15-386 Neural Computation

Problem Set 3: Source separation and Associative Memories

Credit: 12 points.
Out: 2/22, 2023.
Due: 11:59 p.m. 3/15, 2023.

Submission guidelines: Each student needs to submit a type-set write-up to both the gradescope Homework3 folder and the CANVAS folder. Apart from submitting each of the answer to gradescope. You should also create a zip file containing your report, the codes and generated results (graphs) into a zip file and submit to the assignment submission folder in CANVAS. This should include the top level code for each part as part1.m, part2.m along with all supporting scripts, functions, and .mat files. There should be a sub-script (e.g. part1a.m part1b.m, ..) for answer to each part to facilitate grading. Running the top level code should generate all the results and figures included in your report. Note that you do not need to include every graph you have generated in the report. Include only those we explicitly asked or the pertinent ones to illustrate your observations. **Note**: If a question asks you to make plots, please include them in your write-up and select the corresponding pages in gradescope to shorten misgrade - regrade cycle.

Even though the homework seems to be long, the amount of codes you need to write is minimal. We provide much of the wrapper codes for image data manipulation as well as example codes to make it easier. There are two starter code zip files, one for ICA (part 1) and one for Hopfield net (part 2). Your tasks are to study the programs, run them, interpret the results and sometimes write a line or two of codes (implementing an equation) to show your understanding. Part 3 you can perform creative exploration on ICA or Hopfield net or both.

Part 1: Independent component analysis : blind source separation (5 points)

The problem ICA solves is called blind source separation. Imagine you are in a cocktail party, what you hear is a combination of sound sources, many people talking, piano music playing in the background. ICA helps you to discover and separate the different sources of sound, which would allow you to pay attention to one source at a time, listening to what your friend is saying. Although ICA can be implemented using neural circuit model, as in Foldiak's work and Olshausen and Field's works. In this assignment, we will use an optimized fast ICA package for you to understand what it can do.

Mathematically, ICA assumes the following linear generative model of source mixing:

$$X = AS$$

where X is a matrix of data vectors of dimension $N \times M$, called data or mixture matrix, with each of the N rows indicating a particular observation location (a particular microphone or ear), with each of the M columns representing a sample (the sound signals received by all the microphones simultaneously). Hence, it contains N mixture signals, A is called the mixing matrix of dimension $N \times N$, with each column serves as a basis vector. S is a $N \times M$ source matrix with each of the N rows being a particular source, and each column representing a particular value generated by that source in a particular sample (e.g. time instance).

For N source signals (e.g. temporal signals), N observed mixtures, each with M samples, the ICA model is,

$$\begin{bmatrix} \vec{x_1}(t) \\ \vdots \\ \vec{x_N}(t) \end{bmatrix} = \begin{bmatrix} a_{1,1} & \dots & a_{1,N} \\ \vdots & \dots & \vdots \\ a_{N,1} & \dots & a_{N,N} \end{bmatrix} \begin{bmatrix} \vec{s_1}(t) \\ \vdots \\ \vec{s_N}(t) \end{bmatrix}$$

$$\mathbf{X} = \mathbf{A} \qquad \mathbf{S}$$

Each time point t is going to generate one column in X and S respectively, and there are M such columns or time points.

The first objective of ICA is to adapt the column basis vectors in A so that it can generate the observed X based on S. This is analogous to the process of learning. The second objective of ICA is to interpret the data (i.e. identifying the sources). This is analogous to perception or reasoning. ICA also gives you the filter matrix W for computing or inferring the source signals S given the observation X. Hence, $X = AS$ can be considered a synthesis process, while $S = WX$ is an analysis process.

Hidden from these equations but explicit in ICA formulation and implemented in ICA program is that prior that S has to be independent or sparse. To achieve sparsity, the average code length of $< |s| >$ is to be minimized. A popular assumption is that the distribution of the coefficients s follows a Laplacian distribution, $p(s) = e^{-\lambda|s|}$.

In this homework, you can use `fastica.m` in the start code to perform ICA, but you are welcome to implement your version of ICA instead of calling the function. In the following context, we will assume ICA algorithm is implemented in FastICA.

(1a) **Discovering hidden images.** (1 points)

ICA can be used to perform blind source separation, i.e. solving the so-called cock-tail party problem. In the starter code data set, we have 6 mystery mixture images stored in `mystery_images.mat`. `mystery_images{i}` will give you the `i`-th image.

These mixture images are created by weighting six images with 6 sets of weights. For example, the first two images are created by:

$$I_{mix1} = a_{1,1}I_1 + a_{1,2}I_2 + a_{1,3}I_3 + a_{1,4}I_4 + a_{1,5}I_5 + a_{1,6}I_6$$

$$I_{mix2} = a_{2,1}I_1 + a_{2,2}I_2 + a_{2,3}I_3 + a_{2,4}I_4 + a_{2,5}I_5 + a_{2,6}I_6$$

where I_1 is a 2D source image, and I_{mix1} is the 2D mixture images. I_{mix1} is an image with t

indexing every pixel would be the same as the data $\vec{x_1}(t)$ in the above matrix equation. I_1 would be the source $\vec{s_1}(t)$ above.

You can see these images using `imagesc()`.

Your job is to recover the source images using ICA. Run ICA three times. In your report, include the three sets of images that are recovered. Are the results from ICA unique (the same every time)? If not, explain why.

Hint:

These mixture images were generated by multiplying a 6 x 6 mixing matrix (A) and a 6x N source matrix S (or I_i) where each row of S corresponds to an original (source) image reshaped into 1 1D vector. Therefore, your data matrix X (or I_{mix})would be of dimension 6 x N, where N = imgheight x imgwidth.

Each row of X should be an image and you can reshape this 1D vector into a 2-D image. Therefore, N mixtures of images can be used to recover N original images by ICA, after converting images from 2-D to 1-D and applying FastICA to the mixture matrix in the same way as before. Additionally, FASTICA will only work properly with variables of type `double`. Images are usually loaded as type `uint8`. To load a 128x192 image of a cat and make it into a 1-D `double` signal, use: `catsig = reshape( double( imread('cat.png')), 1, 128*192);` You also may need `rgb2gray( )` if loading a color image.

(1b) **Limitations and constraints: Number of Data Channels** (0.5 point)

Next, discard I_{mix6}, and perform ICA to recover five sources based on the remaining five mixtures. Again, repeat three times. What do you observe? Include the results and **explain** your observations in the report.

(1c) **Limitations and constraints: Number of Data Channels** (0.5 point)

Next, construct a new mixture image $I_{mix7} = (1/2)(I_{mix1} + I_{mix2})$, and perform ICA to recover seven sources. Can it work? What do you get? Include the results and **explain** observations in your report.

(1d) **Your own exploration** (1 points) Synthesize mixture images yourself and see if you can separate them correctly. You can choose the number of images you want to mix together, and ask whatever questions you prefer. For example, if you want to combine two images, you can combine them as

$$I_{mix1} = a_1 I_1 + a_2 I_2$$

$$I_{mix2} = b_1 I_1 + b_2 I_2$$

where I_1 and I_2 are the two images. That is, each pixel in mix1 is a weighted sum of two corresponding pixels in image1 and image2 respectively. Image1 and image2 need to have the same

dimensions. Note: You can choose arbitrary a_i and b_i values. And also you can use any images you are interested in to explore!

Include the source images, mixtures, and recovered images in the report. Your goal is to spend an hour to explore and discover the constraints and limitations of ICA, and to understand what images you can mix and separate and what images you can't. What kinds of mixtures are easy to separate and what kinds of mixtures are difficult to separate? How well you can recover the mixing weights a_1, a_2, b_1, b_2?

(1e) **Learning sparse codes of natural image using ICA** (2 points)

We will use ICA to learn sparse codes from natural images. We would like to consider any arbitrary image patch drawn from natural scenes as a mixture of many elementary image components. A 2D image patch can be reshaped into a 1D signal. Therefore, K image patches can be used to recover the basis vectors and the K source signals by ICA, after converting images from 2D to 1D and applying ICA to the data matrix. Your task is to discover these image components (or receptive fields).

The natural scene data are obtained by extracting small image patches from whitened natural scenes, which one can think of as an idealization of the input provided by the retina/LGN. These are stored in the array IMAGES. For instance, if you type

```
imagesc(IMAGES(:,:,1)), axis image
colormap gray
```

you will see one of such whitened image. In the starter code, you should also find a Matlab script for performing whitening that takes an a normal image and output a whitened image by performing a filtering operation in the Fourier domain.

Download FastICA package and other matlab codes from the problem set's starter_code. Use the whitened image set IMAGES.mat in Part 1, sample as many random 12 x 12 patches as your computers RAM allows, and construct the data matrix X of size 144 x N, where here N should be at least 5000, and preferably 100000. Run `fastica.m` using this data matrix to get the mixing matrix A (of size 144 x 144).

In your report, print all the 144 basis vectors in A. You can print them all in one figure or in four figures. Note that each column of A corresponds to a basis vector and can be converted into an image patch by `patchk = reshape(A(:, k), 12, 12).` NOTE: well, if your computer memory can't handle 12 x 12 patches well, you can do the 8 x 8 patches instead. It is just that 12 x 12 patches look better.

The image components are called sparse codes of natural images, and they are supposed to resemble the V1 simple cell receptive fields (see Olshausen and Field 1996).

(1f) **Creative Exploration with ICA** (1 point) (potential up to 1 bonus point) In this bonus question, you are encouraged to explore a problem or question you want to solve or answer about ICA or using ICA. We will award creative novel problems and interesting answers with 1 point, and partial credit for the others. You may use dataset available in the internet, However, if your

question and answers are readily searchable in Google, that would not be interesting, but you might still earn some credit, depending on the effort you put in. The interesting solutions with awards will be released to the class on CANVAS. Exceptionally interesting and novel results would be awarded up to a bonus point.

Part 2: Learning Hopfield Nets (6 pts)

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 $E = -\sum_{i \neq j} v_i v_j w_{i,j}$

Outer Product Rule. In class we discussed the outer product rule for encoding memories in the weights of a Hopfield network. The Outer-Product rule captures the idea of Hebbian learning, but specify the weights for encoding the memory patterns. Here, we will explore the use of alternative Hebbian learning rules to learn a Hopfield Net in an incremental and on-line manner, using starter code in the hopfield2 folder.

1. Hopfield Dynamics (2 point) First, you should revise the program `runHopnet.m` to add in the Hopfield update dynamics.

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)
```

The first variable `learning_rule` have three options: 'hebbian', or 'oja', or 'storkey'. Hebbian implements the outerproduct rule; Oja is the rule you need to implement. Storkey is another learning rule sometimes used to train Hopfield net that we provide the implementation here as an example. Once you implemented the Hopfield dynamics update in `runHopnet.m`, then you should be able to run "mytest" as follows to see the results.

```
mytest('storkey', 2, 50, [''images/bike.jpg'', ''images/face.jpg''],
        'random', [10000, 1000], 'corrupt', [1, 20, 10,10])
```

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: 'random' means update one randomly selected node at a time; 'all' means update all the nodes in parallel, 'checkerboard' means update the nodes in a checkerboard pattern as described above.
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 here is to fill in Hopfield dynamics for the "all" update mode in `runHopnet.m`, and then run `mytest` using the 'storkey' rule as above and observe the results that are generated. You have to look into the code to interpret the various figures generated and what they mean.

For a Hopfield net with memory encoded in the connections, given a certain input configuration, the network will settle into a stable equilibrium state (basin of attraction) corresponding to the memory of the corresponding patterns, if it follows the Hopfield net dynamics:

while the network is not at an energy minimum **do**
  $r \leftarrow$ choose a node in the network
  $v_r \leftarrow sign(\sum_{i \neq r} v_j w_{j,r})$
end while

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

1a. Why is the update rule doing gradient descent? (1 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 stay the same or decrease. 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:

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

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

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

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

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

Now we must show that our update of v_r, following the Hopfield update rule, 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:

$$\Delta E = 2(-v_r^{new} + v_r^{old}) \sum_{i \neq r} v_i w_{i,r} \tag{13}$$

$$= 2(-sign(\sum_{i \neq r} v_i w_{i,r}) + (-1) sign(\sum_{i \neq r} v_i w_{i,r})) \sum_{i \neq r} v_i w_{i,r} \tag{14}$$

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?

1b. Implementing Hopfield dynamics (update) (1 pt)

Implement the Hopfield dynamic update rule in `runHopnet.m`. Note that while there are three versions of Hopfield update dynamics in the code, you should just use 'all', which update all the nodes' states in parallel, and not to worry about 'random' and 'checkerboard' which update the states asynchronously. This is because the parallel update seems to work quite well and is very fast. You can explore the other update modes if you want during creative exploration.

In your report, include some of the generated figures that you think are relevant and comment on them.

2. Implementing Outer-Product rule and Oja's Learning Rule (2 points) Your next task is to fill in the 'hebbian' and 'oja' learning codes in `mytest.m`. Hebbian is implementing the outer-product rule. Oja is implementing the on-line learning rule. How many patterns can each type of network (outer-product versus Oja) learn? Number of epochs is a hyper-parameter you can experiment with.

Note that in the starter code we provided the wrapper code so that you can just fill in a couple lines to get the Outer-product rule and Oja rule working. **Here, we should modify the codes or specify the ones we want to implement more clearly.**

3. Effects of Learning Rates (1 point) Study the effect of learning rates (try at least five rates). How does the learning rate affect the memory encoding and memory retrieval? Choose the iterations and checkpoint parameters properly so that you can monitor the progress of the learning. Include some relevant intermediate and/or result figures in your report to support your claim and observations.

4. Retrieval Performance under Noise and Occlusion (1 point) Choose the best on-line learning strategies, investigate the capacity of the Hopfield network that can be learned when the input during retrieval time is corrupted. Evaluate the capacity of the network when the input image is corrupted with 25% noise during retrieval. Evaluate again the capacity of the network when the

input image is 50% occluded. What do you observe? Include your evaluations and observations supported by key figures in your report. Note, you only need to include the key figures to support your claims (observations) in the report, though you should include all the figures generated in your zip file (submitted to the code assignment). Obviously, the numbers could be different depending on the network you choose to train—we're not looking for a specific "right" number, just a proper investigation.

Part 3: Creative Exploration or Competition (1 point, with 2 possible bonus points). For this part, you can explore either ICA or Hopfield net to answer a question or two that you pose. Interesting and outstanding answers will be rewarded up to 2 bonus points. Limit this part of the report to 4 pages maximum. One legitimate exploration is to develop strategies to maximize the number of patterns that can be encoded in the network. The winner of this competition (maximum number of patterns remembered) will surely be awarded the 2 bonus points.