Artificial Neural Networks and other Learning Systems, 2D1432

Exercise 2: Hopfield Networks

1 Objectives

This exercise is about recurrent networks, especially the Hopfield network and different forms of associative networks. When you are finished you should understand:

- how auto-associative networks work.
- how the Hopfield network is trained.
- how the dynamics of Hopfield networks can be described by an energy function.
- the concept of an attractor state.
- how auto-associative networks can do pattern completion and noise reduction.
- how sparse encoding can increase number of storable patterns

2 Tasks

We will use Matlab to do all calculations in this exercise. Most of the operations of the Hopfield-type of networks can be seen as vector-matrix operations which are easy to express in Matlab.

We will look at some simple networks using the Hebbian learning principle which is often used in recurrent networks. You will construct an auto-associative memory of the Hopfield type, and explore its capabilities, capacity and limitations. Most of the tasks consist of observing the dynamics and analysing why it occurs.

3 Background

A neural network is called *recurrent* if it contains connections allowing output signals to enter again as input signals. They are in some sense more general than feedforward networks: a feedforward network can be seen as a special case of a recurrent network where the weights of all non-forward connections are set to zero.

One of the most important applications for recurrent networks is *associative memory*, storing information as dynamically stable configurations. Given a noisy or partial pattern, the network can recall the original version it has been trained on (*auto-associative memory*) or another learned pattern (*hetero-associative memory*).

The most well-known recurrent network is the *Hopfield network*, which is a fully connected auto-associative network with two-state neurons and asynchronous updating and a *Hebbian learning rule*. One reason that the Hopfield network has been so well studied is that it is possible to analyse it using methods from statistical mechanics, enabling exact calculation of its storage capacity, convergence properties and stability. This exercise will deal mainly with this network and some variants of it.

4 Setting up the Matlab environment

There are a few utility files in the course directory which you should first copy into a fresh directory for this lab. Create a directory and copy the necessary files from the course catalogue.

```
> cd ~/ann06
> cp -r /info/ann06/labbar/lab2 .
> cd lab2
```

Then start Matlab

```
> matlab
```

5 Hebbian learning and Hopfield memory

The basic idea suggested by Donald Hebb is briefly: Assume that we have a set of neurons which are connected to each other through synapses. When the cells are stimulated with some pattern correlated activity causes synapses between them to grow, strengthening their connections.

This will make it easier for neurons that have in the past been associated to activate each other. If the network is trained with a pattern, then a partial pattern of activation that fits the learned pattern will stimulate the remaining neurons of the pattern to become active, completing it (see figure 1).

If two neurons are anti-correlated the synapses between them are weakened or become inhibitory. This way a pattern of activity precludes other learned patterns of activity. This makes the network noise resistant, since a neuron not belonging to the current pattern of activation will be inhibited by the active neurons.

This form of learning is called Hebbian learning, and is one of the most used non-supervised forms of learning in neural networks.

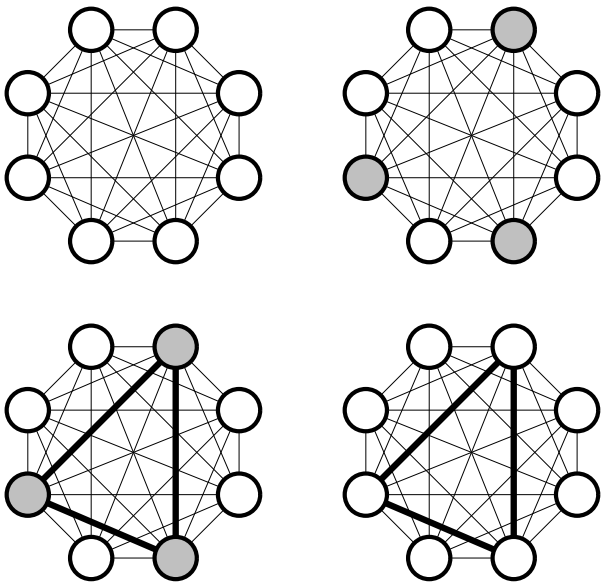

Figure 1: A simple Hebbian fully connected auto-associative network. When three of the units are activated by an outside stimulus their mutual connections are strengthened. The next time some of them are activated they will activate each other.

We may perform this somewhat more mathematically in matrix notation. We index the units by i and call the desired activations of our training patterns for $\bar{x}^\mu$ (a vector with components x_i^μ, where μ is the number of the pattern). We could use for instance 0 and 1 for the activities but the calculations become easier if we choose -1 and 1.

To measure the correlated activities we can use the outer product $W = \bar{x}^T \bar{x}$ of the activity vectors we intend to learn; if the components x_i and x_j are correlated w_{ij} will be positive, if they are anticorrelated w_{ij} will be negative. Note that W is a symmetric matrix; each pair of units will be connected to each other with the same strength.

The coefficients for the weight matrix can then be written as:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} x_i^\mu x_j^\mu$$

where μ is index within a set of patterns, P is the number of patterns, and N is the number of units.

To recall a pattern of activation $\bar{x}$ in this network we can use the following update rule:

$$x_i \leftarrow \text{sign}\left(\sum_j w_{ij} x_j \right)$$

where

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases}$$

The function is undefined for zero; the built-in Matlab function `sign(x)` returns zero for $x = 0$, while the utility function `sgn(x)` returns 1. It does not really matter which function you use, but in order to avoid the appearance of confusing neural states between -1 and 1 sgn is more practical in this lab.

The above dynamics and learning rule form a Hopfield network.

Actually, this variant of the Hopfield network, where all states are updated synchronously, is known as the *Little model*. In the original Hopfield model the states are updated one at a time, allowing each to be influenced by other states that might have changed sign in the previous steps. It can be seen as an asynchronous parallel system. This has some effects for the convergence properties (see section 5.3), but is otherwise very similar in behavior to the synchronous model. The Little model is very easy to implement in Matlab, so we will use it for most of the lab.

In Matlab these operations can be implemented as vector matrix manipulations. To make the pattern vectors as easy as possible to read and write we define them as *row* vectors.

- Translate the calculation of the weight matrix and the update rule into Matlab expressions.

We prefer to make the calculations within the interval $[-1, 1]$ ("bipolar") as this makes the calculations simpler. It is, however, easier to type in and to visually recognize values in the range $[0, 1]$ ("binary"). Therefore, it may be better to use this for input and output. You can use the utility function `vm(x)` to translate a vector from the binary format into the bipolar and `t0(x)` to go from bipolar to binary.

Enter three test patterns

```
>> x1=vm([0 0 1 0 1 0 0 1])
>> x2=vm([0 0 0 0 0 1 0 0])
>> x3=vm([0 1 1 0 0 1 0 1])
```

The next step is to calculate a weight matrix. Use the expression you constructed above. In fact, you don't need to scale the weight matrix with the $\frac{1}{N}$ factor (the scaling becomes important only if there is a bias term, a transfer function with a more complicated shape or if we look at the energy as in section 5.3).

If the network has succeeded in storing the patterns `x1`, `x2` and `x3`, they should all be *fixpoints* when you apply the update rule. This simply means that when you apply the update rule to these patterns you should get the same pattern back.

- Check if the network was able to store all three patterns.

5.1 Convergence and attractors

Can the memory recall the stored patterns from distorted inputs patterns? Define a few new patters which are distorted versions of the original ones:

```
>> x1d=vm([1 0 1 0 1 0 0 1])
>> x2d=vm([1 1 0 0 0 1 0 0])
>> x3d=vm([1 1 1 0 1 1 0 1])
```

`x1d` has a one bit error, `x2d` and `x3d` have two bit errors.

- Apply the update rule repeatedly until you reach a stable fixpoint. Did all the patterns converge towards stored patterns?

You'll probably find that `x1`, `x2` and `x3` are *attractors* in this network.

- How many attractors are there in this network? Hint: automate the searching.
- What happens when you make the starting pattern even more dissimilar to the stored ones (e.g. more than half is wrong)?

The number of iterations needed to reach an attractor scales roughly as $\log(N)$ with the network size, which means there are few steps for a network this small. If you want you can train a larger network to get slower convergence.

5.2 Sequential Update

So far we have only used a very small 8-neuron network. Now we will switch to a 1024-neuron network and picture patterns. Load the file *pict.m*, which contains nine patterns named `p1`, `p2`, `p3`, `p4`, `p5`, `p6`, `p7`, `p8` and `p9`, and learn the first three.

```
>> pict
>> w = p1'*p1 + p2'*p2 + p3'*p3;
```

Since large patterns are hard to read as rows of numbers, we have prepared a function `vis(x)` which displays a 1024 unit pattern as a 32×32 image.

```
>> vis(p1);
```

(one could also use `imagesc(reshape(p1,32,32))`).

- Can the network complete a degraded pattern? Try the pattern `p11`, which is a degraded version of `p1`, or `p22` which is a mixture of `p2` and `p3`.
- Clearly convergence is practically instantaneous. What happens if we select units randomly, calculate their new state and then repeat the process (the original sequential Hopfield dynamics)? Write a matlab script that does this, showing the image every hundredth iteration or so.

why does p22 evolve to p3 rather than p2?

The partial pattern we use is more correlated with p3. The result may change if we shift the right half pattern a little bit.

5.3 Energy

Can we be sure that the network converges, or will it cycle between different states forever?

For networks with a *symmetric connection matrix* it is possible to define an *energy function* or *Lyapunov function*, a finite-valued function of the state that always decreases as the states change. Since it has to have a minimum at

least somewhere the dynamics must end up in an attractor[1]. A simple energy function with this property is:

$$E = -\sum_i \sum_j w_{ij} x_i x_j$$

- How do you express this calculation in Matlab? (Note: you do not need to use any loops!)
- What is the energy at the different attractors?
- What is the energy at the points of the distorted patterns?
- Follow how the energy changes from iteration to iteration when you use the sequential update rule to approach an attractor.
- Generate a weight matrix by setting the weights to normally distributed random numbers, and try iterating an arbitrary starting state. What happens?
- Make the weight matrix symmetric (e.g. by setting `w=0.5*(w+w')`). What happens now? Why?

5.4 Distortion Resistance

How resistant are the patterns to noise or distortion? You can use the `flip(x,n)` function, which flips `n` units randomly. The first argument is a row vector, the second the number of flips.

```
>> p1dist = flip(p1,5);
>> vis(p1dist);
```

(You may have to do this several times to get an impression of what can happen)

Write a matlab script to train a network with `p1, p2, p3`, add noise to a pattern, iterate it a number of times and check whether it has been successfully restored. Let the script run across 0 to 100% noise and plot the result. For speed, use the Little model rather than asynchronous updates.

- How much noise can be removed?

5.5 Capacity

Now add more and more memories to the network to see where the limit is. Start by adding `p4` into the weight matrix and check if moderately distorted patters can still be recognized. Then continue by adding others such as `p5`, `p6` and `p7` in some order and checking the performance after each addition.

- How many patterns could safely be stored? Was the drop in performance gradual or abrupt?

[1] In the Little model it can actually end up alternating between two states with the same energy; in the Hopfield model with asynchronous updates the attractor will always be a single state.

Why the network trained with random patterns has better capacity than that trained with p1-p3?

The random patterns are more dissimilar to each other, which makes the attractors far away from each other. Then, the probability of falling into spurious states will be lowered.

- Try to repeat this with learning a few random patterns instead of the pictures and see if you can store more. You can use `sgn(randn(1,1024))` to easily generate the patterns.
- It has been shown that the capacity of a Hopfield network is around $0.138N$. How do you explain the difference between random patterns and the pictures?

Create 300 random patterns (`sign(randn(300,100))` is a quick way) and train a 100 unit (or larger) network with them. After each new pattern has been added to the weight matrix, calculate how many of the earlier patterns remain stable (a single iteration does not cause them to change) and plot it.

- What happens with the number of stable patterns as more are learned?
- What happens if convergence to the pattern from a noisy version (a few flipped units) is used? What does the different behavior for large number of patterns mean?

The self-connections w_{ii} are always positive and quite strong; they always support units to remain at their current state. If you remove them (a simple trick is to use `w=w-diag(diag(w))`) and compare the curves from pure and noisy patterns for large number of patterns you will see that the difference goes away. In general it is a good idea to remove the self-connections, although it looks like it makes performance worse: actually, they promote the formation of spurious patterns and make noise removal worse.

- What is the maximum number of retrievable patterns for this network?

Why adding bias causes reduction in capacity?

When bias is introduced into the patterns, there will be more +1 and the similarity between different patterns will be increased. Thus, the capacity reduces.

- What happens if you bias the patterns, e.g. use `sign(0.5+randn(300,100))` or something similar to make them contain more +1? How does this relate to the capacity results of the picture patterns?

5.6 Sparse Patterns

The reduction in capacity because of bias is troublesome, since real data usually isn't evenly balanced.

Here we will use binary (0,1) patterns, since they are easier to use than bipolar (± 1) patterns in this case and it makes sense to view the "ground state" as zero and differing neurons as "active". If the average activity $\rho = (1/NP)\sum_\mu \sum_i x_i^\mu$ is known, the learning rule can be adjusted to deal with this imbalance:

$$w_{ij} = \sum_{\mu=1}^{P} (x_i^\mu - \rho)(x_j^\mu - \rho)$$

This produces weights that are still on average zero. When updating, we use the slightly updated rule

$$x_i \leftarrow 0.5 + 0.5 * \text{sign}(\sum_j w_{ij} x_j - \theta)$$

where θ is a bias term.

- Try generating sparse patterns with just 10% activity and see how many can be stored for different values of θ (use a script to check different values of the bias).
- What about even sparser patterns ($\rho = 0.05$ or 0.01)?

There exists a positive causal link between rho and theta. Once rho is determined, the optimal theta could be calculated out theoretically.

Good luck!