Comparative Study of Synchronous and Asynchronous Hopfield Networks for Image Denoising

Piyush Mishra and Clémence Fournié

Aix-Marseille University

1 Introduction

Image denoising refers to the task of removing noise from a noisy image, so as to restore the true image.

Hopfield Network is "a form of recurrent artificial neural network and a type of spin glass system" [2]. It was discribed by Little in 1974 and popularised by John Hopfield in 1982. Hopfield network has an "associative" memory system with binary threshold nodes [2]. It can, however, be extended to work for continuous systems as well.

Our experimentation focuses on two different updation procedures of the Hopfield Network: synchronous and asynchronous.

On the one hand, for the synchronous update, the update is for all nodes at the same time. And on the other hand, for the asynchronous update, the update is for one particular node at a time and there is no coordination between them in time [1]. We elaborate the specific differences in the subsequent sections.

The pipeline, hence, is built with (1) the prepartion of the images, (2) Hopfield training and prediction, (3) energy calculation and finally (4) comparison between the synchronous and asynchronous updation criteria.

2 Methodology

2.1 Preparation of Images

The images used for this study are shown in Fig. 1. Different kinds of images are used to try to throw the machine off, as much as possible. For the sake of brevity, the objective is to use binary images. In doing so, we first convert all images to gray images.

A list of these images is created with the function **get_images()**. Each image, as discussed above, is converted to a gray image matrix. During pre-processing, each image is subsequently converted to a binary image. This is carried out by getting an image threshold, i.e. the central pixel value in an image. Any pixel with value greater than this threshold is assigned the maximum value, while any other pixel with value less that it, is assigned the lowest pixel value. Thus, we attain a binary image matrix. This image matrix is then "flattened" to make it into a vector for further computation.

2.2 Hopfield Training and Prediction: How is the information stored?

Training: Creating the Weights

In order for one to understand how the training happens for denoising, one ought to understand how Hopfield networks work in essence. Training, in this case, is how the network learns to recreate the input signal s. This is achieved by creating a set of weights, in the form of a weight matrix W, each element of which, W_{ij} describes the interaction between the i^{th} and the j^{th} nodes of the network.

Considering s to be the input signal (for this study, we take s to be a row matrix), the weight matrix W can be calculated by:

$$W = s^T.s$$

Fig. 1: Images used for training (clockwise from top-left, Camera, Astronaut, Chelsea and Horse).

Where s^T is the transpose of the s vector.
This is also known as the Hebb's rule of learning [8]. However, in a Hopfield network, there is no interaction of a node with itself, so in essence, the learning rule can be given by,

$$W_{ij} = \begin{cases} s_i^T s_j & i \neq j \\ 0 & i = j \end{cases}$$

In case of multiple images (or signals, for simplicity), the weight matrix is nothing but the summation of the dot products of the transpose of each signal with the signal itself.

$$W = \sum_{i=1}^{n} s_i^T . s_i$$

Once a weight matrix is created and finalised, the training is concluded. We will see in the subsequent sections how the information is converged and how a Hopfield network tackles the issue of noise.

We input our images in the program for training.

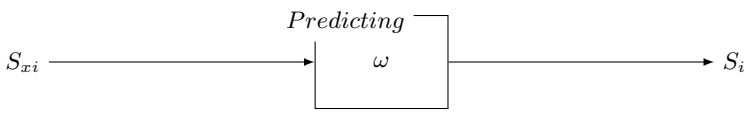
```
def train(self, data):
    print("Training:")
    self.neurons = data[0].shape[0]
    n = len(data)

    # initialising the weights for the Hebb's rule
    W = np.zeros((self.neurons, self.neurons))
    rho = np.sum([np.sum(t) for t in data]) / (n * self.neurons)

    for i in tqdm(range(n)):
      t = data[i] - rho
      W += np.outer(t, t)

    diagW = np.diag(np.diag(W))
    W = W - diagW
    W = W / n

    self.W = W
    return self.W
```

Note: the module for the Hopfield network has functions inside a class structure, which helps in emulating the code as if it were a package.

Prediction: Noise Regulation

For the sake of this experimentation, for all intents and purposes, we refer to prediction as a process in which one inputs a (preferably noisy) image, which goes through the pipeline to output another image, with the anticipation that it does not have the noise in it anymore. Noise, for us, is the presence of undesired features in a particular image. In order to artificially create this noise, we create a function called **crappify**. We do this by randomly sampling numbers from a binomial distribution, with the probability being the level of noise that one wants. We subsequently take these pixels, invert them, and return the "crappified" image.

```
def crappify(image_vector, level = 0.3):
  crappy_image = np.copy(image_vector)
  invert_pixel = np.random.binomial(n = 1,
    p = level,
    size = len(image_vector))
  for i, pixel in enumerate(image_vector):
    if invert_pixel[i]:
      crappy_image[i] = -1 * pixel
  return crappy_image
```



Once this noise is inserted into the image signal, this image is inputted into the pipeline. For a signal, say s_x, the output signal s_y is calculated by:

$$s_y = s_x.W$$

Further, in an attempt to normalise this image, the pixels values are confined within $[-1, 1]$ using the signum function by:

$$sgn(s_{yi}) = \begin{cases} -1 & s_{yi} < \theta_i \\ 0 & s_{yi} = \theta_i \\ 1 & s_{yi} > \theta_i \end{cases}$$

where, s_{yi} is an element of s_y

Example:
Let s be an image, such that, $s = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$. The weight matrix W is calculated as shown,

$$W = s.s^T = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Now, considering s_x as the noisy input, $s_x = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}$ This image is inputted into the pipeline:

$$s_y = s_x.W = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 2 & 0 \end{pmatrix}$$

On applying s_y to the function sgn, we normalise the vector for it to become: $\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$, the original image vector. This is the original signal that we had used to create the weight matrix.

By forcing the diagonal elements to be equal to 0, we eschew the normalising step. However, just to be safe, we add that step in the program so as to not have any ambiguities.

$$W = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow W' = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

So,

$$s_y = s_x.W' = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}$$

Theoretically, making the diagonal elements 0 would mean changing the weight matrix from $s^T.s$ to $s^T.s - Q.I$ where Q is the number of prototype vectors and I is the identity matrix. We find that the prototype vectors continue to be the Eigenvectors of this new weight matrix, which is why, mathematically, it makes sense to make the diagonal 0.

With auto-associative networks in general, setting the diagonals to 0 helps in faster generalisation and often has more biological plausibility. Further, it becomes essential in case of iterative modelling.

2.3 Energy Calculation: Explaining Model Stability and Stopping Criterion

A good question to ask at this point is for how long should one keep the computations running? This is where the model energy comes into play. At each iteration, energy is computed by the following formula, as corroborated from [4],

$$E = -\frac{1}{2} \sum_{i,j} W_{ij} x_i x_j + \sum_i \theta_i x_i$$

As most energy and entropy models [9], [6], [10] work, the energy at each iteration dictates whether or not the computation should terminate. This analogy is taken from statistical physics, which says that in the most usual sense, a fluid tends to stay in the same state when its energy is minimised or its entropy is maximised. This acts as the optimising principle for the entire pipeline. This, hence, dictates that once we see no change in energy, i.e. the model cannot possibly become more stable, the computation stops.

2.4 Synchronous and Asynchronous Hopfield Networks: Associative Memory

In order for one to understand the significance of how a synchronous and an asynchronous Hopfield networks work, one needs to understand how this memory is stored in the network itself. Informally, a Hopfield network takes in a pattern (in this case, an image signal) and gives another pattern as an output. Such patterns that are stable are called memories. Being a memory for a pattern means that it has (or develops) fixed attractive points as the training goes on. It can, thusly, retrieve this memory from a highly corrupted pattern. This is the reason why it is referred to as **associative memory**. Thus for asynchronous update, only one neuron is updated at a time. This "lucky" neuron can be picked randomly, or a pre-defined order can be imposed from the beginning. For our computation, we pick the neuron randomly in case of asynchronous updates. Algorithms 1 and 2 show the steps for these respective updates. Subsequently, there is also the code block which shows how it was achieved.

Algorithm 1 Asynchronous Update of Hopfield Network

Require: Network H
  Choose a neuron i
  Compute activation $\sum_j w_{ij}x_j$
  while logical AND of all stopping criteria is false **do**
    Determine new activity $sgn(\sum_j w_{ij}x_j)$
  end while

Algorithm 2 Synchronous Update of Hopfield Network

Require: Network H
  Compute all activations $\sum_j w_{ij}x_j$
  while logical AND of all stopping criteria is false **do**
    Determine new activity $sgn(\sum_j w_{ij}x_j)$
  end while

```
def compute(self, s0):
    if self.asyn == False:
      s = s0
      energy = self.calculate_energy(s)
      energies = [energy]  # to trace the history of energies
      for i in range(self.iterations):
        s = np.sign(self.W@s - self.threshold)
        next_energy = self.calculate_energy(s)
        energies.append(next_energy)
        if energy == next_energy:  # stopping criterion, no change in energy
          return s, energies
        energy = next_energy
      return s, energies
    else:  # Only one unit is updated at a time. This unit can be picked at
    random, or a pre-defined order can be imposed from the very beginning.
      s = s0
      energy = self.calculate_energy(s)
      energies = [energy]  # to trace the history of energies
      for i in range(self.iterations):
        for j in range(100):
```

```
            lucky_neuron = np.random.randint(0, self.neurons)
            s[lucky_neuron] = np.sign(self.W[lucky_neuron].T@s - self.threshold)
          next_energy = self.calculate_energy(s)
          energies.append(next_energy)
          if energy == next_energy:
            return s, energies
          energy = next_energy
        return s, energies
```

3 Results and Discussion

A varied number of results are obtained from the program. Firstly, a set of energy plots is plotted. This is exhibited in detail in Fig. 2. Each plot is meant for the energy corresponding to each image in the data. So essentially, we aren't studying the energy in general, we are studying the component of energy for each prototype signal. This is done to better understand the behaviour for each image. We see that for both asynchronous as well as synchronous updates, we get a diminishing energy trend with time. This corroborates the notion of minimising energy for stability.

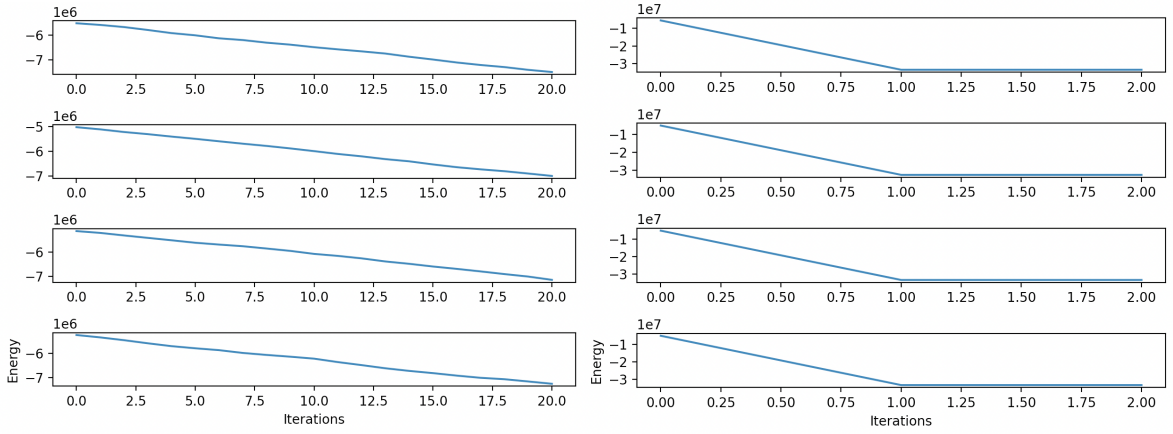


Fig. 2: Evolution of energies by time for the asynchronous (left) and synchronous (right)

Further, we compare the actual output signals of both the asynchronous as well as the synchronous updates in Fig.3. We find that while the asynchronous machine seeks to reconstruct the noisy image, the synchronous machine is able to successfully remove the noise from these signals.

Finally, we observe the weights for the two types of computations in Fig. 4. These are symmetric since in essence we are observing orthogonal matrices here. We observe that the weights look symetric. It makes sense because the weight matrix is computed by taking the dot product of a vector with its transpose. (For square matrix, multiplication of a matrix and its transpose compute a symetric matrix).

Discussion on Storage Capacity

While the synchronous updation works really well in removing the noise, and asynchronous in reconstructing the image, there is bound to be errors. The storage capacity C, for signals of dimension d, for retrieval of patterns without error is given by:

$$C \approx \frac{d}{2log(d)}$$

and the storage capacity for retrieval of patterns with small percentages of error is:

$$C \approx 0.14d$$

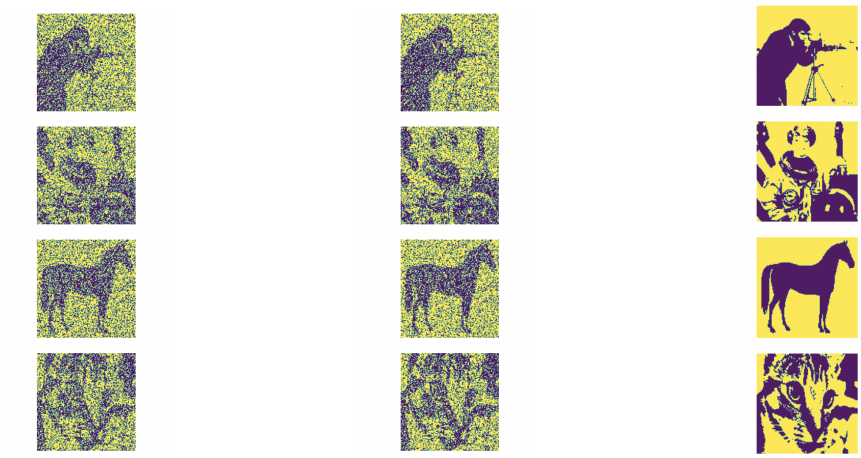


Fig. 3: Input images (left) and Output images for the asynchronous (middle) and synchronous (right)

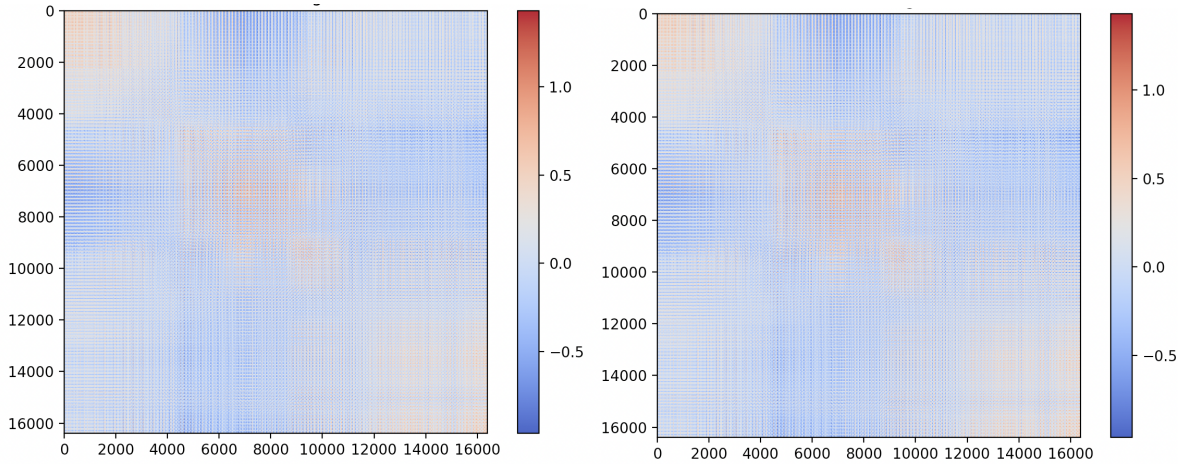


Fig. 4: Weight for the asynchronous (left) and synchronous (right)

So, errors arise, not because there is insufficient storage capacity, but because the input images are correlated. This is a reason why very different images were chosen for this experimentation, and we see that it pays off. Folli et. al [3] show that we can actually have regimes with the ratio C/d being very high, subsequently resulting in a high storage capacity, that is supporting argument of existence of more fixed points. However, it is also argued that while these methods may increase the number of fixed points, these points are not stable attractors [7], [5].

4 Conclusion

Two different methods of updation, asynchronous and synchronous, are employed to see which works better for removing noise from images. It is seen that while asynchronous updation works well for reconstructing input signals, the synchronous updation mechanism works drastically better. An intuitive understanding suggests that since only one neuron is updated at a time, it would unnderstandably take much more time for the asynchronous machine to really learn. So, one could, perhaps "force" the machine to keep carrying out its computations as the iterations go on. However, it also poses a problem of the space-time complexity trade-off and whether it is worth it to move ahead with that method.

We can further carry out the computations with different forms of auto-associative networks, or even branch out to auto-encoders and boltzmann machines for an effective comparison.

References

1. The hopfield net. http://ccy.dd.ncu.edu.tw/~chen/course/neural/ch6/l5/section3_3.html#SECTION00030000000000000000
2. Hopfield network. https://en.wikipedia.org/wiki/Hopfield_network
3. Folli, V., Leonetti, M., Ruocco, G.: On the maximum storage capacity of the hopfield model. Frontiers in computational neuroscience **10**, 144 (2017)
4. Frady, E.P., Somme, F.T.: Robust computation with rhythmic spike patterns (October 2020)
5. Gosti, G., Folli, V., Leonetti, M., Ruocco, G.: Beyond the maximum storage capacity limit in hopfield recurrent neural networks. Entropy **21**(8), 726 (2019)
6. Park, E., Ahn, J., Yoo, S.: Weighted-entropy-based quantization for deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 5456–5464 (2017)
7. Rocchi, J., Saad, D., Tantari, D.: High storage capacity in the hopfield model with auto-interactions—stability analysis. Journal of Physics A: Mathematical and Theoretical **50**(46), 465001 (2017)
8. Sejnowski, T.J., Tesauro, G.: The hebb rule for synaptic plasticity: algorithms and implementations. In: Neural models of plasticity, pp. 94–103. Elsevier (1989)
9. Sethi, I.K.: Entropy nets: from decision trees to neural networks. Proceedings of the IEEE **78**(10), 1605–1613 (1990)
10. Tkačik, G., Marre, O., Mora, T., Amodei, D., Berry II, M.J., Bialek, W.: The simplest maximum entropy model for collective behavior in a neural network. Journal of Statistical Mechanics: Theory and Experiment **2013**(03), P03011 (2013)