Moore Machine implemented purely on multi-layer Hopfield networks

Goran Ivančić *

December 2024

*First year of bachelor program *electrical engineering, communication technologies and computing* at Faculty of Electrical Engineering and Computing, University of Zagreb, first year of bachelor program *applied psychology* at DOBA Faculty of Applied Business and Social Studies

Abstract

We explore a possible implementation of a Moore machine using a neural network paradigm known as multi-layer Hopfield network. We rely purely on this architecture without added computational power usually found in neural networks. Energy minimisation found in Hopfield networks are used to help encode states and outputs directly. Primary objectives are to demonstrate the feasibility of such an architecture and evaluate it's efficiency compared to standard implementations.

We encode states as attractors in energy space with predefined weights between neurons across all layers. Outputs are based only on the state in accordance with Moore's paradigm. Key challenges are finding stable states and achieving reliable transitions based on user input.

Performance for both time and space complexity is analysed through a strictly theoretical approach. We compare the performance and robustness of this system with a traditional implementation of Moore's machine. Since all finite state machines may be represented as a Moore's automata this project shows that any finite state machine may be represented through a Hopfield network. This provides an equivalence tool to connect automata theory and neural networks.

Contents

1 Introduction

1.1 Moore's Machine

Moore machines are one of the most basic types of finite state machines. They have widespread usage in digital systems as a base for sequential logic. Its main characteristic that differentiates it from other finite state machines is having outputs that depend only on the machine's current state, rather than its inputs too.

Components of a Moore machine are:

1. States (Q)
   A finite set of different states a machine can be in (usually denoted by $S_0, S_1...$)

2. Input (Σ)
   A finite set of different inputs which a user can provide to the machine which is used in transition logic

3. Initial state ($S_0 \in Q$)
   A state in which the machine stars

4. Transition function (δ)
   A function of the type $\delta : S \times \Sigma \to S$ which determines the next state from the previous and user input

5. Outputs (O)
   A finite set of different output states

6. Output function (μ)
   A function of the type $\mu : Q \to O$ which determines the output of the machine based only on the state

A graphic way to represent what and how a Moore machine works is like in figure 1.

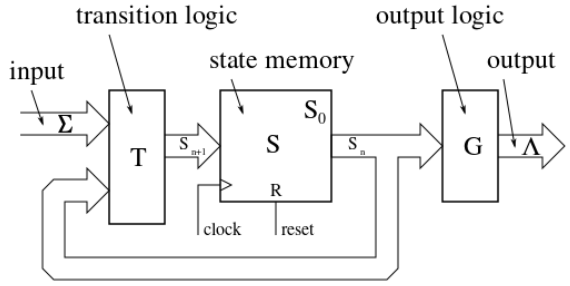

Figure 1: A diagram of what a Moore Machine looks like with added clock and reset functionality

A Moore machine starts at some state S_0 and for each input Σ it decides on the next state based on δ. This process continues indefinitely and usually there exists a loop such that some state S_i leads to S_0 for some input Σ but this isn't required by the paradigm. This kind of architecture is used in a wide variety of cases including digital systems, pattern recognition and automata-based modelling is software engineering. Real implementations usually also include a clock pulse and a reset input which will be omitted from this implementation for simplicity.

1.2 Hopfield network

Hopfield networks or associative memory is a form of recurrent neural network which processes information across multiple time steps. It consists of a set of neurons where each neuron is connected with every other with a certain pre-trained weight. According to those weights the network evolves from an input towards stable states (usually referred to as minimal energy states). This gives the network to recall an exact pattern from any partial or noisy one provided there aren't too many overlaps between stored patters. The learning model is inspired heavily by the Hebbian learning rule.

Components of a Hopfield network are:

1. Neurons (each labelled x_i)
   Each network is made up of individual neurons which have a state x_i which in classical networks either has values in $1, -1$ or $1, 0$ representing on and off states. We will use the first due to a simplification later seen. Every neuron is connected to every other neuron. These kinds of neurons are usually also called perceptrons, and the networks a perceptron network.

2. Weights (each labelled w_{ij})
   Between every two neurons (call them neuron x_i and x_j) the connection has a weight w_{ij}. If this weight is positive the neurons want to be in the same state, otherwise in the opposite. Weights of the form w_{ii} are usually set to 0 but it can also be interpreted as the neurons inertia to stay in the same state or switch states.

3. Energy (usually denoted as H)
   The energy of a system can be thought of as a landscape and the current state a ball rolling across that landscape. H is a function that takes in all the current states and outputs a real number ($H : \mathbf{x} \to \mathbb{R}$ where $\mathbf{x}$ is a vector consisting of all components x_i).

4. Learning, Attractors
   Let us pick some states which we want to save in the network $Q = \{\mathbf{x_i}\}$. Each of these states is called an attractor because we will set them so that if the system is in minimal energy (at least a local minimum). This is the learning process where each weight is set.

5. Processing
   Given some initial state $\mathbf{x_0}$ in each time step we change it to get energy that is equal to or less than what we already had. Assuming some fairly general propositions this process is guaranteed to converge in one of the pre trained states or attractors.

Applying these conditions we means we can recall any saved state from a partial or a noisy one given enough time steps. One way to visualise what happens in the neural network is through the mentioned energy landscape and a ball rolling on it as in figure 2. This is however much more complex than a simple graph as the energy is a function that takes n-dimensional vectors as inputs so the graph is actually $n + 1$-dimensional.

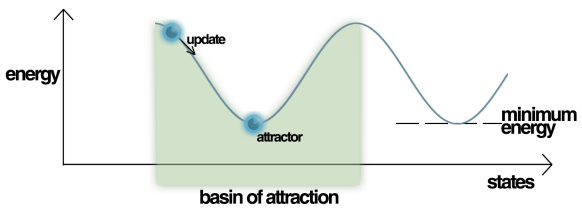

Figure 2: A diagram of what a Hopfield network attractor and energy could be interpreted as. By Mrazvan22 - Own work, CC BY-SA 3.0, link

It is important that all the attractors are different enough. In case they weren't different a "fake" attractor could exists between two similar ones resulting in a new state being encoded (see analysis). However there are ways to ensure this doesn't happen, or at least lower the probability. To move the "ball" from one attractor to another we need to give in "energy". This is done through outside stimulation of some neurons to move it out of what's on the figure called a basin. This will result in the "ball" moving to another attractor and another basin. This is exactly what we will use to achieve a transition between states of Moore's machine.

1.3 Motivation

This started as a university project idea but quickly evolved passed it's original intended design. All of us sometimes think like a finite state machine and we know the brain can work like that so how does it do it? Showing that a model like this can exist shows a way the brain can operate in. For that purpose over-engineered designs found in many other models are not added.

Given what we know sequential memory is a big part of our memory and we know that when learning there are methods based on sequential memory and association. A model like this can help us understand how and why exactly is sequential memory so much better than individual facts. The answer as to why might be obvious because it includes association but the how remains a mystery.

1.4 Application

Besides an interesting perspective in neuronal dynamics this provides a model to implement any finite state machine through a neural network. As bio-processing (computations done on lab grown brains) becomes more and more often encoding big finite state machines on those neural networks becomes a lot more feasible as an option. Along side this keeping the system as plausible as we can also gives us insight into workings of sequential memory in animals

A standard Hopfield-like neural network doesn't have a good capacitance for states but that's because it's a simplification. Changing a few definitions can increase the capacitance for safe storage by incredible amounts. For 3 different states there are about 30 neurons needed for safe storage in the classical Hopfield model. However given those changes which we won't cover with the same 30 neurons an incredible $32,768$ states can be safely stored and retrieved. With standard digital systems with say 30 flip flops the number of possible states is even higher (double the number of revised Hopfield networks), however a system of that size is more prone to errors and if we account for safe encoding and checks the number quickly drops. This is due to perceptrons existing on a spectrum (the total number of states possible to encode is enormous but it stops being safe encoding). The point is that with a perceptron or neuron system which can still emulate finite state machines and binary outputs we can safely encode more information with less infrastructure. The down side is that retraining a neural net is way harder than rewiring a system.

2 Theoretical foundation

2.1 Neuroscience and properties of neurons

Neurons exhibit certain properties which are incredibly useful to us in this scenario. Digital neurons (perceptrons) don't have all of them and that's why in the model later we have to add certain workarounds but we can create a simpler architecture by using these properties

1. All or nothing
   All neurons will either be active or not and will always decide to be in one of the two states in a given time step.

2. Threshold
   Neurons can have a threshold for becoming active. To add to this as neurons exist in space depending on the part of neuron being stimulated the threshold can be different. This is an incredibly useful property perceptrons don't have which we will later use well.

3. Oscillations
   Given a strong enough external input the neuron will enter an oscillatory state. This state can be arbitrarily small but not arbitrarily big. Each neuron has a predefined maximum that depends on it's physical properties.

Perceptrons could be changed to exhibit the same properties but there it's easier to add additional perceptrons and a more "engineered" architecture than what we create from biological neurons. We are striving to achieve a biologically plausible architecture so that means we will use those properties how we can to lower the amount of neurons needed and the complexity of their connections.

2.2 Hopfield Network

Let us look at the theory behind classical Hopfield networks that we will slightly alter so that we get a more convenient structure. All of changes will be seemingly minor but they will reflect strongly into giving us the structure from which it is natural to find a finite state automaton.

2.2.1 Regular

First taking a closer look at the classic definitions so we know what will be changed and how it compares. Symbols will be kept as consistent as possible to those in introduced in the introduction. There will be some mathematical formalisms as to find exact behaviour of the system.

Notice that each weight has 2 indexes meaning we can sort them into a matrix usually denoted W. We know the following properties: $(W)_{ij} = w_{ij}, (W)_{ii} = w_{ii} = 0, (W)_{ij} = (W)_{ji}$. In other words the weights are symmetric and on the diagonal there is only 0. If we define the matrix well than it will represent the energy landscape and attractors well.

Let's start with a single state being encoded. If the state of every neuron i in the state is denoted by x_i what we can do is set the weights using the formula:

$$w_{ij} = x_i \times x_j$$

This will work as we expect it to. if x_i and x_j have the same state in the pattern the weight will be positive (or when noisy they will strive to be the same) and if they are different it will be negative (or they will strive to be different). However given just 1 state we can't do much and especially not encode a finite state machine. So we revise the formula to give the first important equation:

Definition 2.1. *Elements of the weight matrix given n states can be calculated as:*

$$w_{ij} = \frac{1}{n} \sum_{k=1}^{n} x_i^k \times x_j^k \tag{1}$$

This needs some explanation. Given n states $\{\mathbf{x}\}_k$ we calculate the weight like before for each given state and then later take the average of all calculated values for some w_{ij}. This allows us to encode multiple states (x_i^k means i-th neuron on the k-th state pattern).

Theorem 2.1. *Given n states for them to be safely stored they need to be orthogonal to each other.*

This was the training process of creating a neural network. Now given such a weight matrix W we can start retrieving information back from it. The update rule is simple

Definition 2.2. *All neurons will update synchronously based on the current state. The update rule can be written as:*

$$x_i' = sign(\sum_j w_{ij} \times x_j - \theta_i) \tag{2}$$

Here θ_i represents a threshold usually set to 0. This equation looks at the sum of the products of weights and states of corresponding states of neurons. If this sum is non-negative the state is set to $+1$ and -1 otherwise.

Theorem 2.2. *A Hopfield network will always converge to some value, and if stored safely that value is the attractor.*

Looking at the last two theorem we see that given any starting state the Hopfield network will converge to a pre trained state if we choose them correctly.

Theorem 2.3. *The capacity of a network can be given as an equation but it can be estimated as the ratio between the number of vectors stored and number of neurons being approximately* 0.138 *(138 states per 1000 neurons).*

2.2.2 Single layer vs multi-layer

What we saw was a classical Hopfield network that can describe a model of associative intelligence decently fine but we can't create a finite state automaton just from that. There are many modern iterations but we will focus on the *hierarchical associative memory network* although also personalised and edited to fit our needs in the next section.

The important difference between classic and this kind of network is in the fact that we have more networks stacked "on top of each other". Usually we remove the inner layer connections in this model for simplicity. For each neuron in layer i we define it's change function named in definition 2.2 similarly for this. The only difference is that x_j is now the states of all the neurons in the lower layer and w_{ij} also represents cross layer connections.

This architecture has a way higher space requirements but it is also able to process and store more, and more complex patterns than a regular network. What we will look at here is what if we could combine a regular network and a multi-layer one (why each one will be explained in the section 2.3.1.

A more technical definition is that for each neuron we define a weight vector between each layer (or a weight matrix in the end). Here we introduce a new notation $\xi_{ij}^{(AB)}$ (weight between neuron i on layer A and neuron j on layer B). Most sources include a continuous activation state but for now we will remain the discrete version we have already defined. Continuos activation states are a lot more useful and efficient and if a network can be implemented discreetly it can be implemented continuously with less neurons and states but the equations become a lot harder to write and find some properties.

A topic of a follow up paper might be reiterating this idea to continuos state variables as they are a lot more interesting computationally wise. However it is important to note that neurons (the biological kind) have the property "all or nothing" meaning that the neuron will either be active and create an action potential or not (excluding repetitive firing).

Definition 2.3. *The change function can be rewritten now for our case as:*

$$x_i^A = sign(\sum_j \xi_{ij}^{(A,A+1)} \times x_j^{A+1} + \sum_j \xi_{ij}^{(A,A-1)} \times x_j^{A-1} - \theta_i^A) \tag{3}$$

Or alternatively using continuos state variable we will see derivatives but that is out of our current scope.

2.3 Moore's Machine

Let us take a closer look at Moore's machine and some properties it has that are of use to us. Firstly through the classic definitions and perspectives, then through how we can do digital logic with just neurons.

2.3.1 Finite state automaton - mathematical and digital logic definitions

As stated earlier we know all the components and a scheme of how Moore's Machine works. There are however a few questions to be answered. Why Moore's and not a Mealy machine? What about any finite state automaton other than those two?

Moore's machine was chosen because it's outputs depend only on the current state which can exist more simply compared to Mealy's which depends on the current state and input. This connection is harder to achieve but the machines are actually equivalent as we will see.

Theorem 2.4. *Any finite state machine can be remodelled to exhibit the same behaviour but work as a Moore's machine.*

By this theorem any finite state machine we can think of can be modelled as a Moore machine and consequently by this paper as a neural network. This is an incredibly useful theorem because it is the backbone of the applicative part of our proposed architecture. Also by the same theorem if we can prove that a neural network can exhibit the behaviour of an arbitrary finite state machine automata theory starts applying. Given a more permanent memory we also begin to etch a path towards full Turing machines.

Moore machines are defined by registers as memory cells and combinational logic between them. What we can do is create a multi-layer network where the input layer is our base. This base influences the

state register. Our state register will be a classic Hopfield network. With this we have more freedom and escape precise encoding. From there everything else is encoded as layers on layers of networks communicating between them to create the diagram we already had. The output is also a network from which we read data.

A later used term is minterm so it should be clarified. Take any Boole's function of arbitrary many variables. What we call a minterm is a product of all the functions variables or their complements. An example of a function $f(A, B, C)$ would have minterms $ABC, A\overline{B}C, \overline{A}B\overline{C}$. For a function of n variables there are in total 2^n minterms. Adding certain minterms allows us to replicate any Boole's function in it's outputs exactly and that representation is usually called sum of products form.

2.3.2 Biologic?

To apply more complex connections it could be useful to see if we can create some components we would find in digital logic from neurons only.

We will demonstrate this as having two input neurons connected to an output neuron. Just by setting weights we can create some logic so let's see. We don't need the weights to be strictly symmetric here.

1. OR gate
   Have the two input neurons connected to the output neuron with equal positive weights. When only one of them is active the sum from 2.2 will be 0 and the output neuron will be active.

2. AND gate
   We will use almost the same setup as before but our output neuron will now have a higher threshold. The only requirement is that the threshold has a value between $\max(w_1, w_2)$ (if we call those the weights) and $w_1 + w_2$.

3. NOT gate
   We need 1 input neuron with a negative weight to the output neuron.

With this we have the set of operations $\{AND, OR, NOT\}$ which we know is the complete set of all operations to create any component. However let us look at some more components that can be useful:

1. Clock pulse
   Neurons by default (the biological kind) can enter a state where they become active periodically. The frequency of this firing can be manipulated between 0 and the neurons maximum, so we can achieve a simple clock.

2. Write enable
   Using the clock pulse and an AND gate we can achieve a simple write enable function giving us access to synchronous modes of operation. Although by default all neurons (biological) have a "reset time" in which it can't be stimulated by another in any way.

This is some of biological logic. More complex systems can be achieved and as we can see some more complex systems (like clocks) can be achieved really easily with neurons. We won't go through more definitions like these for now.

One more important topic is redesigning the architecture to be more compact. What would happen if we had 2 input neurons and one of those was treated as output as well? Raising this question gives us a simpler view on biologic we have introduced earlier.

1. NOT gate
   Not gate actually doesn't change at all because it doesn't have to it's already two neurons and it's incredibly simple as it's a by-product of neuronal design.

2. OR gate
   If we connect the first neuron to the second with a positive weight and look at the second neuron as the output we achieved an or gate as long as the threshold is 0 or close to 0.

3. AND gate
   Connect the first neuron to the second with a positive weight. We need to clarify that the second input neuron gets stimulated independently closely to where the first neuron is connected. Then by setting a good threshold we achieve an AND gate where on the second neuron we can read the output.

3 Model

3.1 Overview

We have mentioned some pieces of how this model might work but it's time to look at some specifics. Maybe a backwards order but we will firstly look at the output and states and then input and transition but this way we can get rid off the simpler part.

3.2 States and output logic

3.2.1 Training states

This section is actually the simplest. By the definition 2.1 we can make those our weights. In that way no matter the input we will converge to a state we pre programmed it to.

We only need to worry that our conditions for safe storing are met and after that the memory register is prepared for work. This does not account for any transitions just to ensure it works as intended.

3.2.2 Possible outputs

The maximum number of possible outputs is determined by the number of states. We know that the output is a function of only the current state so that's why we can't have more. This can be done in two different ways depending on the preferential of the designer.

1. Single neuron output
   For each state we have a single output neuron which fires when the machine is the specified state. This is equivalent to a decoder. One simple way to determine this is with a simple derivative of the learning equation. We know that for a given state k we know we want that neuron to be in the active state so the learning rule (and consequently it's output) as:

$$x_o^k = sign(\sum_j x_j^k \times x_j^c - \theta_0) \tag{4}$$

   Where $\theta_0 \in < n-1, n >$ and n is the number of neurons (equivalent to the definition of a really big AND gate). Here x_o^k is the state of the k-th output neuron, x_j^k is the state of the j-th neuron in the k-th state and x_j^c the current state of j-th neuron. This is simply visible by setting that for the k-th state we want the x_o^k to be active, or 1 in the equations. This simplifies to what we have in (4).

2. Network output
   Here we have two "subdivisions". We can take the states to be the output directly or for the output to be another network.

   (a) Direct output
   This is pretty self explanatory but is pretty restrictive. Due to the way the states are chosen to interpret them as output data requires additional logic we can build in so we won't spend much time on this.

   (b) Output network
   Using what we know of from the single output neuron and how we generalised one state encoding to multiple state encoding in classic Hopfield networks we may generalise those equations and achieve an output layer which codes from states to more readable and interpretable data as previously defined.

All of output solutions presented here are single layer logic so they are incredibly simple and biologically plausible. We did not require any special use of gates or logic that would complicate the setup.

3.3 Input layer and transition logic

3.3.1 Input layer

We have to make a choice here. We can do direct stimulation on the state register or have a separate layer for user input. Here we opt for a separate layer as by now it might be obvious they are actually equivalent but a separate layer gives us more freedom. Let us explore a new way of connecting multi layer networks.

In passings we might have mentioned that the weight of a neuron to itself is 0 by default but what would happen if we were to set it to a non zero value? This now has significance. Let say that the self connected weight w_{ii} is strictly positive. This is inertia for the neuron to stay in the state it already is. A quick thought might think that changing the threshold does a similar thing but that's actually not correct. A negative self weight presents itself like a clock mechanism so for digital implementation it can be used like so with extra dampening if other neurons are connected to it. For our current purposes we want a strictly positive value or 0 for self weight.

Remark. If we connect each respective neuron from the input layer with a positive weight and all others from the input layer to 0 (that is to say connect only i-th neuron in the input layer to i-th neuron in

state register) and set the self weight to 0 we get our simpler option of direct input.

Any other configuration of weights and self weights gives us an interesting formulation for weights and self weights gives us options for transitioning states. With direct input we can only manually switch states and that's not as fun but marginally useful. However that is not the point of finite state machines at the slightest. It is important to note that we don't necessarily want symmetric weights. In general this means that convergence proofs are a lot harder but here we don't have to worry.

3.3.2 Transition logic

Using the setup we have from input layer architecture we can define some new transition logic. Here into play comes biologic we defined earlier combined with physical properties also described earlier. As we will see we can achieve sum of products architecture without additional layers of neurons when using that instead of the digital implementation.

The system will work by alternating. We know a real neural network can do that, however a digital one has to have additional programs to achieve that. From here on out we will have to differentiate a biological system and a digital one for writing data in.

1. **Digital neural network**
   From the input layer we will insert a simple middle layer. We have shown that an OR gate is quite simple in construction and doesn't require any change to the output neuron (here the neuron in the state register). Before there we will connect an arbitrary amount of inter layer neurons and state register neurons which act as AND gates from the input layer and the output neuron. This makes it equivalent to a minterm. With an arbitrary amount of those connected by an OR gate we have a sum of minterms. This means we can replicate any logical functions. We will see that biological neurons have a certain property that allows them to have two thresholds depending on the connection.

2. **Biological network**
   As mentioned a real neuron can have two separate thresholds depending on the part of neuron where the connection resides. This means we can have those minterms on individual thresholds on seperate parts of the neuron. Have logical functions implemented as sums of products are a simple step from that because any threshold reached causes the neuron to become active. This means any function can be made directly just by having inputs and state registers. With this the connection is so simple that it's biologically probable.

State registers operate in 2 alternating states which can be operated by a clock which we already know is super simple to create. First state is entering data second state is converging to an already stored state. Having the second state means that transition logic can be simpler but given that we have arbitrary Boole functions we don't even need it. That means that the state register doesn't have to have a second mode of functionality that we already mentioned. Given that the transition logic is the most complex part and we showed that any arbitrary transition can be achieved means that we can have an arbitrary Moore machine. By extent we have showed that any finite state machine can be achieved.

3.4 Implementation details

One important note is that of biological plausibility, having exact Boole's function would require that all the neurons are connected twice (once for the Hopfield, once for the input). In the digital version that is not an issue however that's not entirely true for the biological kind. Every neuron is connected only to a subset of all the other neurons which are close to them.

We can't have the neurons reuse connections because of thresholds as discussed previously. But what we can do is leave the connections to the Hopfield network and only take a few double connected neurons. For some neuron x_i all the neurons in its neighbourhood will be connected to it twice. This will allow us to specify partial Boole's functions of transitions but thankfully due to the Hopfield network we will converge to the states we want.

4 Analysis

Now that we have looked at how a model might operate there are questions to be answered. The main one revolving around why would someone do this and is there benefit to using standard digital logic.

4.1 Performance

The main benchmarks used for analysis are speed and space performance. Looking at them we can determine if this architecture is beneficial to us in any way.

4.1.1 Speed performance

Speed of this system is actually incredibly high. This is because neurons on their own are incredibly fast. If one looks at the architecture they will notice that the speed of the network doesn't necessarily depend on the size.

Using the architecture option which doesn't need the so called second mode of operation (no need for minimisation in the state register) the time is constant. The latency we can expect is only three times that of the latency needed for neurons to transmit information. This makes a more complex connection between inputs and state registers but it's incredibly fast.

If we take the simpler connections the latency increases linearly by the number of steps it takes to reach convergence which is not necessarily tied to size still. The independence from size in time is really useful in any case.

4.1.2 Space requirements

Spatially this architecture is not too well designed. However this is only because we used a discrete version of architecture. A lot of the rules don't depend on the fact that there is a binary choice but can be implemented to use continuos variables. This kind of architecture is a lot more spacially efficient.

Spacial requirements are directly tied to the capacity of state registers. If we use the optimal method we can achieve a really high capacity and therefore low spacial requirements as well. Input network size is actually mostly independent on the state register size and could be a single neuron, while the output network depends on the users choice. If the output is taken to be the current state then we don't even have to have an extra network meaning that the spacial requirement is actually only in a single network which again is not too large.

4.2 Stability of attractors

4.2.1 Stable and unstable attractors, "fake" attractors

There was mention of "fake" attractors and fake storage but it's important to clarify for what that actually means. From this we may derive an idea for capacity of a network and work back to the space requirements which were presented as low but that's under the assumption of good storage. Firstly let's clarify the word "fake" attractor. Attractors are states to which a system will converge. They can be stable (if the system is in the neighbourhood of the attractor the system will converge to it) or unstable (if the system is in the neighbourhood of the attractor the system will diverge away from it). And importantly every state we want to encode will be a stable attractor so that the system converges to one of them. But that doesn't mean that every attractor will be a state.

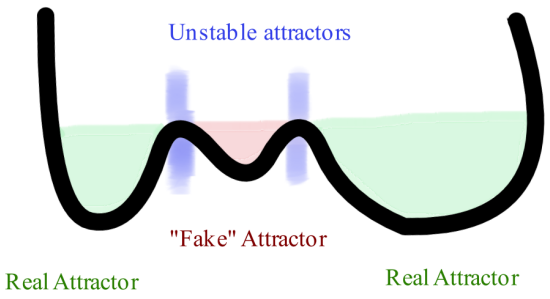

Figure 3: A diagram including real and "fake" attractors

If two states are too close in the energy space a "fake" attractor will form between them. An example is shown in figure 3. Here two real attractors (green) meet in the middle and create a new basin for the "fake" one (red). This is dangerous because the system could settle in that basin instead of a real one giving us the wrong output and wrong transition. Theoretically possible, but practically not so much, are unstable attractors that exist between each two real attractors and the system could converge to them but the chances are incredibly low. We take unstable attractors not to be an issue but the "fake" stable attractors are.

5 Discussion and Conclusion

5.1 Overview

We have shown that any finite state machine can be represented as a neural network. This gives us the ability to apply automata theory onto neural networks, translate those machines to neural networks for faster computation time and insight into how

5.2 Comparison to standard digital logic

In comparison to standard digital logic we will take a look at speed, spacial requirement and error performance. We figure these three metrics are the important metrics when comparing two machines.

1. Speed
   Neurons operate fast, but so do logic gates. Both operate in similar time scales, however the bigger the network gets and the more complex transition function we have the standard implementation starts working slower. However this is not an issue for neurons due to convergence being really fast, and transition logic constant no matter the size.

2. Spacial Requirement
   Here standard implementations have the advantage, sort off. What's important here is how many states can be safely stored. This will depend on the error handling in the next property also. Essentially both systems trade off space for error handling but standard digital systems still have an advantage compared to neural networks.

3. Error Handling
   Error Handling includes ensuring that the state register doesn't get states which aren't encoded and fixing those if it does happen. Standard digital logic for large machines is prone to errors occurring known as static and dynamic hazards, or failures of electronic components. If we add error handling the speed of standard systems also drops due to checks needing to be processed. The neural network has a built in error handling system, and as explained in implementation details we actually use the said system to make it realistic.

Here we see that neural networks trade off some space for speed which is a common type of redesign. This is not ideal for every system and situation but it is more common that the reverse. The inherit error handling of the architecture also makes it a step above the standard system with minimal trade off on time. That is extremely useful in any architecture or project which is why the whole system is based around it.

5.3 Sequential memory in humans

What was the point of doing so many biologically plausible architecture decisions? Considering this model is entirely biologically plausible it could explain how sequential memory in humans, and other organisms with complex brain structures could operate when dealing with sequential data.

Humans are better at memorising connected items than random items and that is due to associative memory. One thing reminds of another. A special case of these are sequences which are even easier to remember. You might also notice that you rarely mess up a sequence when properly remembered. Putting this kind of architecture inside memory blocks of the brain could explain why sequential memory is common and works better than regular associative memory. It might be imagined as steering the associative memory block with the input which can be the clock pulse.

5.4 What's next

This article treats digital neural networks too harshly. Using modern networks which have continuos activation values can make them better than regular neurons. Besides this we should also not have a fully connected Hopfield network as after some point it becomes unrealistic for all of them to be connected.

Proving convergence and properties for the above mentioned situations is rather difficult so we are using a simpler model. With proofs what I can say is that the biological network will get somewhat worse but not too much, and the digital neural network will get a lot better. This is left as a topic for a new paper.

A Proofs

Here we will introduce extra maths such as to clarify some of proposed theorems.

A.1 Theorems in section 2

A.1.1 Theorem 2.1.

Let us recall this theorem first:

Theorem A.1. *Given n states for them to be safely stored they need to be orthogonal to each other.*

This is actually a theorem that in theory of Hopfield networks is introduced incredibly late. Orthogonality requires some sort of inner product but what we can simplify to is to imagine that the state ξ^i is a n-dimensional vector in $\mathbb{R}^n$. Then we want $\xi^i \dot{\xi}^j = 0$ where this is just a regular dot product. If this is true for all states then each state can be recalled without error. For proof please see:

A.1.2 Theorem 2.2.

Theorem A.2. *A Hopfield network will always converge to some value, and if stored safely that value is the desired attractor.*

We can find the energy function as a stability function. What we see is that the energy either decreases or stays the same. If all states are stored safely and there are not fake attractors and given that they will be recalled perfectly it means that the system will always recall information no matter the given state.

A.1.3 Theorem 2.3.

Theorem A.3. *The capacity of a network can be given as an equation but it can be estimated as the ratio between the number of vectors stored and number of neurons being approximately* 0.138 *(138 states per 1000 neurons).*

A lot of maths is being skipped but it shows that for 99% certainty this ration is in fact around 0.138. This isn't a specific number but rather a chosen value and it is shown that the capacity is approximately linear.

A.1.4 Theorem 2.4.

Theorem A.4. *Any finite state machine can be remodelled to exhibit the same behaviour but work as a Moore's machine.*

This theorem is the backbone of this whole paper. This theorem is pretty simple to prove. Any finite state machine has the same components as a Moore machine just the only difference is the possibility of μ depending on the input, as well as the current state. In such a case we introduce "intermediate" states which act to simulate a function depending on the state and the input.

B Code samples in python

Here we will introduce some code samples of Hopfield networks in python which can make the reading abstract ideas simpler.

Code for generating weights. Here *valid* is a list of all valid states which are lists. Here l is the number of neurons, while n is the number of states. If you look at it closely you will see the code generates a matrix W as defined before

```
def createWeight(valid, l, n):
    w = []
    for i in range(l):
        w.append([])
        for j in range(l):
            wij = 0
            if i != j:
                for k in range(n):
                    wij += valid[k][i]*valid[k][j]
                wij /= n
            w[i].append(wij)
    return w
```

Code for changing the current state is as follows. Here s is a list of all the neurons in the current state, s_{new} is the new state, l is the number of neurons and W is the above defined matrix.

```
def energy(s, l, W):
    s_new = []
    for i in range(l):
        s_new.append(0)
        sumTemp = 0
        for j in range(l):
            sumTemp += W[i][j]*s[j]
        if sumTemp >= 0:
            s_new[i] = 1
        else:
            s_new[i] = -1
    return s_new
```

How a regular Hopfield network might work is as follows:

```
while 1:
    input("Press enter to do an itteration")
    print("-------")
    print("Current:", s)
    state = change(valid[s], I)
    f = 0
    while f == 0:
        state = energy(state, l, W)
        for i in range(n):
            if comp(valid[i], state):
                s = i
                f = 1

    print("Balanced:", s)
```

What happens here is a little more complex. s now represents the index of the valid state we're in from the variable *balid* as before. The function change takes in the current stable state and some vector of change I and computes a change in a separate function. Then the energy function is applied to the temporary state until it resembles one of the valid states. This is not the final architecture just how a regular Hopfield network might work.