Introduction to Hopfield Networks With the Package 'hann'

Emmanuel Paradis

January 26, 2026

Contents

---

1 Hopfield Networks

The Hopfield network [3] is a type of neural network with N (input) neurons which are either in state -1 or in state $+1$. These states are determined in order to minimize its global energy level with respect to a list of K "patterns" each of length N.

In practice, the Hopfield network is coupled with other layers neurons which are themselves connected to C classification (or output) neurons whose signals identify the patterns. The different layers are connected through matrices of weights. There are as many layers than there are weight matrices.[1]

The K patterns are used to train the network, both to minimize the energy level of the Hopfield network and to find the parameter values that minimize the loss function (the discrepancy between the observed output signals and their expectations under the known classes of the patterns).

Hopfield networks can "memorize" a large number of patterns. Giving N input neurons, there are 2^N possible patterns, for example for $N = 30$, 60, and 100:

```
> N <- c(30, 60, 100)
> 2^N
[1] 1.073742e+09 1.152922e+18 1.267651e+30
```

Several studies tried to find if a Hopfield network can memorize as many patterns as these numbers, e.g., [1, 4]. Krotov and Hopfield [5] proposed the following formula for the maximum number of these patterns (M):

[1] See `https://www.cs.toronto.edu/~lczhang/360/lec/w02/terms.html` for a very nice introduction to artificial neural networks.

$$M = \frac{1}{2(2n-3)!!} \times \frac{N^{n-1}}{\ln N},$$

where n is a parameter of the energy function. For example for the same values of N above and for $n = 2$, 10, 20, and 30:

```
> ## double factorial (n!!) that we want vectorized to use with
> ## outer() below
> dfact <- function(n) {
+     ## seq() is not vectorized on its 2nd arg.
+     x <- mapply(seq, from = 1, to = n, by = 2)
+     sapply(x, prod)
+ }
> ## eq. 6 in Krotov & Hopfield (2016)
> funM <- function(N, n)
+     N^(n - 1) / (2 * dfact(2 * n - 3) * log(N))
> n <- c(2, 10, 20, 30)
> o <- outer(N, n, funM)
> dimnames(o) <- list(paste("N =", N), paste("n =", n))
> o
            n = 2       n = 10       n = 20       n = 30
N = 30   4.410212 8.396947e+04 2.083464e+05 2.037472e+03
N = 60   7.327180 3.571403e+07 9.074098e+10 9.086758e+11
N = 100 10.857362 3.150767e+09 1.323940e+15 2.192610e+18
```

2 Data Coding

The patterns must be arranged in a matrix where each row represents a single pattern (so there are K rows). The number of columns of this matrix is the number of input neurons (N).

```
> N <- 60L
> K <- 2000L
> xi <- matrix(1L, K, N)
> p <- 0.15 # not smaller than 0.15
> probs <- c(p, 1 - p)
> v <- c(-1L, 1L)
> set.seed(1)
> xi1 <- t(replicate(1000, sample(v, N, TRUE, probs)))
> xi2 <- t(replicate(1000, sample(v, N, TRUE, rev(probs))))
> xi <- rbind(xi1, xi2)
> stopifnot(nrow(unique(xi)) == K)
```

Before simulating the data, we called `set.seed(1)` to repeat consistently the results each time the code of this vignette is executed. If the user wants to simulate other data, just delete the line or give another value to `set.seed()`. If a small value is given to `p`, it is more likely that the number of unique patterns is less than K. It is recommended to use only unique patterns in the subsequent analyses.

3 Building the Hopfield Network

The function `buildSigma()` finds a network with the lowest energy level. The algorithm starts from a random network; convergence to a low energy level depends on the initial state of the network. Thus, the algorithm is repeated 100 times (by default). For this document, we set the number of repetitions to 10 to avoid printing to many lines.[2] We try the function with two values of the energy parameter: $n = 20$ (this is the default) and $n = 30$.

```
> library(hann)
> sigma20 <- buildSigma(xi, nrep = 10)
 1:  Initial energy = -1.02e+25   Updated energy = -4.05e+29
 2:  Initial energy = -2.42e+26   Updated energy = 0
 3:  Initial energy = -2.00e+26   Updated energy = 0
 4:  Initial energy = -3.84e+29   Updated energy = 0
 5:  Initial energy = -4.87e+25   Updated energy = -3.87e+35
 6:  Initial energy = -2.55e+29   Updated energy = 0
 7:  Initial energy = -1.02e+28   Updated energy = 0
 8:  Initial energy = -4.68e+25   Updated energy = 0
 9:  Initial energy = -5.03e+26   Updated energy = 0
10:  Initial energy = -3.60e+29   Updated energy = 0

Final energy = -3.871808e+35
```

```
> sigma30 <- buildSigma(xi, n = 30, nrep = 10)
 1:  Initial energy = -5.51e+37   Updated energy = -2.21e+53
 2:  Initial energy = -3.00e+42   Updated energy = 0
 3:  Initial energy = -2.56e+41   Updated energy = 0
 4:  Initial energy = -3.22e+42   Updated energy = -2.21e+53
 5:  Initial energy = -1.18e+39   Updated energy = -3.13e+44
 6:  Initial energy = -5.91e+42   Updated energy = 0
 7:  Initial energy = -2.78e+41   Updated energy = 0
 8:  Initial energy = -6.08e+40   Updated energy = 0
 9:  Initial energy = -8.17e+43   Updated energy = -2.21e+53
10:  Initial energy = -1.18e+47   Updated energy = 0

Final energy = -2.214794e+53
```

Typically, around 20% of the repetitions convergence to the same (lowest) energy level. It is recommended to leave the default `nrep = 100`.

4 Optimizing the Parameters

The package `hann` has two functions to build neural networks: `hann1` and `hann3`. See their respective help pages where they are described.

We now optimize both types of networks with the data simulated above. We first create a membership variable indicating that the first 1000 patterns belong to the same class, and the last 1000 ones to another class:

```
> cl <- rep(1:2, each = 1000)
```

[2] This function has the `quiet` (`FALSE` by default) to only return the network with the lowest energy level.

Considering that each pattern in the first class has, on average, 15% of -1, while each pattern in the second class has, on average, 15% of $+1$, these patterns are expected to be very similar within a class but very dissimilar between both classes.

We can now optimize the neural nets asking to print the error rate at each iteration:

```
> ctr <- control.hann()
> ctr$trace.error <- TRUE
> nt1 <- hann1(xi, sigma20, cl, control = ctr)
iteration 0        Error rate = 1927 / 2000        obj_fun = 9499.376

iteration 1        obj_fun = 9499.376132
iteration 1        Error rate = 0 / 2000        obj_fun = 0.000
```

For the more complicated 3-layer network, we set a milder target for the convergence of the loss function:

```
> ctr$target <- 0.1
> nt3 <- hann3(xi, sigma20, cl, control = ctr)
iteration 0        Error rate = 1354 / 2000        obj_fun = 4633.223

iteration 1        Error rate = 989 / 2000        obj_fun = 4154.603

iteration 2        Error rate = 0 / 2000        obj_fun = 4.802

iteration 3        Error rate = 0 / 2000        obj_fun = 4.333

iteration 4        Error rate = 0 / 2000        obj_fun = 2.623

iteration 5        Error rate = 0 / 2000        obj_fun = 2.157

iteration 6        Error rate = 0 / 2000        obj_fun = 0.056
```

Both networks perform well and the optimization converged quickly.

5 Classification and Error Rates

The performance of the classification of a network is assessed with the (generic) function `predict`; thus, the help page is accessed with `?predict.hann1` (or `?predict.hann3`). We first assess the error rates with the training data:

```
> table(predict(nt1, xi, rawsignal = FALSE), cl)
   cl
      1    2
  1 1000    0
  2    0 1000

> table(predict(nt3, xi, rawsignal = FALSE), cl)
   cl
      1    2
  1 1000    0
  2    0 1000
```

A trivial test is to assess whether similar classifications could be achieved with random parameters (i.e., unoptimized networks). This can be done by repeating the above analyses after setting the number of iterations to zero:[3]

```
> ctr$iterlim <- 0
> nt0 <- hann1(xi, sigma20, cl, control = ctr)
iteration 0        Error rate = 1924 / 2000        obj_fun = 8929.936

> table(predict(nt0, xi, rawsignal = FALSE), cl)
   cl
      1   2
  1  43 967
  2 957  33

> nt0b <- hann3(xi, sigma20, cl, control = ctr)
iteration 0        Error rate = 1845 / 2000        obj_fun = 5202.224

> table(predict(nt0b, xi, rawsignal = FALSE), cl)
   cl
      1   2
  1 101 898
  2 899 102
```

Clearly, random networks cannot identify our patterns.

We remember that both classes of patterns are fairly homogeneous but different each others. What if these patterns are totally random while still in two different classes? We try to assess this question by setting a small network with $N = 30$ and $K = 200$ patterns.

```
> N <- 30
> K <- 200
> xi <- matrix(sample(v, K * N, TRUE), K, N)
```

The rest of the analyses is very similar to the above ones:

```
> sigma <- buildSigma(xi, quiet = TRUE)
> cl <- rep(1:2, each = 100)
> ctr <- control.hann(iterlim = 1000, quiet = TRUE)
> net1 <- hann1(xi, sigma, cl, control = ctr)
> net3 <- hann3(xi, sigma, cl, control = ctr)
```

We used a larger number of iterations to make sure that the optimizations reached (if possible) a small value of the loss function. We can now assess the (final) error rates:

```
> table(predict(net1, xi, rawsignal = FALSE), cl)
   cl
     1  2
  1 77 34
  2 23 66
```

[3] By default, `hann1()` and `hann3()` initialize the network parameters with random values.

```
> table(predict(net3, xi, rawsignal = FALSE), cl)
   cl
      1   2
  1 100   0
  2   0 100
```

The 3-layer net performed much better than the 1-layer one. Setting `iterlim` to a larger value can make the 3-layer network reach 0% error. The literature on neural networks states that networks with no hidden layer fail to solve some problems even with very small data sets [2, 5].

6 Perspectives

The present document aims to give a quick overview of the possibilities of hann. The package is in its development stage; the current and (possible) future lines of development are:

- Parallelization: there is an early attempt to use multicore based on OMP in `hann1()`. Therefore, this cannot be used together with functions from the package parallel. On the other hand, it is possible to run several optimizations in parallel, for instance with `parallel::mclapply()` if the OMP-based parallelization is off.
- Preparing databases for training: some works have been started on DNA.

References

[1] M. Demircigil, J. Heusel, M. Löwe, S. Upgang, and F. Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168:288–299, 2017.

[2] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[3] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA*, 79:2554–2558, 1982.

[4] I. Kanter and H. Sompolinsky. Associative recall of memory without errors. *Physical Review A*, 35:380–392, 1987.

[5] D. Krotov and J. J. Hopfield. Dense associative memory for pattern recognition. arXiv.1606.01164, 2016.