A Software Framework for Comparing Training Approaches for Spiking Neuromorphic Systems

Catherine D. Schuman
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
schumancd@ornl.gov

James S. Plank
Department of EECS
University of Tennessee
Knoxville, TN, USA
jplank@utk.edu

Maryam Parsa
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
parsam@ornl.gov

Shruti R. Kulkarni
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
kulkarnisr@ornl.gov

Nicholas Skuda
Department of EECS
University of Tennessee
Knoxville, TN, USA
nskuda@vols.utk.edu

J. Parker Mitchell
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
mitchelljp1@ornl.gov

***Abstract*—There are a wide variety of training approaches for spiking neural networks for neuromorphic deployment. However, it is often not clear how these training algorithms perform or compare when applied across multiple neuromorphic hardware platforms and multiple datasets. In this work, we present a software framework for comparing performance across four neuromorphic training algorithms across three neuromorphic simulators and four simple classification tasks. We introduce an approach for training a spiking neural network using a decision tree, and we compare this approach to training algorithms based on evolutionary algorithms, back-propagation, and reservoir computing. We present a hyperparameter optimization approach to tune the hyperparameters of the algorithm, and show that these optimized hyperparameters depend on the processor, algorithm, and classification task. Finally, we compare the performance of the optimized algorithms across multiple metrics, including accuracy, training time, and resulting network size, and we show that there is not one best training algorithm across all datasets and performance metrics.**

***Index Terms*—spiking neural networks, neuromorphic computing, genetic algorithms, decision trees**

I. Introduction

Neuromorphic computing systems have great promise as a future complementary computing paradigm to traditional von Neumann systems [1]. However, there are a variety of issues associated with programming neuromorphic systems. First, most neuromorphic computers implement spiking neural networks (SNNs), and there is currently no obvious way to train or design SNNs to perform different tasks. There are, in fact, a wide variety of different types of training algorithms, and it is not clear *a priori* which algorithm will be most successful in performing a given task. Second, different neuromorphic hardware systems can implement different types of computation (e.g., different types of neuron and synapse models) or different characteristics (e.g., precision of parameters in the network), which means that the same algorithm may have radically different performance on different neuromorphic hardware. Third, neuromorphic algorithms often have hyperparameters that can be difficult to manually tune for a given task. Finally, the performance of a neuromorphic system can be impacted by other factors beyond accuracy on a given task, such as power usage or model/network size, especially if the resulting solution is meant to be deployed in a real-world hardware implementation.

In this work, we present an approach for addressing these four issues. We discuss an updated version of the TENNLab neuromorphic computing framework [2], which enables a common interface across neuromorphic hardware systems to allow for ease of use for application and algorithm developers. We discuss four different ways to train SNNs that are now supported in this framework, including a new approach for mapping decision trees into SNNs and an approach for mapping a separately trained neural network using an approach called Whetstone into SNNs in this framework. We also discuss a newly implemented hyperparameter optimization approach that automatically tunes the hyperparameters of each of these algorithms to produce the best performance for that algorithm, dataset, and processor combination. In Section IV, we present results for classification tasks across four different datasets. We show how the hyperparameter optimization approach tunes the hyperparameters for the algorithm, dataset, and neuromorphic processor, and show the performance differences across algorithms and processors for each dataset.

The contributions of this work are:

- An approach for converting a decision tree into an SNN amenable for deployment on neuromorphic hardware

- A quantitative comparison of different training approaches for common classification tasks in terms of accuracy, time, and network size
- An illustration of the importance of hyperparameter optimization to achieve optimal performance of an algorithm and processor on a particular application
- A common software framework that allows for comparisons across algorithms, processors, and datasets, as well as hyperparameter optimization for each.

II. Background and Related Work

There are relatively few software frameworks in the realm of spiking neural simulators and neuromorphic hardware that support multiple backends. PyNN [3] is a Python interface that supports backends for a variety of computational neuroscience simulators including NEST and Brian, as well as neuromorphic hardware such as SpiNNaker with sPynnaker [4] and the BrainScaleS system [5]. PyNN is primarily focused towards computational neuroscience workloads and less so for machine learning applications. Nengo [6] is another software framework that provides a front-end for different types of simulators and hardware, including NengoFPGA [7] and Loihi [8]. Nengo is built on the philosophy of the neural engineering framework, which can make it difficult to port to new applications. Large-scale neuromorphic hardware efforts often develop their own toolchains, such as the software framework for IBM's TrueNorth [9] and the software framework for Intel's Loihi [10]. Because these software frameworks are customized specifically for each of those hardware platforms, it can be non-trivial to port an algorithm between frameworks, making it difficult to quickly evaluate an algorithm across multiple hardware platforms.

In this work, we use four different training approaches for SNNs, an evolutionary algorithm based approach called EONS [11], [12], a reservoir computing approach, a back-propagation-based training approach called Whetstone [13], and we introduce an approach for mapping a decision tree into an SNN for neuromorphic deployment. There are a wide variety of other training approaches for SNNs. Some approaches train a conventional artificial neural network and then map that network onto an SNN appropriate for neuromorphic deployment [14]–[16]. There are also a variety of approaches that adapt back-propagation in some way to accommodate for spiking neurons [17]–[20], as well as an approach that uses a version of back-propagation to train both weights and delays for SNNs [21]. Reservoir computing or liquid state machines are a common approach for utilizing SNNs, because they do not require training for the SNN component itself [22], [23]. It has been shown that optimizing the SNN reservoir, for example via evolutionary optimization, can improve performance [24]. There are also other approaches for evolving SNNs [25]–[27]. Though there is clearly a tremendous amount of work on algorithms for SNNs and neuromorphic computing, the key issue is that most algorithms are developed in isolation or for one particular neuromorphic hardware implementation. It is often unclear how to map the algorithm to a new hardware platform and difficult to estimate how performance might change because of hardware constraints. We seek to address that issue here by utilizing a common interface that all algorithms are implemented in or mapped to, and evaluate the algorithms across multiple hardware platforms to see how different hardware characteristics affect performance.

III. Approach

In this section, we discuss our overall approach. In Section III-A, we discuss the TENNLab neuromorphic computing software framework to enable a common interface across multiple neuromorphic processors and algorithms. In Section III-B, we discuss the three neuromorphic processors used in this work and their characteristics. In Section III-C, we cover four of the training algorithms evaluated. Finally, in Section III-D, we discuss our Bayesian hyperparameter optimization approach and how we use it to determine the best algorithmic hyperparameters in this work. The overall approach is shown in Figure 1, in which a dataset and neuromorphic processor is selected, and then the algorithms and hyperparameter optimization approach are used to produce candidate SNN solutions for the specified dataset and processor.

A. TENNLab Framework

The TENNLab neuromorphic computing framework [2] specifies a common interface between neuromorphic processors, learning algorithms and applications, so that neuromorphic processors can be exchanged and evaluated easily across multiple algorithms and applications. In the latest version of the TENNLab neuromorphic computing framework, we support applications, algorithms, and neuromorphic processor interfaces written in both C++ and Python.

The framework itself includes modules to perform spike encoding and decoding. Spike encoding supports rate and temporal coding, as well as less common encoding approaches such as binning, charge injection, and spike count encoding (described in more detail in [28]). Spike decoding approaches include winner-take-all, in which the output neuron that spikes the most specifies the output label or action, as well as decoding processes based on number of spikes, timing of spikes, or rate or spikes. In this work, we use the encoding/decoding approaches specific to the algorithm, or, if the algorithm supports multiple encoding approaches, we allow encoding approach as a hyperparameter of the algorithm and fix the decoding approach to winner-take-all.

B. Neuromorphic Processors

In this section, we describe different neuromorphic processors that are included and evaluated in this work. We use the word "processor" in this work to refer to these because they can represent neuromorphic simulators and/or neuromorphic hardware. Other neuromorphic or SNN processors/simulators are also supported in the TENNLab framework, including Brian2 [29], Nengo [6], and BindsNET [30], but we omit those in our study here. It is also worth noting that all of the neuromorphic processors described here have their own sets

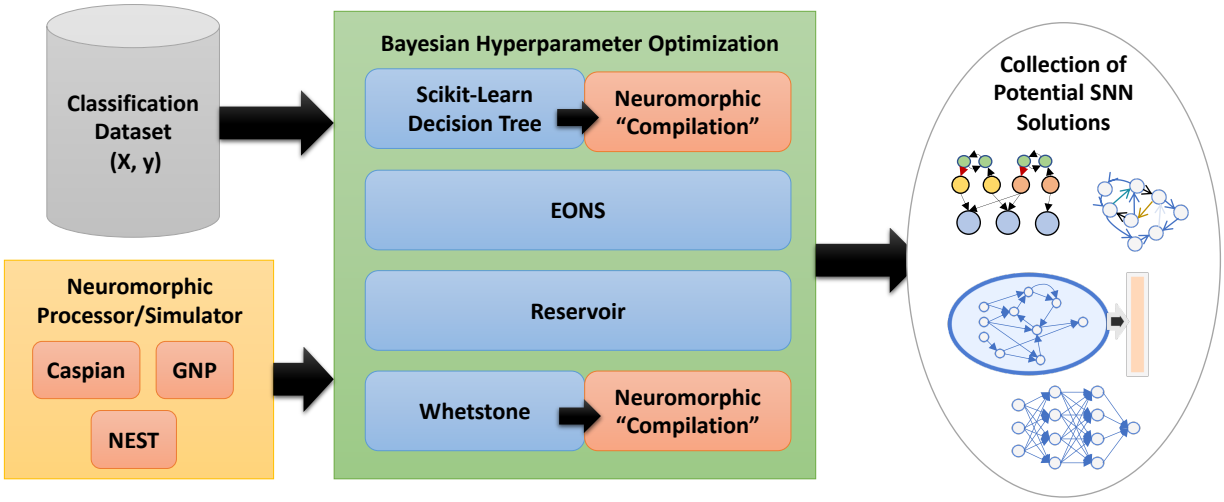

Fig. 1. Overall approach utilized in this work. A classification dataset and neuromorphic processor are selected. Then, four different training algorithms and a Bayesian hyperparameter optimization approach are used to produce a collection of SNN solutions.

of hyperparameters (e.g., neuron models, leak implementation, synaptic plasticity, precision of weights, delays or threshold). We fix the hyperparameters for these processors in this work, but the hyperparameter optimization process described in Section III-D may be used to optimize these parameters as well to inform future hardware design.

1) Caspian: Caspian [31] is a neuromorphic development platform that includes both a physical hardware implementation on an FPGA [32], as well as a hardware accurate software simulator, written in C++. Caspian implements leaky integrate and fire neurons and synapses with delay. In this work, we disable leak. Synaptic weights and neuron thresholds in Caspian are integer values, but the minimum and maximum values in the simulator are parameterizable. In this work, we use hardware-realistic values for the parameters, where the synaptic weights can range from -255 to 255, and the neuron threshold values can range from 0 to 255.

2) Generic Neuromorphic Processor (GNP): The Generic Neuromorphic Processor (GNP) is a parameterizable neuromorphic simulator developed by the TENNLab research group [33]. GNP is an event-driven spiking simulator written in C++ that allows for customizable neuron and synapse models. GNP includes several neuron models with different types of leak implementations, as well as multiple synapse models with different forms of plasticity. The parameters of GNP can be set to be integer or floating point values, based on what is supported by the neuromorphic processor that is being simulated. In this work, we utilize integrate-and-fire neurons with no leak and synapses with no plasticity mechanisms. Synaptic weights and neuron thresholds are continuous (i.e., floating point) values, where the valid ranges are -1 to 1 and 0 to 1, respectively.

3) NEST: The NEST Neural Simulator [34], [35] is simulator for systems of spiking neurons and can be configured as a simulator for a neuromorphic hardware implementation. In this work, we use PyNEST [35] and specify a specific neuron model (iaf_psc_delta) and a simple synapse model; we limit to specific weight (-15 to 15), threshold (1 to 15), and delay (1 to 15) ranges. However, depending on the hardware implementation, these ranges and the neuron and synapse model can be updated. If they are updated, the algorithm mappings described below for Whetstone and decision trees will need to be updated to accommodate for those model changes.

C. Training Algorithms

1) EONS: Evolutionary Optimization for Neuromorphic Systems (EONS) [11], [12] is an evolutionary approach for designing and training SNNs for neuromorphic deployment. EONS begins by initializing a population of randomly generated SNN solutions for a given task and neuromorphic implementation. Then, EONS evaluates all of the networks in that population by loading them into the neuromorphic implementation and testing them on a task to produce a fitness score. In this work, we focus on classification tasks, so evaluating the network entails taking each data instance in the training set, running it through the network, and collecting the output to produce an accuracy score on the training set. Once EONS has evaluated all of the networks (which can be performed in parallel), EONS performs selection to preferentially select better performing networks to serve as parents, and then reproduction operations of duplication, crossover, and mutation are used to produce children networks from those parents. EONS hyperparameters include population size, crossover rate, and mutation rate.

EONS has several advantages over other training approaches for SNNs for neuromorphic deployment. First, it can determine both the structure and parameters of the network, and it can produce networks without any connectivity constraints. Second, it designs directly for a particular neuromorphic hardware implementation and thus does not require a mapping procedure

such as those described below. Third, it can work with a variety of spike encoding and decoding approaches. Fourth, it is not restricted to classification tasks; it can be trivially applied to other types of tasks such as control and anomaly detection because it simply requires a fitness function that scores how well the SNN performs the task. The key disadvantage of using EONS is that it often requires significantly more time to train than other training approaches and in its current form, does not scale well to larger input spaces.

2) Reservoir Computing: We implement a simple reservoir computing approach. We create a reservoir by specifying the input encoding approach, the number of hidden neurons, the number of output neurons, and the probability of connection between any two neurons in the network. Any two neurons can be connected to each other via a synapse. We run each data instance through the encoding process which creates spike events in the network and simulate for a number of time steps. We collect the number of spikes on each output neuron and form a vector from those values. This vector is then sent as output to a (non-spiking) readout layer. Here, we use the SGDClassifier from scikit-learn as our readout layer, with default parameters. We specify the encoding approach, the number of hidden neurons, the number of output neurons, and the probability of connection as hyperparameters of this approach.

3) Whetstone: Whetstone [13] is an approach for training neural networks with binary activation functions that can be mapped to spiking neuromorphic systems. Whetstone begins with a typical activation function throughout the network, such as a rectified linear unit (ReLU) and then over the course of training, it gradually "sharpens" the activation functions throughout the network to be closer to binary activations where the threshold is 0.5. Whetstone allows for a n-hot output encoding to prevent the "dead neuron" issue, and it uses a key to decode the output before passing to a softmax function. In this work, we restrict our attention to training and mapping dense feed-forward neural networks into SNNs, though there are approaches to map convolutional neural networks to SNNs for neuromorphic deployment as well [36]. Whetstone is written in Python and is built on top of Keras.

There are several steps to map a network from one that is produced by Whetstone to one that can be processed by a neuromorphic implementation. First, a quirk of Whetstone is that the input layer itself is not a binary activation. Thus, we cannot directly encode the input layer to first hidden layer computation in an SNN. To accommodate this, we instead perform a "pre-processing" step of calculating the first layer. We perform the weight matrix multiplication on the input vector and add in appropriate biases for that layer. In the converted SNN, our input layer corresponds to the first hidden layer in the Whetstone network, and the pre-processed values are used as the input values directly to that layer. This is the most dramatic change in network structure from the Whetstone network to the corresponding SNN and requires significant pre-processing that is not performed on a neuromorphic system.

Second the synaptic weight values and biases in Whetstone are floating point values and need to be scaled and rounded to the appropriate precision levels for a given neuromorphic implementation. For example, for the Caspian implementation, we multiply weight values by 255 and round to integers. This reduction of precision and perhaps limiting of the range of the values can cause a decrease in the accuracy of the mapped network from the one that is trained in Whetstone. Third, to inject bias values into the network, we create a single neuron that is connected to every other neuron in the SNN and we set the weights of those synapses to be each of those neuron's scaled bias value and the delays to be such that their bias value will arrive when the rest of the data is propagated through the network and arrives at that neuron. Finally, we still have to use the decoding key and softmax layer from Whetstone training to produce an output. This requires that we take the output spikes from the output layer of the SNN, multiply by the key vector, and take the softmax function, all of which occurs outside the neuromorphic implementation as well.

In the case of Whetstone, the hyperparameters are the number of hidden layers, the number of neurons per hidden layer, and n for the output encoding. We also include a hyper-parameter for constraining the max norm of the weights, which can help in the mapping procedure from the ideal Whetstone network to the actual neuromorphic implementation.

4) Decision Tree: We introduce in this work an approach for mapping a decision tree into an SNN suitable for deployment onto a neuromorphic processor. We use scikit-learn's decision tree classifier to produce the decision tree. We also perform a MinMaxScaler transform on the data to scale it into the range $[0, 1]$ prior to training the decision tree. We make an assumption that charge values can be injected directly into the input neurons in the neuromorphic implementation. The TENNLab neuromorphic framework assumes charge values will be between 0 and 1 and that the neuromorphic implementation will scale them appropriately for the input ranges that are valid for that system.

The mapping procedure then parses the decision tree string produced by scikit-learn to determine the decision nodes and the leaf nodes. For each decision node (labeled as $x[i] \leq a$), two input neurons are constructed. One of the input neurons corresponds to the $x[i] \leq a$ case and has its threshold set as 0 or the minimum value possible for the neuromorphic processor. The other input neuron corresponds to the $x[i] > a$ case, and has its threshold set to aT_{max}, where T_{max} is the maximum threshold value that can be realized by the neuromorphic processor. A single output neuron is created for each leaf node, and its threshold value is set to the number of conditions that are required to be met before reaching that node in the decision tree, multiplied by a value ω, which is a small weight value that is allowed in the neuromorphic implementation. A synapse is created between each decision node and the leaf nodes that are further down the tree from that node. That synapse is set to have a small delay value (the smallest possible delay value allowed), and the weight value is set to ω.

We provide as input the values $x[i]$ to each of the decision nodes that require $x[i]$; however, we stimulate the decision nodes corresponding to "greater than" decisions at time 0 and decision nodes corresponding to "less than or equal to" decisions at time 100. We then create a small inhibition loop consisting of two additional neurons to drive the neuronal charge on the "less than or equal to" decision nodes to the minimum charge value if and only if the corresponding "greater than" neuron fires. Thus, for each decision node in the decision tree, only one of the two corresponding input neurons will fire. An example of a decision tree for the iris data set is shown in Figure 2 and its corresponding SNN is shown in Figure 3 to illustrate the mapping procedure.

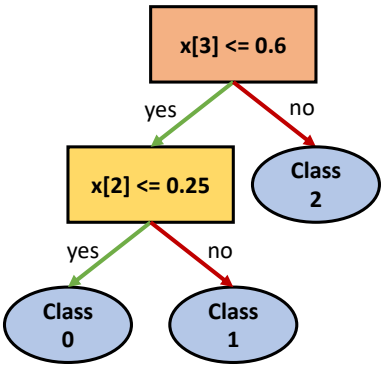

Fig. 2. Example decision tree for the iris dataset.

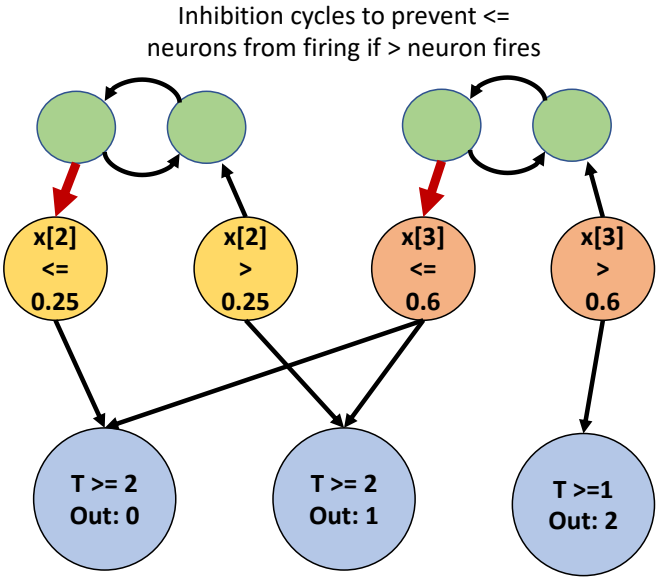

Fig. 3. SNN for the decision tree shown in Figure 2. Each decision node is mapped to two input neurons, one corresponding to less than or equal and the other to greater than. Both are connected to all of their downstream leaf nodes. The leaf nodes have thresholds set to the total number of incoming synapses, so that they only fire if all of the decision neurons they are connected to fire. Finally, the greater than input nodes are stimulated first and if they fire, they stimulate a loop that continuously inhibits the less than or equal to neuron, preventing it from firing when it receives external stimulus. The colors of the decision and leaf nodes correspond to the colors in Figure 2, and the strongly inhibitory synapse is shown in red.

For clarity, we include an example of how a data example would be processed in the SNN shown in Figure 3. We will assume that the thresholds of the "greater than" input neurons are exactly as shown in the figure, while the thresholds of the "less than or equal to" input neurons are 0. Suppose $x = [0.2, 0.3, 0.4, 0.5]$. Since only $x[2] = 0.4$ and $x[3] = 0.5$ are required, we will create a spike with charge value 0.4 on the neuron corresponding to $x[2] \leq 0.25$ at time 100 and a spike with charge value 0.4 on the neuron corresponding to $x[2] > 0.25$ at time 0. Similarly, we will create a spike with charge value 0.5 on the neuron corresponding to $x[3] \leq 0.6$ at time 100 and a spike with charge value 0.6 on the neuron corresponding to $x[3] > 0.6$ at time 0. At time 0, because $0.4 > 0.25$ this will cause neuron $x[2] > 0.25$ to fire, sending a spike along to the output neuron corresponding to output 1. It will also create a spike along the synapse to the inhibition cycle. This inhibition cycle will begin a loop of both of the green neurons firing every other cycle and the second one sending an inhibitory signal of maximum strength to the input neuron corresponding to $x[2] \leq 0.25$. By the time the value of 0.4 is injected into this input neuron at time 100, its charge value will be at an extremely low value, ensuring that the addition of 0.4 will not trigger that neuron to fire. On the other hand, the input value of $x[3] = 0.5$ will not trigger the input neuron corresponding to $x[3] > 0.6$, thus the inhibition cycle there will not be activated. When the value of $x[3] = 0.5$ stimulates the input neuron $x[3] \leq 0.6$ at time 100, it will satisfy the threshold of 0 and cause that neuron to fire a spike to the output neuron corresponding to output 1. Output 1 will have received 2 spikes at that point and will fire, and thus, will be the chosen output label.

It is worth noting that this approach requires that the threshold values of the output neurons be able to realize the number of values corresponding to the maximum depth of the tree. This means that the if the decision tree is sufficiently complex, the neuromorphic implementation may not be able to realize it. In the case of the neuromorphic implementations we evaluate in this work, we have placed an artificial limit on threshold values for NEST to be integers between 1 and 15, inclusive. As such, if the depth of the decision tree is more than 15, the SNN cannot be realized in that implementation.

D. Bayesian Hyperparameter Optimization

Bayesian hyperparameter optimization is a data efficient optimization approach suitable for black-box and expensive objective functions such as performance of a computing system. This technique is based on leveraging prior belief and current observations on the performance trend, estimating a posterior belief, and updating these beliefs using new observations. In this optimization approach, observation is a performance value (e.g., accuracy) for a set of hyperparameter combination. Since each observation is expensive to evaluate and time-consuming, the goal is to predict a posterior distribution close to the

actual performance trend of the system with minimum required observations.

Bayesian hyperparameter optimization begins with sets of random hyperparameters to build an initial prior distribution, and likelihood model, calculating a surrogate model to explore and exploit the search space (which is all possible hyperparameter combinations), and optimizing this surrogate model to determine a set of hyperparameter to explore in the next iteration. In this work we optimized performance of the aforementioned training algorithms (single objective Bayesian optimization) using python's scikit-optimize [37] package. For further details please refer to [38], [39].

IV. Results

We evaluate each of these four algorithms across all three neuromorphic simulators/processors for four different simple classification datasets from scikit-learn's toy dataset collection: iris, wine, breast cancer, and digits. A summary of characteristics of those datasets is given in Table I. Some of the algorithm/processor/dataset combinations could not be completed due to compute time constraints and are omitted in the results. For example, the NEST simulator is significantly slower than the other two processors and EONS requires hundreds of thousands of simulation runs, so none of the NEST simulations completed hyperparameter optimization.

TABLE I
Dataset Characteristics

Dataset	Dimensionality	Classes	Instances
Iris	4	3	150
Wine	13	3	178
Breast Cancer	30	2	569
Digits	64	10	1797

A. Hyperparameter Optimization Results

We completed single objective Bayesian hyperparameter optimization on all processor, dataset, and algorithm combinations that could complete in a reasonable amount of time. We did not perform a hyperparameter optimization on the decision tree algorithm, though we could potentially optimize the hyperparameters in the scikit-learn implementation in the future. For each combination, we completed 75 iterations of Bayesian optimization. For each iteration, five evaluations of that hyperparameter set are completed to obtain an idea of the variation in performance. Figure 4 shows the results of three of those Bayesian optimization runs on the wine dataset, one with Caspian and EONS (top), one with NEST and reservoir computing (middle), and one with Whetstone (bottom). Here, the hyperparameters of the Whetstone algorithm are optimized independently of the processor. As can be seen in Figure 4, there is significant variation in performance based on the selection of hyperparameters. This demonstrates clearly that if hyperparameters are not optimized, the algorithmic performance can suffer significantly. Figure 4 also shows how the Bayesian optimization process performs exploration and exploitation within the search space of potential hyperparameters, showing an improvement in general performance as iterations increase, but still exploring the space (resulting in different hyperparameter combinations and different performance).

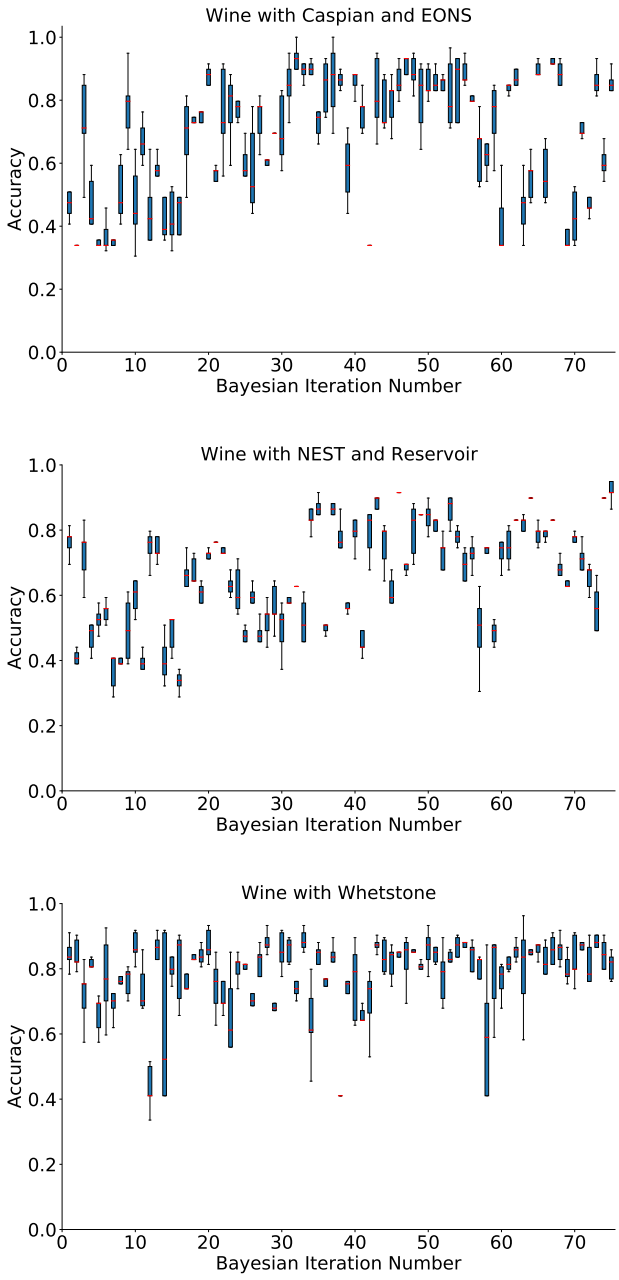

Fig. 4. Hyperparameter optimization results across 75 iterations in the wine dataset for each of the three algorithms in which hyperparameters were optimized (EONS, Reservoir, and Whetstone).

Tables II-IV gives a brief overview of the best hyperparameters for some of the datasets, algorithms, and processor combinations. We omit the full optimized hyperparameter sets here due to space constraints. However, these tables show the

TABLE II
EONS HYPERPARAMETER OPTIMIZATION RESULTS

	Iris		Wine	
	Caspian	**GNP**	**Caspian**	**GNP**
Population	50	40	50	50
Crossover rate	0.9	0.9	0.7	0.4
Mutation rate	0.9	0.9	0.9	0.6
Encoding interval	15	15	25	50
Encoding function	triangle	triangle	flip_flop	flip_flop
No. of bins	2	4	2	2
Min/max values	0.75, 1.0	0.75, 1.0	0.25, 0.75	0, 0.5
# max spikes	2	8	12	10
Sim time	55	100	85	95

TABLE III
RESERVOIR HYPERPARAMETER OPTIMIZATION RESULTS

	Iris		
	Caspian	**GNP**	**Nest**
No of hidden nodes	100	200	150
No. of output nodes	50	40	35
Probability of connection	0.1	0.1	0.15
Encoding interval	30	30	45
Encoding function	triangle	triangle	triangle
No. of bins	4	2	2
Min/max values	0, 0.25	0.25, 0.75	0.5, 0.75
# max spikes	12	8	2
Sim time	65	55	65

importance of optimizing hyperparameters for the dataset, the processor *and* the algorithm. For example, both EONS and reservoir share input encoding hyperparameters, but as we can see in Tables II and III, but the optimal encoding parameters differ even for the same dataset/processor. That is, though all processors and algorithms on the iris dataset share the "triangle" encoding function, the encoding intervals, number of bins, minimum and maximum values and number of spikes differ between datasets and algorithm for the same processors. Similarly, it is important to customize the hyperparameters for each dataset. Table II shows that the best encoding function for the iris dataset using EONS is triangle, but the best encoding function for the wine dataset using EONS is flip-flop (see [28] for a full description of the distinction between triangle and flip-flop). Table IV also shows that it is important to customize the algorithmic hyperparameters to the dataset; in this case, the structure of the feed-forward network used is significantly different across datasets. Table III shows the importance of customizing algorithmic hyperparameters across processors; in this case, we can see that the optimal size of the reservoir used depends on the characteristics of the selected processor. In general, it is clear from these results that we cannot expect one set of hyperparameters for an algorithm to work well in all cases, nor should we dismiss an algorithmic approach simply because one set of hyperparameters results in poor performance. Instead, we should plan to tune or optimize the hyperparameters for the dataset, algorithm, and processor.

B. Algorithm Comparison

Once hyperparameter optimization was complete, we ran 100 different evaluations of each of the algorithm, processor, and dataset combinations that completed a Bayesian optimization run in a reasonable amount of time (five days) with their corresponding best hyperparameters. Figure 5 shows the results for test accuracy for each of those combinations. There are several things to note from this figure. First, the Whetstone algorithm suffers with smaller datasets (iris and wine) and results in significantly larger variation in performance on these tasks when compared with the other training approaches. Second, Whetstone performs the best on the largest dataset with the largest input space (digits), whereas as expected, EONS performs poorly on this task. Third, for all algorithms, we can see a variation in performance across processors, due to the characteristics of the processors themselves. For example, GNP's higher weight resolution can both help performance in some cases (e.g., on the wine dataset with Whetstone), but it can also hurt performance in some cases (e.g., on the wine dataset with EONS). Fourth, the newly described decision tree approach results in very good performance for the smaller datasets, but when the processor has limited precision (in the case of our configuration of NEST), it can suffer greatly (e.g., on both the wine and digits results for NEST).

Tables V-VIII show summaries of the results for the mean training and testing accuracies, mean training time (in seconds), and mean size of the network (in terms of average number of neurons and synapses) across 100 runs for each of the algorithm, processor, and dataset combinations. The best performing algorithm in each category is bolded in each column. With these tables, we can see stark differences across the different training algorithms in terms of different aspects of performance. First, the decision tree approach tends to give the fastest training times, but the test accuracy can suffer, especially for the digits dataset (shown in Table VIII). Second, EONS often results in networks that are competitive with the best approach in accuracy (except on the digits dataset), but the resulting networks are the smallest. However, training

TABLE IV
WHETSTONE HYPERPARAMETER OPTIMIZATION RESULTS

	Iris	**Wine**	**Digits**	**Breast cancer**
Hidden layer structure	25	10,5,10	100	10,5,5,10
N hot output	2	5	10	2
Max norm constraint	1	4	1	1

TABLE V
SUMMARY OF IRIS RESULTS

Algorithm	**Proc**	**Train Acc**	**Test Acc**	**Train Time (sec)**	**Neurons**	**Synapses**
EONS	caspian	0.92	0.93	2.15	**7.85**	9.73
	gnp	0.94	0.94	2.47	8.29	**7.76**
Reservoir	caspian	0.93	0.95	0.07	166.0	2731.0
	nest	0.95	0.95	114.67	193.0	5562.0
Whetstone	caspian	0.78	0.72	0.9	32.0	154.75
	gnp	0.79	0.72	0.9	32.0	156.0
	nest	0.76	0.7	0.9	32.0	134.73
Decision Tree	caspian	**1.0**	0.93	**0.0**	11.0	13.0
	gnp	**1.0**	0.93	**0.0**	11.0	13.0
	nest	0.97	**0.96**	**0.0**	11.0	13.0

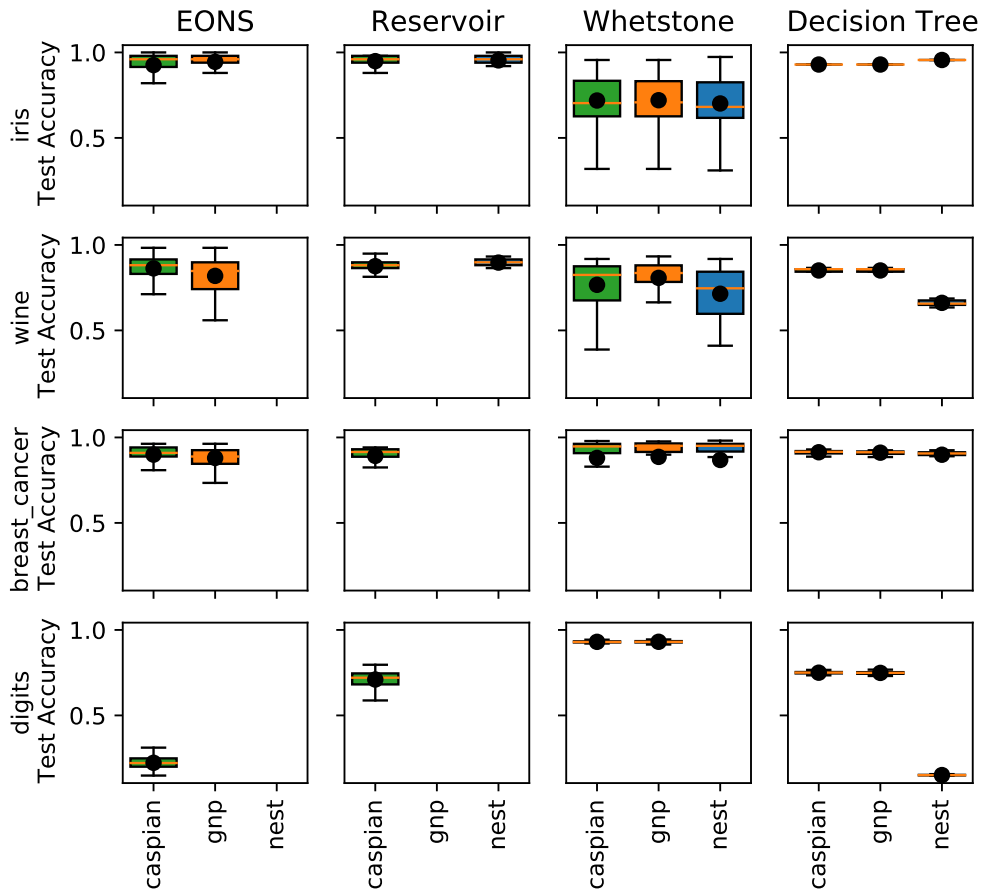

Fig. 5. Test accuracy across 100 runs for each of the algorithm processor, and dataset combinations. The algorithm/processor/dataset combinations that are not included are those that could not be completed because of compute time constraints.

TABLE VI
SUMMARY OF WINE RESULTS

Algorithm	Proc	Train Acc	Test Acc	Train Time (sec)	Neurons	Synapses
EONS	caspian	0.9	0.86	4.43	8.88	7.61
	gnp	0.84	0.82	4.19	**8.01**	**6.52**
Reservoir	caspian	0.89	0.88	0.03	137.0	934.0
	nest	0.96	**0.9**	84.82	176.0	3083.0
Whetstone	caspian	0.85	0.77	1.55	41.0	278.31
	gnp	0.9	0.81	1.55	41.0	280.0
	nest	0.79	0.71	1.55	41.0	250.93
Decision Tree	caspian	**1.0**	0.85	**0.0**	16.0	21.0
	gnp	0.99	0.85	**0.0**	16.0	21.0
	nest	0.78	0.66	**0.0**	16.0	21.0

times for EONS are typically significantly longer than the Whetstone or decision tree approaches. Third, both reservoir and Whetstone networks tend to be much larger than either EONS or reservoir networks.

We can also see in these tables that differences in the processors can result in different accuracy values for the resulting networks. For example, in Table VI, we can see that the testing accuracy results are different across processors for each algorithm. In this case, these differences are due to the different precision level and parameter ranges of the weights, thresholds, and delay values in these processors. In this particular dataset (wine), restricted precision values helped in the performance of both EONS and reservoir, but decreased performance for both Whetstone and the decision tree approach. In this case, we can see an advantage to using training approaches that take into account the actual processor during the training process, rather than performing a training approach and then performing a fixed mapping procedure (as is done in the Whetstone and decision tree approaches). We speculate the reduced precision improved performance in the case of EONS and reservoir because reduced precision can mitigate effects of overfitting during training in some cases.

V. DISCUSSION AND FUTURE WORK

In this work, we have described a software framework for comparing training approaches across spiking neuromorphic systems. We have introduced a new approach for mapping decision trees into SNNs suitable for deployment to neuromorphic hardware, and we have shown that this approach is

TABLE VII
SUMMARY OF BREAST CANCER RESULTS

Algorithm	Proc	Train Acc	Test Acc	Train Time (sec)	Neurons	Synapses
EONS	caspian	0.89	0.9	23.31	**4.95**	**3.67**
	gnp	0.88	0.88	10.95	5.33	3.87
Reservoir	caspian	0.89	0.89	0.33	195.0	3788.0
Whetstone	caspian	0.88	0.88	4.06	35.0	188.01
	gnp	0.89	0.89	4.06	35.0	189.0
	nest	0.87	0.87	4.06	35.0	171.72
Decision Tree	caspian	0.98	**0.91**	**0.0**	46.0	72.0
	gnp	**0.99**	**0.91**	**0.0**	46.0	72.0
	nest	0.91	0.9	0.01	46.0	72.0

TABLE VIII
SUMMARY OF DIGITS RESULTS

Algorithm	Proc	Train Acc	Test Acc	Train Time (sec)	Neurons	Synapses
EONS	caspian	0.24	0.22	69.86	**11.76**	**7.75**
Reservoir	caspian	0.74	0.71	2.76	328.0	5362.0
Whetstone	caspian	**1.0**	**0.93**	3.22	201.0	9880.57
	gnp	**1.0**	**0.93**	3.22	201.0	10100.0
Decision Tree	caspian	**1.0**	0.75	**0.01**	331.0	778.0
	gnp	**1.0**	0.75	**0.01**	331.0	778.0
	nest	0.13	0.15	**0.01**	241.0	514.0

competitive with other more conventional training approaches for SNNs on simple classification tasks. We have also demonstrated a hyperparameter optimization approach for choosing the hyperparameters of each algorithm, and we have shown that the optimized hyperparameters depend on the processor, dataset, and algorithm. Additionally, we have demonstrated the results for each optimized hyperparameter set across algorithms, datasets, and processors. We have shown that there is not one winning algorithmic approach, especially when metrics such as training time or network size are considered. This demonstrates that the future of neuromorphic computing is likely to include a wide variety of algorithmic approaches and that a software framework such as this one will be necessary to enable usability of neuromorphic systems moving forward.

There are several future directions for this work that we intend to pursue. First, we have limited ourselves to optimizing hyperparameters associated with each algorithm in this work. However, neuromorphic processors or simulators often also have hyperparameters that can be tuned. For example, the precision of the weight parameters or the range of values that can be realized can potentially be adjusted in the physical hardware. This process can be used to determine the best values for these processor hyperparameters in the future, and thus can be used as part of an application-hardware co-design process.

In this work, we have also limited our focus to optimization based entirely on accuracy. However, we have previously illustrated a novel hierarchical Bayesian optimization process that can be used to perform multi-objective optimization [39]. As can be seen in Tables V-VIII, the network sizes and training times are radically different across algorithms. Our hyperparameter optimization process can be used to optimize these performance metrics, for example, to minimize training time or minimize the resulting network size. In the future, we also intend to include energy estimates from each processor/simulator and to use our multi-objective optimization approach to maximize accuracy while minimizing energy usage for each algorithm.

It is clear from these results that the different algorithms have different strengths across the datasets, and that the performance for even a single algorithm across the same dataset can differ significantly. We have previously shown that ensembling the results from networks trained using EONS can significantly improve performance on different tasks [40]. In the future, we intend to explore ensembling the SNN results obtained through this framework. With this approach, we can ensemble results obtained across different hyperparameters for the same algorithm, or even ensemble the results across multiple algorithms.

Finally, in this work we have focused our attention entirely on simple classification tasks to demonstrate with large-scale tests the value of a framework that can use multiple training approaches to discover a variety of neuromorphic solutions for different neuromorphic processors. We intend to apply this framework to more complex datasets, and we are interested in expanding this approach to account for control applications as well. We have previously utilized EONS to train networks for a variety of control tasks [12], [41]. However, extending reservoir computing, Whetstone (and other back-propagation-based training approaches), and decision trees to control tasks is non-trivial. We intend to explore reinforcement learning-based approaches in adapting these algorithms for control tasks in the future.

ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725, and in part by an Air Force Research Laboratory Information Directorate grant (FA8750-16-1-0065).

This research used resources of the Compute and Data Environment for Science (CADES) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *arXiv preprint arXiv:1705.06963*, 2017.

[2] J. S. Plank, C. D. Schuman, G. Bruer, M. E. Dean, and G. S. Rose, "The tennlab exploratory neuromorphic computing framework," *IEEE Letters of the Computer Society*, vol. 1, no. 2, pp. 17–20, 2018.

[3] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: a common interface for neuronal network simulators," *Frontiers in neuroinformatics*, vol. 2, p. 11, 2009.

[4] O. Rhodes, P. A. Bogdan, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. Rowley *et al.*, "spynnaker: a software package for running pynn simulations on spinnaker," *Frontiers in neuroscience*, vol. 12, p. 816, 2018.

[5] E. Müller, S. Schmitt, C. Mauch, S. Billaudelle, A. Grübl, M. Güttler, D. Husmann, J. Ilmberger, S. Jeltsch, J. Kaiser *et al.*, "The operating system of the neuromorphic brainscales-1 system," *arXiv preprint arXiv:2003.13749*, 2020.

[6] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. Voelker, and C. Eliasmith, "Nengo: a python tool for building large-scale functional brain models," *Frontiers in neuroinformatics*, vol. 7, p. 48, 2014.

[7] B. Morcos, "Nengofpga: an fpga backend for the nengo neural simulator," Master's thesis, University of Waterloo, 2019.

[8] P. Blouw, X. Choo, E. Hunsberger, and C. Eliasmith, "Benchmarking keyword spotting efficiency on neuromorphic hardware," in *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop*, 2019, pp. 1–8.

[9] J. Sawada, F. Akopyan, A. S. Cassidy, B. Taba, M. V. Debole, P. Datta, R. Alvarez-Icaza, A. Amir, J. V. Arthur, A. Andreopoulos *et al.*, "Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 130–141.

[10] C.-K. Lin, A. Wild, G. N. Chinya, Y. Cao, M. Davies, D. M. Lavery, and H. Wang, "Programming spiking neural networks on intel's loihi," *Computer*, vol. 51, no. 3, pp. 52–61, 2018.

[11] C. D. Schuman, J. S. Plank, A. Disney, and J. Reynolds, "An evolutionary optimization framework for neural networks and neuromorphic architectures," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 145–154.

[12] C. D. Schuman, J. P. Mitchell, R. M. Patton, T. E. Potok, and J. S. Plank, "Evolutionary optimization for neuromorphic systems," in *Proceedings of the Neuro-inspired Computational Elements Workshop*, 2020, pp. 1–9.

[13] W. Severa, C. M. Vineyard, R. Dellana, S. J. Verzi, and J. B. Aimone, "Training deep neural networks for binary communication with the whetstone method," *Nature Machine Intelligence*, vol. 1, no. 2, pp. 86–94, 2019.

[14] P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni, and E. Neftci, "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 2016, pp. 1–8.

[15] X. Ju, B. Fang, R. Yan, X. Xu, and H. Tang, "An fpga implementation of deep spiking neural networks for low-power and fast classification," *Neural computation*, vol. 32, no. 1, pp. 182–204, 2020.

[16] P. Gu, R. Xiao, G. Pan, and H. Tang, "Stca: Spatio-temporal credit assignment with delayed feedback in deep spiking neural networks." in *IJCAI*, 2019, pp. 1366–1372.

[17] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in neuroscience*, vol. 10, p. 508, 2016.

[18] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1311–1318.

[19] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, p. 331, 2018.

[20] S. Yin, S. K. Venkataramanaiah, G. K. Chen, R. Krishnamurthy, Y. Cao, C. Chakrabarti, and J.-s. Seo, "Algorithm and hardware design of discrete-time spiking neural networks based on back propagation with binary activations," in *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 2017, pp. 1–5.

[21] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," *arXiv preprint arXiv:1810.08646*, 2018.

[22] N. Soures and D. Kudithipudi, "Deep liquid state machines with neural plasticity for video activity recognition," *Frontiers in neuroscience*, vol. 13, p. 686, 2019.

[23] P. Wijesinghe, G. Srinivasan, P. Panda, and K. Roy, "Analysis of liquid ensembles for enhancing the performance and accuracy of liquid state machines," *Frontiers in neuroscience*, vol. 13, p. 504, 2019.

[24] J. J. Reynolds, J. S. Plank, and C. D. Schuman, "Intelligent reservoir generation for liquid state machines using evolutionary optimization," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[25] J. D. Schaffer, "Evolving spiking neural networks for robot sensory-motor decision tasks of varying difficulty," in *Proceedings of the Neuro-inspired Computational Elements Workshop*, 2020, pp. 1–7.

[26] E. Eskandari, A. Ahmadi, S. Gomar, M. Ahmadi, and M. Saif, "Evolving spiking neural networks of artificial creatures using genetic algorithm," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 411–418.

[27] J. D. Schaffer, "Evolving spiking neural networks: A novel growth algorithm corrects the teacher," in *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*. IEEE, 2015, pp. 1–8.

[28] C. D. Schuman, J. S. Plank, G. Bruer, and J. Anantharaj, "Non-traditional input encoding schemes for spiking neuromorphic systems," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–10.

[29] M. Stimberg, R. Brette, and D. F. Goodman, "Brian 2, an intuitive and efficient neural simulator," *Elife*, vol. 8, p. e47314, 2019.

[30] H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, "Bindsnet: A machine learning-oriented spiking neural networks library in python," *Frontiers in neuroinformatics*, vol. 12, p. 89, 2018.

[31] J. P. Mitchell, C. D. Schuman, R. M. Patton, and T. E. Potok, "Caspian: A neuromorphic development platform," in *Proceedings of the Neuro-inspired Computational Elements Workshop*, 2020, pp. 1–6.

[32] J. P. Mitchell, C. D. Schuman, and T. E. Potok, "A small, low cost event-driven architecture for spiking neural networks on fpgas," in *International Conference on Neuromorphic Systems 2020*, 2020, pp. 1–4.

[33] N. Skuda, J. S. Plank, and C. D. Schuman, "Gnp: A configurable spiking simulator for rapid learning," 2021, in preparation.

[34] M.-O. Gewaltig and M. Diesmann, "Nest (neural simulation tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[35] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, "Pynest: a convenient interface to the nest simulator," *Frontiers in neuroinformatics*, vol. 2, p. 12, 2009.

[36] J. S. Plank, J. Zhao, and B. Hurst, "Reducing the size of spiking convolutional neural networks by trading time for space," in *IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, December 2020, pp. 115–126.

[37] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, "scikit-optimize/scikit-optimize," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4014775

[38] M. Parsa, J. P. Mitchell, C. D. Schuman, R. M. Patton, T. E. Potok, and K. Roy, "Bayesian-based hyperparameter optimization for spiking neuromorphic systems," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 4472–4478.

[39] ——, "Bayesian multi-objective hyperparameter optimization for accurate, fast, and efficient neural network accelerator design," *Frontiers in neuroscience*, vol. 14, p. 667, 2020.

[40] D. Elbrecht, S. R. Kulkarni, M. Parsa, J. P. Mitchell, and C. D. Schuman, "Evolving ensembles of spiking neural networks for neuromorphic systems," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2020, pp. 1989–1994.

[41] J. S. Plank, C. Rizzo, K. Shahat, G. Bruer, T. Dixon, M. Goin, G. Zhao, J. Anantharaj, C. D. Schuman, M. E. Dean, G. S. Rose, N. C. Cady, and J. Van Nostrand, "The TENNLab suite of LIDAR-based control applications for recurrent, spiking, neuromorphic systems," in *44th Annual GOMACTech Conference*, Albuquerque, March 2019. [Online]. Available: http://neuromorphic.eecs.utk.edu/raw/files/publications/2019-Plank-Gomac.pdf