Neural Computation

Problem Set 4: Association Memory Networks

Credit: 20 points.

Assigned: July 22 Due July 24 8:00 a.m, hard copy. 11:00 p.m. soft copy. 2019.

Please submit hard copy in class if you want to be read carefully by me. The TA will read the submission submitted online with deadline 10:00 p.m.

The objective of this assignment is to familiarize you with the basic mechanics of Hopfield network, to understand how it encodes memory, its dynamics, and capacity. We will experiment with the codes provided in the `ps4_codes` zip file to explore the outer-product rule as well as the Oja rule for learning associative memory. Given the limited amount of time you have, you will be given an implemented network, your job first is to run it and do some experiments with it to see how it work. You should provide a write-up of two pages reporting your observations of studying this implementation, comparing outer-product rule, and Oja rule in two methods of learning.

Background

Notation:

A Hopfield network is defined by the following:

- a set of visible units $V = \{v_1, ..., v_n\}$. $v_k \in \{-1, 1\}$
- a weight matrix W where each element $w_{i,j}$ is the connection between v_i and v_j. $w_{i,j} \in \mathbb{R}$
- the weights are symmetric, i.e $w_{i,j} = w_{j,i}$.
- the **energy** of a visible configuration V is defined as $E = -\sum_{i \neq j} v_i v_j w_{i,j}$

1. Encoding memory using Outer Product Rule

Outer Product Rule. In class we discussed the outer product rule for encoding memories in the weights of a Hopfield network. In `lmy_testm` , the outer product rule is implemented in the two lines of codes under the "hebbian" learning rule by the following one line of codes:

```
T = T + data'*data;
```

Oja Rule. Oja rule is implemented in under the "oja" learning rule in `lmy_testm` as well. There is also another rule we implemented called "storkey" rule that you don't need to worry about.

2. Dynamics of Hopfield network.

The dynamic of Hopfield net is implemented by `runHopnet.m`. Running a Hopfield network consists of searching for a minimum energy configuration. In class we described the following algorithm:

while the network is not at an energy minimum **do**
  $r \leftarrow random\_int(1, n)$
  $v_r \leftarrow sign(\sum_{i \neq r} v_j w_{j,r})$
end while

which selects a random unit and then computes it's state given all the other units. This process is repeated until the energy of the system hits a minimum.

Part 1: Why the update rule is doing gradient descent? (5 pt)

This update algorithm has the nice property that every time we update a unit's state with the rule above the energy is guaranteed to be the same or decrease (or get more negative). In this question you will prove this property. The proof is very simple (so don't worry if you are not familiar with formal proofs, intuitive explanations using the positivity and negativity of the terms is fine).

We will start the proof out and you must complete it.

First let's say that before we update a unit the energy of the current configuration is E^{old}. Let us denote the unit that will be update by v_r which will have value v_r^{old}. Thus:

$$E^{old} = -\sum_{i \neq j} v_i v_j w_{i,j} \tag{1}$$

$$= -\sum_{i \neq j \neq r} v_i v_j w_{i,j} - \sum_{i \neq r} v_i v_r^{old} w_{i,r} - \sum_{j \neq r} v_r^{old} v_j w_{r,j} \tag{2}$$

$$= -\sum_{i \neq j \neq r} v_i v_j w_{i,j} - 2\sum_{i \neq r} v_i v_r^{old} w_{i,r} \tag{3}$$

The last step is correct because the weights are symmetric. When we update v_r by the rule given by the algorithm the new v_r, v_r^{new}, will be:

$$v_r^{new} = sign(\sum_{i \neq r} v_i w_{i,r}) \tag{4}$$

The energy of the new configuration will then be:

$$E^{new} = -\sum_{i \neq j} v_i v_j w_{i,j} \tag{5}$$

$$= -\sum_{i \neq j \neq r} v_i v_j w_{i,j} - \sum_{i \neq r} v_i v_r^{new} w_{i,r} - \sum_{j \neq r} v_r^{new} v_j w_{r,j} \tag{6}$$

$$= -\sum_{i \neq j \neq r} v_i v_j w_{i,j} - 2\sum_{i \neq r} v_i v_r^{new} w_{i,r} \tag{7}$$

Thus the change in energy due to the update will be:

$$\Delta E = E^{new} - E^{old} \tag{8}$$

$$= \left(-\sum_{i \neq j \neq r} v_i v_j w_{i,j} - 2\sum_{i \neq r} v_i v_r^{new} w_{i,r} \right) - \left(-\sum_{i \neq j \neq r} v_i v_j w_{i,j} - 2\sum_{i \neq r} v_i v_r^{old} w_{i,r} \right) \tag{9}$$

$$= -2\sum_{i \neq r} v_i v_r^{new} w_{i,r} + 2\sum_{i \neq r} v_i v_r^{old} w_{i,r} \tag{10}$$

$$= -2v_r^{new} \sum_{i \neq r} v_i w_{i,r} + 2v_r^{old} \sum_{i \neq r} v_i w_{i,r} \tag{11}$$

$$= 2(-v_r^{new} + v_r^{old}) \sum_{i \neq r} v_i w_{i,r} \tag{12}$$

$$\tag{13}$$

Now we must show that our update of v_r will always make the following hold: $\Delta E \leq 0$

There are two cases. The first case is when $v_r^{new} = v_r^{old}$. In this case the configuration does not change and thus the energy does not change. The second case is when $v_r^{new} = (-1)v_r^{old} = sign(\sum_{i \neq r} v_i w_{i,r})$. In this case we have the following change in energy:

$$\Delta E = 2(-v_r^{new} + v_r^{old}) \sum_{i \neq r} v_i w_{i,r} \tag{14}$$

$$= 2(-sign(\sum_{i \neq r} v_i w_{i,r}) + (-1)sign(\sum_{i \neq r} v_i w_{i,r})) \sum_{i \neq r} v_i w_{i,r} \tag{15}$$

$$\tag{16}$$

Complete this proof by showing that with this second case the sought after property still holds, i.e. $\Delta E \leq 0$. We understand that not everyone in this course is familiar with proofs. Therefore it is ok to explain in words why the property will hold true.

After completing the proof answer the following question: If we ran this algorithm so that it went through the while loop an infinite number of times, will the energy of the resulting configuration have lower energy than any other possible configurations? That is, does the network always settle down to the global minimum?

Once we learned some memory into the network using Oja rule or Hebbian (outerproduct rule), we can test memory retrieval. with the following different types of input initialization to the network:

(1) *full*: use the original stored image encoded in the memory as the initial configuration to see if whether you can actually retrieve it.

(2) *corrupted:* : use a noise-corrupted version of the memory image. Here, we can just add some number of dots to the stored image and use that as initial configurations.

(3) *partial image*: use a part of the stored image as input

In `my_test.m` provided, we provided three switches "corrupt", "partial", "full", corresponding to the above three scenarios.

my_test.m to understand the various parameters of the routine.

To run the program, you should run the "mytest" function, with variables specified as follows.

```
mytest(learning_rule, numPatterns, size, custom_load, updatemod,
updatepara, inputmod, inputpara)
```

```
mytest('oja', 2, 50, [``images/bike.jpg'', ``images/face.jpg''],
         'all', [10, 2], 'corrupt', [1, 20, 10,10])
```

Note the image, you need to provide the full path, or you should addpath(genpath(ps4 codes folder)) your entire folder, so that those images are in Matlab's path.

The first variable learning_rule have three options: 'hebbian', or 'oja', or 'storkey'. Hebbian implements the outer-product rule; Oja is the rule you need to experiment now.

The second variable numPatterns is the number of patterns to be remembered.

The third variable size is the size of the input image pattern, 50 means the image size is 50 x 50.

The fourth variable indicates the images to be learned, when it is set to [1], then it will just use the default sequences of images in the images folder, which will be in alphebatical order. In this example, we are selecting bike.jpg and face.jpg to be the two images to remember.

The fifth variable selects the update method or mode: use 'all' which means update all the nodes in parallel, I have deleted the options to avoid confusions.

The sixth variable specifies the number of iterations to be run. It has two parameters [x, y]

where x is the number of iterations to do, and y is the number of iterations per checkpoint where an intermediate result is printed out. Note that parallel update might needs a few iterations. when you use ”all” or ”checkerboard”, [x, y] should be small, e.g. [4, 2], [10, 4].

The seventh variable `inputmod` specifies the type of preprocessing the input to be tested, ‘corrupt’, ‘partial’, or ‘full’.

The eighth variable specifies the parameters for each of the processing input mode. [x,y,z,w] where x is image index to be tested as input. In this example, we choose the 1st image to apply the Hopfield dynamics. The parameters means different things for the different input modes. For the ‘full’ mode, only x is required.
For the ‘partial’ mode, we will specify a patch that is visible, where y indicates the size of the patch, (z,w) specifies the upper left corner coordinate of the visible patch.
For the ’corrupt’ mode, y indicates the number of points of noises to be added to the image to corrupt the images.

Your task is look into the codes to understand the learning paradigms or algoirthms used in the Oja rule implementation, and be able to interpret the various figures generated and what they mean.

Part 2: How many patterns can be stored and retrieved? (10 points)

Compare the results of Hopfield Net learned with outer-product rule (hebbian) and that with Oja rule. By results, we mean the capacity of the network, the ability of the network to retrieve the image under full, noisy and partial scenarios. What do you find? Try to explain what could be the reasons underlying the difference in performance. Make at least three interesting observations and conjectures on what could be happening.

Note that the `my_test.m` train all the patterns you specified, but allow you test the retrieval of patters one a time, even though all the patterns can be learned. You are welcome to write a script or modify `my_test.m` so that you can test multiple images after training them,

rather than running the whole training process all over again every time. That could be tedious particularly when there are 20 objects. If you are not familiar with Matlab to make your algorithm more efficient, it is ok to choose maybe only 10 objects to train, and evaluate based on those 10 objects.

Part 3: What is the algorithm implemented and why it helps? (5 points)

Describe the current learning paradigms used in the Oja rule implementation that appears to yield superior result among all the paradigms we tried. The default paradigm is to train all the patterns together in random interleave. That is, randomly choose a pattern, train it once or multiple times, and then choose another patterns, and do the same, and keep repeating until the network converge. Why do you think the current method training paradigm work (provided to you) work better than our default paradigm. Do you think it is possible to extend this idea to the outer–product rule?