Waveform Design Implemented on Neuromorphic Hardware

Patrick J Farr
Applied Research Solutions
Beavercreek, Ohio 45440
patrick.farr.ctr@us.af.mil

Aaron M Jones
Sensors Directorate
Air Force Research Laboratory
Wright-Patterson AFB, Ohio 45433l

Trevor J Bihl
Sensors Directorate
Air Force Research Laboratory
Wright-Patterson AFB, Ohio 45433

Jayson Boubin
Department of Computer Science
The Ohio State University
Columbus, USA 43210

Ashley DeMange
Sensors Directorate
Air Force Research Laboratory
Wright-Patterson AFB, Ohio 45433

***Abstract*—Neuromorphic computing hardware mimics neurobiological architectures and promises eventual low power operation. Additionally, arbitrary waveform generator hardware permits the realization of complex radar waveform structures. In this paper, we combine these two technologies and investigate the potential of Spiking Neural Networks to generate waveforms and their suitability in dynamic environments where adaptability is paramount. We discuss the process of development, current limitations, and critical assumptions to realizing real-time waveform adaptability with this hardware. Finally, we provide simulation and a novel application of an SNN implemented on Intel's Loihi research processor to classify a matched filter dataset.**

I. Introduction

In environments that contain many radio frequency (RF) transceivers, the spectrum becomes crowded and different devices interfere with each other [1]. This has been observed in traditional RF communications like Radio and Television, and continues today in new technologies concerning Unmanned Aerial Vehicles (UAVs) and mobile devices.

This issue requires a way of designing transmit waveforms in order to avoid said interference. One familiar method is the Error Reduction Algorithm (ERA) [2], which is capable of computing notched waveforms for pseudo-real-time applications. Unfortunately, this algorithm is not as accurate as desired for shared spectrum access. Another method is Reiterative Uniform Weight Optimization (RUWO) [3], which is highly accurate in its waveform computations, but whose accuracy comes at a cost of much slower convergence. An objective of our research is then to achieve (or improve upon) the convergence time of the ERA while maintaining the design quality of algorithms such as RUWO.

A more recent approach is to use artificial neural networks (ANN) to learn attributes of the RF environment and then decide a course of action to optimize the usage of the spectrum [4]. While this method is effective, it requires significant pre/post processing of data, which can be time consuming. Some neural networks have been built to interpret data more directly [5], thus bypassing a significant portion of this data processing. However, these methods require larger networks that necessarily consume more time and power [6].

Spiking neural networks (SNNs) are a class of ANNs whose neurons imitate the temporally spiking nature of biological neurons. Because of this, they naturally lend themselves to time domain signal problems, as they themselves are built on propagating signals over time [7]. Furthermore, the sparse spiking nature of neurons allow for efficient use of energy, and with SNN computational hardware such as the Intel Loihi research processor [8], SNNs will be capable of performing operations at a fraction of the time and power consumption that they have been seen to use on traditional hardware [9].

In this paper, we develop a baseline SNN from an ANN architecture to run on Loihi research hardware, and compare its accuracy and speed to previously implemented methods for radar waveform design. The ANN we reference is based on that presented in [4], which is known to have high accuracy. The SNN will therefore approximate this known accuracy, and can then be used as a baseline for future SNN implementations, which may or may not be built from underlying ANNs. Finally, we show preliminary results for SNNs performing waveform design tasks.

A. Notation and Organization

Our notation for this paper is as follows: Column vectors are denoted using lowercase underlined letters. An entry of a column vector is indicated with square brackets. For example, the k_{th} entry of column vector, $\underline{\mathrm{v}}$ is denoted $\underline{\mathrm{v}}[k]$. Matrices are represented with bold, capital letters. $\mathcal{F}\{\cdot\}$ represents the discrete Fourier transform. Finally, $|\cdot|$, $\lceil\cdot\rceil$, and $\lfloor\cdot\rfloor$ represent magnitude, ceiling, and floor functions respectively.

Section 2 will cover preliminary information about neural networks and signal interference. Section 3 discusses the model of our Neural Network and relevant parameters/background. Section 4 will discuss the results of our SNN implemented on the Loihi processor. Finally, section 5 will cover the overall results and future steps of this line of research.

II. Preliminaries

Let us consider the scenario of a radar system attempting to transmit in a band partially occupied by sources of interference. Our objective is to design the transmit waveform such that the considered band is multi-purpose and we avoid other users. This is a well studied waveform design problem with solutions from traditional signal processing approaches [2], [3].

A. Comparison of Waveform Design Algorithms

For comparison purposes, we reference two methods of waveform design: first being the ERA, known for fast convergence [2], and the second being RUWO, known for yielding high quality waveforms [3]. RUWO gives a more accurate result than ERA, but is also substantially slower to compute. The goal is then to build some system that reaches RUWO accuracy, while computing in a fraction of the run-time of the ERA algorithm.

Another approach that is theoretically both fast and accurate is discussed in [10], which uses Precompute and Lookup (PAL), a hashing function that quickly looks up the best approximate waveform from a library. While this method is both accurate and efficient in time, the storage of precomputed waveforms is unfortunately exponentially complex in space, and so the required library cannot be stored for practical scenarios. This motivates the use of neural network strategies to encode the information about PAL and the given library in a smaller data space, which would achieve a similar speed while using substantially less data.

B. Interference Model and Dataset Generation

For our example problem, let us suppose we have a single Linear Frequency Modulated (LFM) interference signal, s_i, with noise signal substantially lower than the interference. Then we can set a threshold, γ, to create a binary frequency mask

$$\underline{\mathrm{b}} = \begin{cases} 1, & \text{if } |\mathcal{S}_i| \geq \gamma \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where $\mathcal{S}_i = \mathcal{F}\{s_i\}$. $\underline{\mathrm{b}}$ then indicates whether a given frequency bin is occupied by interference.

The information contained in $\underline{\mathrm{b}}$ that is relevant to us is the information about which portions of the spectrum are occupied by interference. Therefore, let us consider a vector, $\underline{\mathrm{p}} \in \mathbb{N}^{l\times 1}$ that contains all indices for which the binary mask outputs 1:

$$\underline{\mathrm{p}} = \{k \in \mathbb{N} | \underline{\mathrm{b}}[k] = 1\}. \tag{2}$$

This vector is more efficient for input to a neural network than the whole binary mask, but it can still be improved for our simple case. To prune the data further, Boubin et al. [4] proposes a data set with an input $\underline{\mathrm{x}}$, as a compressed version of $\underline{\mathrm{p}}$. Simply put, $\underline{\mathrm{x}}$ is a vector of m sample points that indicate which frequency bins are occupied. Each point $\underline{\mathrm{x}}[k] \in \underline{\mathrm{x}}$ is a natural number that is sampled evenly from $\underline{\mathrm{p}}$:

$$\underline{\mathrm{x}} = \left\{ \underline{\mathrm{x}}[k] = \underline{\mathrm{p}} \left[\left\lfloor \frac{kl}{m} \right\rfloor \right] \right\}. \tag{3}$$

This feature extraction constitutes an attempt to use only relevant features for distinguishing between different interference environments, while decreasing the size and complexity of the neural network's input. Relevant features in this context are points in the interference mask that exceed a given threshold; therefore, we only provide the network points from these features.

The dataset's target output space corresponds to the indexing of a predefined RUWO output, allowing us to simplify what the output for the mapping should be. In summary, the dataset consists of a given input vector corresponding to an efficient representation of the interference, and a given output corresponding to an index of a RUWO output library.

C. Feedforward Artificial Neural Networks

The dataset presented above was successfully learned by a shallow feedforward ANN in [4]. This approach allowed for RUWO accuracy in a small set of interference examples, while improving computation time by an order of magnitude over ERA. Supervised ANNs in general are known to be highly successful in learning complex non-linear input/output mappings. The input/output mapping is primarily learned through the selection of weights and biases, which attempt to extract relevant features from the input information in order to map them to the output. There are several supervised learning algorithms that can be used to find satisfactory weights and biases, and additional information can be found in [11].

D. Spiking Neural Networks

While supervised ANNs have been popular in solving a variety of problems in recent years, they become computationally intensive as they scale up, leading to a toll in SWaP performance [6]. SNNs on specialized hardware could improve this, as their neuronal activity is inherently sparse, potentially leading to savings on power consumption of several orders of magnitude [12].

An SNN is a type of neural network in which neurons exhibit spiking behavior when presented an input, unlike ANNs whose perceptrons hold constant values. Because a spiking neuron exhibits behavior over time, different models exist to describe this behavior. One popular neuron model for SNNs is the Leaky Integrate and Fire (LIF) model, which is the simplest model to design in an electronic circuit while still retaining satisfactory practicality (depending on the application). The LIF neuron's primary behavior can be described by a circuit consisting of a capacitor with capacitance c and resistor with resistance r in parallel. With input current, i, the output voltage, v_{cap}, as the voltage across the capacitor, the behavior at time t is:

$$v_{cap}(t) = ri(t) - rc\frac{dv_{cap}}{dt}. \tag{4}$$

As input is applied, the LIF neuron charges. However, the LIF also leaks this charge at an interval determined by rc. Not shown in equation 4 is an additional element in parallel with the resistor and capacitor, that measures v_{cap}. If the voltage across the capacitor reaches some set threshold voltage, then this component discharges the capacitor out as a spike. If constant input is applied, then the LIF will spike at some constant frequency, which will be a function of the input.

E. ANN to SNN

We can use this to build an SNN from a simple feedforward ANN. While there are several methods of coding ANN attributes to SNN properties, the simplest to implement is the rate coding scheme [13]. Rate coding uses the spiking frequency as the information stored and gathered from neurons in the SNN. Thus, if we build the SNN such that a given output of the ANN corresponds to the frequency on a given output of the SNN, then the SNN will replicate the activity of the ANN.

F. Hardware Implementation

While we can operate SNNs on CPUs, it is more efficient to realize their potential by operating them on hardware and software optimized for spiking neural computations [9]. We implement this SNN through the Python package, Nengo, which was developed specifically for SNN design [14]. Nengo is compatible with Intel's neuromorphic research chip, Loihi, and is capable of both programming a Loihi-embedded research device, as well as emulating the Loihi hardware. This is done through the Nengo-Loihi library and follows closely to the class/function syntax of the core Nengo toolset. The Loihi processor contains a set of neuromorphic cores, each of which consists of 1024 neurons. Early implementations of the Loihi have found improved speed and power performance for several tasks previously implemented on traditional hardware [12].

III. Neural Network Model

In this section we cover the specific design of our neural network. The goal is to build a proof of concept SNN that is based on an ANN with known accuracy [4]. This can give us a baseline SNN representation that we can only improve upon in future applications of interference mitigation.

A. ANN Architecture

The initial ANN was developed consistent with [4]. The inputs and output were normalized to the range of -1 to 1, where the maximum value for a given dimension of training data is mapped to 1, the minimum value of each dimension is mapped to -1, and all other values are mapped linearly to some value between -1 and 1. This scaling method can help the convergence of training and reduce over-fitting.

The network consisted of a single hidden layer with $\tanh$ activation function, and the output being connected directly from the hidden layer via a weight matrix. The network was trained using the damped least squares (DLS) algorithm, as it was found to have better performance than standard linear least squares. Table I summarizes our hyper-parameters.

The dataset was split into 80% training data, 10% test data, and 10% validation data. Multiple randomly initialized networks were trained and tested until a network reached satisfactory generalization through validation.

TABLE I: ANN parameters

Parameter	Selection
# Inputs	15
# Hidden Neurons	50
Hidden Activation	Tanh
# Outputs	1
Output Activation	Linear
Training Algorithm	DLS

B. SNN Simulation

The ANN's weights and biases were then transferred to an SNN of equivalent structure. The SNN's behavior is simulated as a function of time. Since the output of the SNN is inherently determined through behavior over time, the SNN will not output its prediction immediately, but at some time $t > 0$ where $t = 0$ is the point at which the SNN is presented an input. Therefore, the SNN's behavior necessarily converges to its prediction, as can be seen in figure 1.

Through trial and error, we determined that the SNN simulation converged after a maximum of 90 simulated ms of being presented an input. Thus, the simulation time chosen for a given SNN input is 100 ms: 90 ms for convergence, and 10 ms for measuring and averaging the output. Note that this time is not the actual time that the hardware takes to compute the output. While there is a relation between the simulation time and the real computation time, the real-time computation speed also depends on the time-step period used in simulating the SNN, as well as the time that the hardware takes to compute the simulation for a single time-step.

The flow of the SNN is as follows: the inputs are transformed via the first weight matrix from the ANN, and the result is then sent as the input to the neuron layer. The general equation describing these neurons are shown in equation 4. However, this function is in continuous time, while both the Nengo software and the Loihi hardware simulate this behavior with discrete time-steps. Thus, with some sampling frequency f_s:

$$\underline{\mathrm{v}}_{cap}[n] = \underline{\mathrm{v}}_{cap}[n-1] + \left(\frac{\underline{\mathrm{i}}[n]}{c} - \frac{\underline{\mathrm{v}}_{cap}[n]}{rc}\right)\Delta t \tag{5}$$

where n is the current time-step, $\underline{\mathrm{i}}[n]$ is the simulated current at time-step n, and $\Delta t = \frac{1}{f_s}$ is the period of a single time-step.

When the voltage across the capacitor reaches some threshold, v_{thresh} The capacitor discharges:

$$\underline{\mathrm{v}}_{pos}[n] = \begin{cases} v_{thresh}, & \text{if } \underline{\mathrm{v}}_{cap}[n] \geq v_{thresh} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

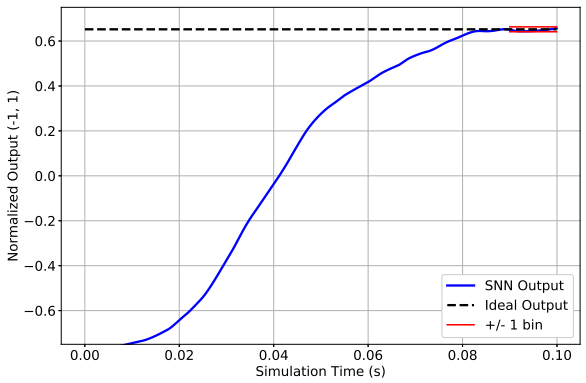

Fig. 1: Example of convergence of the SNN output to the target output. Input (not shown) is presented to the SNN at time $t = 0$. the black line indicates the target that the SNN should reach. After 90 simulated ms, the SNN's output converges around the target output.

where $\underline{\mathrm{v}}_{pos}$ is the voltage of the spike going out from the neuron. Finally, v_{cap} at time-step n is updated, as the capacitor has now lost some of its charge:

$$\underline{\mathrm{v}}_{cap}[n+1] = \begin{cases} \underline{\mathrm{v}}_{cap}[n] - v_{thresh}, & \text{if } \underline{\mathrm{v}}_{cap}[n] \geq v_{thresh} \\ \underline{\mathrm{v}}_{cap}[n], & \text{otherwise} \end{cases}. \tag{7}$$

As the selected resistance r approaches ∞, the leakage rate becomes slower and slower. If we remove the resistor from our circuit altogether such that the capacitor is the only connection, then our LIF model becomes equivalent to a spiking Rectified Linear Unit (ReLU) model. That is, the spiking frequency of this neuron is linear with respect to positive input and zero for negative input. The function for voltage across the capacitor becomes

$$\underline{\mathrm{v}}_{cap}[n] = \underline{\mathrm{v}}_{cap}[n-1] + \underline{\mathrm{i}}[n]. \tag{8}$$

In an ANN, a single neuron can output both positive and negative values. SNNs, on the other hand, cannot directly output a negative value, since the neurons cannot fire at a negative frequency. Therefore, a copy of the first layer is used to represent the ANN's negative neuron values. The input current to the negative layer is then:

$$\underline{\mathrm{i}}_{neg}[n] = -\underline{\mathrm{i}}[n]. \tag{9}$$

Likewise, the final output voltage is updated to be the difference between the first layer and the negative layer:

$$\underline{\mathrm{v}}_{out}[n] = \underline{\mathrm{v}}_{pos}[n] - \underline{\mathrm{v}}_{neg}[n] \tag{10}$$

where $\underline{\mathrm{v}}_{neg}$ derives from equation 6, but with $\underline{\mathrm{i}}_{neg}$ replacing $\underline{\mathrm{i}}$ in equation 5. The output, $\underline{\mathrm{v}}_{out}$, is a train of positive and negative Dirac pulses with magnitude v_{thresh}.

This spike train is passed into a synapse, which in our case is a straightforward first order low-pass filter with impulse response [12]:

$$\underline{\mathrm{h}}[n] = \frac{1}{\tau} e^{-n\Delta t/\tau}. \tag{11}$$

with τ as a time constant for the synapse and a hyperparameter for each neuron layer in the SNN. These synapses regulate the spiking signals and translate a given frequency to a single value. A larger value selection for τ for a given synapse results in a longer convergence time for the network. On the other hand, too small a selection of τ will increase "jitter" in the output, since this synapse does not substantially filter the spiking signal, which can lead to variation in accuracy. Table II gives the values used for this SNN. The outputs of the hidden layer neurons are then plugged into a $\tanh$ activation function and fed through the final weight matrix and synapse to acquire the output.

TABLE II: SNN Parameters

Parameter	Selection
Presentation Time	100 ms
Max. Neuron Firing Rate	600 Hz
Hidden Layer Synapses τ	0.01
Output Synapse τ	0.015

C. Loihi Implementation

The SNN was implemented using the Nengo Python libraries, which act as a class representation of the Loihi hardware. The inputs and output are Node objects in Nengo and correspond to I/O from the host to the Loihi's cores. The spiking layers correspond to Spiking ReLU neurons in the Nengo library, and correspond to a neuron block on the Loihi chip.

It is important to note that a key difference between the Nengo simulation and the Loihi hardware implementation is that Loihi exhibits "reset-to-zero" behavior. That is, instead of the update behavior as in equation 7, the Loihi hardware updates the neurons as the following:

$$\underline{\mathrm{v}}_{cap}[n+1] = \begin{cases} 0, & \text{if } \underline{\mathrm{v}}_{cap}[n] \geq v_{thresh} \\ \underline{\mathrm{v}}_{cap}[n], & \text{otherwise} \end{cases}. \tag{12}$$

In other words, if the voltage stored at a neuron for a given time-step is greater than or equal to the threshold voltage, then all the charge is effectively drained from the neuron. This discrepancy has been addressed in APIs supported directly by Intel, but is still being developed for Nengo implementations. Therefore, it is necessary to find an alternative workaround.

Since the problem has to do with $\underline{\mathrm{v}}_{cap}$ overshooting the threshold, simulating with smaller sampling period, Δt could decrease the amount of charge presented at a given time-step. This would lead to a smaller difference between $\underline{\mathrm{v}}_{cap}$ and v_{thresh} for the case when the accumulated voltage is greater than the threshold.

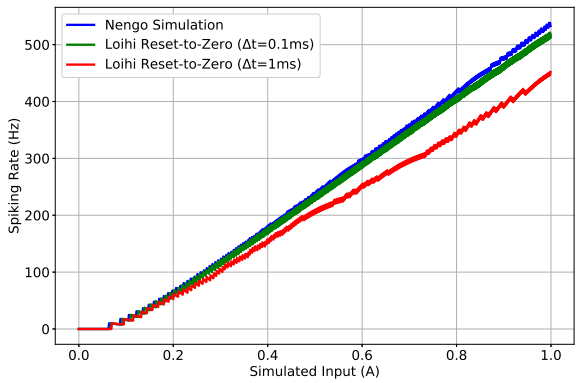

Fig. 2: Loihi Spiking ReLU neuron for different time-steps vs ideal Spiking ReLU neuron. Larger time-steps (i.e. sampling period) leads to lower frequency range, which causes higher aliasing in the neuron's spiking rate due to the reset-to-zero behavior.

On the flip-side, increasing the sampling rate also results in a higher number of computations for the Loihi, leading to longer computation times. Figure 2 shows how the Loihi's neuron's output changes based on different sampling periods. From these results, we selected $\Delta t = 0.1$ ms as the sampling period, as it outputs with satisfactory precision while still remaining reasonably fast to compute. Additionally, with this time-step selection, we found the accuracy of the neuron to decrease substantially after 600 Hz even with sampling period of 0.1 ms. Therefore, this was our selected maximum firing rate.

IV. Experimental Evaluation

We built the SNN model through Nengo and ran the SNN on the Intel Kapoho Beach, a USB device embedded with Loihi processors, as well as Nahuku FPGA board, a board integrated with a set of Loihi processors and tools for measuring energy consumption. While we tested the model on both, we use the time results from the Nahuku, as this board gives details about how much time each hardware component uses, whereas Kapoho Bay only reports the total computation time.

The output corresponded to a given index of the RUWO library. If the output strays too far from the target output, then the selected index becomes some number of bins off from the intended target index. Figure 3 shows how such error can occur. For our case, the simple library was arranged such that similar waveforms were located near each other in the library. Because of this, the error due to a misclassification could be mitigated to some extent through library organization.

A given test input was presented to the SNN for 100 simulated ms. With time-steps of 0.1 ms, the Nahuku would run 1000 time-steps to compute a given dataset example. While the total run-time for a single time-step was 3.51 ms, most of this time was due to I/O, which was primarily due to how Nengo interfaced with the Loihi hardware, and can be remedied in the future. Ignoring host-chip I/O communication, the device computed one time-step in 42 microseconds, leading to a 42 ms computation time per dataset example. Such a time would be comparable to the computation time of the ERA algorithm.

In regards to accuracy, The SNN converged to an output that corresponded to being within 2 indices of the optimal RUWO library waveform 96 percent of the time, with the worst case scenario still remaining within 3 indices. For our case, this translates to an output notch typically being within 10 Hz of the ideal notch or better, and a worst case scenario being about 15 Hz off from the ideal. Figure 4 shows an example of the worst case.

It may be possible to improve the wave quality further. One method would be to increase the null width of the notches. This would guarantee mitigation of interference, but would come with a toll to bandwidth efficiency. As it is, our null widths are wide enough to mitigate the peak interference frequencies for our example, but it would be important to consider this factor for more complex interference environments. Another possibility would be to increase the output range, which was normalized between $+/-1$, to a larger scale such as $+/-2$. It may be more difficult to train the ANN tightly for this case, but it could also allow for more leniency toward the SNN's approximations and improve the performance of the SNN overall.

V. Conclusion

We have developed the first application of an SNN implemented on Loihi hardware to classify a matched filter dataset. The waveform quality was close to that of RUWO, and the main computational work performed on the same time scale as the ERA algorithm. The results here provide a proof of concept baseline for the performance of an SNN on specialized hardware for a matched filter task. Future work should therefore go into training SNNs to learn the RUWO frequency domain directly, and potentially even learn to output an ideal signal directly from a time domain input. Such work has great potential to address important problems involving waveform design and interference mitigation.

Acknowledgment

This work was supported by the United States Air Force Sensors Directorate. However, the views and opinions expressed in this article are those of the authors and do not necessarily reflect the official policy or position of any agency of the U.S. government.

References

[1] M. Wicks, "Spectrum crowding and cognitive radar," *IEEE 2nd International Workshop on Cognitive Information Processing*, 2010.

[2] J. Fienup, "Phase retrieval algorithms: a comparison," *Applied Optics*, 1982.

[3] T. Higgins, T. Webster, and A. K. Shackelford, "Mitigating interference via spatial and spectral nulls," *IET International Conference on Radar Systems*, 2012.

[4] J. Boubin, A. M. Jones, and T. Bihl, "Neurowav: Toward real-time waveform design for vanets using neural networks," *IEEE VNC Conference*, 2019.

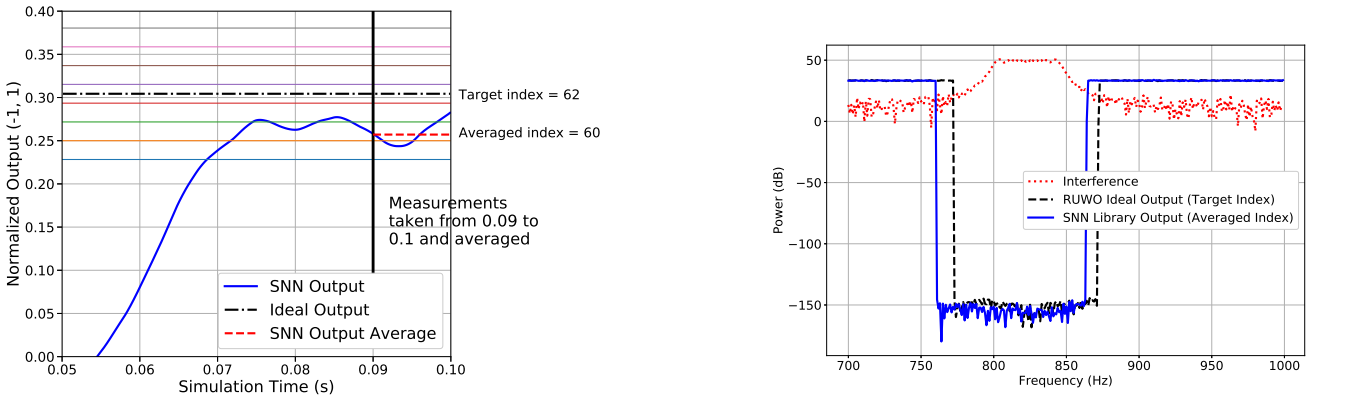

Fig. 3: (a): SNN converges but undershoots due to effects from the "reset-to-zero" behavior. The averaged measurement is 2 indices lower than desired (separation of indices shown by colored horizontal lines). (b): The output waveform is then shifted from ideal.

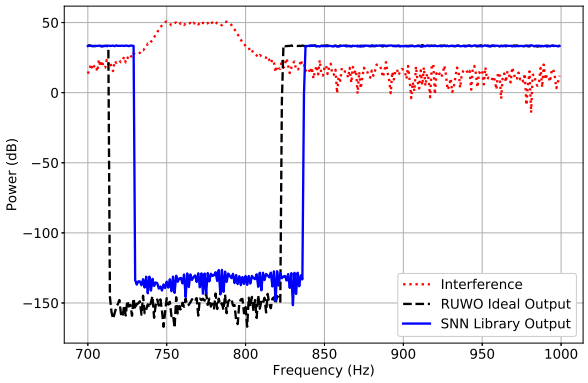

Fig. 4: Worst case scenario for SNN output. The output is shifted approximately 15 Hz off from ideal.

[5] J. Mun, K. Heasung, and J. Lee, "A deep learning approach for automotive radar interference mitigation," *IEEE 88th Vehicular Technology Conference (VTC-Fall)*, 2018.

[6] D. Li, X. Chen, M. Becchi, and Z. Zong, "Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus," *IEEE International Conferences on Big Data and Cloud Computing (BDCloud)*, 2016.

[7] G. Zhang and H. Rong, "An optimization spking neural p system for approximately solving combinatorial optimization problems," *International Journal of Neural Systems*, 2014.

[8] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, 2018.

[9] P. Blouw, X. Choo, E. Hunsberger, and C. Eliasmith, "Benchmarking keyword spotting efficiency on neuromorphic hardware," *arXiv:1812.01739*, 2019.

[10] C. W. Rossler, L. K. Patton, and B. Himed, "Rapid, near-optimal waveform adaptation: precompute and lookup via hash functions," *IET Radar, Sonar and Navigation*, 2013.

[11] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. Wiley, 2000.

[12] A. R. Voelker, "Dynamical systems in spiking neuromorphic hardware," Ph.D. dissertation, University of Waterloo, 2019.

[13] A. Yousefzadeh, S. Hosseini, P. Holanda, S. Leroux, T. Wever, T. Serrano-Gotarredona, B. L. Barranco, B. Dhoedt, and P. Simoens, "Conversion of synchronous artificial neural network to asynchronous spiking neural network using sigma-delta quantization," *IEEE International Conference on Artificial Intelligence Circuits and Systems*, 2019.

[14] T. Beckolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, 2014.