Imperial College London

Using Hopfield Networks to restore corrupted greyscale images

An introduction to Hopfield Networks

Hopfield Networks are a form of Ising model that have many uses in various sectors: computational neuroscience, optimisation, and more. Similar to an Ising model, they are comprised of an array of nodes that are given a node value, a *magnetic spin*, either $+1$ or -1, and edges with edge values, *'weights'*, linking them. We can use this architecture to *'train'* the network's weights on given sets of data, effectively *'storing the memory'* of the data on the network by calculating the weights that best represent the input data. Then, we can recover the stored data by inputting a new set of data and *'updating'* the network using the weights we obtained from training it on our sets of data.

In this project, we will look specifically at their application in image processing: how it can be used to restore and discern between corrupted pixellated images. By training the network on a set of images, we can then input corrupted versions of the images into the network, and updating the network will yield an image that should hopefully resemble the original image to a good degree of accuracy, even if some data is missing from the input.

We will then move on to explore how far we can take this idea - can we apply it to images where the pixels aren't just a binary *black or white*? Could we go even further and apply the same structure to recover corrupted full-colour images?

Basic definitions

We start by defining our n node spins V_i, where $i \in \{0, 1, ..., n\}$. These nodes all have connections to other nodes, and this is denoted by T_{ij}, which represents the connection weight between nodes i and j. We note that, $\forall i, j, T_{ij} = T_{ji}$. Further, $T_{ij} = T_{ji} = 0$ whenever nodes i and j are not connected, or when $i = j$.

Like an Ising model, we *'update'* the network given a set of spins by considering a threshold for each node, denoted by U_i. Then, we say:

$$V_i = \begin{cases} +1, & \sum_j T_{ij}V_j > U_i \\ -1, & \sum_j T_{ij}V_j \leq U_i \end{cases} \tag{1}$$

This threshold vector could be defined specifically to influence how specific nodes tend to behave, but for the purposes of this project we will define the threshold vector to be the zero vector, that is, $U_i = 0, \forall i$.

This update can either be done *asynchronously* or *synchronously*, for the former, each V_i is updated individually and the new V_i is used for the updates of other nodes, and for the latter, all updates use the initial set of node spins before any updates took place. If updating synchronously, some kind of internal clock or way to save previous states of the network must exist, and so most biological systems, such as the brain, will update asynchronously.

The Hebbian learning rule

The core of the Hopfield network lies in its use in *'storing'* states of the network to be recovered later. This is done using the **Hebbian learning rule**. [2]

If we have a set of m states $V^s, s \in \{1, ..., m\}$, we can use the Hebbian learning rule:

$$T_{ij} = \frac{1}{m}\sum_s V_i^s V_j^s \tag{2}$$

Since this equation yields nonzero values for the case where $i = j$, we must then adjust for this by resetting T_{ij} to 0 whenever $i = j$. This will update the weights of the network and effectively *'imprint'* the states into the network.

Now we have our basic tools for applying a Hopfield network to our problem, we can start looking at basic uses of the network.

Maximum storage space

Naturally, Hopfield networks will get less and less accurate at recalling data as the amount of data increases. We use the concept of *maximum storage space* to measure how many states the network can store before it is unlikely to be able to recall individual states.

We can calculate the storage capacity of a network with n nodes by using the following formula:

$$C \cong \frac{n}{2log_2 n} \tag{3}$$

Using this, we see that $C \approx 0.138n$. [1]

Effectively, what this means is that for every 1000 nodes in the network, approximately 138 states can be recalled. For our application, since for an n-by-m image there are nm nodes in the network, we can store $0.138nm$ images with reasonable accuracy.

Note that this means that the storage capacity is *linear* with the shape of the input, which for large sets of data is not preferable. Instead, to improve this, we can utilise a different learning rule, called the **Storkey learning rule**. [4]

This involves using a *'local field'* h^ν. We say a network follows the Storkey learning rule if it satisfies the following for a network with n nodes:

$$T_{ij}^{\nu} = T_{ij}^{\nu-1} + \frac{1}{n}V_i^{\nu}V_j^{\nu} - \frac{1}{n}(V_i^{\nu}h_{ji}^{\nu} + V_j^{\nu}h_{ij}^{\nu}) \tag{4}$$

with $h_{ij}^{\nu} = \sum_{i \neq k \neq j}^{n} T_{ik}^{\nu-1}V_k^{\nu}$ and $T_{ij}^0 = 0, \forall i, j$.

For the context of the project, we will just use the more simple Hebbian learning rule, but Storkey was able to prove that this learning rule offers an improvement in maximum storage capacity without changing the locality of the learning rule.

Initialising and training the network

Suppose we have a set of k n-by-m uncorrupted images to train the model on, where each pixel can either be black or white. We can, without loss of generality, assign the colour black to the spin $+1$ and white to -1. Now, each image can be represented as an array of nm connected nodes with spins, using each pixel as its own node in the network. In this way, we can represent an image with a Hopfield network.

To set up our problem, we can store each image as a vector of its spins, so for the sth image:

$$\boldsymbol{V^s} = \begin{bmatrix} V_1^s \\ V_2^s \\ \vdots \\ V_{nm}^s \end{bmatrix}$$

Similarly, we also store the weights in an nm-by-nm matrix:

$$\boldsymbol{T} = \begin{bmatrix} T_{1,1} & \dots & T_{1,nm} \\ \vdots & \ddots & \\ T_{nm,1} & & T_{nm,nm} \end{bmatrix}$$

Then, we can apply the *Hebbian learning rule* to train the network on our images, using (2):

$$\boldsymbol{T} = \frac{1}{k}\sum_s(\boldsymbol{V^s} \otimes \boldsymbol{V^s} - \boldsymbol{I}) \tag{5}$$

where $\otimes$ indicates the *vector outer product*, and $\boldsymbol{I}$ indicates the *identity matrix*.

This then gives us:

$$\boldsymbol{T} = \frac{1}{k}\sum_s \begin{bmatrix} 0 & V_1^sV_2^s & \dots & V_1^sV_{nm}^s \\ V_2^sV_1^s & \ddots & & \vdots \\ \vdots & & & \vdots \\ V_{nm}^sV_1^s & \dots & \dots & 0 \end{bmatrix}$$

as desired.

Restoring images

Now that the network has been trained on the images, we can use it to *restore* corrupted images. To do this, we take the matrix $\boldsymbol{T}$ that we received from training and an input state $\boldsymbol{V}$, which represents our image to be restored.

By *updating* the state using (1), we receive a new set of spins back, which we can then interpret again as an image by converting $+1$ to black and -1 to white. This will be our restored image.

We can test our trained system by inputting our original images as well as our corrupted ones:

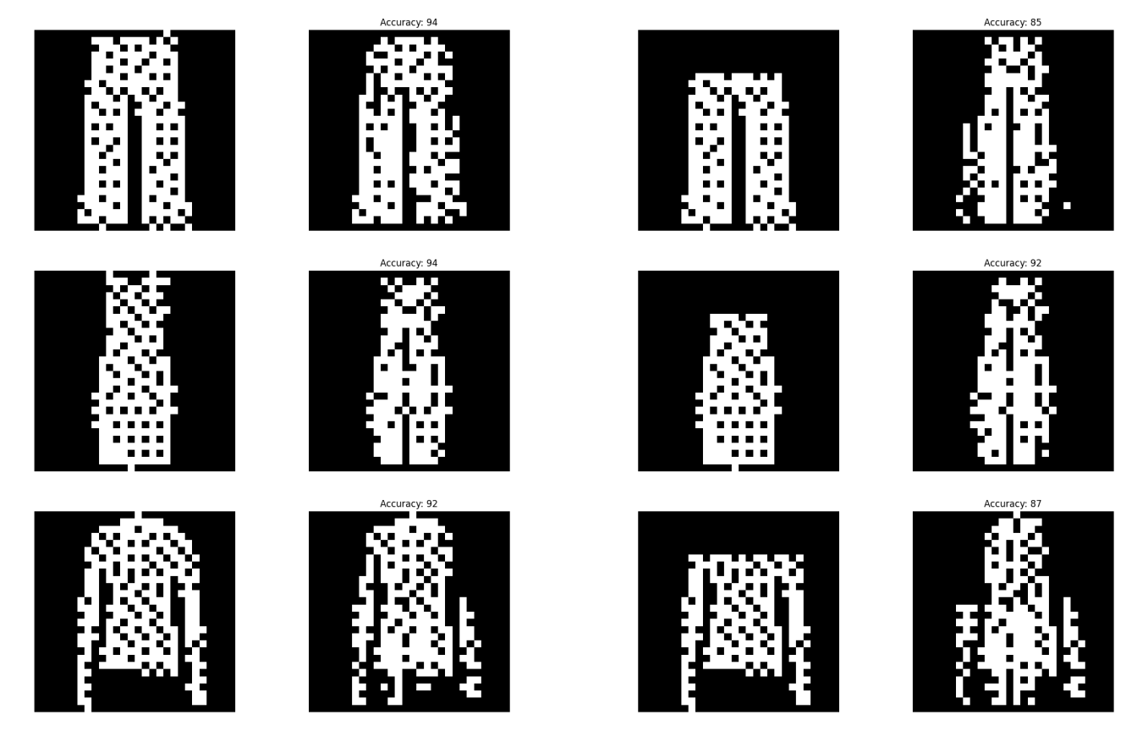
Testing on uncorrupted images
Inputting corrupted equivalents

Figure 1. Testing our network

We can see that our network is able to recover the images with an accuracy of $90\% - 95\%$, or $85\% - 90\%$ for their corrupted variants. The slight errors in recalling the image could be attributed to the fact that the images look very similar, and we could improve its accuracy by using the *Storkey learning rule* mentioned earlier, or by using a modified version of a Hopfield network called a *Modern Hopfield network*. [3]

These types of network aim to increase the storage capacity and memory recall accuracy such that the storage capacity is no longer linear with input shape, but this is outside the scope of this project.

Moving to greyscale images

What if instead of just working with spins of $+1$ or -1, we were to work with spins that take *continuous* values? We would be able to express a continuum of values for each pixel, instead of just black or white.

To do this, we instead model the spin as being a *complex* number of the form $e^{i\theta}$, with $0 \leq \theta \leq \pi$. This includes the values $+1$ and -1, which will still represent the colours black and white respectively, but we can model the pixels as any colour on the greyscale.

To get this working with our model, we need to adjust some of our rules:

New update rule

Since we are now working with continuous values, it doesn't make sense to have a threshold value anymore. Instead, we just take the *weighted average* of all of the connected pixels, and *renormalise* the complex value so its modulus is 1.

$$V_i = \frac{\sum_j T_{ij}V_j}{\left\|\sum_j T_{ij}V_j\right\|} \tag{6}$$

This will inevitably move all of the pixels closer to grey, but we will adjust for that later.

New learning rule

We also need to change the learning rule, as we can't check for a change in sign anymore. Instead, we calculate the *distance* between each pixel's colour by using its argument, and scaling it so that a weight of -1 represents the pixels being as far away as possible, and 1 represents the pixels being the same.
So, for any given state s,

$$a_{ij}^s = \left|arg(\frac{V_i^s}{V_j^s})\right|$$

is the angle between V_i^s and V_j^s. Now, we use this to form our new learning rule:

$$T_{ij} = \sum_s(1 - \frac{2a_{ij}^s}{\pi}) \tag{7}$$

Again, we set T_{ij} to equal 0 whenever $i = j$. Now the ground rules are in place for our new network, we can test it out on some greyscale images.

Training and restoration of greyscale images

Using the same method as before, we get the pixels' values, and scale them to be on the complex unit circle with positive imaginary part. Then, we use these as our node spins and train the network on them using (7). Once the network has been trained on our dataset we can use it to restore our corrupted images using (6). To adjust for the pixels all being averaged towards grey, we scale all the pixels linearly so that the most extreme pixels are the same as the most extreme pixels in the input image. Here are the results of applying the network to some sample greyscale images:

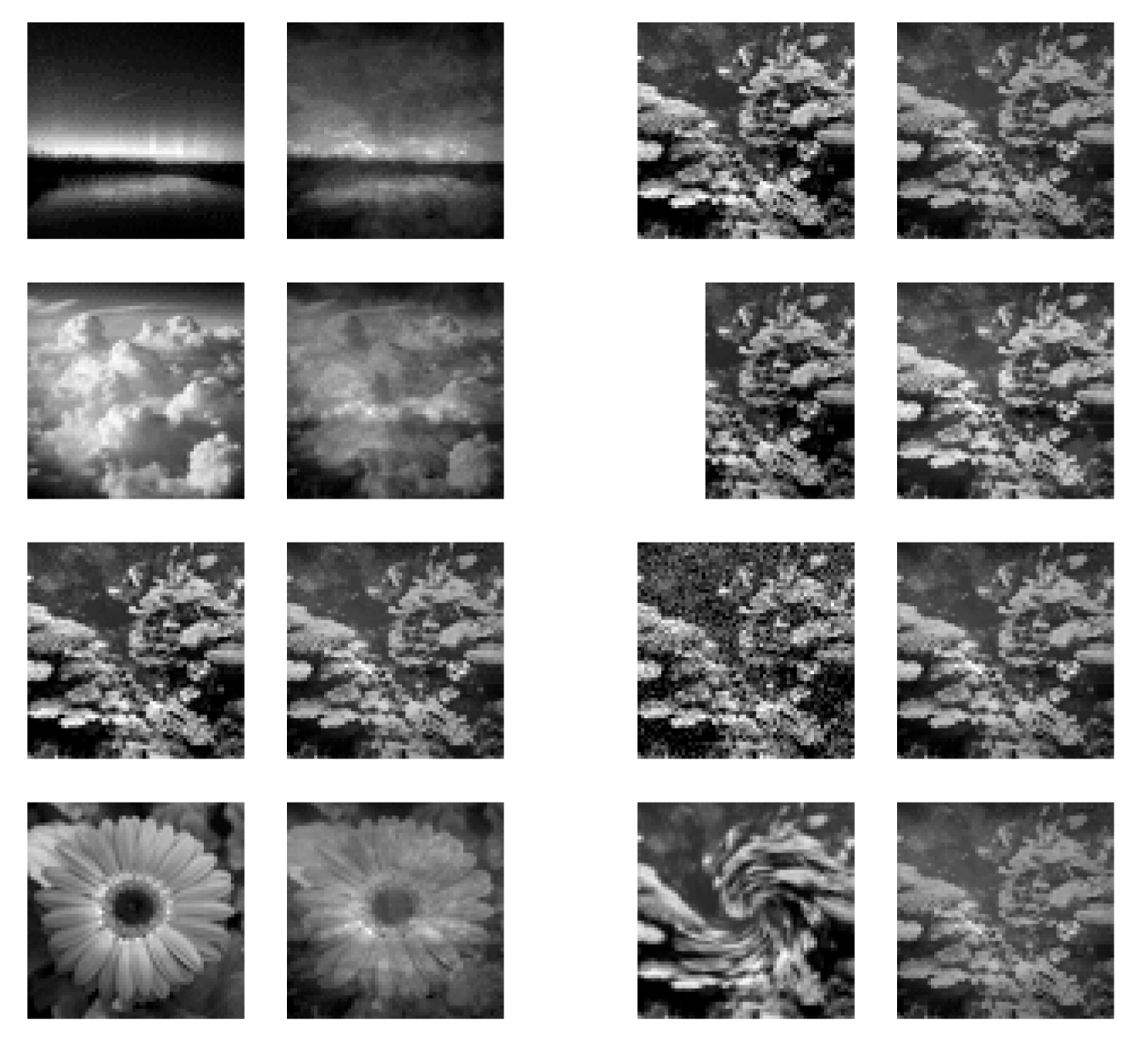

Figure 2. Testing our network again

As we can see, the network is again able to recover the images to a good degree of accuracy, even when portions of the image are removed, or is distorted by applying noise or effects to it. This means that our implementation of a Hopfield network with continuous spins has worked.

Some final thoughts - could we expand further?

Using a similar method that we used to represent the greyscale as a single value, and use that as a spin, we might next consider if we could do the same with *full-colour* images. However, we run into an issue here with associating a single value to a colour. While formats exist to represent colour using one value (ie. RGB or HSV), the Hopfield network relies on the idea of being able to determine the *'difference'* in colour, as the learning rule needs some measure of which colours are 'opposite' to each other. In our greyscale example, we had white and black, but for full colour, this doesn't really make sense. However, we could use a colour system that works like this, for example CIELAB, which is designed to work closely to human vision, with three continuous axes.

References

[1] Viola Folli, Marco Leonetti, and Giancarlo Ruocco.
On the maximum storage capacity of the hopfield model.
Frontiers in Computational Neuroscience, 10, 2017.

[2] J. J. Hopfield.
Neural networks and physical systems with emergent collective computational abilities.
Proceedings of the National Academy of Sciences of the United States of America, 79:2554–2558, April 1982.

[3] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter.
Hopfield networks is all you need, 2021.

[4] Amos Storkey.
Increasing the capacity of a hopfield network without sacrificing functionality.
In Wulfram Gerstner, Alain Germond, Martin Hasler, and Jean-Daniel Nicoud, editors, *Artificial Neural Networks — ICANN'97*, pages 451–456, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

Daniel Haywood