Machine Learning CMPU 395

Assignment 5: Hopfield Networks

1 Tasks

Your task is to use a set of utility functions performing the basic operations of a Hopfield network to study the networks ability to store and recall patterns. You will use three datasets: one small set of short patterns, one set of figurative patterns (binary images), and collections of random patterns.

For the highest grade you will have to measure how the performance degrades when the network is overloaded by storing too many patterns.

2 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 analyze it using methods from statistical mechanics, enabling exact calculation of its storage capacity, convergence properties and stability.

2.1 Hebbian Learning

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, simultaneous activity causes synapses between them to grow, strengthening their connections.

This will make it more likely for neurons that are often simultaneously active to support each other. 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 seldom or never simultaneously active, the synapses between them will become inhibitory. This makes the network noise resistant,

since a neuron not belonging to the currently active pattern 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.

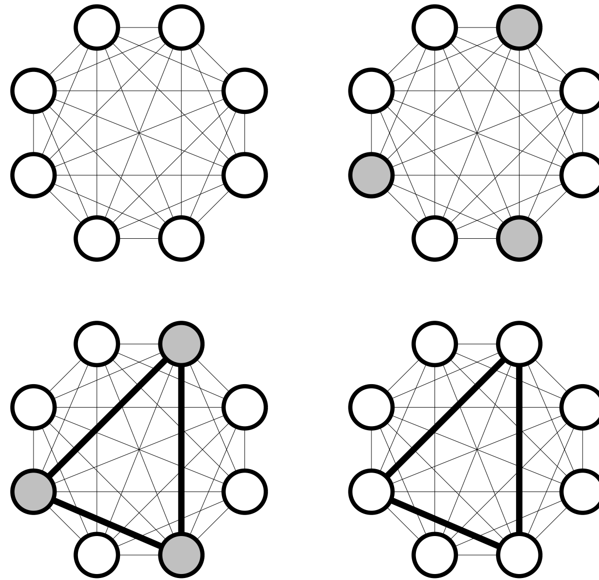

Figure 1: A simple Hebbian fully connected auto-associative network. When three of the units are activated by an outside stimulus their mutual connections are strengthened. The next time some of them are activated they will activate each other.

Mathematically, Hebbian learning can be expressed as the *outer product* of a pattern vector with itself. Let each pattern to be stored be a vector $\vec{x}$ where each element, x_i, is either -1 (meaning inactive) or 1 (meaning active). Since we need to store several patterns we may use an upper index μ to indicate which pattern we are referring to. Thus, we have a set of patterns $\vec{x}^\mu$, where $\mu = 1, 2, \ldots, P$.

To measure the correlated activities we use the outer product $\Delta W = \vec{x}\vec{x}^T$ of the activity vectors we intend to learn; if the components x_i and x_j are correlated w_{ij} will become positive, if they are anti-correlated w_{ij} will become 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 be written as:

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

where μ is index within a set of patterns, P is the number of patterns. Alternatively, we may use vector notation for the same calculation:

$$W = \sum_{\mu=1}^{P} \vec{x}^{\mu} (\vec{x}^{\mu})^T$$

2.2 Memory Recall

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

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

where

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

Actually, the 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 2.3), but is otherwise very similar in behavior to the synchronous model. The Little model is easier to implement using `numpy`, so we will use it for most of this assignment.

2.3 Energy

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:

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

This can be written in matrix form as

$$E = \vec{x}^T W \vec{x}$$

[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.

3 Utility Functions

Most of the operations of the Hopfield network can be seen as vector-matrix operations which are straightforward to express using functions from `numpy`. Note that all the functions listed here are available in the file `utils.py`.

3.1 Test Patterns

3.1.1 Small Patterns

We will need a few small test patterns as `numpy` arrays containing -1 and 1 as elements:

```
    x1 = numpy.array([-1, -1,  1, -1,  1, -1, -1,  1])
    x2 = numpy.array([-1, -1, -1, -1, -1,  1, -1, -1])
    x3 = numpy.array([-1,  1,  1, -1, -1,  1, -1,  1])
```

3.1.2 Figurative Patterns

The file `figs.py` defines nine patterns (`p1`, `p2`, ..., `p9`). Each pattern is 1024 elements long, but is actually a 32×32 binary image. You can look at these patterns by reshaping them into a 32 by 32 matrix:

```
pylab.matshow(figs.p1.reshape((32, 32)))
```

The file also defines two distorted patterns: `p11` and `p22`. In the first, half of pattern `p1` is replaced with random bits. The second is a mixture of two patterns (`p2` and `p3`).

3.1.3 Random Patterns

You will need some random patterns. The following function can be used for generating a random pattern of length n:

```
def rndPattern(n):
    "Create a random pattern of length n"
    return numpy.sign( numpy.random.randn(n) )
```

We use `numpy.sign` which gives -1 for all negative elements and 1 for all positive.

3.2 Learning

Learning is done using Hebbs rule. We use the function `numpy.outer` which calculates the outer product of two vectors. We also remove the self connections, i.e. we set all the diagonal elements to zero. This is done using `numpy.diag`

```
def learn(patterns):
    "Find the weight matrix for a list of patterns"
    n = len(patterns[0])
    w = numpy.zeros((n, n))
    for p in patterns:
        w += numpy.outer(p, p)
    return w - numpy.diag(numpy.diag(w))
```

3.3 Updating

The function for updating an activity pattern makes use of the `numpy` functions `dot` and `sign` for the matrix-vector multiplication, and the thresholding, respectively.

```
def update(w, x):
    "Apply the parallel Hopfield update rule"
    return numpy.sign(numpy.dot(w, x))
```

You may not want to update all elements in parallel. The following function selects a random element and updates only that. Note that the array passed as parameter x is updated *in place*!

```
def updateOne(w, x):
    "Update one element in x"
    i = numpy.random.randint(len(x))
    x[i] = numpy.sign(numpy.dot(w[i,:], x))
```

3.4 Energy

Use the following function to get the energy for an activity pattern:

```
def energy(w, x):
    "Return the energy of state x, using weight matrix w"
    return -numpy.dot(x, numpy.dot(w, x))
```

3.5 Comparing Patterns

You may have use for a function which compares two patterns.

```
def samePattern(x, y):
    "Check if patterns x and y are equal"
    return numpy.all(x == y)
```

3.6 Disturbing Patterns

You will have use for a function which randomly disturbs a pattern. The following function takes a pattern x and a number n and returns a new pattern where n randomly chosen elements have been "flipped" (from -1 to 1 or *vice versa*). The number n must not be larger than the length of x.

```
def flipper(x, n):
    "Flip the values of n elements in pattern x"
    flip = numpy.array([-1]*n + [1]*(len(x)-n))
    numpy.random.shuffle(flip)
    return x * flip
```

4 Experiments

4.1 Small Patterns

- Store the three patterns `x1`, `x2`, and `x3` in a weight matrix and test if they are all *fixpoints*. This simply means that when you apply the update rule to these patterns you should get the same pattern back.
- Test if you the network will recall the stored patterns if you start with distorted versions. Use the `flipper` function to create distorted versions of the stored patterns.
- Do you have any spurious attractors? How many?

  Since this network is rather small you can actually test all possible patterns starting patterns. Alternatively, you can start at random patterns a large number of times and keep track of where you end up.

You will note that very few iterations are necessary for convergence. This is because this network is very small. The number of iterations needed to reach an attractor scales roughly as $\log(N)$ with the network size.

4.2 Restoring Images

Now we will switch to a 1024-neuron network and picture patterns. Load the file *figs.py*, which contains nine patterns named `p1`, `p2`, ..., `p9`, but only store the first three in a weight matrix.

- 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 Python program using the `updateOne` function, showing the image every hundred'th iteration or so.

- Create distorted starting patterns by flipping elements in one of these patterns; iterate a number of times and check whether they are successfully restored.

  How much of the pattern can be destroyed before restoration fails?

- 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.

4.3 Random Connectivity

- Generate a weight matrix by setting the weights to random numbers (use the normal distribution). What happens when you iteratively update from an arbitrary starting point?

- Make the weight matrix symmetric
  (e.g. by setting w=0.5*(w+w.transpose())). What happens now?

4.4 Capacity

Add more and more memories to the network to see where the limit is. Start by including `p4` into the set if training patterns and check if moderately distorted patters can still be recognized. Then continue by including even more (`p5`, `p6`, ...) and checking the performance after each addition.

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

- Repeat this with random patterns instead of the pictures and see if you can store more.

- It has been theoretically shown that the capacity of a Hopfield network is around $0.138N$, but this is only true for random patterns. What was the corresponding figure when you used non-random patterns (pictures).

4.5 Quantitative Measurements

[***This part of the assignment is optional, but required to get the highest grade.***]

Use random patterns of length 100 (or larger) to generate a graph of how many patterns can be successfully retrieved as a function of how many patterns are stored.

To avoid unnecessary random fluctuations, generate a sufficiently large set of random patterns and then use subsets of these patterns throughout the experiment. Construct a weight matrix from 1, 2, ..., etc. of these patterns and

in every case check how many of the stored patterns are actually stable. 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).

4.6 Sparse Patterns

[***This part of the assignment is optional, but required to get the highest grade.***]

- What happens if you bias the patterns to make them contain more -1's than 1's?

  You can test this by defining a modified `rndPattern` which subtracts a small number (e.g. 0.5) from the random numbers before calling `numpy.sign`.

The reduction in capacity because of bias is troublesome, since real data usually isn't evenly balanced. Can this possibly explain the bad capacity results for the picture patterns?

For the final experiment we will use binary $(0, 1)$ patterns, since it makes sense to view the "ground state" as zero and differing neurons as "active". If the average activity $\rho = \frac{1}{NP}\sum_{\mu}^{P}\sum_{i}^{N} x_i^{\mu}$ 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

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

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).

Good luck!