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 *magnetic spin*, either $+1$ or -1, and connections, *'weights'*, linking them. We can use this archetecture to *'train'* the network on given sets of data, effectively *'storing the memory'* of the data on the network. 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 orignal image to a good degree of accuracy.

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 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:

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

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

j [1]

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 a set of nm spins, using each pixel as a node in the network, so we can use a Hopfield network to store them.

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):

$$\boldsymbol{T} = \frac{1}{k} \sum_s (\boldsymbol{V^s} \otimes \boldsymbol{V^s} - diag(V_1^s V_1^s, \dots, V_{nm}^s V_{nm}^s)) \tag{3}$$

where $\otimes$ indicates the vector outer product.

This then gives us:

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

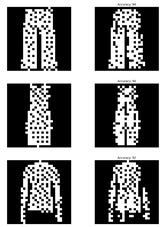
as desired.

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 recieved from training and an input state $\boldsymbol{V}$, which represents our image to be restored.

By updating the state using (1), we recieve a new image back, which we can then interpret again as an image. This will be our restored image.

We can test our trained system by inputting our orignal images $\boldsymbol{V}^s$:



Here, we see that the model is able to distinguish between the various images to a relatively high degree of accuracy, around $90\% - 95\%$.

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.

$$V_i = \frac{\sum_j T_{ij} V_j}{\left| \sum_j T_{ij} V_j \right|} \tag{4}$$

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:

$$T_{ij} = \sum_s (1 - \frac{2a_{ij}^s}{\pi}) \tag{5}$$

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 (5)

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.

Daniel Haywood