```
hopnet = list(weights = matrix(c(0, 1, -1, 0),  2, 2))
```

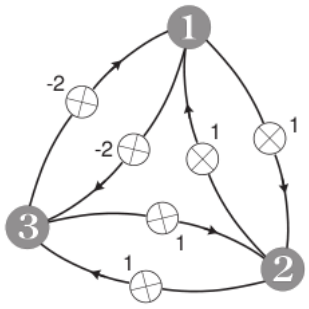

Figure 1: A Hopfield net.

- set the initial state `init.y = c(1,-1)`,
- activate the net by typing

  ```
  run.hopfield(hopnet, init.y, maxit = 10, stepbystep=T, topo=c(2,1))
  ```

  (note that `maxit` is the number of times we pick a neuron to activate it.)

Now, could you conclude anything about the convergence?

Set the weight matrix $\mathbf{W} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ and do all the steps above again. Does the net reach a stable state?

3 Learning with Hebb's rule

A Hopfield net is an associative memory in the sense that it could be used to store patterns and we can retrieve a stored pattern with an incomplete input. The principle behind it is that, the weight matrix is set such that the stored patterns correspond to stable points; and therefore, given an incomplete input pattern, the net will iterate to converge to a stable point which corresponds to a stored pattern most 'similar' to the input.

Given m patterns $\mathbf{p}_k = (p_{k,1}, ..., p_{k,n})^T, k = 1..m$, the Hebbian learning rule can be used to set up the weights as follows

$$w_{ij} = \begin{cases} \sum_{k=1}^{m} p_{k,i} p_{k,j} & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

which could be written compactly:

$$\mathbf{W} = \sum_{k=1}^{m} \mathbf{p}_k \mathbf{p}_k^T - m\mathbf{I}$$

Unfortunately, the learning rule above is not perfect. In order to decide when to apply that rule and which net structure to use, let's claim a store pattern, says $\mathbf{p}_1$, and see when the net converges to that pattern. Recall that the net is stable at $\mathbf{p}_1$ when all neurons are stable, i.e. $\mathbf{v} = sign(\mathbf{u}) = sign(\mathbf{W}\mathbf{p}_1) = \mathbf{p}_1$. Analysing $\mathbf{u}$ we have

$$\begin{aligned} \mathbf{u} &= \mathbf{W}\mathbf{p}_1 \\ &= (\mathbf{p}_1\mathbf{p}_1^T + ... + \mathbf{p}_m\mathbf{p}_m^T - m\mathbf{I})\mathbf{p}_1 \\ &= \mathbf{p}_1\mathbf{p}_1^T\mathbf{p}_1 + ... + \mathbf{p}_m\mathbf{p}_m^T\mathbf{p}_1 - m\mathbf{I}\mathbf{p}_1 \\ &= (n-m)\mathbf{p}_1 + \sum_{j>1} \alpha_j \mathbf{p}_j \end{aligned}$$

where n is the number of neurons, α_j could be considered as the 'similarity' between $\mathbf{p}_1$ and $\mathbf{p}_j$. From this, we can see that, in order to have $sign(\mathbf{u}) = \mathbf{p}_1$, $n > m$ and α_j are small. Intuitively, the number of neurons must be greater than the number of patterns we want to store, and those patterns must be distinguishable.

Exercise 3: In this exercise, you will study how to use the Hebbian learning rule to update a Hopfield net's weights, and its effect.

1. Let's create two 10×10 images, namely `pattern1, pattern2`, and the matrix weight as follows

```
pattern1 = rep(1, 10*10)
pattern2 = rep(-1, 10*10)
weights = matrix(rep(0,10*10*10*10),10*10, 10*10)
```

2. Then, use the Hebbian learning rule to update the weight matrix

```
weights = weights + pattern1 %o% pattern1 + pattern2 %o% pattern2 - 2*diag(10*10)
```

3. Examine whether the net with that weight matrix stores those two patterns by executing

```
run.hopfield(list(weights=weights), pattern1,
                    stepbystep=T, topo=c(10,10), replace=F)
```

and

```
run.hopfield(list(weights=weights), pattern2,
                    stepbystep=T, topo=c(10,10), replace=F)
```

4. Create other two patterns, which are `pattern1` but some lines are filled with -1

```
pattern3 = matrix(pattern1, 10, 10)
for (i in 1:3)
      pattern3[i,] = -1
dim(pattern3) = NULL

pattern4 = matrix(pattern1, 10, 10)
for (i in 1:5)
      pattern4[i,] = -1
dim(pattern4) = NULL
```

If these patterns are set as the initial state for the net, which patterns will the net retrieve?

5. Now, use the Hebbian learning rule to store `pattern3`, by updating the weight matrix as follows

```
weights = weights + pattern3 %o% pattern3 - diag(10*10)
```

Does the net now remember this pattern?

6. If `pattern4` is set as the initial state for the net, which patterns will the net retrieve?

7. One interesting property of the Hebbian learning rule is that we can use its reverse (i.e,. addition becomes subtraction and vice versa) to 'erase' a pattern out of the memory. To make the net 'forget' `pattern3`, execute

```
weights = weights - pattern3 %o% pattern3 + diag(10*10)
```

Now, check if the net still remembers `pattern3`. Check if `pattern4` is set as the initial state for the net, which patterns will the net retrieve?

Exercise 4: In this exercise, you will train a Hopfield net to store some digits. The files you need are 'hopfield.R' for training a Hopfield net, 'hopfield_digit.R' for the whole experiment, and '0.png', '1.png',..., '9.png' which are images of the ten digits.

1. First of all, execute the file 'analyse_digit.R' to examine the similarities between the digits. Which pair is the most similar? Which is the least? Which digit is the most distinguishable from the others? (Note the digit 0 is indexed 10.)
2. Open 'hopfield_digit.R' and set `digits` to a set of digits you want to store in a Hopfield net. First try with all ten digits. Then with all odd digits, and finally with all even digits. After executing the 'hopfield_digit.R' file, you should see a plot, in which the first row contains input images, the second row output images, and the third row expected output images. What do you see? Can you explain?
3. Find the largest set of digits that the net can store.
4. Now, we consider only even digits. The parameter `noise_rate` decides the probability that a pixel is flipped, e.g. `noise_rate = 0.1` means any pixel is flipped with the probability 0.1. The higher `noise_rate` is, the more noisy the input is. Gradually increase the value of `noise_rate` from 0 to 1, report the maximum value that the network correctly retrieves all inputs.
5. In some cases, the retrieved digit looks quite good, but not perfect. Can you explain why?

4 Submission

You have to submit a file named 'your_name.pdf'. The deadline is 15:00 Monday 25 Nov. If you have any questions, contact Phong Le (p.le@uva.nl).

References

Kröse, B. and van der Smagt, P. (1996). An introduction to neural networks.