Implementing a Hopfield Network for Digit Recognition

Elizabeth McKinnie

Department of Computer Science and Department of Applied Math

University of Washington

August 2018

Abstract

Hopfield Networks are a simple form of an artificial neural network, which are vital for machine learning and artificial intelligence. In this project I've implemented a Hopfield Network that I've trained to recognize different images of digits. The mathematical model behind the network — how the network is initialized, trained, and then tested — is explained, and I've proved why the network is able to recognize patterns as well as how the network evolves to the recognizable pattern. After testing my network on a very small scale, my results are promising, but I've identified several flaws in the Hopfield Network itself that makes it difficult to obtain good results on larger and more complex data sets.

Problem Description

A Hopfield Network is a type of artificial neural network (ANN). ANNs loosely resemble biological memory: they can be trained to remember patterns and can apply those patterns to new cases. Although initially intended to simulate human memory, ANNs have become the backbone of artificial intelligence and machine learning. They've been responsible for speech, text, and image recognition, medical diagnosis, and social media filtering. As our society becomes more technologically advanced, the speed and accuracy of these networks will become even more crucial.

Hopfield Networks are content-addressable memory systems. A Hopfield Network typically contains n nodes, each of which has a binary data value (often a 0 or a 1). Each node is connected to the others with bidirectional and symmetric connections, called "weights", w_{ij}, such that $w_{ij} = w_{ji}$. Since there are no connections connecting a node to itself, $w_{ii} = 0$ for all elements [1]. Each state of the network (a snapshot of the values of the nodes) has an "energy" value, which monotonically decreases to a state that is a local minimum as the network is updated [2]. This local minimum often represents an "attractor" state, which is one the network has been trained to recognize, although sometimes there are local minima called "spurious states" that the model was not trained to recognize [3].

For this project, my focus is on implementing a Hopfield Network as a Java program, training it on a data set, testing it, and analyzing the network and its possible applicability to other data sets.

Simplifications

I chose to train and test my Hopfield Network on an image, represented as an array of pixels that are either black (represented by the digit 1) or white (represented by the digit 0). Grayscale images have pixel values from 0 to 255, so for greyscale images I set a threshold of 128 to decide if the pixel should be white or black. Although this may lose some of the detail, if the training and testing datasets are filtered through the same criteria the efficiency of the network shouldn't be heavily influenced.

Mathematical Model

I used the standard Hopfield Network as outlined by J. J. Hopfield in his introductory paper [2]. A Hopfield Network typically contains n nodes, where each node is connected to the others with bidirectional and symmetric connections. Given a set of states v, where each state is a unique pattern we want our network to recognize, we can calculate the weights as

$$w_{ij} = \sum_{l=1}^{n} (2v_i^l - 1)(2v_j^l - 1)$$

That is, for a weight w_{ij} that connects nodes i and j, we set it equal to the sum of

$$(2v_i^l - 1)(2v_j^l - 1)$$

Where v_i^l is the value of the ith node in the vector v^l. This is how we train the network before testing it. Now given a new vector that we want our Hopfield network to interpret, v^p, we must update every individual node. We calculate

$$v_i in = \sum w_{ji} v_j^p$$

That is, we multiply each weight between nodes i and j by the value at j in every state and sum the result. Then we set $v_i = 1$ if $v_i in \geq 0$ and $v_i = 0$ if $v_i in < 0$ [2]. We continue to update the network, choosing a random node to update each time, until the state of the nodes is a pattern that the network has been trained to recognize. This is asynchronous updating, because we only update one node at a time. Although synchronous updating (where you update all of the nodes simultaneously) is possible, it's difficult to execute in practice [1].

Another interesting feature of the Hopfield Network is the energy equation:

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} v_i v_j$$

That is, the energy of the network is the sum of the product of every weight with its respective end node values, multiplied by $-\frac{1}{2}$ [2]. We can arrange our attractor states to be at low energy states, so that our network will naturally devolve to them [3].

Solutions of Features of the Mathematical Model

I An important aspect of the Hopfield Network is that the energy function monotonically decreases as time increases. This lets us prove that the state converges to some equilibrium state (an attractor state or a spurious state) because the set of states is finite.

First, consider the state of a node i at time t:

$$v_i(t) = \begin{cases} 1: \sum_j w_{ij} v(t)_j \geq 0 \\ 0: otherwise \end{cases}$$

And note that our equation for energy of the state at time t is

$$E(t) = -\frac{1}{2}\sum_{i,j} w_{ij} v_i(t) v_j(t)$$

Without loss of generality, we will update the ith node. At time $t+1$ we've updated the ith node but no other nodes, and the energy of the state is

$$E(t+1) = -\frac{1}{2}\sum_{i,j} w_{ij} v_i(t+1) v_j(t)$$

Which we can rewrite as

$$E(t+1) = -\frac{1}{2}\sum_{j\neq i, k\neq i} w_{jk} v_j(t) v_k(t) - \frac{1}{2} v_i(t+1) \sum_j w_{ij} v_j(t)$$

That is, we've separated out the sum of all terms that don't depend on the ith node from those that do. For simplicity, let

$$c(t+1) = c(t) = -\frac{1}{2}\sum_{j\neq i, k\neq i} w_{jk}v_j(t)v_k(t)$$

So

$$E(t+1) = c(t) - \frac{1}{2}v_i(t+1)\sum_j w_{ij}v_j(t)$$

Now we can consider three cases:

If $v_i(t+1) = v_i(t)$ then clearly $E(t+1) = E(t)$.

If $v_i(t+1) = 1$ and $v_i(t) = 0$, then we know because of the updating rule that

$$\sum_j w_{ij}v_j(t) \geq 0$$

So

$$E(t+1) = c(t) - \tfrac{1}{2}\textstyle\sum_j w_{ij}v_j(t) = E(t) - \tfrac{1}{2}\textstyle\sum_j w_{ij}v_j(t) \leq E(t)$$

Conversely, if $v_i(t+1) = 0$ and $v_i(t) = 1$, then because of the updating rule

$$\sum_j w_{ij}v_j(t) < 0$$

So $E(t) = c(t+1) = c(t) < c(t) - \frac{1}{2}\sum_j w_{ij}v_j(t) = E(t)$

So in all cases,

$$E(t+1) \leq E(t)$$

And thus the energy function is monotonically decreasing [4][5][6].

II The following is not a rigorous proof, but more of an explanation as to why the weights will properly "pull" the state of their related nodes to the correct attractor state.

Without loss of generality, consider the weight $w_{ij} = w_{ji}$ between the nodes i and j.

If $w_{ij} > 0$, when $v_j = 1$ then $w_{ij}v_j > 0$ and so the contribution to $v_i in$ (in the updating rule) is positive, pulling v_i towards $v_j = 1$. When $v_j = 0$ then $w_{ij}v_j = 0$ and so the contribution to $v_i in$ is 0, pulling v_i towards $v_j = 0$.

Conversely, if $w_{ij} < 0$, when $v_j = 1$ then $w_{ij}v_j < 0$ and so the contribution to $v_i in$ is negative, pulling v_i towards 0 and away from $v_j = 1$. When $v_j = 0$ then $w_{ij}v_j = 0$ and so the contribution to $v_i in$ is 0, pulling v_i towards 1 and away from $v_j = 0$. (It seems counterintuitive that it's pulling towards 1, but instead we can consider it "not as strongly" pulling towards 0 compared to when the weight is negative.)

If $w_{ij} = 0$ we can say there is no contribution by v_j to the updated v_i.

Thus, the weights serve to pull the updated node i values towards the node j values. The cumulative effect is that nodes are pulled toward attractor states, as every j node is one step closer to the attractor than the i node being updated [6].

Implementation of the Network

My implementation of a Hopfield Network consists of 4 Java classes: State, Network, DataParser, and Client. The State class is used to store information about the attractor states (created from the train data set) and the DataParser reads in the name of a training set file or a testing set file and stores the data appropriately. The Network class has functions to initialize the network to a given set of node values, to train the network, to update the network, and to get the current energy of the network. The Client class is where the code is run: it creates the network, trains it, tests all the test cases on it, and prints statistics on the results.

I chose to use arrays in my implementation of Network. For a network with n nodes, an array of length n stores the values, and a two-dimensional array with size n by n stores the weights. Although typically such an implementation would be slow, and most programmers prefer more robust data structures, arrays have fast access because they use random-access memory. I tested the speed of my program and concluded that even on a very large data set (such as the MNIST data set) the network isn't significantly slow.

Results and Discussion

After training my network on two patterns representing 5x3 pixel images of the digits 0 and 1, I tested my network on 30 cases, which were the original training patterns with one pixel flipped. My network can correctly classify all 30 of these cases.

However, I started out attempting to train my network on larger cases. Originally, I wanted to train my network on the MNIST data set, a database of handwritten digits with 60,000 training patterns and 10,000 testing patterns [7]. When I was unable to get a single test to pass, I tried creating my own data set of the digits 0-9 which were 5x3 pixels. Although some of the tests passed, the success rate was poor: my network could only correctly classify about 6 out of 20 cases on average.

There are three major problems that led to the poor results on these large data sets.

I One is a general deficiency of the Hopfield Network, which allows for "spurious states" to exist. These are low-energy states that the network was not trained to recognize, but because they are low-energy, the state of the network can sometimes evolve to them [3]. This meant that sometimes a state would run forever, as it would not be able to get past the spurious state (as the energy function monotonically decreases) so it would continue to update, even though it could never find a matching attractor state. This problem may have been further exacerbated by my choice of attractor states. As explained earlier, it's ideal for the attractor states to be lower-energy than the test cases [3]. If an attractor state has higher energy, there's no way for a given test case to evolve to that state. But in a real-life scenario, this isn't realistic, as you're not able to choose the data you want your network to identify based on its energy-level.

II Another problem I've identified concerns how many attractor states you're training your network to recognize. According to the USC Brain Project [3], if you have more patterns stored than 15% of the number of nodes, the network will randomize the energy minima. For the MNIST data set, my network only had 400 nodes but I was trying to train it to recognize 60,000 different patterns (exceeding the 60-pattern capacity). The successful data set, which has 15 nodes and 2 attractor states, works because the capacity for a 15-node network is 2.

III Finally, another flaw in the Hopfield Network is in how the nodes are updated asynchronously. Even if you've carefully chosen your test cases to have initial energy states higher than the attractor states, and you don't have more attractor states than nodes in your network, it's possible that the order the network updates in causes the state to evolve to the wrong attractor state.

These flaws in the standard Hopfield Network explain why I was unable to successfully apply my network to the MNIST data set or to my slightly larger test set, and explain why my results on the small data set are perfectly reasonable.

Improvements

Through my testing on the small data sets as well as the MNIST data set, it seems that the fewer attractor states there are, the better it is for results. Choosing fewer attractor states would be easy to implement and may improve results, but it could also mean your network must be more specific in what it's trained to analyze.

Conclusions

In this paper, and in my attached Java files, I've presented the background and importance of Hopfield Networks, the mathematical model behind it, and discussed the results and problems I encountered while implementing my own Hopfield Network. This was my first foray into the world of machine learning, and I learned a lot about the mathematical basis that allows us to prove that these models work, as well as possible applications. As a Computer Science major, I enjoyed the opportunity to build my own program based off a mathematical concept and to apply what I've been learning in my CSE classes about program organization and testing. I've seen first-hand how powerful this sort of network can be, and I'm looking forward to explore in greater depth how I can apply machine learning to other problems.

References

[1] “42.2 Definition of the Binary Hopfield Network.” *Information Theory, Inference, and Learning Algorithms*, by David J. C. MacKay, Cambridge University Press, 2003, p. 506.

[2] Hopfield, J. J. “Neural Networks and Physical Systems with Emergent Collective Computational Abilities.” *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, 1 Apr. 1982, pp. 2554–2558., doi:10.1073/pnas.79.8.2554.

[3] “Hopfield Simulation.” *The USC Brain Project*, University of Southern California, 2018, neuroinformatics.usc.edu/resources/hopfield-simulation/.

[4] alto. “Derivation of the Energy Function of a Hopfield Network.” *Computer Science Stack Exchange*, 13 July 2009, cs.stackexchange.com/questions/13132/derivation-of-the-energy-function-of-a-hopfield-network.

[5] Dennis, Simon. “The Hopfield Network: Descent on an Energy Surface.” *The BackPropagation Network: Learning by Example*, 1997, staff.itee.uq.edu.au/janetw/cmc/chapters/Hopfield/.

[6] Edalat, Abbas. “Hopfield Networks.” *Abbas Edalat's Home Page*, Imperial College London, 2015, www.doc.ic.ac.uk/~ae/papers/Hopfield-networks-15.pdf.

[7] LeCun, Yann, et al. “THE MNIST DATABASE.” *MNIST Handwritten Digit Database*, yann.lecun.com/exdb/mnist/.