ENPH 455 Final Report

In Situ Training of Deep Neural Networks Using Continuous Dynamics

Adam Grace

A thesis submitted to the

Department of Physics, Engineering Physics and Astronomy

in conformity with the requirements for

the degree of Bachelor of Applied Science

Queen's University

Kingston, Ontario, Canada

April 2021

Abstract

Advances in artificial neural networks and deep learning have supported some of the most exciting solutions to modern engineering challenges. Supervised learning of deep networks is a popular machine learning subfield, which can take up to weeks to perform on power-intensive, state-of-the-art digital hardware. Neuromorphic computing presents an exciting opportunity to accelerate training using the physics of real systems. This thesis proposes a novel approach to training deep neural networks in situ without ever converting to the digital domain. An additional feedforward neural network and multiple continuous-time recurrent neural networks are also built in situ to compute and control parameter updates according to the direct feedback alignment algorithm. Principles of the neural engineering framework are leveraged to continuously evolve and decode the network parameters according to their defined dynamics. The proposed approach to training was tested using the benchmark XOR gate training task. Unfortunately, the hidden layer parameter dynamics could not be successfully implemented with the neural engineering framework, so an intermediate digital computation was included. This flaw was attributed to a poorly selected encoding strategy and is not fundamental to the proposed approach. The continuous dynamics of the neuromorphic system was simulated using digital computing and discrete timesteps. The XOR gate network successfully trained in 800 simulated seconds using an arbitrary learning rate of 50. This indicates that this approach has the potential to accelerate supervised learning if the selected neuromorphic implementation allows for very large learning rates.

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Dr. Bhavin Shastri for his unending support over the past year. I've never had the complete freedom to take a project in any direction, and I appreciate your faith in my steering. You taught me that no topic is too complicated to learn and no idea is too inventive to explore. I look forward to joining Shastri Lab as a graduate student this fall and continuing our exciting investigations.

I began this project with very little background knowledge and was fortunate to learn from some extraordinary people along the way. In particular, thank you Dr. Alex Tait and Matthew Filipovich for sharing your invaluable ideas and research. I am also grateful to all the teaching staff at Queen's University who's unwavering dedication to student success has guided me through these last four years.

Thank you to my family for your unconditional love and support throughout my entire life. I could never be in my current position without it. Lastly, I would like to thank my friends Matt, Travis and Adam for unintentionally leading me towards this thesis topic. Your interest and enthusiasm towards artificial intelligence encouraged me to look outside my comfort zone and explore a new field.

Contents

List of Tables

List of Figures

1. Introduction

In recent times, artificial neural networks (ANNs) have re-emerged as popular information processing models for research and real-world applications. In particular, deep neural networks have shown great success in learning patterns from abstract data representations, enabling them to perform complex tasks such as classification, speech and image recognition, prediction and recommendation [1]. Recently, DeepMind's AlphaGo computer program was even able to beat a world-class human player at Go by using deep neural networks [2]. Although ANNs are most commonly used for machine learning, they can also perform a variety of computing tasks such as optimization and function approximation [3] [4].

Supervised learning is the most common form of machine learning, where a deep network is trained using a set of known inputs and desired outputs. For example, the network may be presented with the greyscale pixel values of an image and the network should indicate if there is a person in the image or not. After training on a known dataset, the network should function properly even for inputs that it has not encountered before. Training presents the most computationally intensive step in creating a deep neural network, with some programs requiring hours, days or even weeks of training on state-of-the art hardware [1] [5].

Neuromorphic computing presents an interesting alternative to digital implementations of ANNs. It is often referred to as brain-inspired because it attempts to achieve similar performance and energy efficiency as human brains by adopting some principles of neural biology [6]. The networks are built using real, physical systems instead of abstract digital representations in computer memory. Data is represented by analog signals so that fast computations can be performed in parallel. Established approaches to neuromorphic computing include analog electronics [7], CMOS electronics [8] and silicon photonics [9].

The goal of this thesis is to leverage neuromorphic computing to implement an existing supervised learning algorithm. Training should be performed entirely in situ, without requiring conversions to the digital domain. The validity of the neuromorphic system and training strategy will be testing on the benchmark XOR gate training task, using digital simulations with discrete timesteps.

2. Neural Networks Background

2.1 The Neuron Model

Artificial neural networks consist of individual neurons that receive and transmit signals to one another. Each node (synonymous with neuron) receives signals from several others, so that the input to a node is considered to be a vector. Each input connection has a scalar weight associated with it, so that a weight vector exists with the same rank as the input vector. The dot product of the input vector and the weight vector is the weighted input to the node, which is offset by a scalar bias term to produce the node's linear activation. A non-linear activation function is then applied to this linear activation to generate the node's output. The following equation describes the behaviour of a neuron denoted by *j*, where a_j is the output, $\vec{w}_j$ is the weight vector, $\vec{x}_j$ is the input vector, θ_j is the bias term, z_j is the linear activation and σ_j is the non-linear activation function [10].

$$a_j = \sigma_j(\vec{w}_j \cdot \vec{x}_j + \theta_j) = \sigma_j(z_j) \tag{1}$$

Non-linear activation functions will usually output about zero if the linear activation is less than zero, and about one if the linear activation is greater than zero. A way to interpret this is that the neuron is "activated" if its weighted input is great enough to overcome the bias term. In many networks, all neurons will share identical non-linear activation functions. A depiction of the neuron model and the common sigmoid activation function are shown below.

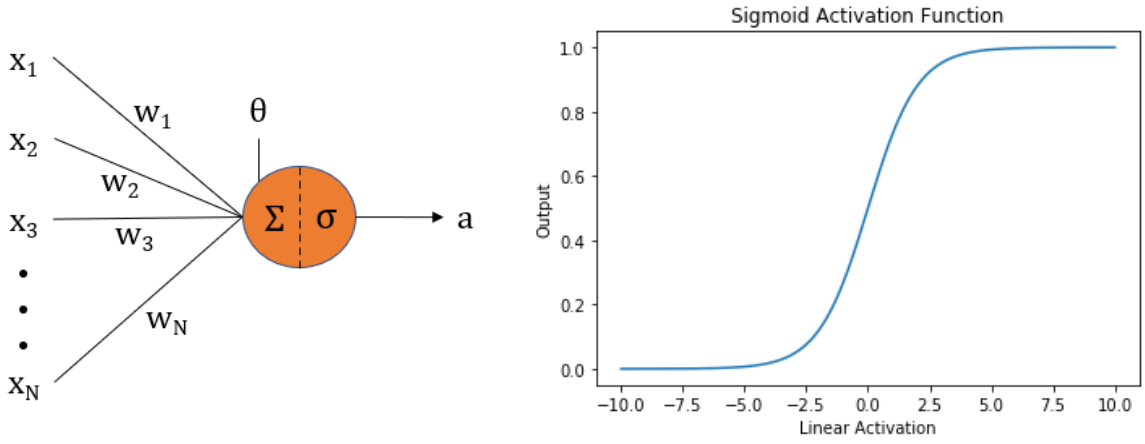

Figure 1 – Left: Model of an artificial neuron. Right: Sigmoid activation function.

The output of each neuron will either become an input to one or more other neurons or an output of the entire network. The network architecture determines which neurons in the network each neuron is allowed to connect to.

2.2 Feedforward Neural Networks

2.2.1 General Feedforward Architecture

Feedforward architectures require that neurons are grouped into sequential layers. A neuron in a certain layer can only receive inputs from neurons in the previous layer and can only send outputs to neurons in the next layer [11]. Signals can therefore only propagate in one direction and the network will quickly stabilize. Neurons in the input layer do not have a non-linear activation function. They simply distribute the external inputs to the next layer. The last layer, called the output layer, provides the result of the computation. There can be any number of layers between the input and output layers, which are called hidden layers (synonymous with deep layers). There can also be any number of nodes in each layer, however the size of the input and output layers is determined by the computing task itself. Since each node has a weight vector and a bias term, and each layer includes multiple nodes, we associate each hidden and output layer with its own weight matrix and bias vector.

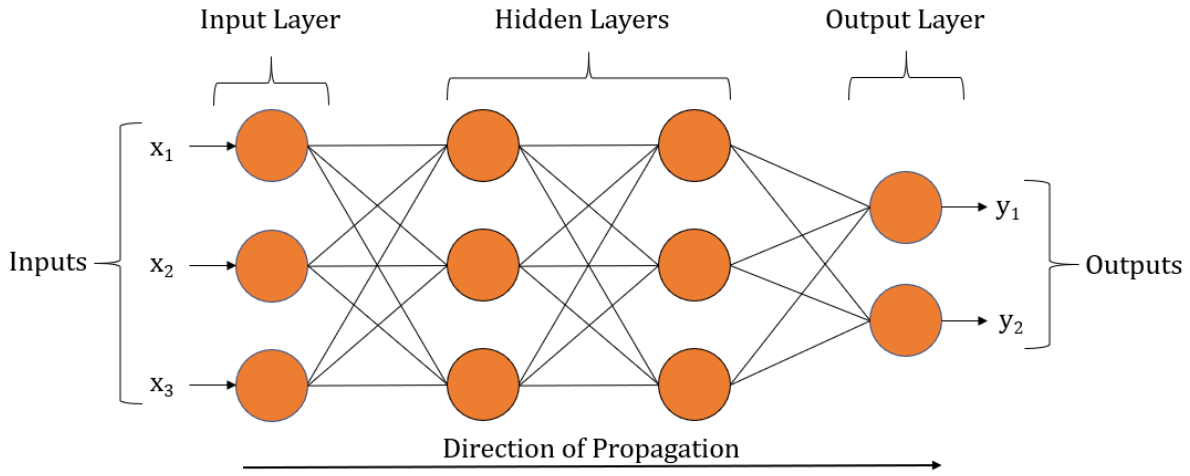

Figure 2 – Depiction of a feedforward neural network with two hidden layers.

These networks are often used to extract information from a given set of inputs [12]. Classification is a common example, where a known input is provided, such as the normalized RGB pixel values of an image, and a binary output is produced, such as if there is a cat in the image or not.

2.2.2 Direct Feedback Alignment

Before a feedforward neural network can function as intended, its weights and biases must be adjusted through training. The network will need a diverse set of inputs and desired outputs so that it can learn patterns from the data. For a given pair of inputs and desired outputs, the following equation defines a quadratic cost function for the network, where *L* denotes the output layer, *j* denotes the specific node in that layer, a is the node output and y is the desired output.

$$C = \sum_j \left(a_j^L - y_j^L\right)^2 \tag{2}$$

The goal of training should be to minimize C over the entire data set. Backpropagation is the most widely-used training algorithm, which iteratively adjusts the weights and biases along the negative gradient of the cost function (stochastic gradient descent) [11] [13]. An important observation about backpropagation is that it makes use of the transpose of each layer's weight matrix to propagate error from the output layer towards the input layer. Every iteration of training will change these weight matrices, making this algorithm difficult to implement using fixed neuromorphic hardware.

Direct feedback alignment (DFA) is an uncommon alternative that has shown similar results to backpropagation on the MNIST data set [14]. It makes use of constant, random feedback matrices that propagate the error in the output layer directly to each hidden layer simultaneously [15]. The first step in the algorithm is to compute the error in the output nodes using the following equation, where σ' is the derivative of the non-linear activation function and $\odot$ represents the Hadamard product.

$$\vec{e} = \frac{\partial C}{\partial \vec{z}^L} = 2(\vec{a}^L - \vec{y}) \odot \sigma'(\vec{z}^L) \tag{3}$$

Weights and biases in the output layer are adjusted using the following equations, where *L-1* denotes the last hidden layer, *j* denotes the node in the output layer, *k* denotes the node in the last hidden layer and μ is a constant learning rate.

$$\Delta w_{jk}^L = -\mu a_k^{L-1} \vec{e}_j \tag{4.1}$$

$$\Delta b_j^L = -\mu \vec{e}_j \tag{4.2}$$

The error in the output nodes is used to calculate the error in the hidden nodes using the following equation, where $\boldsymbol{B}^l$ is a constant, random feedback matrix with appropriate dimensions for layer l.

$$\vec{\delta}^l = (\boldsymbol{B}^l \vec{e}) \odot \sigma'^{(\vec{z}^l)} \tag{5}$$

Weights and biases in the hidden layers can now be updated using the following equations.

$$\Delta w_{jk}^l = -\mu a_k^{l-1} \vec{\delta}_j^l \tag{6.1}$$

$$\Delta b_j^l = -\mu \vec{\delta}_j^l \tag{6.2}$$

Individual updates are kept small using an appropriate learning rate. After iterating through all training samples multiple times, the trained network is tested using a separate data set. The training procedure pseudocode is shown below for conventional digital computing.

```
Import training_data and testing_data
Initialize neural_network with random weights & biases
Create a constant, random B matrix for each hidden layer
for epoch in num_epochs:
        for sample in training_data:
                Compute output of neural_network using inputs from sample
                Compute error in output layer nodes
                Compute error in hidden layer nodes
                Update all weights & biases
Test trained neural_network on testing_data
```

Figure 3 – Digital direct feedback alignment training program pseudocode.

2.2.3 Universal Approximation Theorem

According to this theorem, a feedforward neural network with at least one hidden layer can approximate any continuous function [16]. The number of hidden layers and nodes in the hidden layers will affect the achievable approximation accuracy. For example, one could create a network with a single hidden layer with five hidden nodes to approximate the function $f(x, y) = x^2 + y$. The output node must have a linear activation function, or no activation function at all.

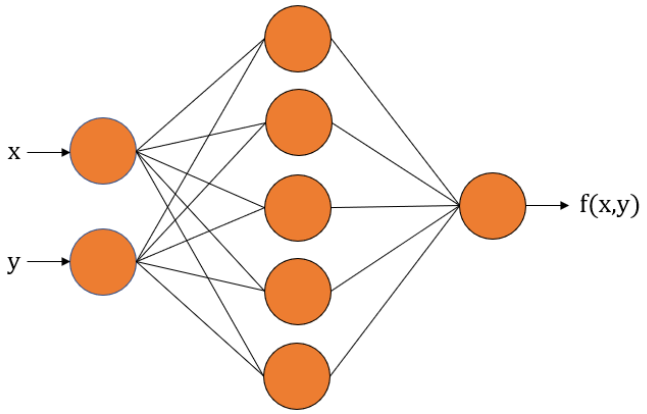

Figure 4 – Example feedforward neural network approximates a scalar function of two variables.

To train such a network, a data set must be generated using the known analytic expression for the function over a range of relevant inputs. Any appropriate training algorithm can be used, such as backpropagation or DFA. This approach even allows for the approximation of solutions to certain differential equations [17] [18].

2.3 Continuous-Time Recurrent Neural Networks

2.3.1 General Continuous-Time Recurrent Architecture

Continuous-time recurrent architectures allow a hidden neuron to send and receive signals from any other hidden neuron, including itself. Because recurrent connections introduce feedback, these networks will not immediately stabilize to a constant output, contrasting with feedforward architectures. The input and output layer neurons can connect to any number of the hidden layer neurons. The following figure represents an all-to-all connected CTRNN.

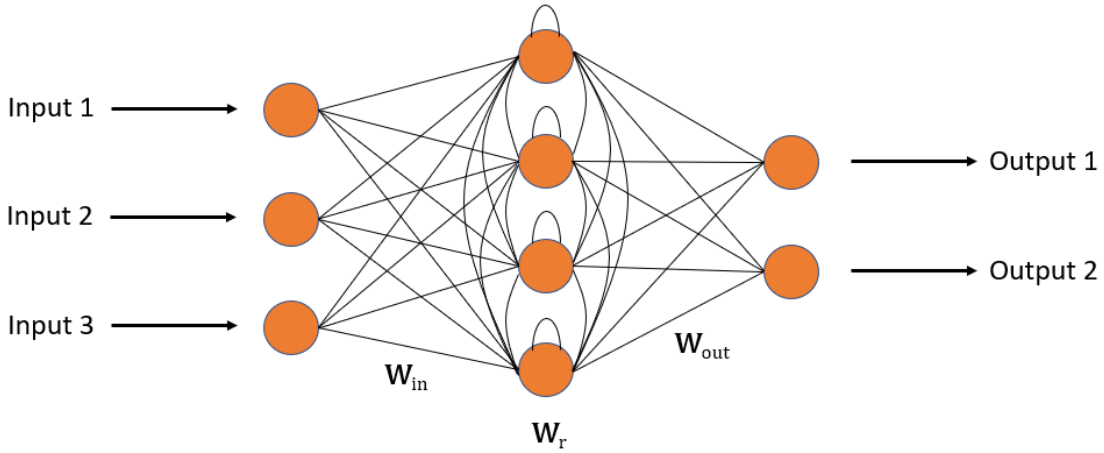

Figure 5 – Depiction of a continuous-time recurrent neural network.

The state of the hidden neurons, $\vec{s}(t)$, is controlled by the following system of coupled ordinary differential equations (ODEs), where τ is the neuron time-constant, $\boldsymbol{W}_r$ is the recurrent weight matrix, $\boldsymbol{W}_{in}$ is the input weight matrix and $\vec{x}(t)$ is the input vector [19].

$$\tau \frac{d\vec{s}(t)}{\mathrm{d}t} + \vec{s}(t) = \boldsymbol{W}_r \sigma\big(\vec{s}(t)\big) + \boldsymbol{W}_{in}\vec{x}(t) \tag{7}$$

The continuous state of a CTRNN hidden neuron is analogous to the discrete linear activation of a traditional neuron. If the net input to a hidden neuron is zero, that neuron's state will decay back towards zero at a rate determined by τ. Simulating a CTRNN with conventional digital computing often involves employing the Euler method with small, discrete timesteps.

2.3.2 The Neural Engineering Framework

Although it was developed as an approach to designing and simulating neurobiological systems, the Neural Engineering Framework (NEF) has recently been used to perform more general computing tasks [20]. Information is encoded into and decoded from the hidden neurons of a CTRNN and the recurrent matrix is designed to produce specific dynamics [21]. Consider a vector $\vec{x}$ with dynamics that are analytically known as $\dot{\vec{x}} = f(\vec{x})$. The vector $\vec{x}$ is represented by the CTRNN hidden neuron states $\vec{s}$ at any given time. The encoding strategy is given by the following equations, where for the i^{th} hidden neuron, g_i is a constant gain term, $\vec{e}_i$ is a constant encoding vector and b_i is a constant bias term [22].

$$s_i = g_i \vec{e}_i \cdot \vec{x} + b_i \tag{8.1}$$

$$\vec{s} = \boldsymbol{E} \cdot \vec{\boldsymbol{x}} + \vec{\boldsymbol{b}} \tag{8.2}$$

The neurons apply a non-linear activation function to their state to generate their outputs. A decoding strategy is then applied to estimate the vector $\vec{x}$ and its derivative vector $f(\vec{x})$ at any given time using linear combinations of the neuron outputs.

$$\hat{\vec{x}} = \boldsymbol{d}^{(\boldsymbol{x})} \cdot \sigma(\vec{s}) \tag{9.1}$$

$$\hat{f}(\vec{x}) = \boldsymbol{d}^{(\boldsymbol{f})} \cdot \sigma(\vec{s}) \tag{9.2}$$

An arbitrary decoding matrix $\boldsymbol{d}^{(\boldsymbol{h})}$ depends on the encoding strategy, σ, and the function h. Nengo is a useful Python package that implements the NEF and uses the Moore-Penrose pseudoinverse

method to find these decoding matrices [21]. The recurrent weight matrix is then designed so that the complete CTRNN is described by the following system of coupled ODEs.

$$\tau\dot{\vec{s}} + \vec{s} = \boldsymbol{W}_r \cdot \sigma(\vec{s}) \tag{10.1}$$

$$\boldsymbol{W}_r = \boldsymbol{E}\left(\tau\boldsymbol{d}^{(f)} + \boldsymbol{d}^{(x)}\right) \tag{10.2}$$

The neuron states now evolve so that the encoded vector $\vec{x}$ can be continuously decoded using $\boldsymbol{W}_{out} = \boldsymbol{d}^{(x)}$. The important benefit of this strategy is that $\boldsymbol{W}_r$ and $\boldsymbol{W}_{out}$ are computed just once and then the CTRNN automatically solves the dynamics $\dot{\vec{x}} = f(\vec{x})$.

3. Proposed Approach to Training

The proposed solution is centered around the main neural network being physically implemented with neuromorphic computing so that the output of each node can be continuously measured in real-time. Modifying a network weight or bias will then change the node outputs and their measured values almost instantaneously. To train this network, the DFA training algorithm will be uniquely implemented using several other in situ neural networks. As discussed earlier, the first step in the algorithm is to calculate the error in the output nodes. For simplicity, we will assume that the main network exclusively uses the sigmoid non-linear activation function and has a single output node. The sigmoid function, $\sigma(z)$, is convenient because its derivative is simply $\sigma(z)\big(1-\sigma(z)\big)$. We can therefore compute the error in the output node with the following equation.

$$e = 2(a^L - y)\sigma'(z^L) = 2(a^L - y)a^L(1 - a^L) \tag{11}$$

This computation could be performed using digital computing, however that would require converting from the analog domain to the digital domain. Instead, we make use of the Universal Approximation Theorem and approximate this function using another feedforward neural network built in situ. Once trained, this network will output the approximate error in the output node as an analog signal in almost real-time. Next, the DFA weight and bias update equations are made continuous to produce analytic expressions for their derivatives.

$$\dot{w}_k^L = -\mu a_k^{L-1} e \tag{12.1}$$

$$\dot{b}^L = -\mu e \tag{12.2}$$

$$\dot{w}_{\mathrm{jk}}^l = -\mu a_k^{l-1}\big(B_j^l e\big)a_j^l(1 - a_j^l) \tag{12.3}$$

$$\dot{b}_{\mathrm{j}}^l = -\mu\big(B_j^l e\big)a_j^l(1 - a_j^l) \tag{12.4}$$

Every non-constant term in these expressions (node outputs and e) is available as a real-time analog signal. Using the NEF and neuromorphic computing, each weight and bias can be encoded into its own in situ CTRNN that obeys the appropriate dynamics [22]. The following figure depicts such a CTRNN for a hidden layer weight.

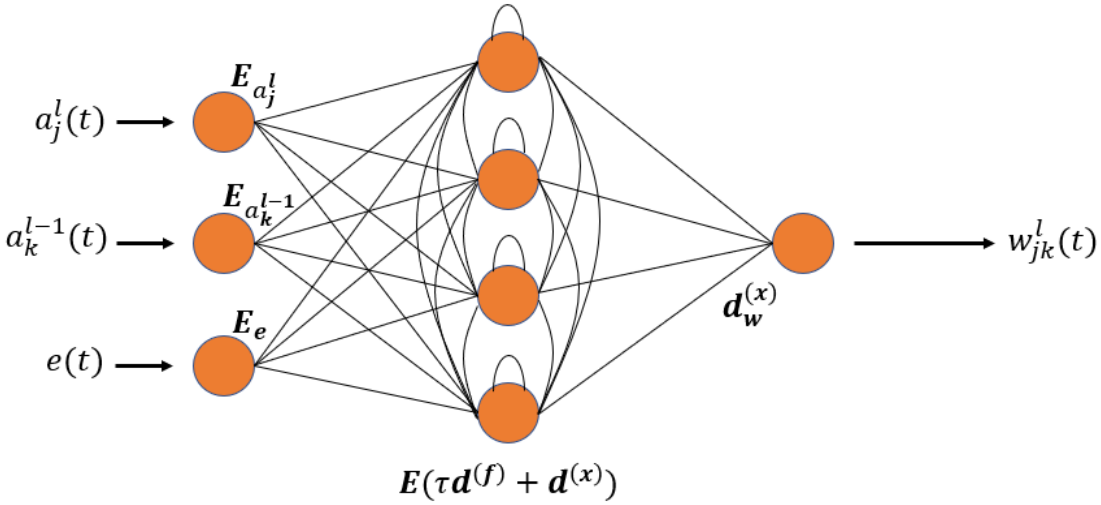

Figure 6 – Depiction of a CTRNN for a hidden layer weight in the main network.

For example, w_{13}^2 can be continuously represented by a population of neurons such that it obeys the expression $\dot{w}_{13}^2 = -\mu a_3^1 (B_1^2 e) a_1^2 (1 - a_1^2)$, so long as the CTRNN takes a_3^1, a_1^2 and e as analog inputs. With one CTRNN for each weight and bias, the necessary updates can be measured from their outputs decoded outputs. The summarized training steps are shown below with a diagram depicting the training approach that includes just one of the required CTRNNs.

1. Provide training sample inputs to the main neural network and desired output to the error approximating neural network.
2. Allow CTRNNs to evolve the encoded weights and biases according to their respective expressions for just enough time to create small changes in the encoded values.
3. Set the main network weights and biases to equal the output of their respective CTRNNs.
4. Move to the next training sample and return to step 1. Stop after several epochs of the training samples.

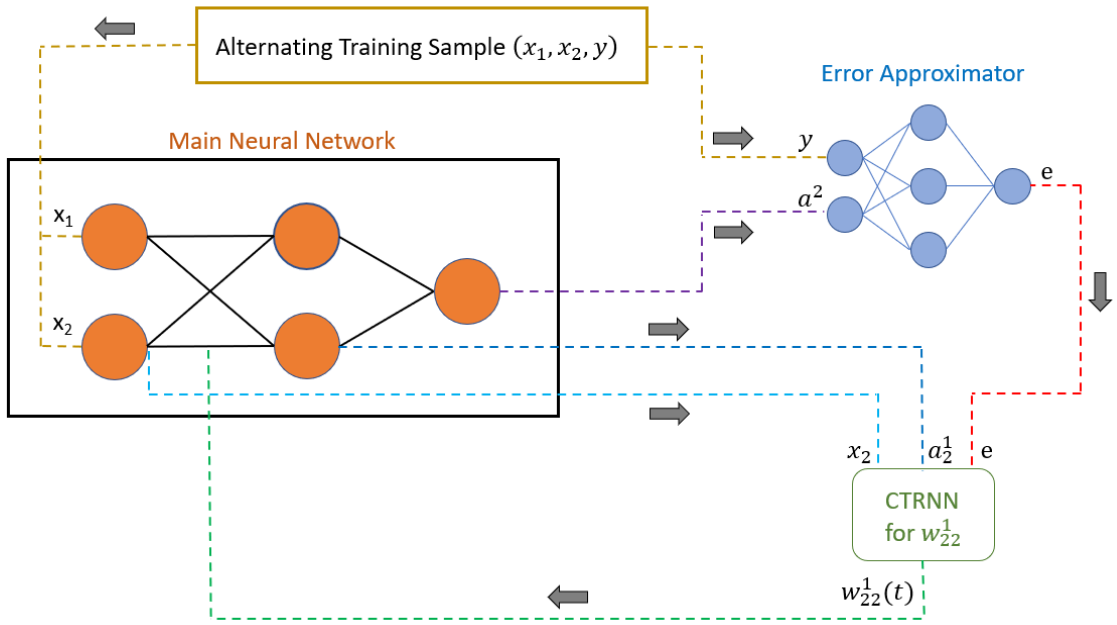

Figure 7 – Depiction of the proposed training approach, showing just one of the necessary CTRNNs.

This approach scales poorly with large main networks since each weight and bias requires a CTRNN of its own. It would be convenient if one CTRNN could be used for all weights and biases in the hidden layers, but the relevant B matrix elements change depending on which weight or bias is in consideration. Additionally, the error approximating network would need to increase in size if the main network has multiple output nodes.

4. XOR Gate Training Task

4.1 Digital DFA Test

The linearly-inseparable XOR gate problem is a benchmark training task for neural network training algorithms. The complete data set for both training and testing only includes the following four samples.

Sample No.	First Input (x_1)	Second Input (x_2)	Desired Output (y)
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Table 1 – Complete data set for an XOR gate neural network.

Standard backpropagation can successfully solve the XOR gate problem with a single hidden layer with 2 hidden nodes. It was unclear if DFA would be successful with the same network size, so this was investigated using conventional digital computing. With just one output node, the hidden layer feedback matrix $\boldsymbol{B}$ actually becomes a vector $\vec{B}$ with just one element for each of the hidden nodes. Using a normal distribution with a standard deviation of one, $\vec{B}$ was taken as [0.56, -0.78]. The output node and two hidden nodes all used the sigmoid non-linear activation function. With the learning rate and number of epochs set to 0.01 and 100 000 respectively, the trained network output approximately [0.1, 0.9, 0.9, 0.1] for samples 1-4 in order. This clearly indicates that training was successful and a neural network with a single hidden layer with 2 hidden nodes is appropriate for DFA training of the XOR gate problem.

4.2 Error Approximating Neural Network

From Equation 3, the output node error is given by $e = 2(a^L - y)a^L(1 - a^L)$, which is a simple function of two variables. In Python, the TensorFlow machine learning library with the Keras interface was used to construct a feedforward neural network with 2 input nodes, 1 output node and 2 hidden layers with 20 nodes each. All hidden nodes were given the sigmoid non-linear activation function and the output node was not given any activation function.

The a^L training data inputs were generated somewhat randomly by taking values between 0 and 1. If the value was greater than 0.5, the distance between that value and 1 was decreased by a factor of 2. If the value was less than 0.5, the distance between that value and 0 was decreased

by a factor of 2. The purpose of this was to more closely match the expected outputs of the sigmoid function, which are often either near 0 or near 1. The y training data inputs were randomly made either 0 or 1, corresponding with the only two desired outputs of the XOR gate problem. The desired outputs of the error approximating network, not to be confused with the desired outputs of the main network y, were generated using the known analytic expression. A total of one million training samples were generated.

After 5 training epochs using the mean squared error loss function and the Adam optimizer, the network achieved a net training loss of 1.6635. Considering the very large number of training samples, this is an acceptable value. From manually testing the trained neural, the difference between the error approximator and the analytic expression was at most 0.002. For reference, the output node error is limited to the range of -0.3 to 0.3. The model was saved so that the trained weights and biases could be imported into a custom Python class that digitally implements the error approximating neural network.

4.3 Nengo CTRNN Simulations

Nengo allows users to both create and simulate custom CTRNNs using discrete timesteps [21]. With this tool, the CTRNN neurons were given sinusoidal non-linear activation functions, which provides a Fourier basis to represent the encoded values. Nengo does not include this neuron type by default, so its implementation was added to the library from a different research project [22]. More information on the selected encoding strategy is given in the Appendix. The CTRNNs for the hidden weights and biases had a population of 96 neurons each, whereas the CTRNNs for the output weights and bias only had 48 each. This is because the equations describing the output layer dynamics involve less variables and therefore have less information to encode. The same single hidden layer, 2 hidden node network shape from the digital DFA Test was selected, again with $\vec{B} = [0.56, -0.78]$.

Selecting an appropriate encoding strategy for the CTRNNs proved to be the greatest challenge. Looking at a specific case, the CTRNN for w_{21}^1 needs to implement the function $\dot{w}_{21}^1 = -\mu x_1 (B_2 e) a_2^1 (1 - a_2^1)$. The real-time analog signals for x_1, a_2^1 and e are continuously encoded into the CTRNN using real-time external inputs. Including w_{21}^1 itself, there are then four values that must be encoded into the neural population at all times. To improve the CTRNN performance, the encoded values were linearly scaled so that they tend to exist in the range -1 to 1. Unfortunately,

the function for hidden weight time dynamics could not be properly implemented, even after experimenting with different encoding vectors, gain coefficients and bias terms. Specifically, the function's $a_j^l(1 - a_j^l)$ term was too difficult for any of the attempted encoding strategies. It is very likely that this problem could be overcome with a better understanding of the NEF because other projects have successfully implemented more complicated dynamics with Nengo [22]. The following figure shows the poor hidden layer weight dynamics, where x_k and e are held constant and a_j^l is set to 0 half way through the simulation. By inspection of Equation 12.3, the encoded weight should stop changing in time when a_j^l is 0 or 1.

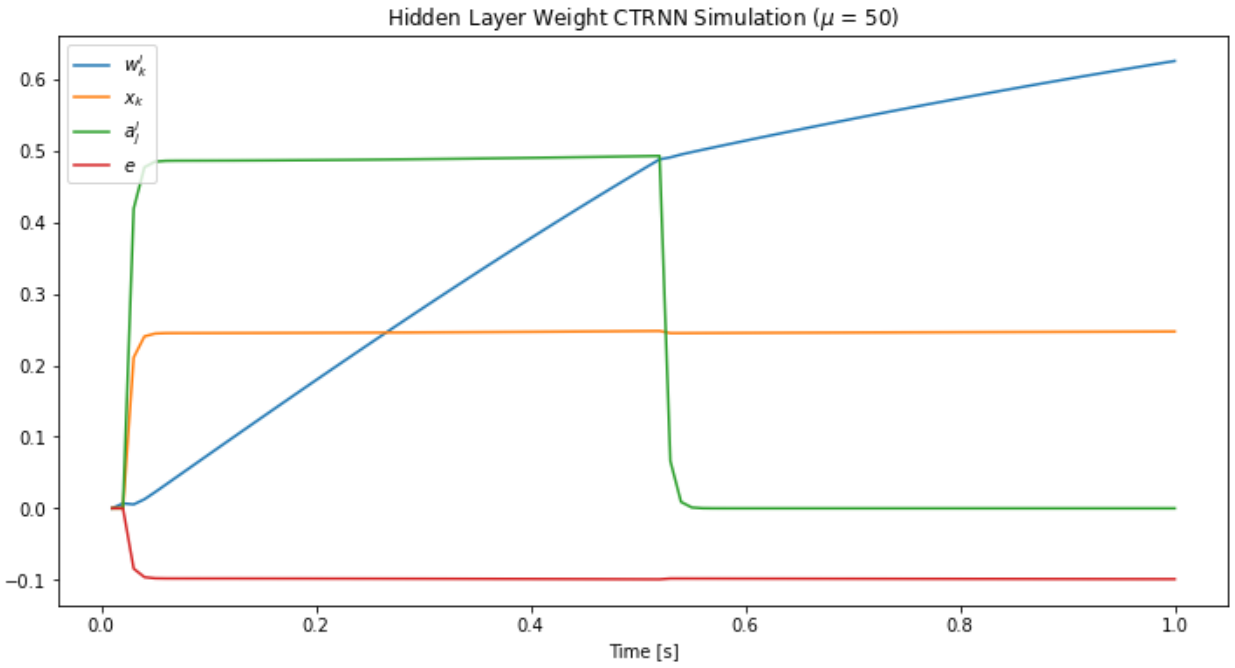

Figure 8 - Example simulation of a hidden layer weight CTRNN where x_k and e were held constant and a_j^l is set to 0 half way through the simulation. The desired dynamics are poorly implemented.

To work around this issue, an invalid assumption was intentionally made. Instead of providing a_j^1 to the hidden weight and bias CTRNNs as an external input, $\sigma'(z_j^l) = a_j^l(1 - a_j^l)$ was computed digitally and provided in its place. Although the number of values to encode in the neural population remains the same, the dynamics are simplified so that for w_{jk}^l, the function to be implemented becomes $\dot{w}_{jk}^l = -\mu x_k(B_j e)\sigma'(z_j^l)$. This function only involves multiplication, and was successfully implemented with the final selected encoding strategy.

The neuron timeconstant (τ) was arbitrarily set to 10 s. The learning rate controls how quickly the weight or bias evolves. It also determines the length of time that each training sample should be provided to the system, since small parameter changes are desired at each iteration. After general experimentation, the learning rate (μ) was set to 50 to overcome an observable drift in the encoded weight or bias. When the learning rate was too small, this drift significantly impaired the desired dynamics. The following figures demonstrate this phenomenon using a hidden layer weight CTRNN, where x_k and $\sigma'(z_j^l)$ were held constant and the sign of e was changed half way through the simulation. With perfectly implemented dynamics, the weight should return to its initial value at the end of the simulation.

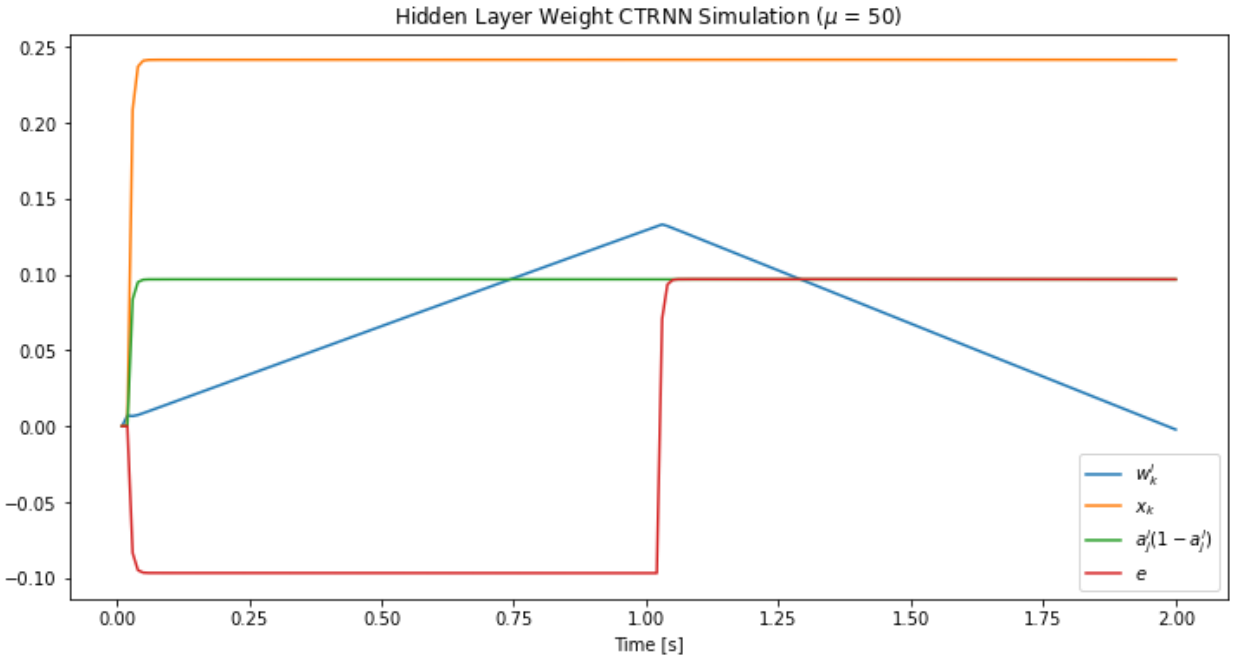

Figure 9 - Example simulation of a hidden layer weight CTRNN where x_k and $\sigma'(z_j^l)$ were held constant and the sign of e was changed half way through the simulation. The learning rate was set to 50, and the desired dynamics are well implemented.

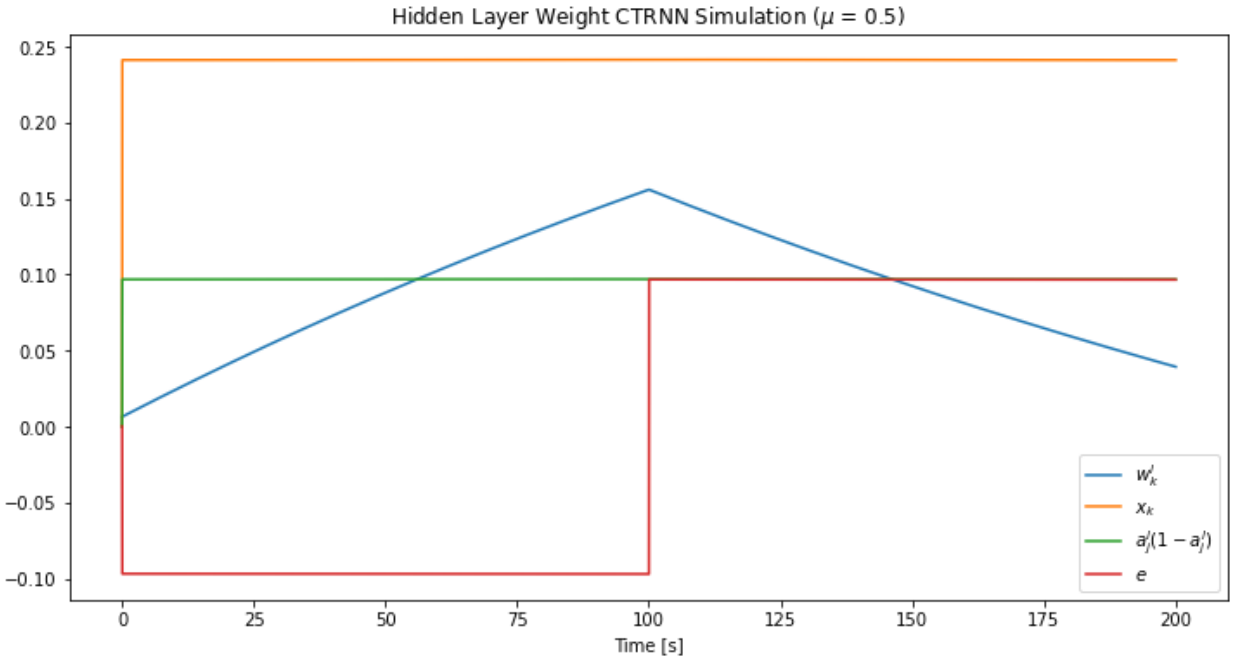

Figure 10 - Example simulation of a hidden layer weight CTRNN where x_k and $\sigma'(z_j^l)$ were held constant and the sign of e was changed half way through the simulation. The learning rate was set to 0.5, and the desired dynamics are poorly implemented.

4.4 Complete System Training Results

A Python class was constructed and instantiated to implement the feedforward XOR gate neural network that requires training. The network's weights and biases would normally be initialized to random values, but in this case they were initialized to 0. This is acceptable only because the imperfect CTRNNs that control the weight and bias updates will naturally drift the parameters away from their initial values. The weights and biases of the trained error approximating neural network were imported into a separate Python object. Both of these digital neural networks compute their outputs instantaneously, however in a real neuromorphic system they would have some inherent delay. An appropriate CTRNN object was initialized using Nengo for all 9 of the XOR gate network's weights and biases.

The CTRNN simulation timestep was set to 0.001 s and the time period that each training sample is provided to the system was set to 0.05 s. Two similar parameter update approaches were simulated. First, the weights and biases were updated once at the end of each training sample iteration. In a real system, this would involve measuring the output of each CTRNN after each training sample iteration and adjusting the XOR gate network parameters accordingly. With the second approach, the weights and biases are updated 10 times throughout each training sample

iteration. Instead of one small step at each iteration, the parameters would now change almost continuously. In a real system, this would involve multiple CTRNN output measurements and XOR gate network parameter updates throughout each training sample iteration.

The number of epochs was set to 4000, therefore with 4 training samples per epoch and 0.05 seconds per sample, the system completes its training attempt in 800 seconds. The complete training time is expected to scale linearly with inverse of the learning rate, so the benefit of the proposed approach to training is entirely dependant on the maximum learning rate allowed for a specific neuromorphic implementation. The following table summarizes the simulation results, which clearly indicates proper training using both approaches.

Sample No.	First Input (x_1)	Second Input (x_2)	Desired Output (y)	Approach 1 Output	Approach 2 Output
1	0	0	0	0.003	0.12
2	1	0	1	0.91	0.92
3	0	1	1	0.89	0.92
4	1	1	0	0.01	0.06

Table 2 - Results from simulating the complete system, where parameters are updated once per training sample iteration in Approach 1 and 10 times in Approach 2.

It is curious why Approach 1 yields different outputs for samples 2 and 3. All digital tests of the DFA training algorithm on the XOR gate network resulted in nearly identical weights leading to each of the two hidden nodes, which predicts identical outputs for the output-1 samples. It may be due to the natural drift in parameters encoded into the CTRNNs, which is likely pushing the parameters away from their optimal values.

5. Conclusion

Artificial neural networks are invaluable computing tools that enable some of the most exciting engineering solutions in modern times. Unfortunately, deep networks require significant training time on power-intensive hardware. This thesis proposed and investigated a novel approach to training in situ deep neural networks using direct feedback alignment and several other in situ neural networks. All main network node outputs are continuously measured and regenerated as real-time analog signals. To implement DFA, the error in the output layer nodes is approximated using a feedforward neural network and each weight and bias is updated using its own CTRNN and the Neural Engineering Framework.

Due to an imperfect encoding strategy, the correct hidden layer parameter dynamics could not be implemented with a CTRNN. To temporarily overcome this issue, some dynamics were digitally computed as an intermediate step. Future work should aim to improve the encoding strategy and implement the complete dynamics. The inherent system latency of a promising neuromorphic approach also must be investigated. It will likely determine the greatest acceptable learning rate that can be used by the CTRNNs, which in turn determines the length of time spent on each of the training sample iterations. Lastly, the most important focus of future work should be to enable training with a set number of CTRNNs. With one CTRNN per network parameter, the current proposed approach to training scales very poorly with network size. Alternative training algorithms to DFA may need to be investigated.

The proposed training approach was tested using the benchmark XOR gate problem and digital simulations of the neuromorphic system. The main network included a single hidden layer with 2 hidden nodes and the error approximating network included 2 hidden layers with 20 nodes each. The hidden layer weights and biases each required a CTRNN with 96 hidden neurons, whereas the output layer weights and bias each required a CTRNN with just 48 hidden neurons. With a learning rate of 50 and 0.05 second training sample iterations, the complete system successfully trained the XOR gate network using two similar parameter update approaches. The first approach updated the weights and biases just once at the end of every training sample iteration and the second approach updated the weights and biases 10 times at even intervals throughout.

References

[1] Y. LeCun, Y. Bengio and G. Hinton, "Deep Learning," *Nature,* vol. 521, pp. 436-444, 2015.

[2] David Silver et Al., "Mastering the game of Go with deep neural networks and tree search," *Nature,* vol. 529, pp. 484-489, 2016.

[3] T. Ferreira de Lima, H.-T. Peng, A. N. Tait , M. A. Nahmias, H. B. Miller , B. J. Shastri and P. R. Prucnal, "Machine Learning With Neuromorphic Photonics," *Journal of Lightwave Technology,* vol. 37, no. 5, pp. 1515-1533, 2019.

[4] A. Kratsios and I. Bilokpytov, "Non-Euclidean Universal Approximation," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020),* Vancouver, 2020.

[5] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *Conference on Computer Vision and Pattern Recognition,* 2016.

[6] S. Furber, "Large-scale neuromorphic computing systems," *Journal of Neural Engineering,* vol. 13, no. 5, 2016.

[7] A. Fumarola et al., "Accelerating Machine Learning with Non-Volatile Memory: exploring device and circuit tradeoffs," in *Ineternational Conference on Rebooting Computing,* San Diego, 2016.

[8] M. Davies et Al., "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro,* vol. 38, no. 1, pp. 82-99, 2018.

[9] B. Shastri et Al., "Photonics for artificial intelligence and neuromorphic computing," *Nature Photonics,* vol. 15, pp. 102-114, 2021.

[10] A. Jain et Al., "Artifial neural networks: a tutorial," *Computer,* vol. 29, no. 3, pp. 31-44, 1996.

[11] D. Svozil et Al., "Introduction to multi-layer feed-forward neural networks," *Chemometrics and Intelligent Laboratory Systems,* vol. 39, no. 1, pp. 43-62, 1997.

[12] Abiodun et Al., "State-of-the-art in artificial neural network applications: A survey," *Heliyon,* vol. 4, no. 11, 2018.

[13] B. J. Wythoff, "Backpropagation neural networks: A tutorial," *Chemometrics and Intelligent Laboratory Systems,* vol. 18, no. 2, pp. 115-155, 1993.

[14] A. Nøkland, "Direct Feedback Alignment Provides Learning in Deep Neural Networks," in *30th Conference on Neural Information Processing Systems,* Barcelona, Spain, 2016.

[15] B. Crafton et Al., "Direct Feedback Alignment with Sparse Connections for Local Learning," *Frontiers in Neuroscience,* vol. 24, 2019.

[16] D. A. Winkler and T. C. Le, "Performance of Deep and Shallow Neural Networks, the Universal Approximation Theorem, Activity Cliffs, and QSAR," *Molecular informatics,* vol. 36, p. 1600118, 2017.

[17] M. C. M. and M. Kiener, "Solving Differential Equations Using Neural Networks," 2013.

[18] A. K. S. E. Modjtaba Baymani, "Artificial Neural Networks Approach for Solving Stokes Problem," *Applied Mathematics,* vol. 1, pp. 288-292, 2010.

[19] K.-i. Funahashi and Y. Nakamura, "Approximation of Dynamical Systems by Continuous Time Recurrent Neural Networks," *Neural Networks,* vol. 6, no. 6, pp. 801-806, 1993.

[20] T. C. Stewart, "A Technical Overview of the Neural Engineering Framework," Centre for Theoretical Neuroscience, 2012.

[21] Bekolay et Al., "Nengo: a Python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics,* vol. 7, no. 48, 2014.

[22] A. Tait, T. Ferreira de Lima, E. Zhou, A. Wu, M. Nahmias, B. Shastri and P. Prucnal, "Neuromorphic Photonic Networks using Silicon Photonic Weight Banks," *Scientific Reports,* vol. 7, pp. 1-10, 2017.

Appendix: Encoding Strategy

The complete project source code can be found at:
https://github.com/adamg0709/UndergraduateThesis.

The following cell defines the encoding strategy used for all hidden layer weight and bias CTRNNs. The only difference for output layer weights and biases is that there are 4 encoding vectors instead of 8, which leads to 48 neurons instead of 96.

```
# Encoding strategy

s_pi = 0.1 # Half of the hidden neuron response's wavelength
max_transmission = 1 # Max output of a hidden neuron

intercept_vals = [0,1/8,1/4,3/8] # Intercept values that correlate with hidden neuron bias terms
rate_vals = s_pi*np.array([1/2,1,3/2]) # Gain coefficients that scale encoded inputs to the hidden neurons

# Selected encoding vectors for the 4 values (w,x,a,e) that are encoded into the neural population
encoder_vals = [[1,1,1,1],[1,1,1,-1],[1,1,-1,1],[1,-1,1,1],[1,1,-1,-1],[1,-1,1,-1],[1,-1,-1,1],[1,-1,-1,-1]]

num_intercepts = len(intercept_vals)
num_rates = len(rate_vals)
num_encoders = len(encoder_vals)
num_neurons = np.int_(num_rates*num_intercepts*num_encoders)

intercepts = np.zeros(num_neurons)
rates = np.zeros(num_neurons)
encoders = np.zeros(shape=(num_neurons, 4))

j = 0
for ir in range(num_rates):
    for ii in range(num_intercepts):
        for ie in range(num_encoders):
            encoders[j,:] = encoder_vals[ie]
            intercepts[j] = intercept_vals[ii]
            rates[j] = rate_vals[ir]
            j += 1

print(num_neurons)
```

```
96
```

The next cells define how recurrent and external connections are made, specifically for hidden layer weight CTRNNs. The only difference for hidden layer bias CTRNNs is that “input_x” provides a constant external input of one.

```
# The feedback connections
def feedback(vec):
    w, x, a, e = vec
    dw = -B*e*x*a*(1-a)*mu
    return [w + dw*tau]
```

```
seed=np.random.randint(1, 10000000)
model = nengo.Network()
with model:
    # The ensembles and recurrent connections for the CTRNNs
    CTRNN = nengo.Ensemble(num_neurons, dimensions=4, radius=2., n_eval_points=100000,
                encoders=encoders, intercepts=intercepts, max_rates=rates,
                neuron_type=nengo.neurons.FourierSinusoid(max_overall_rate=max_transmission, s_pi=s_pi))

    # Recurrent connection
    nengo.Connection(CTRNN, CTRNN[0], function=feedback, synapse=tau)

    # External inputs
    input_x = nengo.Node(lambda t: x_val)
    input_a = nengo.Node(lambda t: a_val)
    input_e = nengo.Node(lambda t: e_val)

    # External connections
    nengo.Connection(input_x, CTRNN[1])
    nengo.Connection(input_a, CTRNN[2])
    nengo.Connection(input_e, CTRNN[3])

    # Record values
    CTRNN_probe = nengo.Probe(CTRNN, synapse=0, sample_every=0.01)
```

The following figure depicts the sinusoidal response curves of some CTRNN hidden neurons. The complete neural population provides a Fourier basis to represent encoded values. A neuron's response frequency and phase offset is determined by its intercept and gain coefficient that is defined early in the program.

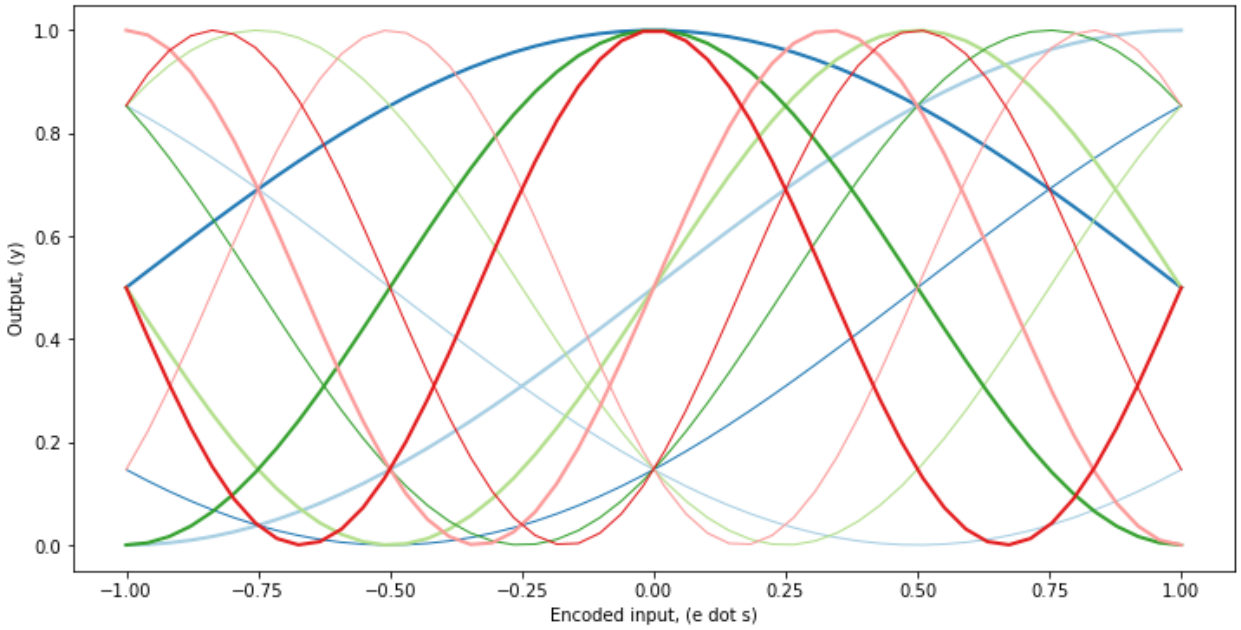

Figure 11 - Sinusoidal response curves of some of the CTRNN hidden neurons.