Design and Control of a Photonic Neural Network Applied to High-Bandwidth Classification

Ethan Gordon '17
egordon@princeton.edu

Advisor: Paul R. Prucnal
prucnal@princeton.edu

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

May 8, 2017

Honor Statement

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

Ethan K. Gordon '17

Design and Control of a Photonic Neural Network Applied to High-Bandwidth Classification

Ethan Gordon '17
egordon@princeton.edu

Abstract

Neural networks can very effectively perform multidimensional nonlinear classification. However, electronic networks suffer from significant bandwidth limitations due to carrier lifetimes and capacitive coupling. This project investigates photonic neural networks that can get around these limitations by performing both the activation function and weighted addition in the optical domain using microring resonators. These optical microring resonators provide both nonlinearity and superior fan-in without compromising bandwidth. The ability to thermally calibrate networks of cascaded axons and dendrites and train such a network to solve nonlinear classification problems are demonstrated using theory and simulations. The former is also demonstrated experimentally on a two-channel axon cascaded into a two-channel dendrite, showing good agreement between simulation and experiment. In addition, the use of transverse modes to increase the size of each photonic layer is examined. Simulations that determined the optimal waveguide geometry for using these modes were experimentally validated.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Alex Tait 'GS, my primary day-to-day mentor and advisor. Without him, absolutely none of this work would have been possible. I would also like to extend my thanks to the entire Lightwave Communications Laboratory for their advice, support, and infrastructure. This project was also made possible by generous funding from the School of Enginering and Applied Science and the Department of Electrical Engineering. Finally, I would like to acknowledge all of my friends and peers: the fighting ELE Class of 2017, off which I could always bounce ideas, and my friends and roommates, who were willing to listen to my ramblings and help edit this document. Thank you to everybody for helping make this a success.

This work is dedicated to the loving memory of my parents, Lauren and David Gordon.

Contents

1 Background

1.1 Motivation

Neural networks have captured the public imagination. Very large, deep networks like Google's AlphaGo have harnessed these nonlinear systems to perform impressive feats on human timescales: the order of seconds or minutes [1]. However, software neural networks in general become impractical as the speed of the input signal increases, assuming the computation needs to be completed in real-time. There hardware networks can step in to fill the niche. While they cannot scale infinitely by definition, moderately-sized networks (on the order of hundreds of neurons or smaller) can provide useful, high-speed computation.

This thesis focuses on nonlinear classification: the problem of separating data into N different classes when the optimal curve of separation is more complicated than a straight line. As shown in Figure 1, a neural network accomplishes this by using the nonlinear *activation function* in each layer to map the previous space onto a new, warped space. In this warped space, the different classes are linearly separable, and the last neurons draw this optimal line. Linear classification in curved space is equivalent to nonlinear classification in uncurved space. In terms of speed, a layered network's latency is the sum of the latency of each layer, but since each layer can operate in parallel, the total throughput of the network is limited by the throughput of the smallest layer.

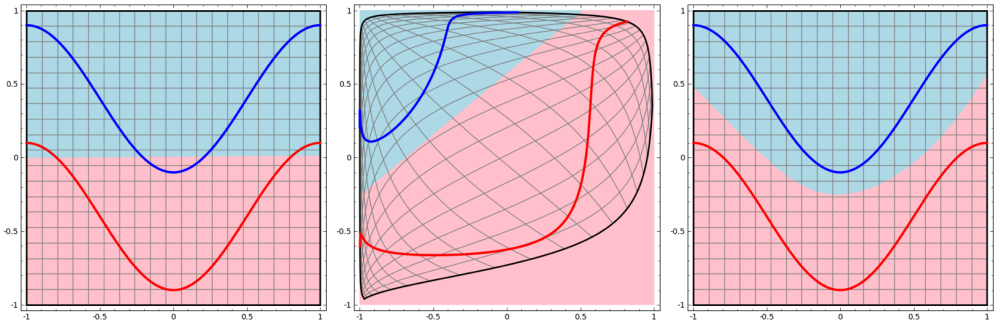

Figure 1: (Left) An optimal linear classifier for the provided data set. **(Middle)** Example of using a 2-2-1 feed-forward neural network. The hidden (middle) layer warps the euclidean space, and the last layer performs linear classification in this distorted space. **(Right)** The resulting effective nonlinear classification curve. Image Credit: [2]

High-speed nonlinear classification can have applications in scientific computing (such as CERN's L1 trigger) and radio (where even simple digital demodulation requires nonlinear classification). However, as the frequency of the input signal increases, even hardware

networks begin to run into problems. Existing electronic networks [4][5] are fundamentally limited by a trade-off between the bandwidth of the input signal and the size of each layer of the network. Adding more neurons in a layer allows for processing more information at once, but the increased fan-in also decreases the bandwidth of each neuron proportionally. The total throughput remains approximately constant. For example, the TrueNorth [3] network design has a layer size of 256 neurons, but each layer updates at a speed of 1kHz (which then limits the input signal bandwidth). The product is 256kHz, which is the total effective maximum throughput of the network. For example, this value can be achieved with audio by having the network operate on 256 audio samples at once. TrueNorth could increase the bandwidth by decreasing the layer size, but the total throughput would remain about the same.

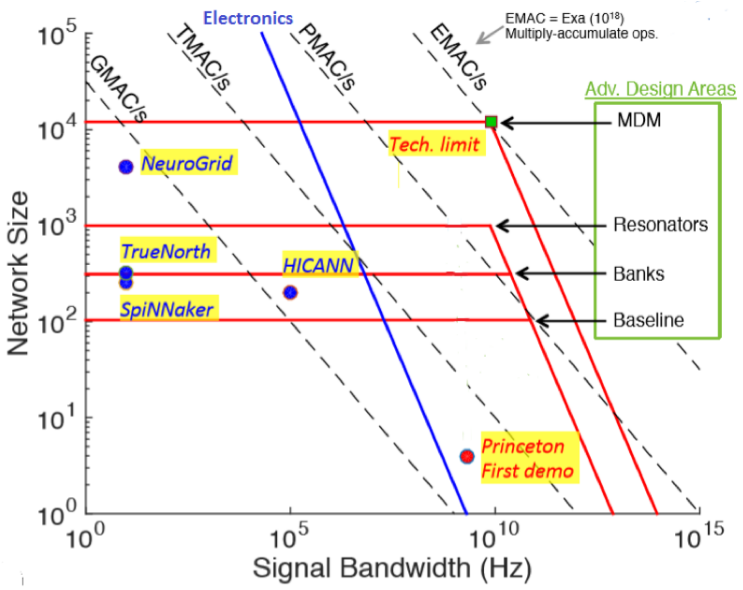

Figure 2: Trade-off between the size of a network layer and the input signal bandwidth for different electronic and proposed photonic networks. Marked are dashed lines of constant throughput. A point is marked for the Lightwave Communication Lab's first neuron demonstration, covered in Section 1.2. Adding weighted addition to the optical domain is discussed in Section 2, and expanding network size through mode-division-multiplexing (MDM) is discussed in Section 4.

Figure 2 demonstrates this trade-off in existing electronic networks. Such systems have trouble reaching terahertz throughput. Here is the motivation for switching to optical-electronic-optical (OEO) networks. By moving the interconnects between neurons into the optical domain, this trade-off can be beaten, allowing for large signal bandwidths for a given

network size. Such networks would be even more useful in applications requiring high-speed nonlinear computation.

1.2 Previous Work

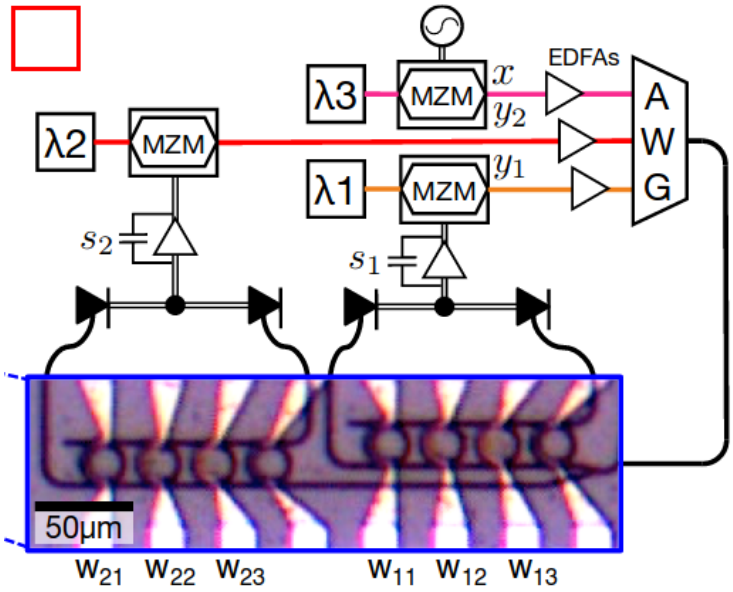

Figure 3: Diagram of the components in Princeton's first neural network demonstration using wavelength division multiplexing (WDM) to distinguish between different channels. The network produced the results in Figure 4. Note how the weight banks are the only integrated components, leaving the nonlinear modulation to happen off-chip.

The first proof of concept for an OEO neural network came from Princeton [6] and demonstrated neuromorphic dynamics. An outline of the network can be seen in Figure 3. In this demonstration, the weight banks making up the dendrites (described in more detail in Section 2) were the only integrated component. These banks, combined with a pair of balanced photodiodes, carried out the weighted addition equivalent to a dot product with a weight matrix. This current signal was then amplified and fed into an off-chip Mach-Zahnder modulator, whose saturation provided the nonlinear activation function feeding back into the optical network.

Figure 4 provides evidence of neuromorphic dynamics. Modeling the activation function as being similar to hyperbolic tangent, the neuron output as $y(t)$, the self-weight as W_f, and

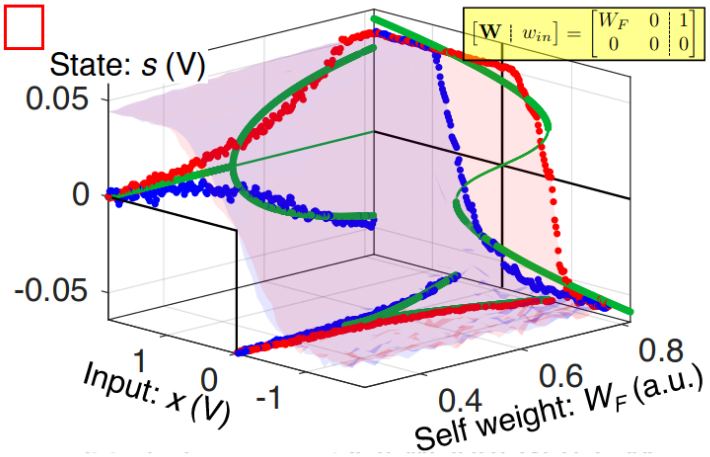

Figure 4: Evidence of neuromorphic dynamics of a single-neuron in feedback with itself. As the self-weight increases, an intermediate input signal produces a bistable state, creating the hysteresis loop shown at the back of the graph (a self-weight of 0.8).

the external input as x, we get the following network dynamics:

$$y_{t+1} = c\tanh(W_f * y_t + x) \tag{1}$$

In the steady state, we set $y_{t+1} = y_t = y$ and solve for y. As a function of W_f, $y(W_f)$ bifurcates when $W_f * c > 1.0$, leading to the image on the back-left of Figure 4. Once the weight is high enough to reach bifurcation, the relation $y(x)$ ceases to be a function, forming instead a hysteresis loop where the value of y is dependent upon the history of x. This leads to the S-shaped function seen on the back-right of Figure 4.

While a good proof of concept, this first network demonstration suffered from quite a few drawbacks. The activation function was applied off-chip on external Mach-Zehnder interferometers, a scheme which is in general not scalable. The definition of positive and negative optical signals relative to some base positive optical power made for difficult network dynamics and required biases to be *weight-dependent*, making any possible experiments in future plasticity very difficult. Finally, while a control algorithm was demonstrated for this grouping of dendrites [7], that algorithm was incapable of working with different network topologies and was left untested on different examples of the same topology.

This thesis seeks to expand upon this previous work by looking at possible fixes to these limitations: investigating integraded axons in the form of microring resonators, looking at different network topologies, attempting to show nonlinear classification with strictly positive signals, and modifying the control algorithm to extend to all of these new additions.

2 Network Design

2.1 Operating Principle

To create a neural network, each neuron operates with an activation function on a weighted sum of fanned-in inputs and produces a single fanned-out output. For a neuron input $\vec{x}$, a weight vector $\vec{w}$, a bias b, an activation function f, and an output y, a single neuron can be modeled as:

$$y = f(\vec{w} \cdot \vec{x} + b) \tag{2}$$

The basic unit of computation in the optical domain is the microring resonator, shown in Figure 5. Fortunately, this unit can accomplish both weighted addition and the nonlinear transfer funciton. The useful power transfer characteristics are derived here.

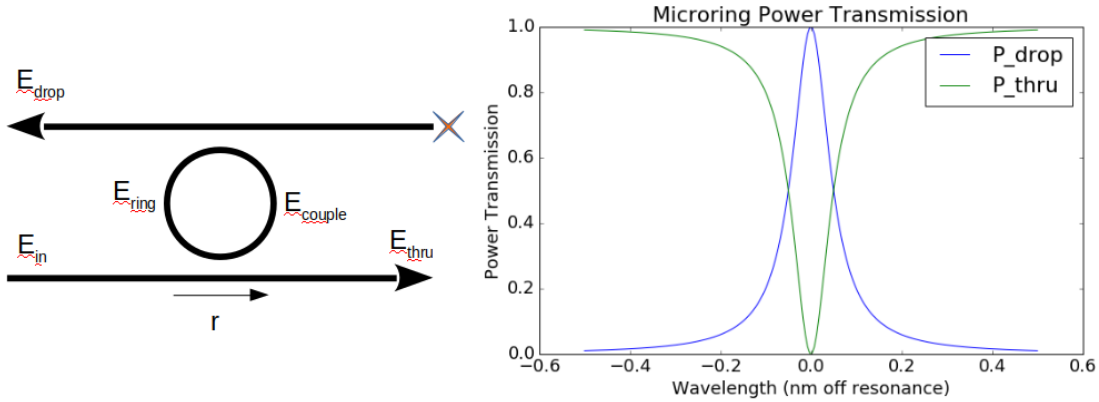

Figure 5: (Left) Layout of a microring resonator. **(Right)** Power transmission spectra between the input port and the drop and thru ports. The drop port spectrum, to the second order, matches a Cauchy-Lorentz distribution.

First consider a generic optical coupler made of two parallel waveguides, taking two optical inputs $\vec{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$ and generating two outputs $\vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$. In the continuous wave approximation, we assume that all operations occur independently of the frequency of the light. Therefore, each input a_i and output b_i can be represented as a simple complex number containing the amplitude and phase of the wave. We can model this optical coupler as a scattering matrix:

$$\vec{b} = S\vec{a} \tag{3}$$

Assuming a lossless coupler, our matrix should also be unitary. Therefore, we can write S

generally as:

$$S = \begin{bmatrix} r_c & t_c \\ -e^{i\theta}t_c^* & e^{i\theta}r_c^* \end{bmatrix} \tag{4}$$

where $|r_c|^2+|t_c|^2= 1$. Taking the limit as $t \to 0$, which is equivalent to an isolated waveguide, we know that the incoming light should experience no phase shift. Therefore, we take $r_c = r$ to be real. Additionally, due to the physical symmetry of the system, we expect this matrix to be both symmetric and persymmetric. Therefore, we have $r = e^{i\theta}r \implies e^{i\theta} = 1$ and $t_c = -t_c^*$. This last statement implies t_c is purely imaginary, and can then be written as $it, t \in \mathbb{R}$. We now write our scattering matrix as:

$$S = \begin{bmatrix} r & it \\ it & r \end{bmatrix} \tag{5}$$

The important takeaway is that coupled power in an optical coupler picks up a $\frac{\pi}{2}$ phase shift. For the rest of the derivation, t will be replaced with $\sqrt{1-r^2}$.

Now consider a microring resonator with an input amplitude and phase E_{in}. This input couples with the wave E_{ring} already inside of the ring. We start by writing the equation for the wave coupling into the ring:

$$E_{couple} = rE_{ring} + i\sqrt{1-r^2}E_{in} \tag{6}$$

This wave is attenuated by r due to the optical coupler on the other side of the ring and picks up a phase shift ϕ from the optical path length of the ring, allowing us to write a feedback equation for β:

$$E_{ring} = E_{couple} * r * e^{i\phi} = (rE_{ring} + i\sqrt{1-r^2}E_{in})re^{i\phi} \implies E_{ring} = \frac{i\sqrt{1-r^2}E_{in}re^{i\phi}}{1-r^2e^{i\phi}} \tag{7}$$

We can now look at the other side of the optical coupler from Equation 6 to determine the field transmission at the thru port:

$$T_{thru} = \frac{E_{thru}}{E_{in}} = \frac{1}{E_{in}}(r\alpha + i\sqrt{1-r^2}\beta) = r - \frac{(1-r^2)re^{i\phi}}{1-r^2e^{i\phi}} = \frac{r(1-e^{i\phi})}{1-r^2e^{i\phi}} \tag{8}$$

From here, we can derive the power transmission to each port of the microring.

$$P_{thru} = |T_{thru}|^2= \frac{2r^2(1-cos(\phi))}{1+r^4-2r^2cos(\phi)} \tag{9}$$

$$P_{drop} = 1 - P_{thru} = \frac{(1-r^2)^2}{1+r^4-2r^2cos(\phi)} \tag{10}$$

We can find ϕ with respect to the perimeter of the ring L, the index of refraction n (creating the optical path length nL), and the wavelength of the light λ:

$$\phi = \frac{2\pi nL}{\lambda} \tag{11}$$

The system is in resonance when $\phi = 2m\pi$, $m \in \mathbb{Z}$, giving us the resonant wavelengths:

$$\lambda_0 = \frac{nL}{m},\ m \in \mathbb{Z} \tag{12}$$

Finally, we replace $cos(\phi)$ with its second-order Taylor approximation around one of these resonant frequencies, yielding:

$$P_{drop} \approx \frac{\gamma^2}{\gamma^2 + (\lambda - \lambda_0)^2} \tag{13}$$

$$P_{thru} \approx \frac{(\lambda - \lambda_0)^2}{\gamma^2 + (\lambda - \lambda_0)^2} \tag{14}$$

Where:

$$\gamma = \frac{(1 - r^2)nL}{2\pi r m^2} \tag{15}$$

This shows that the power transmission spectrum of a microring around a single resonance peak can be approximated by a nonlinear Cauchy-Lorentz distribution. Especially important is the fact that this center frequency of the distribution λ_0 is proportional to the index of refraction n in the medium. Silicon's index of refraction is temperature-dependent. Therefore, by using a current heater around the perimeter of the ring, the center frequency of the distribution can be shifted to any desired frequency.

2.2 Components and Topology

The first component of each neuron is the dendrite, charged with performing weighted addition on the optical inputs. In a WDM network, each input is an analog optical power on a given wavelength, and each weight is applied by a single microring resonator slightly detuned from the wavelength channel. Even though these microrings are cascaded together, if the tuning wavelenghts are sufficiently far apart, the continuous-wave approximation from the previous section holds, and each microring can be considered separately. After the weight bank, the power from the input is split between the combined drop port and the combined thru port. Each port then feeds into a photodiode. The current output from the photodiode is effectively independent of wavelength for the wavelengths used; it is proportional to

the sum of all optical power entering the photodiode. By putting the photodiode on the drop port and the photodiode on the thru port in a balanced configuration (see Figure 8 and the explanation in Section 2.3), the output electrical current can be described as being proportional to the difference between the thru port and the drop port optical powers.

This give us positive and negative weights. When the microring is on-resonance ($P_{drop} \approx 1$), all of the optical power on that channel goes into the drop photodiode, creating a positive elctrical current. When the microring is completely detuned ($P_{drop} \approx 0$), all the optical power on that channel goes to the thru photodiode, creating a negative electrical current. 50% power transfer corresponds with a weight of 0.

After weighted addition, the next step is to feed the signal into some nonlinear element: the axon. Here, we can actually take advantage of the nonlinear power transfer characteristic of the microring resonator to use it as an axon as well. Cascading another microring tuned to the same channel as the microring in the dendrite would cause the continuous-wave approximation to break down. However, the axon can be configured to only connect to the rest of the network via the thru port, leaving dropped power to dissipate. This preserves the analysis done in the previous section and creates the inverted Cauchy-Lorentz transfer-function used in training, as described in Section 3.3.

The next system-level problem is to determine the best configuration of dendrite-microrings and axon-microrings. Two possible technologies are shown in Figure 6. First, in the optical domain, let a *branch* be defined as optical transmission from an axon into a dendrite network. In the star topology, which was used the previous OEO neuron demonsration, each branch has optical power flowing from the axon into a splitter, dividing the power up equally between all dendrite weight banks. While easier to design and control, such a network runs into issues of scalability as the number of dendrites increases, since each dendrite needs to connect directly to a single point on the network.

A possibly better solution is a folded bus, or "hairpin," topology. Here, each branch starts out with the axon rings, which drop optical power onto a single bus. This bus folds into the dendrite network. Each neuron then consists of two cascaded weight banks: one pulls optical power off of the bus, and the second does the actual weighting. For an n-dendrite branch, the first dendrite can pull $1/n$ power off of the bus, the second can pull $1/(n-1)$, and so on, making sure each dendrite gets the same amount of optical power to weight. The benefits of a hairpin topology include scalability and configurability. On the first front, since each dendrite need only connect directly to the previous dendrite rather than a star point, the amount of waveguide required per dendrite is constant, rather than increasing linearly

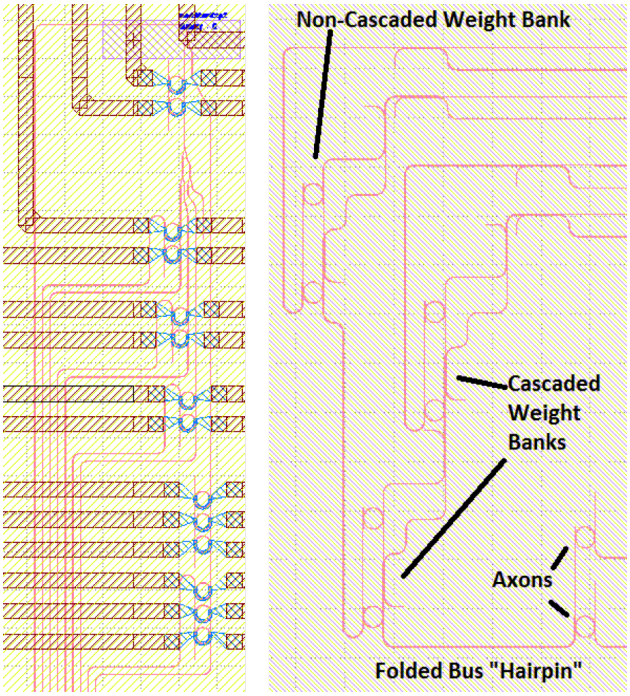

Figure 6: (left) Two-branch network in star topology. Optical power for all dendrites in a branch, as well as all branches, originates at a single central point, hindering scalability. (right) One-branch network in hairpin topology. Intermediate weight banks pull power off the bus and shepherd it to each weight bank.

with the number of dendrites. Additionally, since the system uses microrings to drop power from the bus before weighting, the amount of dropped power can be configured. This, for example, can allow one dendrite to have magnified weights at the expense of other dendrites.

In addition, the hairpin topology is necessary for extension into Mode-Division Multiplexing (MDM) networks, discussed further in Section 4.

2.3 Experimental Design: A 2-3-1 Feed-Forward Network

While the hairpin topology makes the most sense long-term, the problems of control, calibration, and training are easier to initially solve on a star network. To demonstrate nonlinear classification, a 2-3-1 feed-forward network topology was chosen, and the optical components can be seen in Figure 7. The input optical power, a combination of three wavelength channels, is split into two branches. On the first branch, two axons modulate the input signal onto two of the wavelength channels. These in turn are split and transmitted through three dendrite weight banks, creating three sets of complementary optical outputs to biased

photodetectors. Note how, in this first branch, there is an unhandled optical wavelength channel, and this will show up as an effective extra optical bias in the training algorithm. The second optical branch operates similarly, with three axon microring-resonators modulating the three wavelength channels and feeding into a single dendrite weight bank to produce the final output.

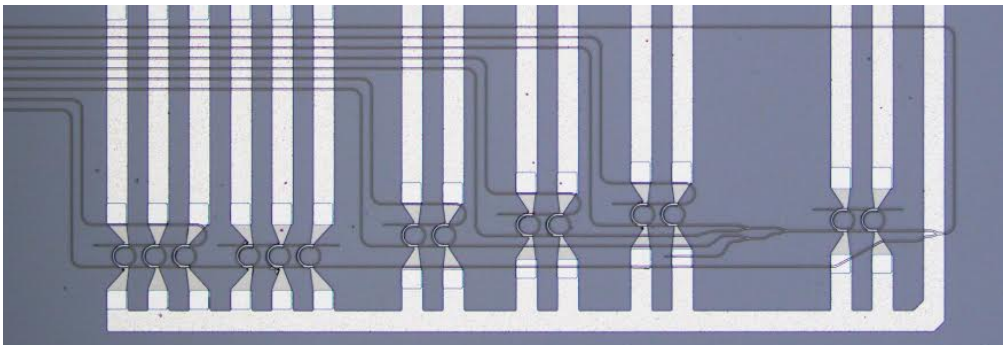

Figure 7: Optical portion of 2-3-1 Feed Forward Network. From the right: optical pump power splits into two branches. An nxm layer starts with n axons to modulate the signal from the previous layer (or input) onto the bus. There are then m weight banks to perform the weighted addition.

The relatively simple circuitry involved in the electrical portion of the network is shown in Figure 8. For each weight bank, the two optical outputs are captured by two balanced photodiodes. Due to the flat frequency response of the diodes, the output current for each diode is essentially proportional to the sum of the input optical powers. By placing two diodes in series and measuring the output current from the intermediate node, the output becomes the difference between the sum of optical powers at the thru port and the sum of optical powers at the drop port. This is what allows for both positive and negative weights. From here, we can write the expression for the current output c of a given weight bank as

$$c = R(\vec{w} \cdot \vec{x} + b_p) = R((2\vec{t} - \vec{1}) \cdot \vec{x} + b_p) \tag{16}$$

where x is the input optical power on each channel, $t \in [0, 1]$ is the transmission through the tuned microring, b_p encompasses any bias optical power from any unweighted channels, and R is the responsitivity of the photodiodes (i.e. the proportionality between input optical power and output current).

This output current signal is then amplified and converted to a voltage through a transimpedence amplifier. The gain is set by the resistor R_t and the offset is set by b_v. In practice, since the voltage b_v is reflected backwards to the node between the photodiodes, it is

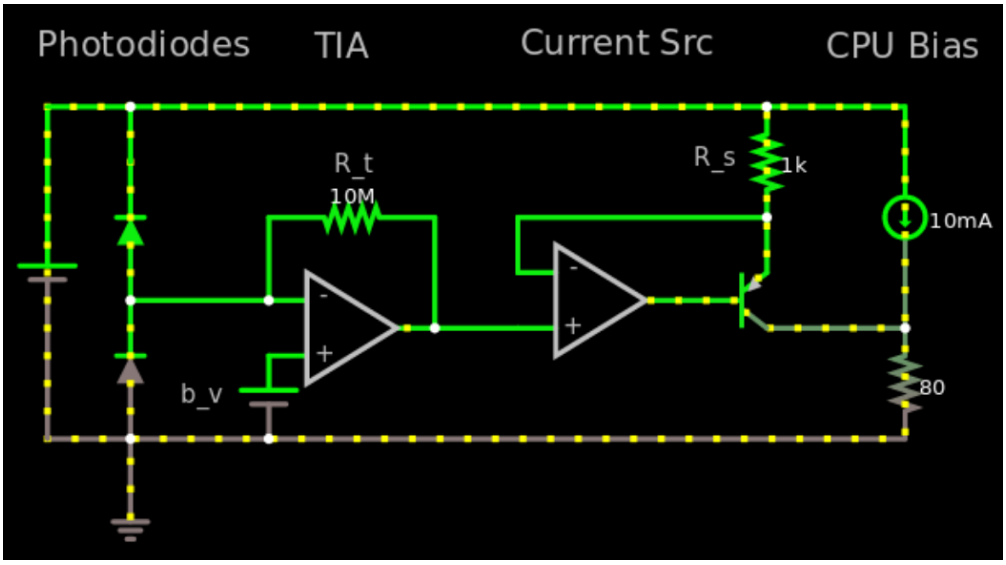

Figure 8: Electrical component of 2-3-1 feed forward network. One circuit is required per dendrite weight bank.

best to fix it at $V_{dd}/2$ so that both photodiodes have the same reverse bias. This ensures that the photodiodes operate symmetrically with the same responsivity. The resulting voltage is fed through a voltage follower into a PNP transistor that to thre first order acts as an ideal current source. This current signal is attenuated by the resistor R_s and is combined with the bias current controlled directly by an external CPU before feeding into the corresponding axon. Combining all of these circuit elements, the final circuit transfer function is:

$$c_{out} = \frac{R_t * R(\vec{w} \cdot \vec{x} + b_p) + b_v}{R_s} + b_c \tag{17}$$

Where b_c is the current bias from the CPU. How this circuit's transfer function applies to a backpropagation algorithm is discussed with Network Training in Section 3.3, along with a full simulation of this network design.

3 Calibration, Control, and Training

Before building the physical network described in the previous section, it is necessary to construct a procedure to thermally calibrate and train it.

The goal of thermal calibration is to create an invertible map between the input current vector and the location of the resonance peaks (and by extension power transmission) for every microring in the system. In other words, it becomes possible to determine how much current to send to each microring to realize a given weight vector $\vec{w}$ that has be mapped to the power transmission vector $\vec{t}$ via Equation 16. Meanwhile, the orthogonal problem of training involves using backpropagation to determine the optimal weights and biases for a given input dataset.

The following three subsections discuss both procedures and provide the results of their application on simulated devices.[1] To demonstrate good agreement between simulation and experiment, empirical results of thermal calibration are provided in the final subsection. Experimental validation of network training was not done here, but it is the immediate next step in future work.

3.1 Basic Calibration Procedure

As stated before, the goal of thermal calibration is to provide a map between the state of electrical currents sent to the on-chip heaters and the power transmission (or, equivalently, the position of the resonance peak) of each microring. The annotated code to carry out this procedure in simulation can be found in Appendix B.1, and an outline of the procedure in terms of its effect on a simulated spectrum through a single dendrite is shown in Figure 9.

In general, testing calibration code in simulation involves first creating a simulated model meant to act as the "real" system. The parameters of this module are set manually and remain hidden from the calibration code at all times. It is the goal of the calibration code to initialize an empty model of the same type and fill it with parameters that match those of the "hidden" model. In principle, running an inverse simulation of the calibration model will yield input currents from desired spectra. These currents can then be used to control the "hidden" model with predictable effects.

Recall from Section 2.1 that the resonance wavelength of each microring is temperature-

[1] While the creation of simulation models for both the optical devices and laboratory instruments was an extensive part of this thesis, that code does little to enhance understanding of these procedures. Documented code can be requested directly from the Lightwave Comunications Laboratory.

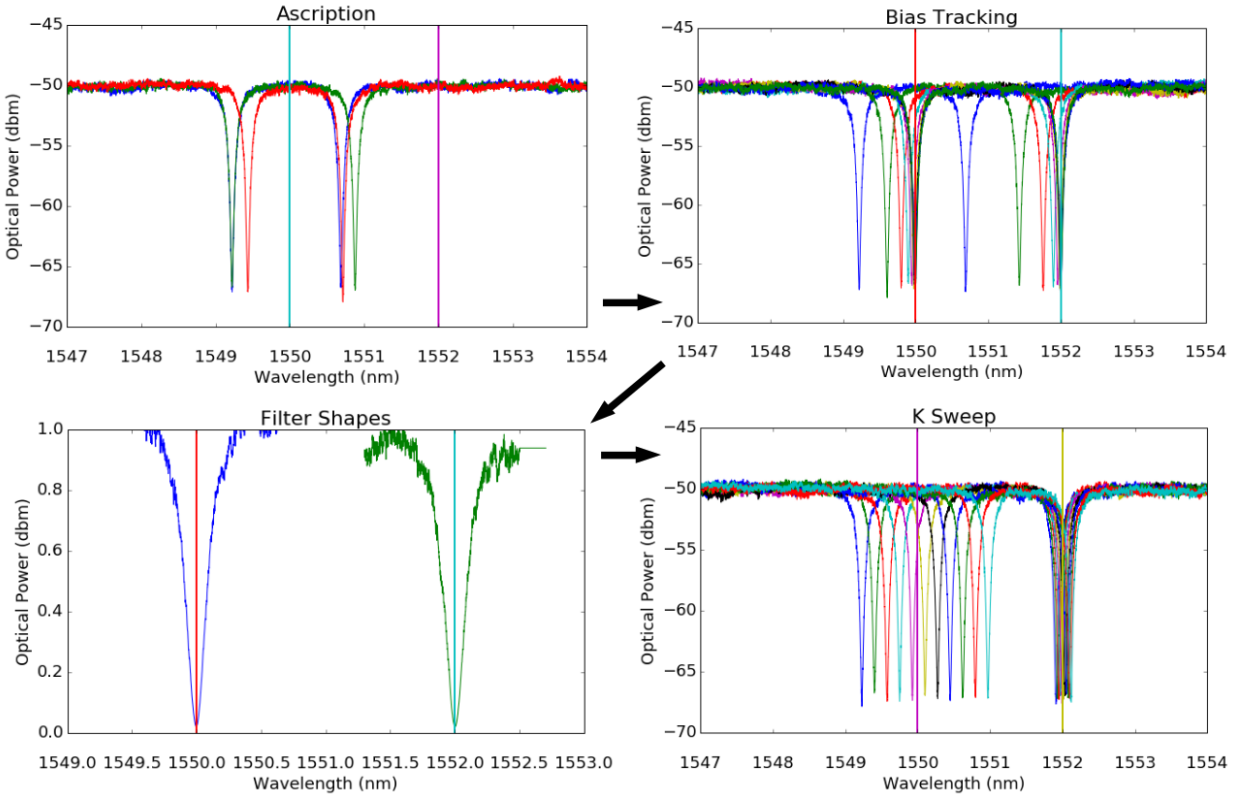

Figure 9: Basic calibration procedure performed on a 2-microring dendrite without axons. From upper-left to lower-right: send a small current to determine primary mapping between filaments and microrings, use feedback control to direct the resonance peaks to the input channels (marked with vertical lines), pull the shape of each peak for use in backpropagation, and sweep around resonance to pull the proportionality constants between dissipated heat and wavelength shift. Note how, even though only one current channel is being swept, the other peak moves due to thermal cross-talk. These are the off-diagonal elements of the K matrix described in the section.

dependent. The heat supplied to the chip is proportional to the square of the current sent to the heater. Therefore, we can linearize the system around a bias current, creating the following system model:

$$(\lambda_j - \lambda_{0_j}) = K_{jk}[(i_k + i_{b_k})^2 - (i_{b_k})^2] \tag{18}$$

Where i_k are the currents sent to each microring, i_{b_k} are the bias currents that set each microring's resonance to the bias wavelengths λ_{0_j}, and λ_j are the resulting resonance wavelengths of each microring. Diagonal elements of the matrix K are the proportionality constants relating each microring's resonance peak and its own heater, while the off-diagonal

matrix represents the effect heaters have on adjacent microrings due to thermal cross-talk. In practice, the magnitude of the off-diagonal elements tend to be about 5% the magnitude of the diagonal elements. Together, the K matrix, bias wavelengths λ_0 and currents i_0, the approximate shape of each microring Cauchy-Lorentz distribution, and the global attenuation of the system (from effects including insertion loss and scattering) are all the model parameters that the calibration procedure must determine.
The description of the calibration procedure is as follows:

1. **Ascription:** Initially, the calibrator does not know which current channel corresponds to which resonance. In this step, send a small amount of current to each current channel and see which resonance moves the most. Also use this step to estimate the diagonals of the K matrix.

2. **Background Removal:** While the noise in the simulation is zero-mean pink noise, the real system could have some non-random noise. Tune the resonances off to see the background spectrum behind them, and then subtract this background spectrum from each successive spectrum measurement. Also use this step to determine total system attenuation and store in the calibration model.

3. **Track to Bias:** The wavelength biases are fixed to the wavelengths of the input channels. Use proportional feedback control on the supplied currents to adjust each resonance to the wavelength bias. The diagonals of the control matrix is proportional to the estimated diagonals of K, decreasing convergence time. This leads to the controller:

$$\Delta i_k^2 = k_p * K_{kk}\text{err}_k \tag{19}$$

   When done, store the wavelength and current biases in the calibration model.

4. **Pull Filter Shapes:** Take a spectrum isolating each Cauchy-Lorentz distribution, and store that distribution in the calibration model to estimate the power transfer as a function of detuned wavelength.

5. **Determine the K Matrix:** Sweep each current channel around the bias current and determine the best linear fit for each resonance. Store the slope of this fit as the element of the K matrix.

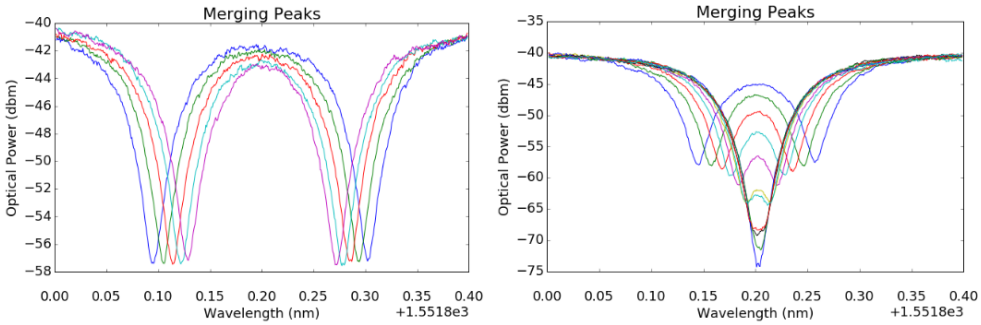

Figure 10: Example of moving one axon peak and one dendrite peak onto the same resonance. **(Left)** The two peaks are brought together until the peak-finding algorithm cannot distinguish them from a single trough. **(Right)** A feedback controller attempts to minimize the full-width half-maximum (i.e. the width) of the combined trough, bringing both troughs onto the same resonance.

3.2 Cascaded Calibration and Experimental Results

The difficulty with calibrating a real star-topology network is that it is only possible to measure the spectra of optical power passing through the axon *and* the dendrite weight banks, returning a spectrum product. This would normally not be an issue, except for the fact that each axon microring needs to be biased to the same wavelenegth resonance as a dendrite microring. Two superimposed troughs or peaks in a spectrum become difficult to distinguish from a single larger peak or trough. As such, an extra step needs to be added during the track to bias (the third step of the basic procedure), as illustrated in Figure 10. One dendrite and one axon are brought to wavelengths immediately below and immediately above the desired resonance through the same mechanism as the previous section. Then, the troughs are moved closer and closer until our peak-finding algorithm begins to mistake them for a single peak. From there, a new feedback control loop takes hold, trying to minimize the width of this double-peak while keeping it on center. This leads to the following controller:

$$\Delta i^2 = \pm k_p * K * (\text{err}_{fwhm} + \text{err}_\lambda) \tag{20}$$

Where the right most peak uses the minus sign, err_{fwhm} is the width of the trough, and err_λ is the deviation of the center of the double-trough. After this slight change to the bias calculation, the rest of the calibration procedure is carried out similarly, except for making sure the two peaks are slightly detuned during the sweep for the K matrix so their trough locations can be distinguished. An annotated procedure can be found in Appendix B.2.

Both simple and cascaded calibration procedures were carried out on simulated devices

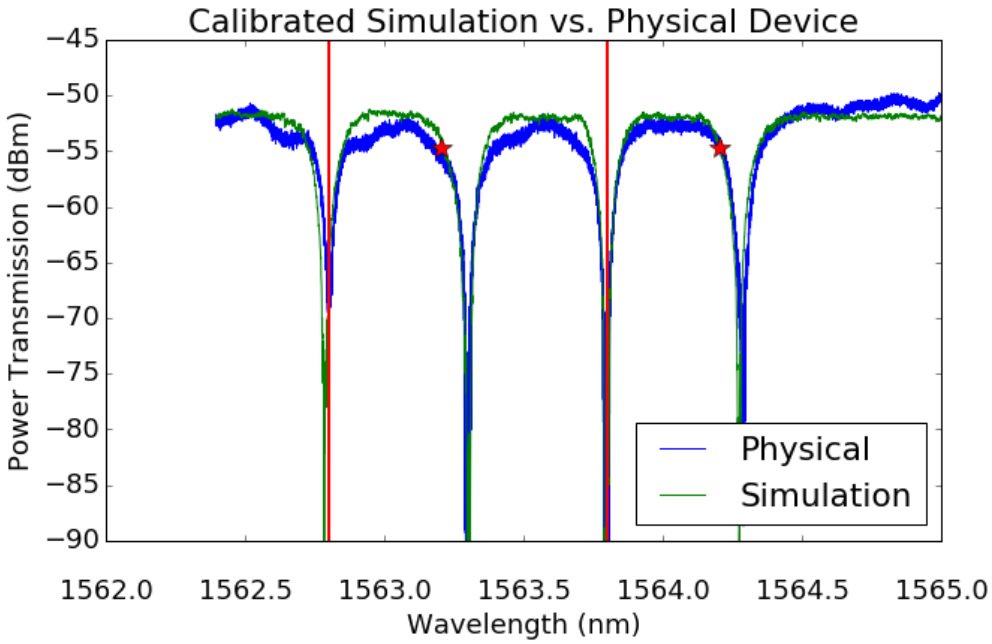

Figure 11: Spectrum comparison between a simulation of the calibrated model and real cascaded microrings. The vertical lines show the requested axon wavelengths, and the red stars show the requested dendrite transmission (-3dBm relative to the maximum, or about 50% transmission). Note the good agreement between calibrated simulation and experiment.

to great success, but to test the accuracy of those simulations, the more difficult cascaded procedure was also tested on a physical 2-axon, 2-dendrite network branch. After determining all the physical parameters and loading them into the calibration model, the model was used to control the physical device. Initial results are shown in Figures 11 and 12. In the former, the calibration model was asked for the electrical current vector that would allow the axons to be detuned by 0.5nm and the dendrites in order to allow 50% optical power transmission (equivalent to -3dBm relative to maximum). These currents were then fed into both a simulation of the calibration device and the physical device, and the resonances ended up very close to their requested locations. Figure 12 sweeps through axon location requests between -0.5nm and 0.2nm relative to bias. Note both how the dark blue axon trough follows the requested trajectory very closely and how the dendrite is held at a constant resonance despite the presence of thermal cross-talk. To quantify this, Figure 13 compares the total wavelength errors between a fully calibrated model and one that neglected thermal cross-talk. Factoring in cross-talk decreases errors significantly, showing that the thermal calibration procedure reasonably estimates the off-diagonal elements of the K matrix.

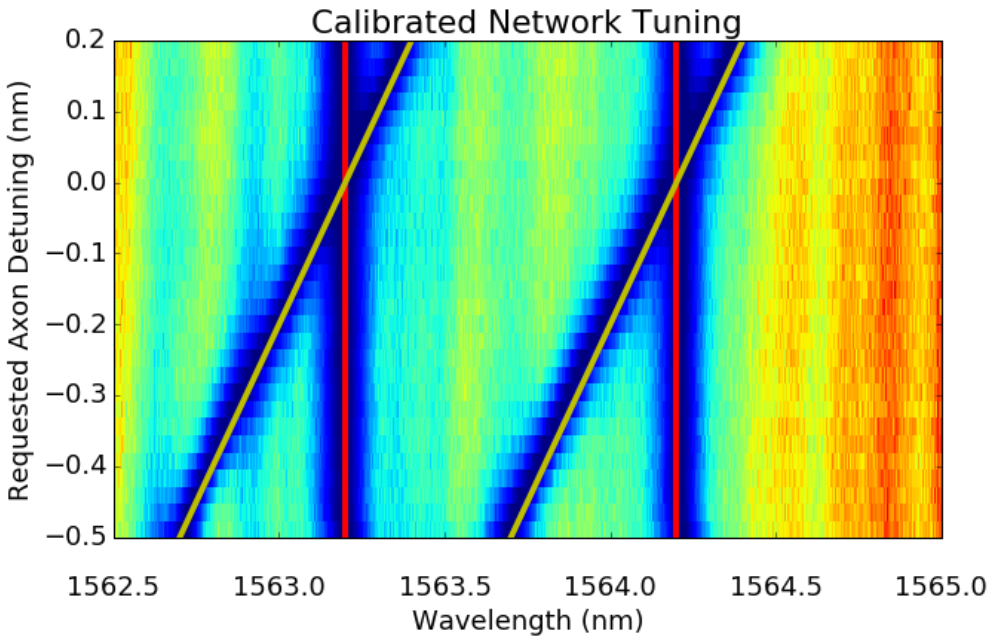

Figure 12: Evidence of successful axon tuning. The yellow line corresponds to the requested axon tuning, and the red line corresponds with the requested dendrite tuning. The colormap represents the physical spectrum data (imagine Figure 11 flipped with the troughs pointing into the page). Note how the axon trough (dark blue) very closely matches the requested trough location.

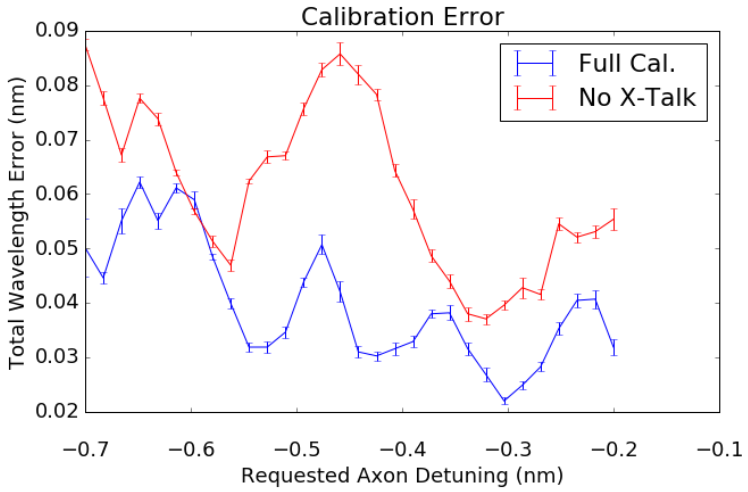

Figure 13: Comparison of total wavelength error between using a fully calibrated model and a model that neglects thermal cross-talk (a more naive calibration algorithm).

3.3 Network Training and Simulated Results

With network calibration mapping the electrical state of the network to optical transmission through each weight bank, the next step is the determine the optimal transmissions for nonlinear classification of a given dataset. This optimization problem is generally solved using *backpropagation*, which is stochastic gradient descent applied to a large composition of functions. The first step is to determine the nonlinear activation function in each layer. For the photonic network, this is the axon, where the relationship between the shift in wavelength and optical power transmission is the nonlinear Cauchy-Lorentz distribution:

$$h(\Delta\lambda) = \frac{\Delta\lambda^2}{\gamma^2 + \Delta\lambda^2} \tag{21}$$

However, $\Delta\lambda$ itself has a *quadratic* dependence on the current. This second nonlinear function is:

$$\Delta\lambda(i) = K[(i^2 + I_0)^2 - I_0^2] \tag{22}$$

Where I_0 is the bias current where the microring is on resonance with the wavelength channel. Here, it is assumed that the current is small enough that thermal cross-talk is negligible, so K is just the *diagonal* of the K matrix. The final activation function $f(i) = h(\Delta\lambda(i))$, is therefore a composition of these two nonlinear functions. This function and its gradient, both of which are required for backpropagation, are shown in Figure 14.

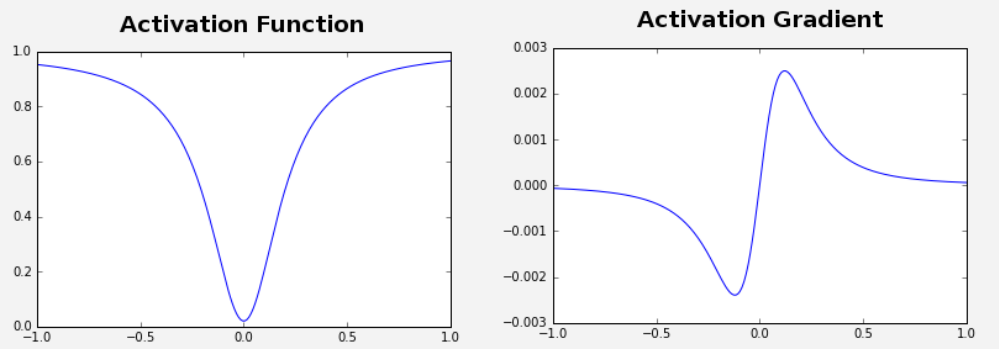

Figure 14: (Left) Activation function used in backpropagation and **(Right)** its gradient. This is a composition of a quadratic to convert current to temperature and the upside-down Cauchy-Lorentz distribution of the axon.

The output of the activation function is power transmission in the domain $(0, 1)$, so there are many physical quantities in each layer that go into converting the previous layer's power transmission into the next layer's current. Using the transfer function of the amplification circuit (Equation 17) and the attenuated optical pump power p, we can construct the

entire composition of functions that maps the optical input $x_j^0 \in (0,1)$ to the hidden layer x_k^1:

$$x_k^1 = f(\frac{R_t^0 * R(\sum_j w_{kj}^0 x_j + b_p^0) + b_v^0}{R_s^0} + b_k^0) \tag{23}$$

where $j = 1, 2$ for the two inputs and $k = 1, 2, 3$ for the three outputs. The second layer of the network is simply a single weight-bank dendrite, as the final signal is a scalar that does not need to go back into the optical domain. In addition, this layer does not have any extra optical bias b_p, since all three channels are used. The final voltage output y can therefore be written as:

$$y = R_t^1 * R(\sum_k w_k^1 x_k^1) + b_v^1 \tag{24}$$

The final binary classificaiton is simply the sign of y. While these equations provide a full

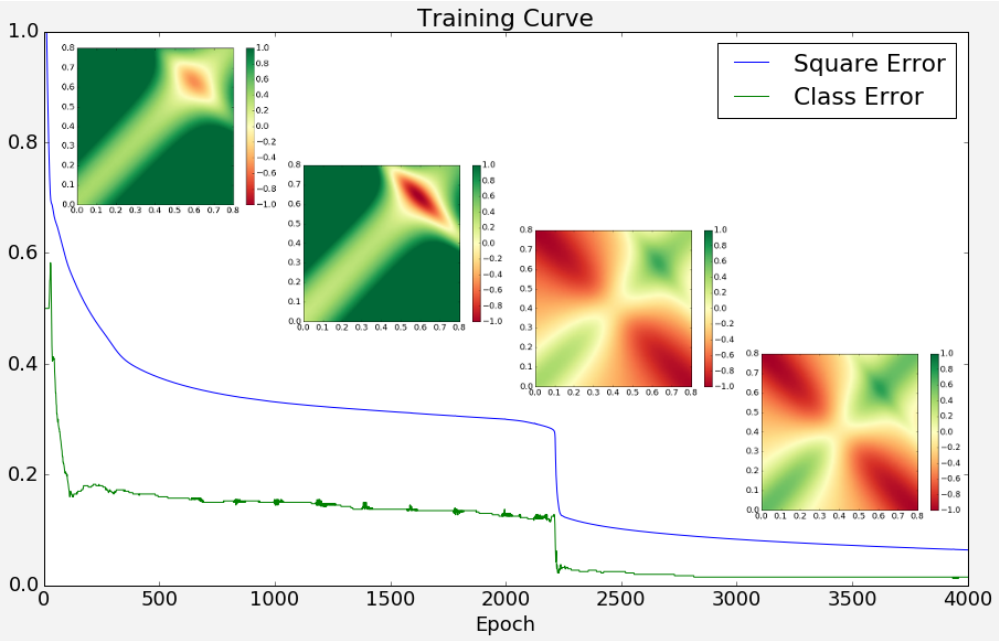

Figure 15: Backpropagation training trajectory on XOR data. Square error is the difference between the continuous variable y and perfect values of -1 and 1, while the class error is the percentage of data that is completely mis-classified due to y being the wrong sign. A straight, diagonal line is a metastable state approached in the first couple thousand epochs. Then, the system is knocked into the more stable, more accurate classification state.

description of the network, the extra physical constants make the backpropagation algorithm unnecessarily tedious and more sensitive to parameters such as learning rate and initial conditions. One solution is to distill the network dynamics to isolate just the activation

function by introducing "virtual" weights and biases:

$$W_{kj}^0 = \frac{R_t^0 * R * w_{kj}^0}{R_s^0} \tag{25}$$

$$W_k^1 = R_t^1 * R * w_k^1 \tag{26}$$

$$B_k^0 = f(\frac{R_t^0 * R * b_p^0 + b_v^0}{R_s^0} + b_k^0) \tag{27}$$

$$B^1 = b_v^1 \tag{28}$$

Then, the total composition of functions then distills to:

$$x_k^1 = f(\sum_j W_{kj} x_j^0 + B_k^0) \tag{29}$$

$$y = \sum_k W_k x_k^1 + B_k^1 \tag{30}$$

It were these simplified functions that were fed into the backprop algorithm to find the optimal values for W and B. This algorithm was tested on a generalized XOR dataset, where opposite quadrants are classified the same, and managed to find weights and biases that achieved greater than 99% classification accuracy. The training curve can be seen in Figure 15, and the final training results are in Figure 16. The backprop code, which used the TensorFlow framework, can be found in Appendix B.3, but a more complete description of the backprop algorithm can be found in Appendix A.1.

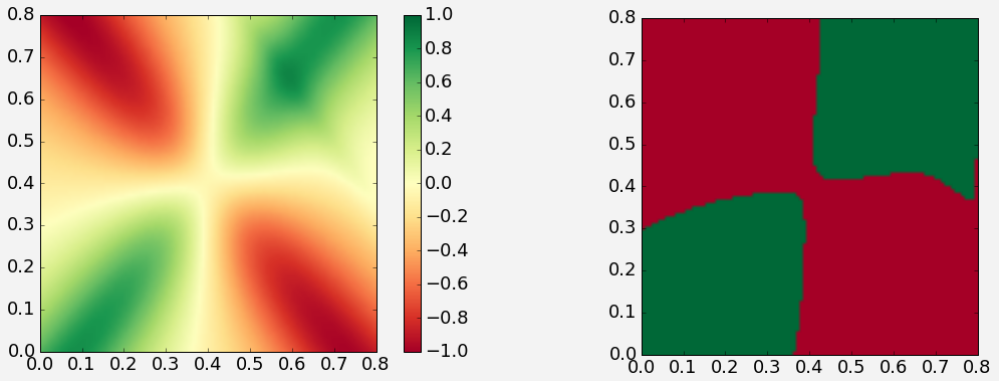

Figure 16: Final results of XOR backpropagation on the abstract network model and activation function. **(Left)** Continuous output variable. **(Right)** Output variable thresholded for binary classification.

After backpropagation, the resulting virtual weights and biases were converted back into the physical weights and biases. Resistance values were fixed such that the physical

weights were in the interval $[-1, 1]$ and the physical biases were at reasonable currents and voltages. These final parameters were loaded into a pre-calibrated network and the network was simulated. Annotated simulation code can be found in Appendix B.4. The resulting classification space is shown in Figure 17, showing decent agreement with the ideal backprop output. Performance was somewhat degraded due to noise, cross-talk, and calibration parameters not exactly matching the parameters used in backprop.

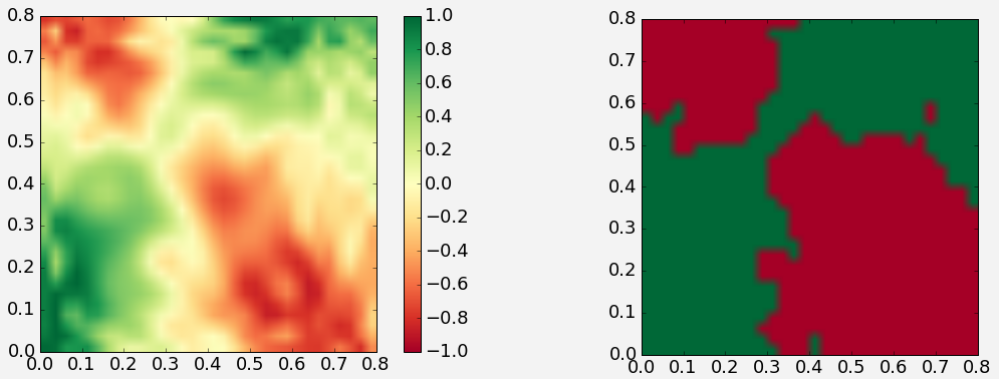

Figure 17: Final results feeding the backpropagation weights into the simulated network after these weights were converted to non-abstract, physical weights. **(Left)** Continuous output variable. **(Right)** Output variable thresholded for binary classification.

4 Next Steps: Mode Division Multiplexing

4.1 Motivation and Operating Principle

Looking back at Figure 2, the main draw of OEO networks is their ability to handle larger signal bandwidths for a given layer size. However, with the network scheme investigated in this thesis (labeled "Resonators" in Figure 2), there exists a hard limit on the size of each optical layer that is independent of input signal bandwidth. From Section 2.1, the Cauchy-Lorentz distribution is only a valid approximation of the microring spectrum around a single resonant wavelength. In reality, a microring has multiple resonant wavelengths, one for each integer multiple of the fundamental wavelength of the ring, and each resonance has its own Lorentz trough. The gap between these resonances is called the *free spectral range* of the microring. Since microring resonance troughs have a finite spectral width, only a finite amount of them can fit effectively in this free spectral range. This limits the total amount of wavelength channels that can be used in a single axon-dendrite branch.

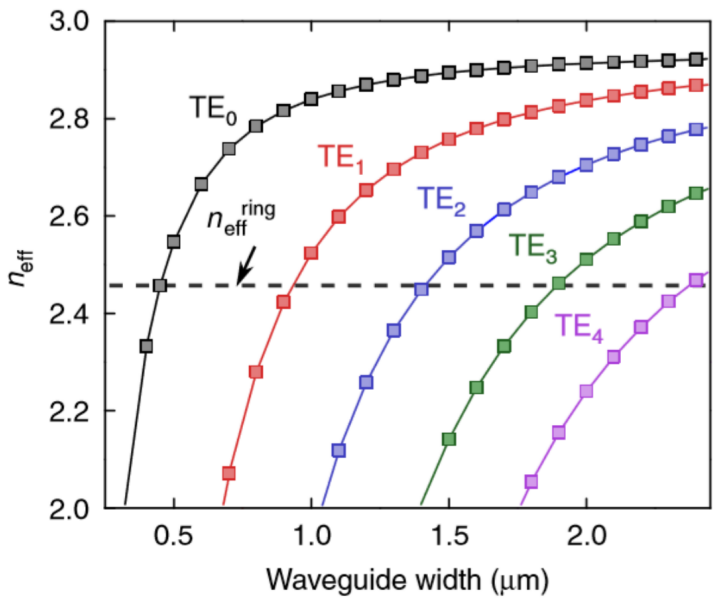

Figure 18: Graph of the effective index for each transverse mode as a function of the waveguide width. The dashed horizontal line is the effective index of a standard-width single-mode waveguide used in all of the microrings in this project. Graph Credit: [8]

One way around this limitation is to expand the network to use both wavelengths and *transverse spatial modes* to differentiate between channels. Transverse modes look at how

the magnitude of the electric fields changes across the waveguide in the transverse direction relative to the direction of propagation. Most integrated optics operate in single-mode waveguides, where the electric field is strongest in the center of the waveguide and tapers off towards the edges. This is called the fundamental mode. However, if the waveguide is sufficiently wide, other possible solutions exist with the magnitude of the electric field peaking more than once. The electric field peaks twice in the first excited mode, three times in the second excited mode, and so on. Since these modes are orthogonal, channels of a given wavelength in two separate transverse modes do not mix in a straight waveguide, allowing them to be distinguished. The total number of channels in a layer is now the product of the number of wavelength channels and the number of modes, theoretically increasing the layer size by an order of magnitude.

The process of adding multiple single-mode channels onto a multi-mode waveguide is called *mode-division multiplexing* (MDM) and relies on the fact that optical coupling can only happen effectively between two media that have the same index of refraction. Each mode experiences a different index of refraction which is dependent on the width of the waveguide, as shown in Figure 18. By adiabatically varying the width of the waveguide, the index of refraction for the desired mode can be adjusted to match the index of the fundamental mode in a single-mode waveguide, allowing for light to couple back and forth. Power oscillation occurs over the course of one *beat length,* so it is necessary to separate the two waveguides after half a beat length for maximum coupling. An example of coupling between the fundamental mode and the second excited mode for a matched width and an unmatched width is shown in Figure 19.

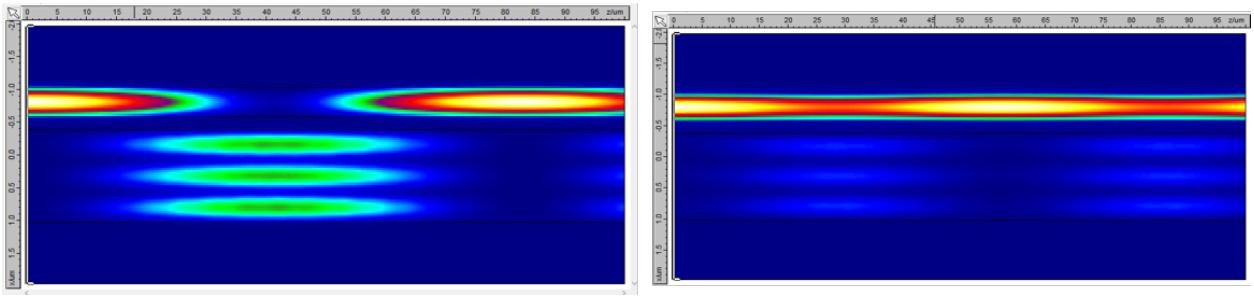

Figure 19: Simulated power coupling between a single-mode waveguide and the second excited mode of a larger waveguide. **(Left)** Full coupling when the effective indices of refraction match. **(Right)** Significantly diminished coupling when the effective indices do not match.

4.2 Experimental Validation

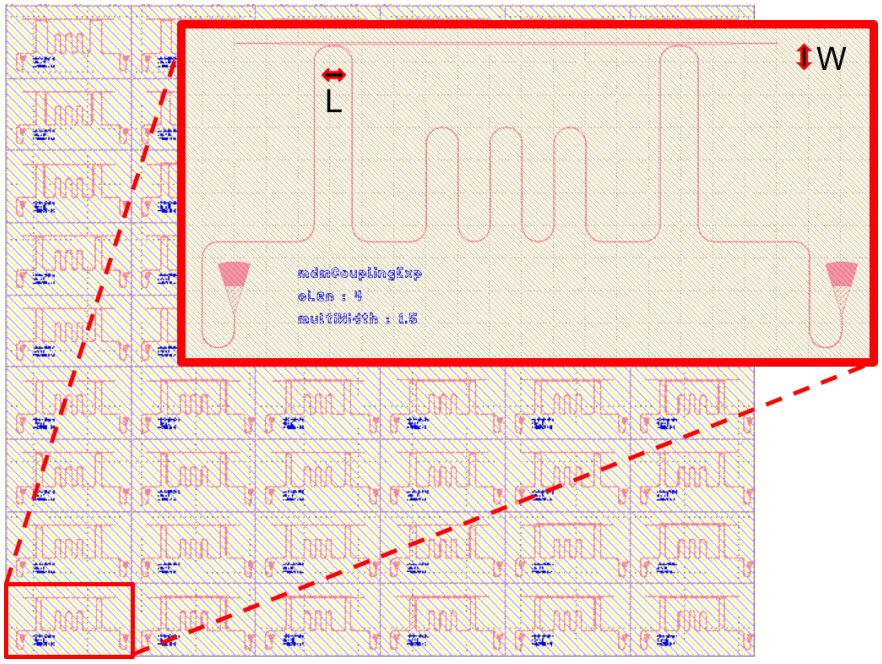

Figure 20: Asymmetric Mach-Zehnder interferometer. This experiments sweeps over different coupling lengths and waveguide widths in order to determine the optimal geometry for coupling to each excited mode.

Because the coupling strength is so strongly dependent upon the waveguide width and the coupling length, it was necessary to construct a physical experiment to verify the accuracy of the simulations. This experiment, shown in Figure 20, is an asymmetric Mach-Zehnder interferometer, which provides a way to estimate the coupling coefficient for a given length and width. The exact mechanism is detailed in Appendix A.2, but suffice it to say here that the magnitude of the oscillations in the spectrum of this device (called the extinction ratio) is large oscillations for 50% coupling and small for 0% and 100% coupling. By setting the coupling length to approximately a quarter of the beat length (50% coupling), the optimal width maximizes the extinction ratio. An example of this part of data from this part of the procedure can be seen in Figure 21. Once at the optimal width, the optimal coupling length of a half beat length will minimize the extinction ratio.

While the parameter space was too coarse and the devices too noisy to hone in on the exact optimal widths and lengths, aggregate data (as shown in Figure 22) was consistent with simulated results. Therefore, the next step is to look into repeating the results of previous network experiments using MDM channels.

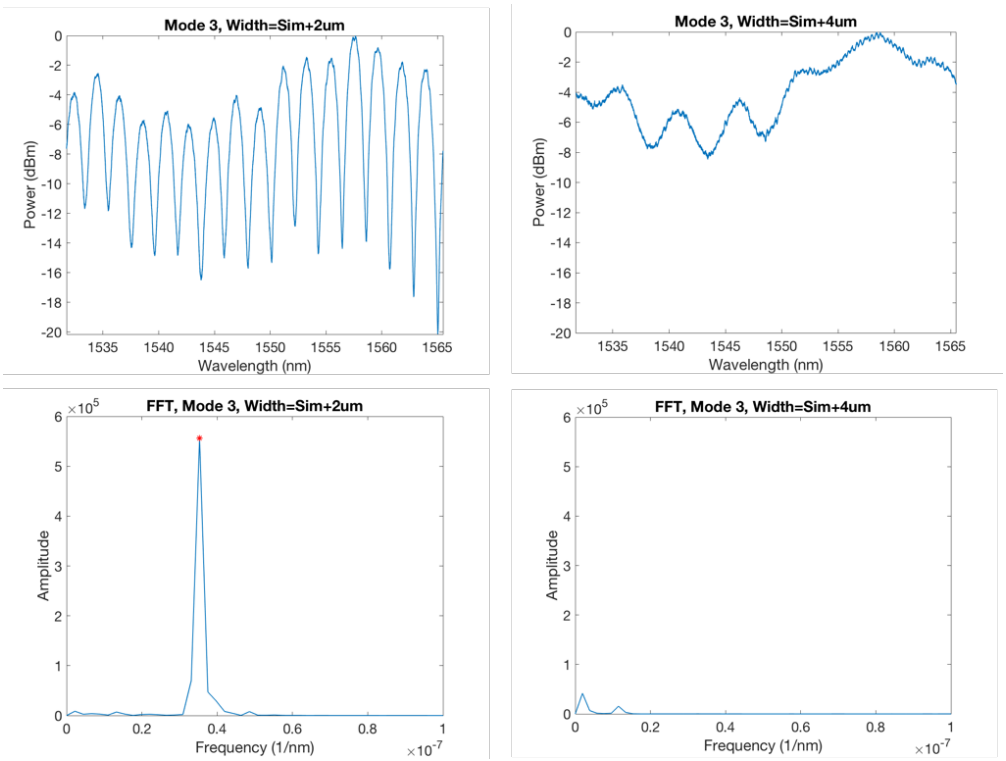

Figure 21: Example of transmission spectra from a device with 50% coupling **(Left)** and a high extinction ration, and 0% coupling **(Right)** and a low extinction ratio due to mismatched widths.

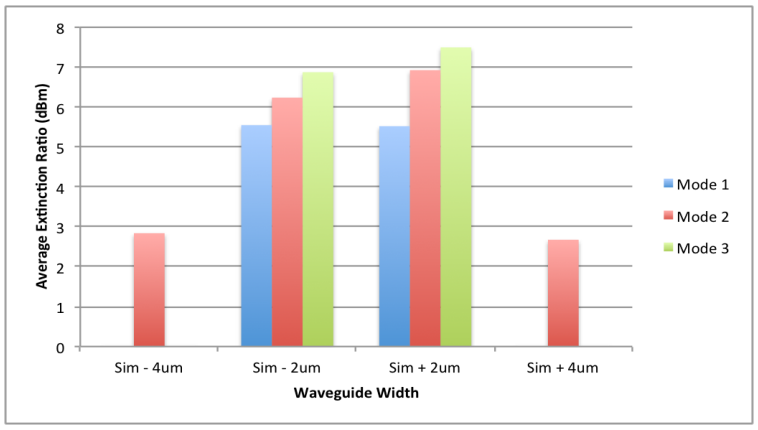

Figure 22: Aggregate data across all coupling lengths, comparing the extinction ratio of different waveguide widths for each mode. Note that the waveguide widths closer to the simulated optimal width have a higher extinction ratio. This is evidence consistent with simulated results.

4.3 Challenges

Even with optimal coupling geometry, MDM is not without further challenges. First and foremost, standard split waveguides do not split power evenly on each mode channel. It will be impossible to use a star topology network with MDM channels, necessitating a switch to the more-difficult-to-control hairpin topology. This is not really a limitation in itself, but it necessitates the challenge of switching topologies.

Secondly, spatial modes become less orthogonal as the spatial geometry changes, such as when the waveguide needs to bend. As shown in Figure 23, even if optical power starts out only in the fundamental mode, it will mix into other modes in anything but a straight waveguide. Assuming minimal attenuation, such *intermodal mixing* can be modeled as a unitary matrix M, and it is extremely sensitive to initial fabricaiton conditions, requiring direct measurement. Fortunately, once M is determined, its inverse can just be added to the weight bank to yield the desired weights, but having to determine M in the first place could make the calibration procedure exceedingly complicated. That said, increasing network throughput by an order of magnitude is worth pursuing MDM channels in future work.

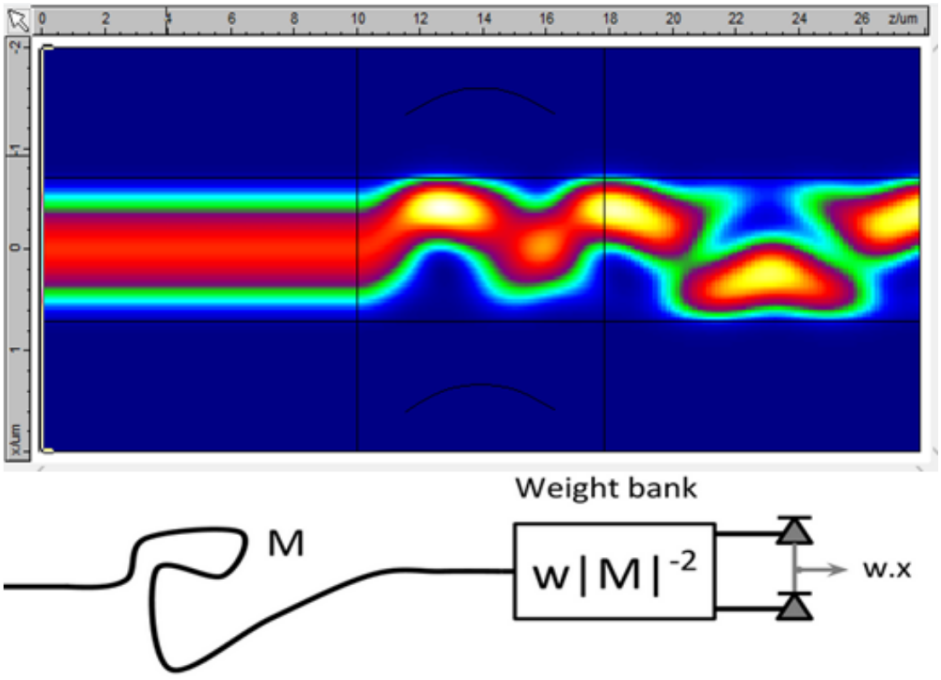

Figure 23: (Top) Simulation of the fundamental mode of a multi-mode waveguide mixing with the first excited mode. **(Bottom)** Since this mixing matrix is unitary (assuming minimal attenuation), it can be inverted by the weight bank to yields the desired weights.

5 Discussion and Future Work

The work in this thesis demonstrated the ability to calibrate a feed-forward WDM network and train it to solve nonlinear classification problems. The immediate next step is to demonstrate a feed-forward network on a physical device. Fortunately, experimental verification of the calibration procedure also demonstrated a good agreement between simulation and experiment. Therefore, it is reasonable to deduce from the successful simulation of the 2-3-1 feed-forward network that the physical device will be equally successful.

However, these networks still face many practical limitations that could prevent them from reaching their full potential throughput. Even before moving to MDM networks, the current WDM scheme has plenty of room for improvement. Most pressingly, thermal tuning is simply a slow way of adjusting the index of refraction of silicon. This is fine for applying the large, stable biases and weights, but puts a significant limit on signal bandwidth when used in the path of actual computation. All of this is in addition to degraded performance due to the thermal cross-talk between the active network signal and all of the network weights. Other faster methods of modifying the index of refraction of silicon, such as by carrier depletion, need to be investigated.

Finally, future work should include an application study. Possible applications in radio frequency real-time computing and scientific computing were mentioned in Section 1.1, but it is currently unknown which of those applications would most benefit from the extra throughput that OEO networks can provide. This is especially important considering that it will be difficult to keep OEO networks as power-efficient as existing electronic networks, a problem that has yet to be tackled.

Overall, photonic neural networks have the potential to benefit applications with high bandwidth demand, and this thesis took steps towards its realization. There are still many more problems to solve to finish crossing the gap from curiosity into utility, but that also leaves plenty of room for future research.

6 References

[1] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.

[2] http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

[3] Akopyan, Filipp, et al. "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 34.10 (2015): 1537-1557.

[4] Friedmann, Simon, et al. "Reward-based learning under hardware constraints-using a RISC processor embedded in a neuromorphic substrate." arXiv preprint arXiv:1303.6708 (2013).

[5] Benjamin, Ben Varkey, et al. "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations." Proceedings of the IEEE 102.5 (2014): 699-716.

[6] A. Tait, et al., "Demonstration of a silicon photonic neural network," *Photonics Society Summer Topical Meeting Series (SUM)*, IEEE, 2016.

[7] A. Tait, et al., "Multi-channel control for microring weight banks," Opt. Express 24, 8895-8906 (2016).

[8] L. Luo, et al., "WDM-compatible mode-division multiplexing on silicon chip," Nature Communications, vol. 5, no. 3069, Macmillian Publishers Limited, 2014.

A Math Appendices

A.1 Backpropagation

Given a neural network model, the goal of backpropagation is to determine the weights and biases that minimize some cost function using stochastic gradient descent. In the 2-3-1 neural network performing XOR classification, for a given input x_j^0, there is some output y and a desired output $d = -1, 1$. The network model is as follows:

$$x_k^1 = f(\sum_j W_{kj}^0 x_j^0 + B_k^0) \tag{31}$$

$$y = \sum_k W_k^1 x_k^1 + B^1 \tag{32}$$

One popular cost function is the square of the error between y and d:

$$E = \frac{1}{2}(d - y)^2 \tag{33}$$

In stochastic gradient ascent, the weights and biases are updated in the direction of the gradient of E at some sample pair x_j^0 and y. Equivalently, each individual weight or bias is updated by an amount proportional to the partial derivative of E with respect to that weight. Because the cost function needs to be minimized, the update happens opposite the gradient, or with a negative sign in front of the partial derivative.

$$\Delta W_{jk}^l = -\eta \frac{\partial E}{\partial W_{jk}^l} \tag{34}$$

$$\Delta B_k^l = -\eta \frac{\partial E}{\partial B_k^l} \tag{35}$$

Here η is called the *learning rate*, and it is set manually prior to training. If the learning rate is too small, the network will take a long time to converge on the minimum. However, if the learning rate is too large, the network will never converge, jumping back and forth across the minimum.

Using the chain rule, it is possible to write out these partial derivatives in terms of known quantities:

$$\begin{aligned} \Delta W_k^1 &= -\eta \frac{\partial E}{\partial y} \frac{\partial y}{\partial W_k^1} \\ &= \eta (d - y) x_k^1 \end{aligned} \tag{36}$$

$$\begin{aligned}\Delta B^1 &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial B^1} \\ &= \eta(d-y)\end{aligned} \tag{37}$$

$$\begin{aligned}\Delta W^0_{jk} &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial x^1_k}\frac{\partial x^1_k}{\partial W^0_{jk}} \\ &= \eta(d-y)W^1_k f'(\sum_j W^0_{jk}x^0_j + B^0_k)x^0_j\end{aligned} \tag{38}$$

$$\begin{aligned}\Delta B^0_k &= -\eta \frac{\partial E}{\partial y}\frac{\partial y}{\partial x^1_k}\frac{\partial x^1_k}{\partial B^0_k} \\ &= \eta(d-y)W^1_k f'(\sum_j W^0_{jk}x^0_j + B^0_k)\end{aligned} \tag{39}$$

One update is performed for each point in the dataset, and exhausting all points in the training dataset is called an *epoch*. A learning curve (Figure 15) plots the change in the average value of the error function (and the binary classification error) for each epoch.

A.2 Multi-Mode Interferometer

The following experiment was designed as part of the Junior Independent Work: "Mode division multiplexing (MDM) weight bank design for Use in photonic neural networks."

The asymmetric mach-zehnder interferometer has an input waveform E_{in} and produces the output E_{out} and an unused output E_{taper}. The device's optical transfer function of the device can be written as:

$$\begin{bmatrix} E_{out} \\ E_{taper} \end{bmatrix} = M \begin{bmatrix} E_{in} \\ 0 \end{bmatrix} \tag{40}$$

Where M is a composition of both single-mode to multi-mode couplers and a phase shift $\Delta\phi$ due to the difference in optical length of the two paths through the interferometer.

$$M = M_{coupler} * M_{\Delta\phi} * M_{coupler} \tag{41}$$

$M_{coupler}$ takes on the same form as the coupler in Section 2.1, where the cross coupling term α is used instead of the self-coupling term r.

$$M_{coupler} = \begin{bmatrix} \sqrt{1-\alpha} & i\sqrt{\alpha} \\ i\sqrt{\alpha} & \sqrt{1-\alpha} \end{bmatrix} \tag{42}$$

Combined with the phase shift $\Delta\phi = k * \Delta L$, where $k = 2\pi/\lambda$ is the wavenumber of the light, the rest of the matrix can be determined:

$$M = \begin{bmatrix} \sqrt{1-\alpha} & i\sqrt{\alpha} \\ i\sqrt{\alpha} & \sqrt{1-\alpha} \end{bmatrix} \begin{bmatrix} e^{ik\Delta L} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{1-\alpha} & i\sqrt{\alpha} \\ i\sqrt{\alpha} & \sqrt{1-\alpha} \end{bmatrix} \tag{43}$$

$$\left| \frac{E_{out}}{E_{in}} \right|^2 = \left| (1-\alpha)e^{ik\Delta L} - \alpha \right|^2 = \alpha^2 + (1-\alpha)^2 - 2\alpha(1-\alpha)cos(k\Delta L) \tag{44}$$

Therefore, a power spectrum of the device should exhibit sinusoidal oscillations ontop of a constant offset. The amplitude of these oscillations, called the *extinction ratio*, is only dependent on the coupling coefficient α between the single-mode and multi-mode waveguides. It reaches its maximum value when $\alpha = 0.5$ and its minimum value when $\alpha = 0, 1$.

B Code Appendices

B.1 Dendrite Calibration Procedure

B.1.1 Step 1: Initialization

Import all classes and set up global parameters

```
In [1]: # Imports and reservations
        import numpy as np
        import matplotlib.pyplot as plt
        import lightlab.instruments as inst
        import lightlab.model as m
        from lightlab.util.modeling import kOSAPwr, dbm2lin,
            kOSASpacing, CurrentUnit, MrrOut
        from lightlab.util.calibrating import SpectrumMeasurementAssistant
        from lightlab.util import io
```

```
In [2]: # Global Parameters, Available to Calibration Model
        wlChannels = [1550, 1552, 1554, 1556]
        wlRange = [1530, 1559]
        currentChan = [5, 3, 6, 4]
        numRings = 4
        minPeakDist = 0.5 / kOSASpacing # 0.5nm spacing for peak-finder
```

B.1.2 Step 2: Initialize Hidden Model

Create a virtual model that will mimic the instruments. This will remain hidden during calibration.

In this case, we are making a 4-ring Lorentzian Filter Bank. Then, register it to the Instruments Module.

```
In [3]: # Thermal Group, Uses 4 current channels
        therm = m.ThermalGroup(currentChan)

        # Model Parameters, NOT available to calibration model
```

```
        attenuation = 0.0001 # 40dB baseline attenuation
        lfwhm = 0.2          # 0.1nm fwhm of lorentzian
        latten = 0.98        # Attenuation at resonance
        heatBias = {currentChan[0]: 2.5, currentChan[1]: 2.0,
            currentChan[2]: 1.5, currentChan[3]: 1.0}

        # Random K with extra on diagonal to denote primary filament
        K = np.multiply(np.absolute((0.1 * np.random.randn(numRings,
            len(currentChan)) + 0.1) + 0.5*np.eye(numRings)), 200)

        # FilterBank Module
        fb = m.FilterBank(therm, numRings)
        fb.setAttenuation(attenuation)

        fb.setBiasParams(wlChannels, latten * np.ones(numRings),
            lfwhm * np.ones(numRings))

        # Set K and bias for Thermal Group
        therm.setK(K)
        therm.setHeatBias(heatBias)

In [4]: # Reserve Current Channels, Add Current Channels
        inst.togglePhony(True, fb)
        token = inst.reserveCurrentChan(currentChan)

In [5]: fb.setOsaOut(MrrOut.kThru.value)
        nm, dbm = inst.spectrum(wlRange)
        plt.plot(nm, dbm)
        plt.show()
```

B.1.3 Step 3: Ascription

Map primary filament to each peak.

```
In [6]: # Initialization
        calCurrentChan = np.array([3, 4, 5, 6])
        calCurrentState = dict() # in mW
        for ch in calCurrentChan:
            calCurrentState[ch] = 0

        spctAssist = SpectrumMeasurementAssistant(nChan=numRings,
            arePeaks=False, visualize=False)

In [7]: # Set Base Numbers
        baseTune = calCurrentState
        baseLams = np.array([r.lam for r in spctAssist.resonances()])
        tuneBy = 0.01 # in mW/Ohm"
        ascrBuilder = np.arange(len(calCurrentChan))
        kEstBuilder = np.arange(len(calCurrentChan))

In [8]: from time import sleep
        # Run Ascription
        for iChan, ch in enumerate(calCurrentChan):
            inst.setCurrentChanTuning({ch:
                baseTune[ch] + tuneBy}, token, CurrentUnit.mW)
            spect = spctAssist.fgSpect()
            presLams = np.array([r.lam
                for r in spctAssist.resonances(spect)])
            inst.setCurrentChanTuning({ch: baseTune[ch]}, token)
            shifts = presLams - baseLams
            ascrBuilder[iChan] = np.argmax(shifts)
            kEstBuilder[iChan] = max(shifts) / tuneBy

        # Re-Order Current Channels
        newCalCurrentChan = [calCurrentChan[j]
            for j in np.argsort(ascrBuilder)]
        kEstTemp = kEstBuilder[np.argsort(ascrBuilder)]

In [9]: # Validation, make sure re-ordered list is
```

```
        for j in range(len(newCalCurrentChan)):
            assert newCalCurrentChan[j] == currentChan[j]
```

B.1.4 Step 4: Create Calibration Model

Create an empty calibration model to be filled, using the ascribed channels for validation purposes.

```
In [10]: calTherm = m.ThermalGroup(newCalCurrentChan)
         calFB = m.FilterBank(calTherm, numRings)
```

B.1.5 Step 5: Background Removal and Tracking

PI-Controller to move channels onto wavelengths.

At the end, we should be able to set the **biases.**

```
In [11]: ### Background Removal

         avgOnSpect=3
         detuneByFwhms = 3

         # Get resonance FWHMs.
         resFwhms = np.array([r.fwhm for r in spctAssist.resonances()])
         displacedWls = detuneByFwhms * resFwhms

         # Get Raw Spectrum
         baseRawSpct = spctAssist.fgSpect(avgCnt=avgOnSpect,
            bgType='smoothed')
         # Tune to displace resonances, look at new spectrum, tune back
         displTuning = dict()
         for j in range(len(displacedWls)):
             displTuning[newCalCurrentChan[j]] =
                displacedWls[j] / kEstTemp[j]
         inst.setCurrentChanTuning(displTuning, token, CurrentUnit.mW)
         displacedRawSpct = spctAssist.fgSpect(avgCnt=avgOnSpect,
            bgType='smoothed')
```

```
        # Return to base
        inst.setCurrentChanTuning(baseTune, token, CurrentUnit.mW)

        # Update Background
        spctAssist.setBgTuned(baseRawSpct, displacedRawSpct)
```

In [12]:
```
# See Peaks w/ Background Removed
spctAssist.fgResPlot()
plt.show()
```

In [13]:
```
### Tracking
precision = 0.005 # Threshold
propCoef = 0.5 # kP
avgCnt = 4
targets = np.array(wlChannels)
nowTune = baseTune
appxThrmCoefs = kEstTemp
inst.setCurrentChanTuning(baseTune, token, CurrentUnit.mW)

for i in range(100):
    spect = spctAssist.fgSpect(bgType='tuned', avgCnt = avgCnt)
    spect.simplePlot()
    actualPeaks = np.array([r.lam
       for r in spctAssist.resonances(spect)])
    errs = targets - actualPeaks
    io.printProgress('i=', i, ', error=', max(abs(errs)))
    if max(abs(errs)) < precision:
        print('\nTracking complete')
        break
    # recenter wlRange to avoid other FSR resonances
    wlRangeTight = np.array([min(min(actualPeaks), min(targets)),
       max(max(actualPeaks), max(targets))])
    newwlRange = np.mean(wlRangeTight) +
       np.diff(wlRangeTight) * np.array([-1, 1]) / 2 * 2
```

```
            spctAssist.wlRange = newwlRange
            try:
                for j in range(len(newCalCurrentChan)):
                    ch = newCalCurrentChan[j]
                    nowTune[ch] = nowTune[ch] +
                       propCoef*errs[j] / appxThrmCoefs[j]
                inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
            except io.RangeError as err:
                print('Out of range during tracking. See plot')
                spctAssist.fgResPlot()
                raise err

        plt.show()


Tracking complete


In [14]: spctAssist.fgResPlot()
         plt.show()

In [15]: calTherm.setHeatBias(nowTune, CurrentUnit.mW)

In [16]: # Validation
         threshold = 0.01

         calBias = np.array(sorted([CurrentUnit.toVolt(b, CurrentUnit.mW)
            for b in calTherm.heatBias]))
         virtBias = np.array(sorted(list(heatBias.values())))
         diff = np.absolute(calBias - virtBias)
         print(diff)
         for d in diff:
             assert d < threshold

[ 0.00272099  0.00028576  0.00065551  0.00018863]
```

```
In [17]: calFB.setBiasParams([r.lam for r in spctAssist.resonances()])

In [18]: ## Validation
         threshold = 0.01

         calBias = [r.lam for r in spctAssist.resonances()]
         virtBias = np.array(wlChannels)
         diff = np.absolute(calBias - virtBias)
         print(diff)
         for d in diff:
             assert d < threshold
```

```
[ 0.00202127  0.00264774  0.00487436  0.00730028]
```

B.1.6 Step 6: Take Filter Shapes

```
In [19]: spect =spctAssist.fgSpect(avgCnt=5, bgType='tuned')
         filtShapes = np.empty(numRings, dtype=object)
         filtCurves = [None] * numRings
         for i, r in enumerate(spctAssist.resonances(spect)):
             relWindow = 7 * r.fwhm * np.array([-1,1])/2
             proximitySpect = spect.shift(-r.lam).crop(relWindow)
             filtShapes[i] = proximitySpect
             nm, dbm = proximitySpect.shift(r.lam).getData()
             lin = np.clip(dbm2lin(dbm), 0.0, 1.0)
             filtCurves[i] = (nm,lin)
             plt.plot(nm, lin)

         # Store in assistant
         spctAssist.filtShapesForConvolution = filtShapes
         # Store in model
         calFB.setCurve(filtCurves, MrrOut.kThru)
         plt.show()
```

```
In [20]: # Hot-Swap in Calibration Module and make sure things look good
         inst.lockPhony(calFB)
         calFB.setOsaOut(MrrOut.kThru.value)

         # Assume we can pick this up easily
         calFB.setAttenuation(attenuation)

         spctAssist.fgResPlot()
         plt.show()
         inst.releasePhony()
         biasTune = nowTune.copy()
```

B.1.7 Step 7: Fill K Matrix

K is the coefficients between mW and deltaWL.

```
In [21]: nPts = 11
         avgCnt = 5

         nowTune = biasTune.copy()
         biasWL = np.array([r.lam for r in spctAssist.resonances()])
         nowWL = np.copy(biasWL)

         newK = np.zeros((numRings, len(newCalCurrentChan)))

         # For Each Current Channel
         for ich in range(len(newCalCurrentChan)):
             ch = newCalCurrentChan[ich]

             # Shift in a 1nm range around bias WL
             dB = 1.0 / kEstTemp[ich]
             x = np.linspace(max(0.0, (biasTune[ch]-dB)),
                biasTune[ch]+dB, nPts)
             y = np.zeros((numRings, nPts))
```

```
            for ipt, pt in enumerate(x):
                nowTune[ch] = pt
                inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
                nowWL = np.array([r.lam for r in spctAssist.resonances()])
                diff = nowWL - biasWL
                y[:, ipt] = diff

            # Make Linear Fit

            for r in range(numRings):
                yr = y[r, :]
                p = np.polyfit(x, yr, 1)
                newK[r, ich] = max(p[0], 0.0)


            inst.setCurrentChanTuning(biasTune, token, CurrentUnit.mW)
            nowTune = biasTune.copy()

        calTherm.setK(newK)
```

In [22]:
```
        # Percent Error
        err = np.round(np.absolute(100 *
           np.divide(np.subtract(newK, K), K)))

        for i, e in enumerate(err.flatten()):
            if e > 10:
                if K.flatten()[i] < 10:
                    print("Small K: " + str(K.flatten()[i]))
                    continue
            assert e <= 10
```

```
Percent Error:
[[ 0.  1.  2.  0.]
 [ 0.  0.  3.  0.]
```

```
 [ 0.  1.  0.  1.]
 [ 0.  7.  0.  0.]]

Avg K: 45.3534028127


If you see an error >10, that means we made a normal mistake on a small K.
```

And calFB is a calibrated device!

Validation was done at the following points: * Ascription is guaranteed to be correct. * HeatBias is within 0.01mW * Wavelength bias is within 0.01nm * Significant K cross-terms have less than 10% error

B.2 Cascaded Calibration Procedure

B.2.1 Step 1: Initialization

Import all classes and set up global parameters

```
In [1]: # Imports and reservations
        import numpy as np
        import matplotlib.pyplot as plt
        import lightlab.instruments as inst
        import lightlab.model as m
        from lightlab.util.modeling import kOSAPwr,
            dbm2lin, kOSASpacing, CurrentUnit, MrrOut, lin2dbm
        from lightlab.util.calibrating import SpectrumMeasurementAssistant
        from lightlab.util import io
        from bidict import bidict
        from time import sleep
```

```
In [2]: # Global Parameters, Available to Calibration Model
        wlChannels = np.array([1550, 1552, 1554])
        wlRange = np.array([1545, 1560])
        axonChan = np.array([0, 1, 2])
        dendriteChan = np.array([3, 4, 5])

        numRings = 3
```

B.2.2 Step 2: Initialize Hidden Model

Create a virtual model that will mimic the instruments. This will remain hidden during calibration.

In this case, we are cascading 2 2-ring Filter Banks.

```
In [ ]: axonMixed = list([axonChan[1], axonChan[2], axonChan[0]])
        dendriteMixed =
            list([dendriteChan[1], dendriteChan[2], dendriteChan[0]])
        currentChan = axonMixed + dendriteMixed
```

```
        # Thermal Group, Uses 4 current channels
        therm = m.ThermalGroup(currentChan)

        # Model Parameters, NOT available to calibration model
        attenuation = 0.0001 # 40dB baseline attenuation
        lfwhm = 0.1          # 0.1nm fwhm of lorentzian
        latten = 0.98        # Attenuation at resonance
        heatBias = dict()
        for i in range(len(axonMixed)):
            heatBias[axonMixed[i]] = 2.0
            heatBias[dendriteMixed[i]] = 1.5


        # Random K with extra on diagonal to denote primary filament
        K = np.absolute(np.random.randn(2*numRings, 2*numRings) +
            20.0*np.eye(2*numRings))
        print(K)

        # FilterBank Module
        axon = m.FilterBank(therm, numRings)
        axon.setAxon()
        fb = m.FilterBank(therm, numRings)

        fb.setBiasParams(wlChannels, latten * np.ones(numRings), lfwhm *
            np.ones(numRings))
        axon.setBiasParams(wlChannels, latten * np.ones(numRings), lfwhm *
            np.ones(numRings))

        csc = m.Cascade(axon, fb)

        # Set K and bias for Thermal Group
        therm.setK(K)
        therm.setHeatBias(heatBias)
```

In [4]:
```
# Reserve Current Channels, Add Current Channels
```

```
        inst.togglePhony(True, csc)
        token = inst.reserveCurrentChan(currentChan)

In [5]: csc.setOsaOut(MrrOut.kThru.value)

        # Manually Set to Separate Peaks
        baseTune = dict()
        zeroTune = dict()
        for j in range(len(currentChan)):
            zeroTune[j] = 0.0
        baseTune[1] = 0.0
        baseTune[4] = 0.0
        baseTune[2] = 0.0
        baseTune[5] = 0.0
        baseTune[0] = 0.0
        baseTune[3] = 0.0

        inst.setCurrentChanTuning(zeroTune, token, CurrentUnit.mW)
        nm, dbm = inst.spectrum(wlRange)
        plt.plot(nm, dbm), 'b'
        inst.setCurrentChanTuning(baseTune, token, CurrentUnit.mW)
        nm, dbm = inst.spectrum(wlRange)
        plt.plot(nm, dbm, 'g')
        plt.show()
```

B.2.3 Step 3: Ascription

Map primary axon and dendrite filaments to peaks.

```
In [26]: spctAssist = SpectrumMeasurementAssistant(nChan=2*numRings,
    arePeaks=False, visualize=False)
         def ascribe(aChans, dChans, bTune, tuneBy=0.01):

             baseLams = np.array([r.lam
                for r in spctAssist.resonances()])
             pMap = bidict()
```

```
tEstMap = bidict()

nowTune = bTune.copy()

# Run Ascription
for iChan, ch in enumerate(aChans):
    nowTune[ch] = bTune[ch] + tuneBy
    inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
    spect = spctAssist.fgSpect()
    presLams = np.array([r.lam for r in
       spctAssist.resonances(spect)])
    nowTune[ch] = bTune[ch]
    inst.setCurrentChanTuning(nowTune, token)
    shifts = presLams - baseLams
    pNum = np.argmax(shifts)
    pMap[pNum] = ch
    tEstMap[ch] = shifts[pNum] / tuneBy

for iChan, ch in enumerate(dChans):
    nowTune[ch] = bTune[ch] + tuneBy
    inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
    spect = spctAssist.fgSpect()
    presLams = np.array([r.lam for r in
       spctAssist.resonances(spect)])
    nowTune[ch] = bTune[ch]
    inst.setCurrentChanTuning(nowTune, token)
    shifts = presLams - baseLams
    pNum = np.argmax(shifts)
    pMap[pNum] = ch
    tEstMap[ch] = shifts[pNum] / tuneBy

try:
    assert len(list(pMap.keys())) == len(aChans) + len(dChans)
except:
```

```
                print(pMap)
                assert False

            return pMap, tEstMap
```

In [27]:
```
# Run Ascription
pMap, tEstMap = ascribe(axonChan, dendriteChan, baseTune)

calAxonChan = list()
calDendriteChan = list()

for p in sorted(list(pMap.keys())):
    ch = pMap[p]
    if ch in axonChan:
        calAxonChan.append(ch)
    else:
        calDendriteChan.append(ch)

calCurrentChan = calAxonChan + calDendriteChan
```

In [28]:
```
# Validation
for j in range(len(currentChan)):
    try:
        assert calCurrentChan[j] == currentChan[j]
    except:
        print("ERROR: Should be the same...")
        print(currentChan)
        print(calCurrentChan)
        break
```

B.2.4 Step 4: Create Calibration Model

Create an empty calibration model to be filled, using the ascribed channels for validation purposes.

In [29]:
```
calTherm = m.ThermalGroup(calCurrentChan)
```

```
calAxon = m.FilterBank(calTherm, numRings)
calAxon.setAxon()
calFb = m.FilterBank(calTherm, numRings)

calCsc = m.Cascade(calAxon, calFb)
calCsc.setOsaOut(MrrOut.kThru.value)
```

B.2.5 Step 5: Background Removal

Also sets the OSA attenuation in the calibration model.

```
In [30]: avgOnSpect=3
         detuneByFwhms = 3

         # Get resonance FWHMs. The extra sweep is technically
           unnecessary, but code below is cleaner
         resFwhms = np.array([r.fwhm for r in spctAssist.resonances()])
         displacedWls = detuneByFwhms * resFwhms

         # Get Raw Spectrum
         baseRawSpct = spctAssist.fgSpect(avgCnt=avgOnSpect,
            bgType='smoothed')
         # Tune to displace resonances, look at new spectrum, tune back
         displTuning = dict()
         for j in range(len(displacedWls)):
             ch = pMap[j]
             tEst = tEstMap[ch]
             displTuning[pMap[j]] = baseTune[ch] + displacedWls[j] / tEst
         inst.setCurrentChanTuning(displTuning, token, CurrentUnit.mW)
         displacedRawSpct = spctAssist.fgSpect(avgCnt=avgOnSpect,
            bgType='smoothed')

         # Return to base
         inst.setCurrentChanTuning(baseTune, token, CurrentUnit.mW)
```

```
        # Update Background
        spctAssist.setBgTuned(baseRawSpct, displacedRawSpct)
```

In [31]:
```
        # See Peaks w/ Background Removed
        spctAssist.fgResPlot()
        plt.show()
```

In [37]:
```
        # Set Attenuation and Validate
        calAtten = spctAssist.getAtten()
        err = 100*np.absolute(np.divide(calAtten-attenuation,
           attenuation))
        try:
            assert err < 4
        except:
            spctAssist.rawSpect().simplePlot()
            plt.show()
            print(10*np.log10(calAtten))
            print(err)
        # Less than 4% error
```

B.2.6 Step 5.5: Pull Filter Shapes

In [38]:
```
from lightlab.util.data import Spectrum

        # Pull Filter Shapes
        spect = spctAssist.fgSpect(avgCnt=5, bgType='tuned')
        axonCurves = list()
        dendriteCurves = list()
        for i, r in enumerate(spctAssist.resonances(spect)):
            relWindow = 8 * r.fwhm * np.array([-1,1])/2
            proximitySpect = spect.shift(-r.lam).crop(relWindow)
            nm, dbm = proximitySpect.shift(r.lam).getData()
            plt.plot(nm, dbm)
            lin = np.clip(dbm2lin(dbm), 0.0, 1.0)
            if pMap[i] in calAxonChan:
                axonCurves.append((nm,lin))
```

```
        else:
            dendriteCurves.append((nm, lin))

    assert len(axonCurves) == numRings
    assert len(dendriteCurves) == numRings
    plt.show()

    # Don't add them until AFTER bias calculation
```

B.2.7 Step 6: Determine Heat Bias

1. Put peaks on AD order (should already be done manually).
2. Move to within FWHM of targets
3. Focus on each target, track 2 peaks together.

```
In [39]: # Set Targets
         maxfwhm = np.max(np.array([r.fwhm
            for r in spctAssist.resonances()]))
         targets = list()
         for wl in wlChannels:
             targets.append(wl - maxfwhm)
             targets.append(wl + maxfwhm)

         targets = np.array(targets)

In [40]: ### Tracking to targets

         precision = 0.005 # Threshold
         propCoef = 0.5 # kP
         avgCnt = 4
         nowTune = baseTune.copy()
         appxThrmCoefs = tEstMap
         inst.setCurrentChanTuning(baseTune, token, CurrentUnit.mW)

In [41]: for i in range(100):
             spect = spctAssist.fgSpect(bgType='tuned', avgCnt = avgCnt)
```

```
            actualPeaks = np.array([r.lam
               for r in spctAssist.resonances(spect)])
            errs = targets - actualPeaks
            if max(abs(errs)) < precision:
                print('\nTracking complete')
                break
            # recenter wlRange to avoid other FSR resonances
            wlRangeTight = np.array([min(min(actualPeaks), min(targets)),
               max(max(actualPeaks), max(targets))])
            newwlRange = np.mean(wlRangeTight) +
               np.diff(wlRangeTight) * np.array([-1, 1]) / 2 * 2
            spctAssist.wlRange = newwlRange
            try:
                for p, ch in pMap.items():
                    nowTune[ch] = nowTune[ch] +
                       propCoef*errs[p] / appxThrmCoefs[ch]
                inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
            except io.RangeError as err:
                print('Out of range during tracking. See plot')
                spctAssist.fgResPlot()
                raise err

        spctAssist.fgResPlot()
        plt.show()
```

```
Tracking complete
```

```
In [42]: import copy
         def mergePeaks(wlTarget, newPMap, spctAssist,
            currentTune, fwhmThresh, tEstMap, skipStart):
             targets = np.array([wlTarget, wlTarget])
             newTune = currentTune.copy()
             newSp = copy.deepcopy(spctAssist)
```

```
newSp.wlRange = [wlTarget-2*fwhmThresh, wlTarget+2*fwhmThresh]
newSp.nChan = 2

### Tracking to single detected peak

precision = 0.005 # Threshold
propCoef = 0.1 # kP
avgCnt = 4
appxThrmCoefs = tEstMap
inst.setCurrentChanTuning(newTune, token, CurrentUnit.mW)

for i in range(100):
    if skipStart:
        break

    spect = newSp.fgSpect(bgType='tuned', avgCnt = avgCnt)
    try:
        actualPeaks = np.array([r.lam
           for r in newSp.resonances(spect)])
        actualfwhms = np.array([r.fwhm
           for r in newSp.resonances(spect)])
    except:
        print("Peaks have merged! (exception)")
        break

    # Check for merged peaks
    flag = False
    for f in actualfwhms:
            if f > 2*maxfwhm:
                print("Peaks have merged! (fwhm)")
                flag = True
    if flag:
        break
```

```
    # Run PID
    errs = targets - actualPeaks
    if max(abs(errs)) < precision:
        print('\nTracking complete')
        break
    try:
        for p, ch in newPMap.items():
            newTune[ch] = newTune[ch] + propCoef*errs[p] /
               appxThrmCoefs[ch]
        inst.setCurrentChanTuning(newTune, token,
           CurrentUnit.mW)
    except io.RangeError as err:
        print('Out of range during tracking. See plot')
        newSp.fgResPlot()
        raise err
#Peaks should have merged
newSp.nChan = 1

### Tracking to minimum FWHM (actually 1 peak)

leftCh = newPMap[0]
rightCh = newPMap[1]
errfwhm = 10000
propCoef = 0.11
prevTune = newTune.copy()
strike = False

for i in range(100):
    spect = newSp.fgSpect(bgType='tuned', avgCnt = avgCnt)

    aCenter = np.array([r.lam
       for r in newSp.resonances(spect)])[0]
    aFwhm = np.array([r.fwhm
       for r in newSp.resonances(spect)])[0]
```

```
# Run PID
prevErr = errfwhm
errfwhm = aFwhm - maxfwhm
errCenter = aCenter - targets[0]

if errfwhm > prevErr:
    newTune = prevTune
    inst.setCurrentChanTuning(newTune, token,
       CurrentUnit.mW)
    if strike:
        break
    else:
        # Maybe the two peaks have crossed
        strike = True
        tmp = leftCh
        leftCh = rightCh
        rightCh = tmp

if abs(errfwhm) < precision:
    print('\nTracking complete')
    break
try:
    prevTune = newTune.copy()
    newTune[leftCh] = newTune[leftCh] +
       propCoef*(errfwhm-errCenter) /
           appxThrmCoefs[leftCh]
    newTune[rightCh] = newTune[rightCh] -
       propCoef*(errfwhm-errCenter) /
           appxThrmCoefs[rightCh]
    inst.setCurrentChanTuning(newTune, token,
       CurrentUnit.mW)
except io.RangeError as err:
    print('Out of range during tracking. See plot')
```

```
                    newSp.fgResPlot()
                    raise err

            # Peaks should really have merged
            newSp.fgResPlot()
            return newTune
```

In [43]:
```
### MAY HAVE TO RUN A FEW TIMES TO GET GOOD RESULTS

inst.setCurrentChanTuning(nowTune, token, CurrentUnit.mW)
newTune = nowTune.copy()

# Initial Run-thru
for i, wl in enumerate(wlChannels):
    newPMap = dict()
    newPMap[0] = pMap[2*i]
    newPMap[1] = pMap[2*i+1]
    newTune = mergePeaks(wl, newPMap, spctAssist,
       newTune, maxfwhm, tEstMap, False)
    plt.show()
    sleep(1)

# Another run-thru to smooth out any errant cross-talk
for i, wl in enumerate(wlChannels):
    newPMap = dict()
    newPMap[0] = pMap[2*i]
    newPMap[1] = pMap[2*i+1]
    newTune = mergePeaks(wl, newPMap, spctAssist,
       newTune, maxfwhm, tEstMap, True)
    plt.show()
    sleep(1)
```

```
Peaks have merged! (fwhm)
```

```
In [44]: # There should now be two peaks
         spctAssist.nChan = len(wlChannels)
         spctAssist.fgResPlot()
         plt.show()

In [45]: calTherm.setHeatBias(newTune, CurrentUnit.mW)

In [47]: # Validation
         threshold = 0.01

         calBias = np.array(sorted([CurrentUnit.toVolt(b, CurrentUnit.mW)
            for b in calTherm.heatBias]))
         virtBias = np.array(sorted(list(heatBias.values())))
         diff = np.absolute(calBias - virtBias)
         print(diff)
         for d in diff:
             assert d < threshold
```

```
[ 0.0028115   0.00295138  0.00819335  0.00469781  0.0002451   0.0007584 ]
```

```
In [48]: calAxon.setBiasParams([r.lam for r in spctAssist.resonances()])
         calFb.setBiasParams([r.lam
            for r in spctAssist.resonances()])

In [49]: ## Validation
         threshold = 0.01

         calBias = [r.lam
            for r in spctAssist.resonances()]
         virtBias = np.array(wlChannels)
         diff = np.absolute(calBias - virtBias)
         print(diff)
         for d in diff:
             assert d < threshold
```

```
[ 0.00308762  0.00493711  0.00198601]
```

```
In [50]: # NOW Add filter shapes, after shifting to new bias
         for i in range(len(axonCurves)):
             oldCurve = axonCurves[i]
             shift = calBias[i] - np.mean(oldCurve[0])
             newCurve = (np.add(oldCurve[0], shift), oldCurve[1])
             axonCurves[i] = newCurve

         for i in range(len(dendriteCurves)):
             oldCurve = dendriteCurves[i]
             shift = calBias[i] - np.mean(oldCurve[0])
             newCurve = (np.add(oldCurve[0], shift), oldCurve[1])
             dendriteCurves[i] = newCurve

         calAxon.setCurve(axonCurves, MrrOut.kThru)
         calFb.setCurve(dendriteCurves, MrrOut.kThru)
```

```
In [51]: # Hot-Swap in Calibration Module and make sure things look good
         inst.lockPhony(calCsc)
         calCsc.setOsaOut(MrrOut.kThru.value)
         spctAssist.fgResPlot()
         plt.show()
         inst.releasePhony()
```

B.2.8 Step 7: Fill K Matrix

K is the coefficients between mW and deltaWL.

```
In [52]: biasTune = newTune.copy()
```

```
In [75]: def partialK(wlTarget, primNum, axonChan, dendChan,
    tEstMap, biasTune, spctAssist, avgCnt, nPts):
             '''
             For each wlTarget, retuns 2 rows of K.
```

```
rows: axon, dendrite
cols: a-d alternating in order provided by
axonChan and dendChan
'''
newSp = copy.deepcopy(spctAssist)
newSp.wlRange = [wlTarget-1, wlTarget+1]
newSp.nChan = 2
nowTune = biasTune.copy()
biasWL = np.multiply(wlTarget, np.ones(2))
KList = [None] * len(axonChan)
assert len(axonChan) == len(dendChan)

# Step 1: Do primary square
square = np.zeros((2, 2))
for ich, ch in enumerate([axonChan[primNum],
   dendChan[primNum]]):
    # Shift in a 0.75nm range around bias WL
    dB = 0.75 / tEstMap[ch]
    x = np.linspace(max(0.0,
       (biasTune[ch]-dB)), biasTune[ch]+dB, nPts)
    y = np.zeros((2, nPts))
    for ipt, pt in enumerate(x):
        if pt == biasTune[ch]:
            continue
        nowTune[ch] = pt
        inst.setCurrentChanTuning(nowTune, token,
           CurrentUnit.mW)
        nowWL = np.array([r.lam
           for r in newSp.resonances()])
        diff = np.sort(nowWL - biasWL)
        # Catch Negative Case
        if np.absolute(diff[1]) < np.absolute(diff[0]):
            tmp = diff[1]
            diff[1] = diff[0]
```

```
            diff[0] = tmp
        y[ich, ipt] = diff[1]
        y[1-ich, ipt] = diff[0]

    # Make Linear Fit

    for r in range(2):
        yr = y[r, :]
        p = np.polyfit(x, yr, 1)
        square[r, ich] = max(p[0], 0.0)

    if ich == 1:
        # Change bias to separate two peaks
        biasTune = nowTune.copy()
        biasWL = nowWL

    nowTune[ch] = biasTune[ch]


KList[primNum] = square

# Step 2: Do all Other Squares
nPts = nPts + 6

for num in range(len(KList)):
    if num == primNum:
        continue
    square = np.zeros((2, 2))
    for ich, ch in enumerate([axonChan[num], dendChan[num]]):
        # Shift in a 1.0nm range around bias WL
        dB = 1.25 / tEstMap[ch]
        x = np.linspace(max(0.0,
           (biasTune[ch]-dB)), biasTune[ch]+dB, nPts)
        y = np.zeros((2, nPts))
```

```
                    for ipt, pt in enumerate(x):
                        if pt == biasTune[ch]:
                            continue
                        nowTune[ch] = pt
                        inst.setCurrentChanTuning(nowTune, token,
                           CurrentUnit.mW)
                        nowWL = np.array([r.lam
                           for r in newSp.resonances()])
                        diff = nowWL - biasWL
                        y[:, ipt] = diff

                    # Make Linear Fit

                    for r in range(2):
                        yr = y[r, :]
                        p = np.polyfit(x, yr, 1)
                        square[r, ich] = max(p[0], 0.0)

                    if ich == 1:
                        # Change bias to separate two peaks
                        biasTune = nowTune.copy()

                    nowTune[ch] = biasTune[ch]

                KList[num] = square

            return np.hstack(KList)
```

In [76]:
```
rows = list()
for j, wl in enumerate(wlChannels):
    retRow = partialK(wl, j, calAxonChan, calDendriteChan,
       tEstMap, biasTune, spctAssist, avgCnt=5, nPts=7)
    rows.append(retRow)
newK = np.vstack(rows)
```

```
        def swapK(i, j):
            newK[:,[i, j]] = newK[:,[j, i]]
            newK[[i, j], :] = newK[[j, i], :]

        # At ADADAD
        # Want AAADDD
        # Swap rows/cols 1 and 2 (AADDAD)
        swapK(1, 2)
        # Swap rows/cols 3 and 4 (AADADD)
        swapK(3, 4)
        # Swap rows/cols 2 and 3 (AAADDD)
        swapK(2, 3)

        # Swap Middle Rows and Columns to order axons then dendrites

        # Add to thermalgroup
        calTherm.setK(newK)
```

In [77]:
```
        print("K:")
        print(np.round(K))
        print()

        print("New K:")
        print(np.round(newK))
        print()


        print("Round Error:")
        print(np.round(np.absolute(newK - K)))
        print()

        print("Absolute Error:")
        print(np.absolute(newK-K))
```

```
K:
[[ 19.   1.   1.   1.   0.   0.]
```

```
 [  1.  21.   2.   1.   0.   0.]
 [  0.   2.  20.   0.   1.   0.]
 [  0.   1.   1.  20.   1.   0.]
 [  1.   1.   1.   0.  20.   1.]
 [  0.   1.   0.   0.   0.  20.]]

New K:
[[ 19.   1.   1.   1.   0.   0.]
 [  1.  21.   2.   0.   0.   0.]
 [  0.   2.  20.   0.   0.   0.]
 [  0.   1.   1.  20.   1.   0.]
 [  1.   1.   0.   0.  20.   1.]
 [  0.   1.   0.   0.   0.  20.]]

Round Error:
[[ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]

Absolute Error:
[[ 0.02004703  0.00661128  0.00038421  0.0148798   0.00423898  0.02573439]
 [ 0.00783713  0.02285229  0.11606766  0.60466732  0.05026481  0.44121404]
 [ 0.00027619  0.00252188  0.04469797  0.01197408  1.06470847  0.02240962]
 [ 0.00851921  0.18221379  0.01600566  0.03894927  0.00990297  0.01298121]
 [ 0.0043928   0.02412916  1.11338314  0.05033487  0.04016287  0.47670709]
 [ 0.00217652  0.00233603  0.02640214  0.02192149  0.20880726  0.00609685]]
```

And calCsc is a calibrated device!

Validation was done at the following points: * Ascription is guaranteed to be correct. * HeatBias is within 0.01mW * Wavelength bias is within 0.01nm * Check K cross-terms to make sure error is 1 or 0, ESPECIALLY along diagonals

Note that these validation constraints **degrade** as the cross terms between axons and dendrites increase. Fortunately, those cross-terms will not be significant on the actual chip.

In [ ]:

B.3 2-3-1 Network Backpropagation

B.3.1 Step 1: Initialization

- Import anything necessary.
- Pull Neural Network measured thermal parameters (thK and Ib).
- Define activation function and load it into tensorflow.

In [43]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from lightlab.util.modeling import lorentz
from tensorflow.python.framework import ops
```

In [34]:
```
# Step 1: Required Parameters

# K Diagonals
thK = 20.0
# 0 Current Bias (mA)
Ib = 6.0
```

In [42]:
```
# Step 2: Transfer Function, approximate as lorentzian
nm, lin = lorentz([-0.5, 0.5], 0, 0.1, 0.98)
lin = 1.0-lin

def h(x):
    return np.interp(x, nm, lin)

def dh(x):
    return np.interp(x, nm, np.gradient(lin))

def g(x):
    return np.divide(thK*np.add(np.power(x, 2.0), 2*Ib*x), 1000)

def dg(x):
    return np.divide(np.add(2*thK*x, 2*thK*Ib), 1000)
```

```
        def f(x):
            return h(g(x))

        def df(x):
            return np.multiply(dh(g(x)), dg(x))

        x = np.linspace(-1, 1, 200)
        plt.plot(x, f(x), 'b')
        plt.xlim([-1, 1])
        plt.show()
        plt.plot(x, df(x), 'b')
        plt.show()
```

In [50]:
```
        # Step 3: Add to TensorFlow

        # Py_Func Hack
        (http://stackoverflow.com/questions/39921607
        def py_func(func, inp, Tout, stateful=True, name=None, grad=None):

            # Need to generate a unique name to avoid duplicates:
            rnd_name = 'PyFuncGrad' + str(np.random.randint(0, 1E+8))

            tf.RegisterGradient(rnd_name)(grad)
            g = tf.get_default_graph()
            with g.gradient_override_map({"PyFunc": rnd_name}):
                return tf.py_func(func,
                    inp, Tout, stateful=stateful, name=name)

        np_df = np.vectorize(df)

        np_df_32 = lambda x: np_df(x).astype(np.float32)

        def tf_df(x,name=None):
            with ops.op_scope([x], name, "df") as name:
                y = tf.py_func(np_df_32,
```

```
                                [x],
                                [tf.float32],
                                name=name,
                                stateful=False)
                return y[0]

        def fgrad(op, grad):
            x = op.inputs[0]

            n_gr = tf_df(x)
            return grad * n_gr

        np_f = np.vectorize(f)

        np_f_32 = lambda x: np_f(x).astype(np.float32)

        def tf_f(x, name=None):

            with ops.op_scope([x], name, "f") as name:
                y = py_func(np_f_32,
                                [x],
                                [tf.float32],
                                name=name,
                                grad=fgrad)
                return y[0]
```

```
In [51]: with tf.Session() as sess:

            x = tf.constant([-0.4,-0.2,0.0,0.2,0.4])
            y = tf_f(x)
            tf.initialize_all_variables().run()

            print(x.eval(), y.eval(), tf.gradients(y, [x])[0].eval())
```

B.3.2 Step 2: Set Up Neural Network

- Create random XOR Data
- Create tensorflow model

In [273]:
```
# Generate XOR Data
tData = np.random.uniform(-1.0, 1.0, (400, 2))
tData[0:100, :] = tData[0:100, :]*0.2 + 0.2
tData[100:200, :] = tData[100:200, :]*0.2 + 0.6
tData[200:300, 0] = tData[200:300, 0]*0.2 + 0.2
tData[200:300, 1] = tData[200:300, 1]*0.2 + 0.6
tData[300:400, 1] = tData[300:400, 1]*0.2 + 0.2
tData[300:400, 0] = tData[300:400, 0]*0.2 + 0.6
tData = np.clip(tData, 0.0, 0.8)
tLabels = np.zeros((400, 2))
tLabels[0:200, 0] = np.ones(200)
tLabels[200:400, 1] = np.ones(200)
plt.plot(tData[0:200, 0], tData[0:200, 1], 'r*')
plt.plot(tData[200:400, 0], tData[200:400, 1], 'bs')
plt.show()
```

In [274]:
```
# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50
```

In [275]:
```
# Network Parameters
n_hidden = 3 # 1st layer number of features
n_input = 2 # XOR data input
n_classes = 2 # total classes (0 or 1)

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])
```

In [276]:
```
# Create model
def multilayer_perceptron(x, weights, biases):
```

```
    # Hidden layer with Photonic Axon
    hidden = tf.add(tf.matmul(x, weights['w0']), biases['b0'])
    hidden = tf_f(hidden)
    # Output layer with linear activation
    out_layer = tf.matmul(hidden, weights['w1']) + biases['b1']
    return out_layer

# Store layers weight & bias
weights = {
    'w0': tf.Variable(tf.random_normal([n_input, n_hidden])),
    'w1': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'b0': tf.Variable(tf.random_normal([n_hidden])),
    'b1': tf.Variable(tf.random_normal([n_classes]))
}
```

In [277]:
```
# Construct model
pred = multilayer_perceptron(x, weights, biases)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
  logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(
  learning_rate=learning_rate).minimize(cost)

# Initializing the variables
init = tf.global_variables_initializer()
```

B.3.3 Step 3: Train

- Train the network.
- Check accuracy and see the 2D sweep.
- Pull weights to pass back into the network simulation.

In [303]:
```
# Launch the graph
```

```
            #sess = tf.Session()
            #sess.run(init)

            # Training cycle
            try:
                for epoch in range(training_epochs):
                    # Run backprop and cost func
                    _, c = sess.run([optimizer, cost], feed_dict={x: tData,
                           y: tLabels})
                    # Display logs per epoch step
                    if epoch % display_step == 0:
                        print("Epoch:", '%04d' % (epoch+1), "cost=", \
                            "{:.9f}".format(c))
                print("Optimization Finished!")
            except:
                print("Training Interrupted")
```

```
Epoch: 0001 cost= 0.059564065
Epoch: 0051 cost= 0.058935978
Training Interrupted
```

```
In [304]: with sess.as_default():
              # Test model
              correct_prediction =
                tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
              # Calculate accuracy
              accuracy = tf.reduce_mean(
                tf.cast(correct_prediction, "float"))
              print("Accuracy:", accuracy.eval({x: tData, y: tLabels}))
```

```
Accuracy: 0.9875
```

```
In [296]: cList = np.linspace(0.0, 0.8, 100)
          yMat = np.zeros((100, 100, 2))
```

```
            with sess.as_default():
                for i, c1 in enumerate(cList):
                    for j, c2 in enumerate(cList):
                        cIn = np.zeros((1, 2))
                        cIn[0, 0] = c1
                        cIn[0, 1] = c2
                        yMat[i, j, :] = pred.eval({x: cIn})
```

In [297]:
```
            # Plot Normalized Output
            im = plt.imshow(yMat[:, :, 0],
              interpolation='bilinear', cmap=cm.RdYlGn,
                                origin='lower', extent=[0, 0.8, 0, 0.8],
                                vmax=abs(yMat).max(), vmin=-abs(yMat).max())
            plt.colorbar()
            plt.show()
            # Classification
            cls = np.sign(yMat[:, :, 0])
            im = plt.imshow(cls, interpolation='bilinear', cmap=cm.RdYlGn,
                                origin='lower', extent=[0, 0.8, 0, 0.8],
                                vmax=1, vmin=-1)


            plt.show()
```

In [301]:
```
            # Pull Weights
            with sess.as_default():
                allweights = sess.run(weights)
                w0 = allweights['w0'].T
                w1 = allweights['w1'].T[0, :]
                print(w0)
                print(w1)
```

```
[[ 1.30109513  0.90975827]
 [-0.79916418 -0.85981083]
 [ 0.9742766  -1.02000749]]
```

```
[ -4.16287088  11.24845219  -9.0137167 ]
```

```
In [302]: # Pull Biases
          with sess.as_default():
              allbiases = sess.run(biases)
              b0 = allbiases['b0']
              b1 = allbiases['b1'].T[0]
              print(b0)
              print(b1)
```

```
[ 1.0609535   0.65707952  0.01618425]
2.17837
```

```
In [ ]:
```

B.4 2-3-1 Network Simulation

B.4.1 Step 1: Initialize 2-3-1 Virtual Network (Calibrated)

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import lightlab.instruments as inst
import lightlab.model as m
from lightlab.util.modeling import kOSAPwr, dbm2lin, CurrentUnit
from lightlab.util.calibrating import SpectrumMeasurementAssistant
from scipy import interpolate
import matplotlib.cm as cm
```

In [2]:
```
# Thermal Group, Uses 14 current channels
numCurrents = 14
currentChan = range(numCurrents)
therm = m.ThermalGroup(currentChan)

# Axons first, two branches, 2 in 1st branch, 3 in second branch
axonChannels = [[0, 1], [2, 3, 4]]

# Master Variables
numBranches = 2
numRings = [2, 3]
numBanks = [3, 1]
wlChannels = np.array([1550, 1552, 1554])
lorentzAtten = 0.99
lorentzfwhm = 0.1
fullAtten = 0.01

# Make Axons
axons = list()
for i in range(numBranches):
    axon = m.FilterBank(therm, numRings[i])
    axon.setAxon()
    axon.setBiasParams([wlChannels[j] for j in range(numRings[i])],
```

```
            lorentzAtten*np.ones(numRings[i]),
            lorentzfwhm * np.ones(numRings[i]))
    axons.append(axon)

# Make Dendrites
fbs = [None] * numBranches
for i in range(numBranches):
    fbs[i] = list()
    for j in range(numBanks[i]):
        fb = m.FilterBank(therm, numRings[i])
        fb.setBiasParams([wlChannels[k] for k in range(
            numRings[i])], lorentzAtten*np.ones(numRings[i]),
            lorentzfwhm * np.ones(numRings[i]))
        fbs[i].append(fb)

# Connect Components
cscs = list()
for i in range(numBranches):
    cscs.append(m.Cascade(axons[i], m.Splitter(fbs[i])))
network = m.Splitter(cscs)
network.setAttenuation(fullAtten)


# Set K and bias of hidden phony module
K = 20.0*np.eye(numCurrents)
therm.setK(K)

heatBias = dict()
for c in range(numCurrents):
    heatBias[c] = c*0.2 + 1  # In Volts

#heatBias[2] = heatBias[3] = 1.5
#heatBias[4] = heatBias[3]
```

```
therm.setHeatBias(heatBias, CurrentUnit.V)

# Register Network
inst.togglePhony(True, network)
token = inst.reserveCurrentChan(currentChan)

wlRange = [1547, 1555]
```

B.4.2 Step 2: Network Simulation Function

```
In [3]: # Global Parameters
        Resp = 0.9
        Rt = 15000 # (15MOhm)
        Rs = 1.5 #(1kOhm)
        bv = 4 # 4V

In [4]: def getWeightsBias(weight23, weight31):
            # Generate transmission matrices
            # Note: Thru port is positive
            # t_thru = (w+1)/2

            # Start with all axons 0 deltaLambda
            dummyAxonBias = [np.array([0, 0]), np.array([0,0,0])]

            # First Branch
            tM1 = np.ones((6, 3)) # Third Channel keep 1, acts as bias
            for bank in range(3):
                for channel in range(2):
                    tM1[2*bank, channel] =
                        np.divide(weight23[bank, channel] + 1, 2)
                tM1[2*bank+1, :] = np.subtract(1.0, tM1[2*bank, :])

            # Second Branch
            tM2 = np.ones((2, 3))
            tM2[0, :] = np.divide(np.add(weight31, 1.0), 2.0)
```

```
        tM2[1, :] = np.subtract(1.0, tM2[0, :])

        tM = list()
        tM.append(np.divide(tM1, 3.0)) # Don't forget splitter.
        tM.append(tM2)

        return network.control(dummyAxonBias, tM, wlChannels)
```

```
In [5]: from time import sleep
        def netSim(axon2, axon3, weightBias, bOut, debug = False):
            '''For a given set of weights and axon biases,
            simulate the neural network.
            '''

            biasCurrent = weightBias.copy()

            # Add In Axon Biases
            for i, ch in enumerate(axonChannels[0]):
                biasCurrent[ch] += axon2[i]

            # Convert to mA
            nowCurrent = dict()
            for ch, v in biasCurrent.items():
                nowCurrent[ch] = CurrentUnit.voltTo(biasCurrent[ch],
                    CurrentUnit.mA)

            # Write Current to instruments
            inst.setCurrentChanTuning(biasCurrent, token)

            # See what our normalized input is:
            x0 = np.zeros(2)

            inst.lockPhony(axons[0])
            axons[0].setAttenuation(fullAtten)
            nm, dbm = inst.spectrum(wlRange)
```

```
inst.releasePhony()

lin = dbm2lin(dbm) / (kOSAPwr * fullAtten)
x0 = np.clip(np.interp(wlChannels[:2], nm, lin), 0.0, 1.0)
# Store for Return


# Get three Currents in mA
c0 = np.zeros(3)
for i in range(3):
    network.setOsaOut(2*i)
    nm, dbm = inst.spectrum(wlRange, avgCnt=10)
    thru = np.sum(dbm2lin(np.interp(wlChannels, nm, dbm)))


    network.setOsaOut(2*i + 1)
    nm, dbm = inst.spectrum(wlRange, avgCnt=10)
    drop = np.sum(dbm2lin(np.interp(wlChannels, nm, dbm)))

    c0[i] = Resp*(thru - drop)

# Pre-Amplification
#if debug:
#    return x0, c0

# Send through amplifiers
c0 = (Rt*c0 + bv)/Rs + axon3

if debug:
    # Give Effective Weight Matrix
    return x0, c0


# Send to axons
```

```
#print(c0)
for i, ch in enumerate(axonChannels[1]):
    nowCurrent[ch] += c0[i]

inst.setCurrentChanTuning(nowCurrent, token, CurrentUnit.mA)

# See what our normalized hidden layer is:
x1 = np.zeros(3)

inst.lockPhony(axons[1])
axons[1].setAttenuation(fullAtten)
nm, dbm = inst.spectrum(wlRange)
inst.releasePhony()

lin = dbm2lin(dbm) / (kOSAPwr * fullAtten)
x1 = np.clip(np.interp(wlChannels, nm, lin), 0.0, 1.0)
# Store for Return

#return x0, x1

#'''
# Get current in mA
network.setOsaOut(6)
nm, dbm = inst.spectrum(wlRange)
thru = np.sum(dbm2lin(np.interp(wlChannels, nm, dbm)))

network.setOsaOut(7)
nm, dbm = inst.spectrum(wlRange)
drop = np.sum(dbm2lin(np.interp(wlChannels, nm, dbm)))

c1 = Resp*(thru - drop)

y = Rt2*c1 + bOut
```

```
        return x0, y
```

In [6]:
```
### Function to Plot Sweep of Network

def sweepNetwork(weight23, weight31, axonBias, outBias, sweepNum):

    # Calculate currents that give said weights
    weightBias = getWeightsBias(weight23, weight31)

    # Run Sweep
    cList = np.linspace(0, 0.15, sweepNum)
    z = np.zeros((sweepNum, sweepNum))
    x = np.zeros(sweepNum)
    y = np.zeros(sweepNum)

    for i, c1 in enumerate(cList):
        print("Step: " + str(i))
        for j, c2 in enumerate(cList):
            x0, out = netSim([c1, c2],
                axonBias, weightBias, outBias)
            x[i] += x0[0]
            y[j] += x0[1]
            z[i, j] = out

    # Average value of x0[0], x0[1] for values of cList
    x = np.divide(x, sweepNum)
    y = np.divide(y, sweepNum)

    f = interpolate.interp2d(x,y,z,kind='cubic')

    delta = 0.025
    maxVal = 0.8
    X = Y = np.arange(0.0, maxVal, delta)
    Z = f(X, Y)
    return maxVal, Z
```

Test on some sample weights:

```
In [7]: # Global Parameters
        Resp = 0.9
        Rt = 7500 # (7.5MOhm)
        Rt2 = 7500 # (7.5MOhm)
        Rs = 2 #(2kOhm)
        bv = 5 # 5V

        # Should be equivalent of 2-Neuron Perceptron
        weight23 = np.array([[-0.8, 0.8], [0.8, -0.8], [0.0, 0.0]])
        weight31 = np.array([[0.8, -0.8, 0.0]])
        axonBias = -2.8*np.array([1.0, 1.0, 1.0])
        outBias = -1

        maxVal, surface = sweepNetwork(weight23,
            weight31, axonBias, outBias, 10)

In [8]: im = plt.imshow(surface, interpolation='bilinear', cmap=cm.RdYlGn,
            origin='lower', extent=[0, maxVal, 0, maxVal],
            vmax=abs(surface).max(), vmin=-abs(surface).max())
        plt.colorbar()
        plt.show()
        # Classification
        cls = np.sign(surface)
        im = plt.imshow(cls, interpolation='bilinear', cmap=cm.RdYlGn,
            origin='lower', extent=[0, maxVal, 0, maxVal],
            vmax=1, vmin=-1)
        plt.show()
```

B.4.3 Step 3: Run Backprop for Virtual Weights and Biases

(See 2-3-1_FFNet_Backprop)

```
In [11]: # Pull Network Params
         print(np.diag(K)[2:5])
```

```
        print([CurrentUnit.voltTo(heatBias[c], CurrentUnit.mA)
           for c in axonChannels[1]])
```

```
[ 20.  20.  20.]
[5.6, 6.4, 7.2]
```

```
In [60]: ### Output of 231 Backprop:

        ### Best Output So Far:
        virt23 = np.array([[1.30109513, 0.90975827],
           [-0.79916418, -0.85981083], [0.9742766,  -1.02000749]])
        virt31 = np.array([ -0.416287088, 1.124845219, -0.90137167])
        virtBias = np.array([ 1.0609535, 0.65707952,  -0.01618425])
        outBias = 0.4359157047300002
```

B.4.4 Step 4: Convert to "Real" Weights

Basically multiply by physical parameters to get the actual 2t-1 we send to the network.

```
In [61]: p = kOSAPwr * fullAtten
        # Global Parameters
        Resp = 0.9
        Rt = 15000 # (15MOhm)
        Rt2 = 3000 # (3MOhm)
        Rs = 1 #(1kOhm)
        bv = 4 # 4V
```

```
In [62]: weight23 = np.multiply(virt23, Rs*6/(Rt*Resp*p))
        print(weight23)
```

```
[[ 0.5782645   0.40433701]
 [-0.35518408 -0.38213815]
 [ 0.43301182 -0.45333666]]
```

```
In [63]: weight31 = np.multiply(virt31, 2/(Rt2*Resp*p))
         print(weight31)
```

```
[-0.30836081  0.83321868 -0.66768272]
```

```
In [64]: axonBias = np.zeros((1, 3))
         axonBias = np.subtract(virtBias, (Rt*Resp*p/6) + (bv/Rs))
         print(axonBias)
```

```
[-5.1890465  -5.59292048 -6.26618425]
```

```
In [65]: # Check Virtual Weights
         weightBias = getWeightsBias(weight23, weight31)
```

```
In [66]: # Check Virtual Weights
         x0, c0 = netSim([0.1, 0.1], axonBias, weightBias, outBias, True)
         print("Actual: " + str(c0))
         cWant = np.dot(virt23, x0) + virtBias
         print("Target: " + str(cWant))
```

```
Actual: [ 2.61785122 -0.48164702 -0.18952191]
Target: [ 2.60217079 -0.52201333 -0.17577987]
```

```
In [69]: maxVal, surface =
    sweepNetwork(weight23, weight31, axonBias, outBias, 20)
```

```
In [71]: # Plot Normalized Output
         im = plt.imshow(surface, interpolation='bilinear', cmap=cm.RdYlGn,
            origin='lower', extent=[0, maxVal, 0, maxVal],
            vmax=1, vmin=-1)
         plt.colorbar()
         plt.show()
         # Classification
         cls = np.sign(surface)
```

```
    im = plt.imshow(cls, interpolation='bilinear', cmap=cm.RdYlGn,
            origin='lower', extent=[0, maxVal, 0, maxVal],
            vmax=1, vmin=-1)

    plt.show()
```

The thing about formatting code is that it always feels like perfectionless effort.