

# Hopfield/Hopular Neural Networks in Clash

Rad Tudor-Andrei

Radu Tudor-Eduard

Serie: CTIen | Grupa: 30432

## 1 Contents

1. Clash Syntax & Data Structures
2. Clash Modus Operandi
3. Hopfield Network
4. Modern Hopfield Networks
5. Hopular Network
6. Implementation Overview

## 2 Clash Syntax & Data Structures

Clash is a functional hardware description language based on Haskell. Key architectural constraints include:

- **Static Sizing:** All bit-widths and vector lengths must be known at compile time.
- **Time Modeling:** Time is modeled via the `Signal` keyword, representing values received at every clock cycle (translated as wires).
- **Explicit Clocks:** Clash does not assume a default clock; it must be explicitly defined.

### 2.1 Signal Type and Lifting

A `Signal#(T)` represents a stream of values. Operations on signals must be **lifted**:

- **fmap** (`< $ >`): Applies a function at every clock cycle (combinational logic).
- **Applicative** (`< * >`): Lifting in the case of multiple signals.

Operation	Hardware Meaning
<code>Lifted (&lt; \$ &gt;, &lt; * &gt;)</code>	Combinational Logic
<code>register</code>	Sequential Logic (Flip-Flops)

## 2.2 Standard Primitives

- **Mux:** `mux cond a b` (standard multiplexer).
- **Bundling:** `bundle` and `unbundle` for converting between tuples and signals.
- **Vectors:** `Vec n a` provides fixed-size, synthesizable arrays.
- **State Machines:** `mealy` and `moore` functions define stateful logic.

## 3 Clash Modus Operandi

The transformation from Haskell code to Hardware Description Language (HDL) follows these steps:

1. **Parsing:** GHC parses Haskell + Clash primitives.
2. **Normalization:** Functions are inlined, recursion is eliminated, and higher-order functions are resolved.
3. **Clash Core:** The final IR before hardware interpretation.
4. **Netlist Generation:** A graph of wires, registers, and combinational blocks is created.
5. **HDL Emission:** VHDL/Verilog code is generated.

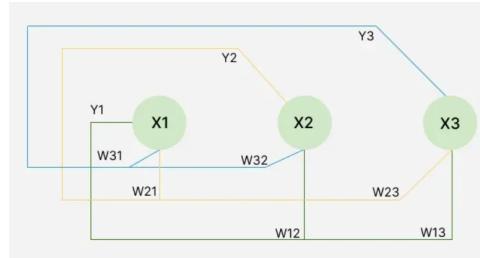
## 4 Hopfield Neural Network

A Hopfield network is a recurrent, fully connected neural network used for **Auto-Associative Memory**.

### 4.1 Architecture

The network consists of  $N$  neurons, connected to every other neuron with the following constraints:

- No self-connections:  $w_{ii} = 0$
- Symmetric weights:  $w_{ij} = w_{ji}$



For a specific neuron  $i$ , the local field  $h_i$  and activation  $x_i$  are defined as:

$$h_i = \sum w_{ij}x_j - \theta_i \quad (1)$$

$$x_i^{(t+1)} = \text{sign}(h_i) \quad (2)$$

## 4.2 Energy Function

The network minimizes a **Lyapunov function** to reach a stable state:

$$E = -\frac{1}{2} \sum w_{ij} s_i s_j - \sum I_i s_i \quad (3)$$

Here,  $s_i$  is the state of the neuron and  $I_i$  is the external input. Updates are performed asynchronously to ensure stability.

## 4.3 Training Steps

1. Initialize the weights  $w_{ij}$  to store the desired patterns.
2. For each input vector  $\mathbf{x}$ , initialize the activations of the network as:

$$y_i = x_i, \quad \text{for } i = 1, 2, \dots, n.$$

3. Compute the total input to each neuron by combining the external input and the weighted feedback from all other neurons:

$$y_i^{\text{in}} = x_i + \sum_j y_j w_{ji}.$$

4. Apply the activation function to each output:

$$y_i = \begin{cases} 1, & \text{if } y_i^{\text{in}} > \theta_i, \\ y_i, & \text{if } y_i^{\text{in}} = \theta_i, \\ 0, & \text{if } y_i^{\text{in}} < \theta_i, \end{cases}$$

where the threshold  $\theta_i$  is typically set to 0.

5. Feed back the updated outputs  $y_i$  to all neurons in the network and update the activation vector accordingly.
6. Repeat the computation of the total input and activation update until the network converges (the activation vector remains unchanged between successive iterations).

Due to its architecture, it has a maximum number of patterns to store:

$$P \approx 0.138N, \quad N \rightarrow \text{number of nodes}$$

## 4.4 Behavior example

In case of a small network with 3 binary neurons,  $\{-1, +1\}$ : Stored pattern:

$$\xi = \begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}$$

For each weight:

$$w_{ij} = \frac{1}{N} \xi_i \xi_j, i \neq j$$

$$w_{ii} = 0$$

Compute the weight matrix:

$$W = \frac{1}{3} \begin{bmatrix} 0 & -1 & +1 \\ -1 & 0 & -1 \\ +1 & -1 & 0 \end{bmatrix}$$

Give a noisy input, with one bit wrong:

$$y^{(0)} = \begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix}$$

Apply (1) to update the neurons, with a  $\theta_i$  of 0.

$$y_i = \text{sign}(\sum_j w_{ij} y_j)$$

For neuron #1:

$$\begin{aligned} h_1 &= \frac{1}{3}[0 \cdot 1 + (-1) \cdot 1 + 1 \cdot 1] = 0 \\ y_1 &= +1 \quad (h_i \text{ equal to } \theta_i) \end{aligned}$$

For neuron #2:

$$\begin{aligned} h_2 &= \frac{1}{3}[(-1) \cdot 1 + 0 \cdot 1 + (-1) \cdot 1] = -\frac{2}{3} \\ y_2 &= -1 \end{aligned}$$

For neuron #3:

$$\begin{aligned} h_3 &= \frac{1}{3}[1 \cdot 1 + (-1) \cdot (-1) + 0 \cdot 1] = \frac{2}{3} \\ y_3 &= +1 \end{aligned}$$

If we update again, all neurons keep their values. Thus, the pattern is stable, and the network's energy converged.

## 5 Modern Hopfield Networks

Modern Hopfield Networks introduce non-linearity, either in the energy function or neurons' activation function, leading to super-linear (even exponential, depending on the chosen activation function) scaling of the memory capacity as a function of the number of neurons.

## 6 Hopular Networks

### 6.1 Introduction

Hopular is a modern **Hopfield** network-based model for **tabular** data, able to retain associative memory properties. It competes with other tabular ML models, such as Gradient-Boosted Decision Trees (e.g. XGBoost) and MLPs.

## 6.2 Performance Comparison

The model exceeds in small-medium ( $\lesssim 10000$  samples) sized datasets. This is due to the network being able to identify local neighborhoods in the data, which is useful for regression tasks when the underlying function is hard to determine without large amounts of data.

The drawback of this recurrent model is the *iterative refinement* step: re-accessing the entire training set at every layer. The performance of this network comes at the cost of refining the prediction in an iterative manner.

Table 1: Performance and computational comparison on tabular data

Model	Median Rank	Accuracy	Training	Inference
Hopular	$\approx 1.5$	Highest	High	Medium-High
CatBoost	$\approx 2.0$	Very High	Medium	Low
LightGBM	$\approx 2.2$	Very High	Medium	Low
XGBoost	$\approx 4.5$	High	Medium-High	Low
MLP	$\approx 5.0$	Medium	Low	Low

## 6.3 Hopfield vs. Hopular

Before commencing with the implementation in Haskell/Clash, it may be helpful to present a side-by-side comparison of the original Hopfield network and the advanced version, Hopular network:

Table 2: Comparison of architecture and behavior of the two networks

Feature	Hopfield	Hopular
States	Binary states: $\{-1, 1\}$	Continuous states (floats)
Update	Hard update - sign()	Smooth non-linearity (softmax/exp)
Energy function	Quadratic energy	Log-sum-exp energy*
Use case	Associative memory	Supervised learning for tabular data
Scalability	Small networks	Scales to modern datasets
Memory capacity	$O(N)$ patterns	Exponential in dimension**

\* Hopfield networks use a log-sum-exp energy function:

$$E(x) = -\frac{1}{\beta} \log \sum_{i=1}^P e^{\beta x^\top M_i}, \quad \text{where } \beta \text{ controls sharpness, } x \in \mathbb{R}^d, M_i \in \mathbb{R}^d$$

\*\* For  $d =$  dimensionality of the state vector, and  $N =$  number of neurons, there is:

$$P_{max} \approx 0.138N \text{ for Hopfield, and } P_{max} = O(e^d) \text{ for Hopular}$$

## 7 Implementation Overview

### 7.1 Top Entity

The implementation of the Hopular Network starts off with the *TopEntity*, a component similar to the *main()* function of a computer program. The input will be a noisy row and the output either the closest pattern stored or a spurious state.

```
1 Project.hs
2     topEntity :: 
3         Clock Dom50 ->
4         Reset Dom50 ->
5         Enable Dom50 ->
6         Signal Dom50 State ->
7         Signal Dom50 State
8     topEntity =
9         exposeClockResetEnable hopularCore
```

### 7.2 Hopular Functions

`hopularStep` performs attention-based update by computing dot-product similarities between current state and all stored patterns. It uses scaling and exponentiation to obtain softmax weights. It, then, computes the new neuron values using those weights in a weighted sum. Thus, it brings the current state to the most similar stored pattern.

```
1 hopularStep :: Patterns -> State -> State
2 hopularStep pats x =
3     let scores = map ('dotWide' x) pats
4     scoresScaled = map (clampScore . ('shiftR' softmaxShift)) scores
5     exps = map expApprox scoresScaled
6     sumExp = sum exps
7     col i = map (\p -> p !! i) pats
8     combine i =
9         let ws = zipWith
10             (\e p -> (resize e :: AccWide) * (resize p :: AccWide))
11             exps
12             (col i)
13         wsum = sum ws
14         out = safeDiv wsum sumExp
15         in resize out :: Acc
16     in imap (\i _ -> combine i) x
```

In short,

$$x' = \sum_i softmax(p_i \cdot x) \cdot p_i$$

`hopularCore` wraps `hopularStep` into a Mealy machine that updates the network's state on each clock cycle. When a new input arrives, the internal state and iteration counter are reset. The iteration happens sequentially until convergence or the maximum number of iterations is reached.

```

1 Hopular.hs
2     hopularCore :: 
3         HiddenClockResetEnable Dom50 =>
4             Signal Dom50 State ->
5                 Signal Dom50 State
6     hopularCore x0 =
7         mealy step initState x0
8         where
9             maxIter :: Iter
10            maxIter = 8
11            initState = (repeat 0, 0 :: Iter)
12            step (st, it) x =
13                if x /= st then
14                    ((x, 0), x)
15                else
16                    let st' = hopularStep patterns st
17                        done = st' == st || it == maxIter
18                        it' = if done then it else it + 1
19                        in ((st', it'), st')

```

## 7.3 Types

Since the data had thirteen columns, the dimension `Dim` has been set to 13. Although the initial data was quite large in values, after preprocessing (z-factored and scaled by 256), a width of 16 bits (-32,768 to 32,767) for the `Acc` was sufficient.

```

1 type Dim = 13
2 type Acc = Signed 16
3 type AccWide = Signed 32
4 type State = Vec Dim Acc
5 type NumPats = 2
6 type Patterns = Vec NumPats State
7 type Iter = Unsigned 4

```

## 7.4 Utility functions

### 7.4.1 Dot product

The `dot` function was used for similarity between two state vectors using 16-bit & 32-bit arithmetic (`dotWide`, safer for larger intermediate sums). In the context of Hopfield update, the dot products represent attention scores between input and stored patterns.

```

1 dot :: State -> State -> Acc
2 dot a b = sum (zipWith (*) a b)

```

### 7.4.2 Exponential computation

For the softmax  $\alpha_i$  value computation, a look-up table of  $e^x, x \in [-8, 8]$  values was used:

$$\text{expLUT}[i] \approx 256 \cdot e^{(i-8)/2}$$

Instead of computing  $e^x$ , the implementation approximates  $e^{x/2}$ , reducing the growth rate.

```

1  expLUT :: Vec 17 AccWide
2  expLUT =
3      5 :> 8 :> 13 :> 21 :> 35 :> 57 :> 94 :> 155 :>
4      256 :> 422 :> 696 :> 1147 :> 1892 :> 3119 :> 5142 :> 8478 :>
5      13977 :> Nil
6
7  expApprox :: AccWide -> AccWide
8  expApprox s =
9      let idx = fromIntegral (s + 8) :: Unsigned 5
10     in expLUT !! idx

```

#### 7.4.3 Error metrics

The `sqError` function computes SSE between two states, corresponding to the squared Euclidean distance. Provides a magnitude-sensitive measure of the deviation.

```

1  sqError :: State -> State -> AccWide
2  sqError y t =
3      sum (map (\d -> let w = resize d :: AccWide in w*w) (zipWith (-) y t))

```

The `linfDist` function, or the Chebyshev distance,  $L^\infty$  metric, captures the worst-case deviation across all neurons.

```

1  linfDist :: State -> State -> Acc
2  linfDist y t =
3      foldl max 0 (map abs (zipWith (-) y t))

```

## 8 Simulation Results

Example of pattern stored in the Hopular network:

```

ghci> p0
384 :> -128 :> 64 :> -256 :> 512 :> 128 :> -64 :> 0 :> 256 :> -192 :> 96 :> 320 :>
-32 :> Nil

```

Noisy vector derived from previous pattern example:

```

ghci> x0
350 :> -120 :> 80 :> -220 :> 500 :> 100 :> -40 :> 10 :> 240 :> -180 :> 120 :> 300
:> -20 :> Nil

```

Closest row given by the Hopular network:

```

ghci> let y1 = hopularStep patterns x0
ghci> y1
383 :> -128 :> 63 :> -256 :> 511 :> 127 :> -64 :> -1 :> 255 :> -192 :> 95 :> 319
:> -32 :> Nil

```

The small SSE and the maximum deviation of one unit indicate only minor errors.

```

ghci> sqError y1 p0
8
ghci> linfDist y1 p0
1

```