Evolutionary Optimization for Neuromorphic Systems

Catherine D. Schuman, J. Parker Mitchell,
Robert M. Patton, Thomas E. Potok
schumancd@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

James S. Plank
jplank@utk.edu
Department of Electrical Engineering and Computer Science, University of Tennessee
Knoxville, Tennessee

ABSTRACT

Designing and training an appropriate spiking neural network for neuromorphic deployment remains an open challenge in neuromorphic computing. In 2016, we introduced an approach for utilizing evolutionary optimization to address this challenge called Evolutionary Optimization for Neuromorphic Systems (EONS). In this work, we present an improvement to this approach that enables rapid prototyping of new applications of spiking neural networks in neuromorphic systems. We discuss the overall EONS framework and its improvements over the previous implementation. We present several case studies of how EONS can be used, including to train spiking neural networks for classification and control tasks, to train under hardware constraints, to evolve a reservoir for a liquid state machine, and to evolve smaller networks using multi-objective optimization.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Genetic algorithms**; • **Hardware** → **Neural systems**.

KEYWORDS

spiking neural networks, neuromorphic computing, genetic algorithms

ACM Reference Format:
Catherine D. Schuman, J. Parker Mitchell, Robert M. Patton, Thomas E. Potok and James S. Plank. 2020. Evolutionary Optimization for Neuromorphic Systems. In *Neuro-inspired Computational Elements Workshop (NICE '20), March 17–20, 2020, Heidelberg, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3381755.3381758

NICE '20, March 17–20, 2020, Heidelberg, Germany

ACM ISBN 978-1-4503-7718-8/20/03…$15.00
https://doi.org/10.1145/3381755.3381758

1 INTRODUCTION

In order to evaluate a neuromorphic hardware implementation, neuromorphic computing researchers must first be able to define a spiking neural network (SNN) for their hardware. Ideally, this SNN not only runs on the hardware successfully but also performs some task, such as data classification or a control task. It remains an open question for SNNs as to how to appropriately train those networks, both in general and specifically for neuromorphic deployment.

Though there have been significant advances in traditional neural network training in recent years, there are a variety of characteristics associated with training spiking recurrent neural networks that make them difficult to train using these approaches, including the requirements for input and output data to be in the form of spikes, as well as the internal spike-based activation function. Moreover, neuromorphic computing systems can present additional difficulties for training, as they may include the presence of noise in the operation of the network, physical connectivity restrictions, and limited synaptic weight resolution.

Several potential training approaches have been presented to accommodate for one or more of these issues. Those approaches include traditional gradient descent or back-propagation-like approaches that have been adapted to deal with spike-based activations, spiking input and output and/or hardware limitations; liquid state machine approaches, which may rely on a non-spiking readout layer; and and neuroscience-inspired plasticity mechanisms such as spike-timing dependent plasticity. Though these approaches have compelling applications spaces, they also each have their own limitations that can make them difficult to use in practice.

We have previously presented an evolutionary optimization-based approach for training SNNs for neuromorphic deployment, called EONS [31]. In this work, we present an overview of the different features of EONS, as well present the updated version of EONS, which allows for more rapid application development. We discuss how the new approach is implemented, but our primary focus is on how it is utilized from a neuromorphic researcher perspective. There are several key reasons that motivate using EONS as an approach for training SNNs for neuromorphic deployment:

- EONS can be applied to multiple types of tasks (e.g., classification, control, etc.) without altering the underlying algorithm.
- EONS can train networks for different hardware implementations. In this work, we focus on one hardware implementation, but EONS has previously been used to train networks directly for memristive [5], digital [9, 23], optoelectronic [3], and biomimetic [16] implementations.
- EONS can operate within the constraints and characteristics of a hardware implementation and can be used as part of the co-design process in understanding design trade-offs.

- EONS can determine the structure of the required SNN and/or the required parameters.
- EONS can be used in combination with other training approaches, such as back-propagation or reservoir computing [27]. In Section 4.4, we illustrate how we can use EONS to train the spiking reservoir in a liquid state machine.
- EONS can be parallelized in order to get to better solutions faster. We have previously presented a scalable version of EONS [30], scaling up to nearly 300,000 cores on the Oak Ridge National Laboratory Titan supercomputer [32].

Here, we present results using EONS for a variety of practical use cases, including training for both classification and control problems, training within hardware constraints, training a reservoir for a liquid state machine, and multi-objective optimization to minimize network size.

2 BACKGROUND AND RELATED WORK

Given the success of conventional neural networks in recent years, it is unsurprising that a common approach for training SNNs has been to use some variation on back-propagation or gradient descent. Some of these approaches focus on training an SNN in general [1, 20, 26, 33, 34], while others focus on training an SNN for neuromorphic deployment by taking into account restrictions present in neuromorphic systems, such as reduced weight precision [7, 12]. Others still, focus on back-propagation tailored to specific neural network approaches and specific application domains such as sparse coding for adversarial image attacks [18].

All of these training approaches provide a compelling way to to train SNNs, and for several of the approaches, there are open-source software packages available and/or they utilize common machine learning frameworks (PyTorch, TensorFlow, Keras). However, there are also several issues associated with these approaches. Some of the approaches do not utilize all potential computational capabilities of an SNN or a neuromorphic system, e.g., by not taking advantage of axonal or synaptic delay. Some of the approaches do not take into account the corresponding hardware restrictions during the training process, which can require significant effort in order to embed the resulting SNN onto a new neuromorphic system and which may result in a loss in performance. Finally, as is the case in general for gradient descent-based neural network training, these approaches can be difficult to extend to non-classification tasks. For example, to use these approaches for control tasks, it may be required to include a reinforcement learning approach, which can add greatly to the training time and may not be successful *(may want to quote that paper from Purdue from John's dissertation).*

Reservoir computing, or for SNNs, liquid state machines (LSMs) [28], provide another approach for defining a network to be deployed onto neuromorphic hardware. In the case of reservoir computing, the SNN (which becomes the reservoir or the "liquid") is random – that is, the connectivity and parameters such as weights and delays are randomly defined for the reservoir. It has been shown that there are metrics that a reservoir must have in order to be successful (such as input separability and fading memory [11, 19, 25]). However, it is not clear how to build a reservoir from these properties. In practice, often random reservoirs are tried until one is successful. Thus, even though there is no explicit training of the reservoir, there is often significant computational power required to determine the appropriate reservoir.

Plasticity mechanisms such as spike-timing dependent plasticity (STDP) have been proposed as mechanisms for training SNNs for neuromorphic deployment as well. Though there has been some limited success in using variations of STDP [13, 36], there is still no clear evidence that they are broadly applicable and even if they are applicable to the task, they may require significant hand-tuning, e.g., to find appropriate the network structure to use.

In this work, we apply an evolutionary algorithm approach for training SNNs. Using evolutionary algorithms for training neural networks in general is sometimes called neuroevolution [14] and has been utilized as an approach for training neural networks since the late 1980's. Yao notes several major reasons for using GAs or evolutionary approaches to train neural networks rather than using gradient-descent based approaches, including that they are not reliant on gradient information (which may be unavailable or difficult to calculate), they can be applied to any neural network architecture, and they are less sensitive to initial conditions [37]. Though neuroevolution approaches can be applied simply for weight training [15], we are more interested in methods that utilize GAs to train both network topology (number of neurons, number of layers, connectivity patterns, etc.) and network parameters, such as the NEAT method [35]. Neuroevolution approaches have been applied to many different types of neural networks, including recurrent neural networks [35], deep neural networks [21, 38], and SNNs [17, 31]. In this work, we are concerned primarily with training SNNs specifically for neuromorphic implementation, which come with additional challenges due to hardware constraints. GAs have also been investigated to help overcome the challenges associated with designing SNNs for neuromorphic systems [4, 31].

3 EONS

We have previously introduced the Evolutionary Optimization for Neuromorphic Systems (EONS) approach in [31]. However, both the algorithmic approach and the corresponding software implementation have undergone significant shifts and expansions since that time. In the following subsections, we detail the key features of the latest EONS implementation and how they can be used for neuromorphic implementation. The overall workflow of EONS is shown in Figure 1. The first step for EONS is to generate a population of potential solutions. This initial population can be randomly generated, or it can be seeded with pre-existing network solutions from which to start the evolution. Once a population of networks is defined, there are three key steps: evaluation, selection, and reproduction. We detail each of those steps in the following subsections.

Though EONS was created with neuromorphic systems in mind, it is also simply operating on a network structure and can be used for any task that requires network optimization. We restrict our attention to use on neuromorphic systems here. EONS optimizes a flexible network structure that is made up of nodes and edges. This network type can have input nodes, output nodes, and hidden nodes. The only way that EONS distinguishes input/output nodes from hidden nodes is that EONS leaves the input and output nodes fixed, while hidden nodes may be added or deleted. This network structure allows for networks, nodes, and edges each to have any number

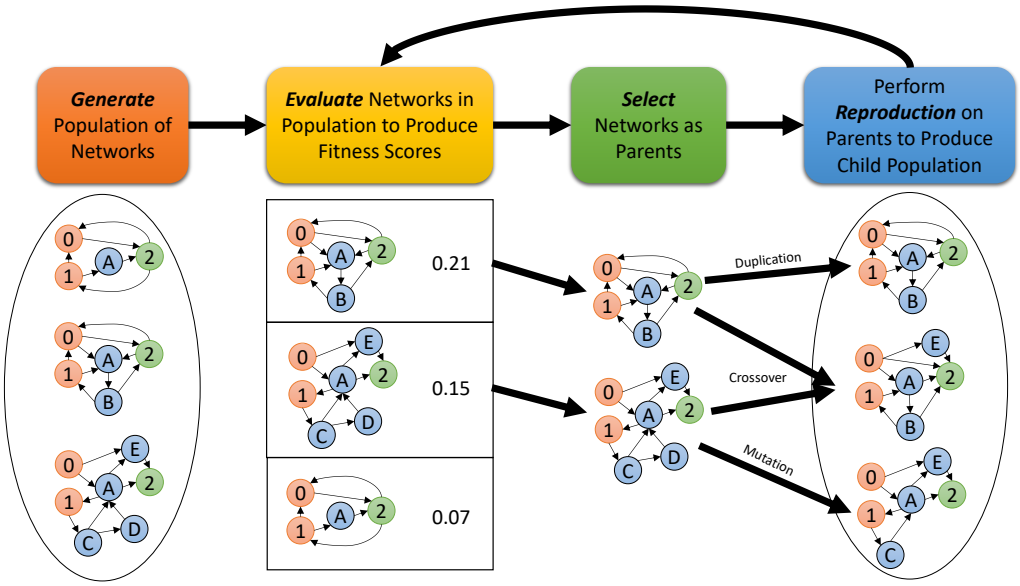

Figure 1: EONS Overview.

```
evolve = eons.EONS(eons_params)

# EONS creates a template network based on the
# neuromorphic processor, and the number of
# inputs and outputs required for the task
evolve.make_template_network(processor, nin, nout)

pop = evolve.generate_population(eons_params)
for i in range(num_epochs):
  # fitness() is a task-specific function
  fits = [fitness(processor, net) for net in pop]
  pop = evolve.do_epoch(pop, fits, eons_params)
```

Figure 2: Example of EONS code in Python

of parameters. For example, a node may have a leak parameter, a threshold parameter and/or an axonal delay, while an edge may have a weight parameter or a delay parameter. EONS is written in C++, but there are Python bindings for EONS for ease of use. An example of how EONS is used in Python is given in Figure 2.

3.1 Evaluation

In the evaluation step of the EONS process, each network in the population is assigned a fitness score. It is up to the user to define how that score is determined, and it can be as simple or complex as the user requires. The only requirement for EONS is that higher scores correspond to better performing networks. Typically, for neuromorphic systems, the network that is being evaluated is loaded onto neuromorphic hardware or into a neuromorphic simulator and evaluated on a given task (by converting inputs into spikes and interpreting spiking output as decisions).

For classification tasks, the fitness score is typically calculated as the accuracy on the data, but the user may adapt this score to the dataset as needed, for example, by weighting certain classes as more important. Unlike back-propagation-style algorithms, there is no requirement that anything associated with the task be differentiable, so the score can be adjusted depending on the goal. For control tasks, the fitness evaluation is also related to how well the task is being performed. For example, if the neuromorphic system is learning to play a game, then the fitness score could simply be the score on the game. However, depending on the types of behaviors one would like to see, the fitness score may be adjusted. For example, in playing an Atari game, one may be interested in maximizing the score and minimizing the number of times a button is "pushed." This can be accomplished through a weighted fitness score that considers both the score and the number of buttons pushed.

Finally, the score can be made more complex by adding or multiplying factors that take in account different characteristics of the network being evaluated. For example, one may subtract a factor related to a network's size in order to evolve smaller networks, as we demonstrate in Section 4.5, or one can add factors that take into account energy or power usage or resiliency of the network.

3.2 Selection

EONS provides a variety of common selection algorithms from the genetic algorithm literature, including tournament, Roulette wheel or fitness proportionate, and truncation selection [29]. The default selection approach is tournament selection. Tournament selection has two parameters: tournament size and p, the probability that the best member of the tournament is selected. EONS also allows users to implement custom selection approaches (either natively in C++ or in Python) so that more advanced selection functionalities can be implemented as needed for their tasks.

3.3 Representation and Reproduction

In the original EONS implementation described in [31], the representation and reproduction of instances in the EONS population

were specific to the neuromorphic hardware implementation. In other words, for each new neuromorphic hardware implementation, a custom network representation and custom reproduction operators (e.g., crossover and mutation) had to be created, which made it difficult to extend EONS to new hardware implementations, as operations such as crossover are non-trivial to define and implement on a network-like structure.

In the new EONS implementation, a generic network representation is used. This network implementation is made up of nodes and edges, and each node and edge, as well as the network itself, can have any number of parameters associated with it. All of these parameters can be evolved using EONS. When using EONS to evolve an SNN for neuromorphic deployment, the hardware specifies the parameters that should be optimized. For example, a neuron may have an associated threshold, leakage parameter(s), etc., and a synapse may have an associated weight, learning parameters, etc.. The hardware will have some bounds on what values are attainable for these parameters, and those bounds are specified to EONS as "properties" of the network. Then, the user converts the corresponding EONS network to and from the network specification required by the hardware for evaluation on the hardware system.

Unlike the previous implementation, reproduction operators for EONS are no longer hardware implementation-specific. Instead, these reproduction operators operate directly on the generic network implementation, but utilize the hardware-specific properties to tailor the network design to the hardware implementation. There are two major types of reproduction operations: those that take two or more parent networks to produce one or more child networks and those that operate solely on a single parent network to produce a child network. The first type includes operations such as crossover (or recombination) and merging, and the second type includes duplication and different forms of mutation. When building the next generation, we also include some percentage of random networks to inject diversity into the population (governed by the `random_factor` parameter). To guarantee that we also keep the best performing networks, we specify a parameter `num_best`, which is the number of best networks from the previous population to directly duplicate into the next generation.

Multi-parent reproduction operations that are currently implemented in EONS include crossover and merging. The previous implementation of crossover relied on spatial coordinates in the network in order to function. In the current version of crossover, no spatial coordinates are required, though each node within a network is required to have a unique ID. In crossover, two parents are selected and two children are generated. In particular, the nodes and edges of each parent are distributed into each child such that each node/edge in each parent has a corresponding node/edge in exactly one of the children. For merging, two parents are selected and one child is generated such that all of the nodes and edges in each parent appear in the single child network (essentially performing a union operation on the two networks). The user sets the probability of crossover or merge occurring in the production of children, setting the `crossover_rate` and `merge_rate`, respectively.

The other type of reproduction operations are those that produce one child from one parent. These operations currently include duplication and mutation. EONS allows for parameterized structural and parameter mutations. In particular, the user can specify the frequency of each type of mutation (e.g., adding a node, deleting a node and all of its associated edges, adding an edge, deleting an edge, and randomizing network, node, and edge parameters), as well as turn off mutations that should not occur.

Table 1: EONS Hyperparameters

Hyperparameter	Value
crossover_rate	0.5
merge_rate	0
mutation_rate	0.9
num_mutations	7
add_node_rate	0.09
delete_node_rate	0.07
add_edge_rate	0.16
delete_edge_rate	0.14
node_param_rate	0.27
edge_param_rate	0.27
selection_type	tournament
random_factor	0.1
num_best	3
population_size	500
num_generations	100

4 RESULTS

We present several case studies to illustrate the diversity of ways in which EONS can be used to produce SNNs for neuromorphic deployment,. All of these approaches train networks for the Caspian neuromorphic computing system [22]. The input/output encoding for data to/from spikes is provided by the TENNLab neuromorphic software framework [24]. We note, though, that EONS is not bound to Caspian or the TENNLab framework; it is an independent optimization module that works on graphs that are converted to Caspian networks, and executed via TENNLab framework. Unless otherwise noted, each of the tests below uses the EONS default hyperparameters, given in Table 1.

4.1 Evolving Networks for Classification

We begin with simple data classification tasks and show the performance of EONS with default parameters. We use three classification datasets from scikit-learn's toy datasets collection: iris, wine, and breast cancer. For each of these datasets, we use a 2/3 training, 1/3 testing split on the dataset, and the train/test split is consistent across all techniques evaluated. We compare the resulting EONS network results with results for a reservoir computing approach (using a randomly generated reservoir and a linear regression implementation from scikit-learn for the readout layer), the scikit-learn MLP implementation with a single hidden layer of 100 neurons, and Whetstone [33], in which we use the default parameters (except for a batch size of 8), a hidden layer of 100 neurons, and 10 times the number of outputs for 10-hot encoding. For Whetstone, we also normalize the input data. For the EONS evaluation, the fitness function is simply the accuracy on the training set. That is, in order to evaluate the fitness function, for each network in the population,

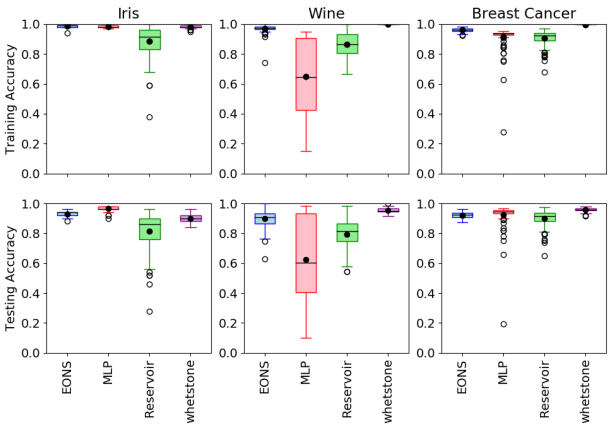

Figure 3: Comparison of Classification Results

each of the training examples are run through the SNN, the output is recorded, and the output is compared to the correct label.

Figure 3 gives the results for the performance of EONS on the testing set as compared with the other approaches. For each of the four approaches, 100 different runs were completed, showing variation in performance of each of the different algorithms for each of the datasets. It is worth noting that the MLP, reservoir, and Whetstone approaches all use fixed size networks, while EONS also optimizes the network size for each of the different networks.

EONS produces comparable results with each of the other approaches. It has more consistent performance than the random reservoir approach and the MLP approach, and is outperformed only by Whetstone. However, it is worth noting that the Whetstone results do not take into account mapping onto a specific neuromorphic platform and may have some performance degradation as part of that mapping process (which will need to accommodate for spiking input and perhaps for limited weight resolution).

To illustrate that EONS does not evolve "traditional" network structures, we show the best networks evolved for EONS for the wine dataset in Figure 4 and for the breast cancer dataset in Figure 5. We do not eliminate any input or output neurons in these figures, even though they may not be used. As can be seen in these figures, the network structures do not conform to traditional ANN structures and are relatively simple. The best wine network has only two hidden neurons and the best breast cancer network has only one hidden neuron. Both networks leverage input-to-input and output-to-input connections and include at least one cycle.

4.2 Evolving Networks for Control

To illustrate the versatility of EONS on different types of tasks, we also demonstrate the results on two control tasks from the OpenAI gym [2]: CartPole-v1 and LunarLander-v2. We leave the EONS default parameter values the same as in the classification task, but we decrease the population size from 500 to 200 and increase the number of epochs from 100 to 200. Otherwise, the only other change is to the fitness function. That is, rather converting data examples into spikes and using the neuromorphic network

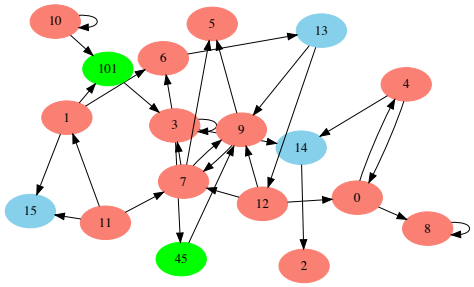

Figure 4: Best network for wine dataset. Neurons are pink (input), blue (output) and hidden (green). Labels correspond to the internal EONS representation of the network.

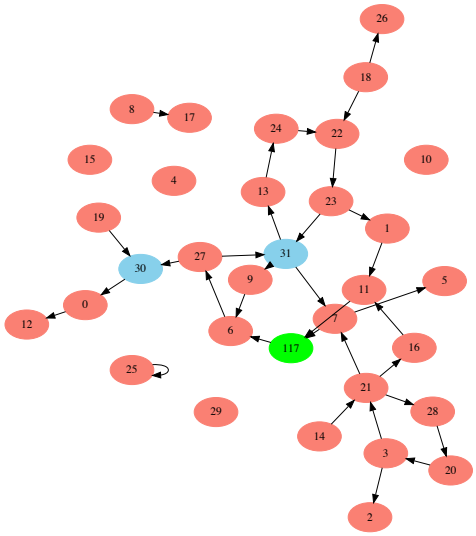

Figure 5: Best network for breast cancer dataset. Neurons are pink (input), blue (output) and hidden (green). Labels correspond to the internal EONS representation of the network.

under evaluation to produce a classification decision, we instead convert the observation from the OpenAI gym to spikes and use the neuromorphic network to come up with an action. We then update the OpenAI gym environment based on that action to get a new observation. The fitness score for these tasks is simply the score obtained from the OpenAI gym. For the fitness value in training, we use 10 random episodes. For the testing value, we use 100 episodes (seeding OpenAI gym with 0).

Figure 6 gives the results for this task. Unlike the classification task, there is significant variation in the performance of EONS for the LunarLander task. However, it is clear that EONS is able to discover a solution in some cases; thus, it is likely that EONS simply needs more generations or a larger population in order to converge consistently for these harder tasks.

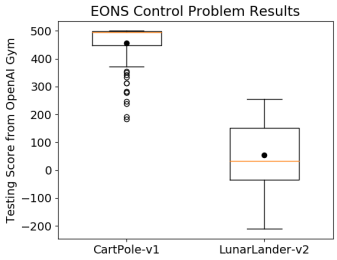

Figure 6: Results for control problems from OpenAI Gym.

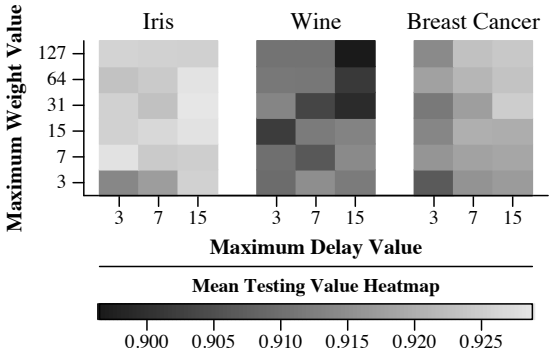

Figure 7: Mean testing results for different hardware constraints (limited weight and delay values) for three datasets.

4.3 Evolving Under Hardware Constraints

In this work, we are demonstrating different use cases of EONS for a single neuromorphic hardware platform, Caspian. However, EONS can be utilized on different neuromorphic systems with different constraints, and it will attempt to optimize within the constraints of that system. A common hardware constraint on neuromorphic systems is limited weight resolution or limited delay values on the synapses or neurons. The default implementation of Caspian has an 8-bit weight resolution and represents integer values between -127 and 127, inclusive, and a maximum synaptic delay of 15. To illustrate that EONS is capable of optimizing within different hardware constraints, we evaluate different constraint combinations: restricting the weight resolution to 3 bits, 4 bits, 5 bits, 6 bits, and 7 bits, and restricting the maximum delay values to 3 and 7.

For this case study, the only change that was made with respect to tests in Section 4.1 is that the default parameter values of Caspian were changed. Nothing about the EONS approach was updated to accommodate for differences in the hardware, including the fitness function. Because the hardware parameters have changed, EONS automatically accommodates for those changes in the networks that are randomly generated and the mutations that are allowed.

Figure 7 shows the results for the mean testing value across 100 EONS runs for different hardware constraints. As can be seen in this figure, there is relatively little difference in terms of mean performance across all of the different hardware constraints, and in fact, constraining the space can sometimes help the performance of EONS. These results demonstrate that EONS is capable of optimizing within the hardware constraints provided, but it also shows that EONS can be used as part of a hardware co-design process. In particular, because settings like maximum weight value and maximum delay value can have a substantial effect on the efficiency of the hardware implementation, using the minimum acceptable settings for those values could potentially result in significantly more efficient hardware. By performing studies using EONS, different hardware features can be evaluated for inclusion. A previous version of EONS has been used for a similar co-design study in [3] for an optoelectronic neuromorphic system.

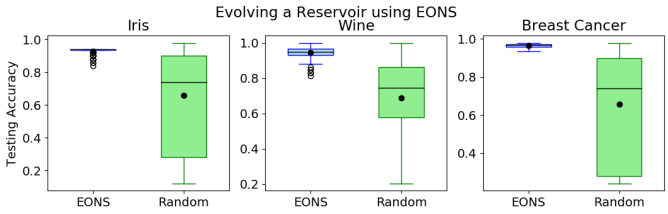

Figure 8: Testing results for evolving a reservoir using EONS as compared with randomly generated reservoirs.

4.4 Evolving a Reservoir

We have previously demonstrated that EONS can be used to intelligently generate reservoirs for liquid state machines [27]. For this work, we demonstrate that, using the same number of network evaluations, a better reservoir can be found using EONS than through random network generation. We use the same toy datasets from Section 4.1. To allow for evolving a reservoir (rather than an entire network solution), the fitness function of EONS was adapted to include the training of the readout layer (a linear regression implementation from scikit-learn). Unlike the other examples, we use a population size of 100 and evolve for 20 generations. To compare with reservoir performance, rather than using a fixed size reservoir as in Section 4.1, we sweep over different numbers of hidden neurons (10 to 200 in increments of 10) and synapses (50 to 500 in increments of 50), and we randomly generate 1000 networks for each of those sizes. This is equivalent to doing a grid search and allowing for the same number of evaluations as our EONS searches. A similar approach was taken by Reynolds [27].

Figure 8 gives the results for the random reservoir generation as compared with using EONS to evolve an appropriate reservoir. As can be see in this figure, there is much more variation in using random reservoir generation than there is for the EONS generated approach. In particular, though the starting best reservoir for EONS may have low accuracy, it can refine the reservoir to produce a well-performing solution. It is also worth noting that, on average, the best performing reservoir sizes for the randomly generated reservoirs were 10 hidden neurons and 150 synapses for iris, 20 hidden neurons and 150 synapses for wine, and 50 hidden neurons and 250 synapses for breast cancer. In contrast, the best performing EONS-evolved "reservoirs" had an average of 14.96 synapses for iris, 22.63 synapses for wine, and 21.1 synapses for breast cancer, and no hidden neurons at all for each of those datasets. It is worth noting

that the logistic regression layer alone cannot achieve the results achieved by our best reservoirs, indicating that these reservoirs, though small, are still performing valuable computation.

4.5 Multi-Objective Optimization

We have previously demonstrated that EONS is capable of multi-objective optimization through augmenting the genetic algorithm's fitness function. In particular, in [8], we show that by adding weighted values that take into account network size and network performance in the presence of synaptic variation to the typical fitness function of accuracy, we can produce smaller, more resilient networks than we can otherwise.

In this work, we focus on demonstrating how EONS can easily be adapted to evolve significantly smaller networks at little or no performance cost to the actual networks. We demonstrate this result on the same three datasets as the results presented in Section 4.1: iris, wine, and breast cancer. The original fitness, which is simply the accuracy on the training set or ($accuracy(net)$), is augmented to include a penalty for larger networks as follows:

$$fit(net) = accuracy(net) - \alpha(n_{synapses}(net) + n_{neurons}(net)) \quad (1)$$

where α is a weighting parameter, $n_{synapses}(net)$ is the number of synapses and $n_{neurons}(net)$ is the number of neurons. To demonstrate the effect that adding this penalty has on performance, we ran EONS 100 times for each of the datasets, using the same EONS random number generator seeds for the original fitness function ($fit(net) = accuracy(net)$), which corresponds to no multi-objective optimization, and the fitness function shown above, which has the penalty associated with network size, which corresponds to a multi-objective optimization approach to maximize the accuracy on the training set while minimizing network size. In this case, α was chosen experimentally ($\alpha = 10^{-7}$) and used for all three datasets. This value is a hyperparameter for this approach and will need to be customized to the given application. We ran the 100 test cases for each approach using the same random number generator seeds so that the initial populations are the same for each approach. As such, the variation in performance is entirely due to the difference in the way that the fitness function is evaluated.

Figure 9 shows the results for the two approaches. It shows box plots for the testing accuracy, the number of neurons, and the number of synapses for each data set. As can be seen in this figure, there is very little difference in the testing accuracy performance of the resulting networks produced by EONS. However, the networks for the multi-objective approach that includes a penalty for network size produces significantly smaller networks in terms of number of neurons and number of synapses. This sort of reduction in network size can result in more power efficient and area efficient deployments in neuromorphic systems. It is worth noting that this drastic difference in network size is a result of changing a single line of EONS code – the line that specifies the fitness function.

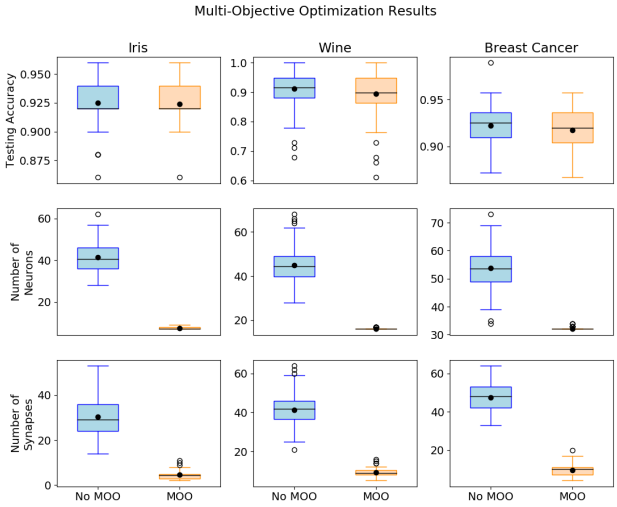

Figure 9: Multi-objective optimization results, where results without multi-objective optimization (No MOO) are shown in the blue box plots and the results with multi-objective optimization (MOO) are shown in the orange box plots. The mean is plotted as a black dot on each of the box plots.

5 DISCUSSION AND CONCLUSIONS

In this work, we demonstrate five different use cases of EONS, showcasing several of its features. The development of EONS has several different motivations and user groups in mind. One of the EONS user groups is new users of neuromorphic computing. With this group in mind, EONS is meant to be a tool that can quickly and easily get a user started with applying neuromorphic computing to their problems of interest. We demonstrate in Sections 4.1 and 4.2 that, unlike other algorithmic approaches, which have to be adapted for different types applications, EONS can be used essentially out-of-the-box for two different types of applications (classification and control, respectively). In the future, we plan to continue building out EONS capabilities to enable further ease of use, including hyper-parameter optimization features for EONS and tools that allow the user to seamlessly utilize high performance computing or cluster resources from the Python interface, expanding on our previous work that implements EONS for supercomputers [30].

Another user group that we envision benefiting from EONS is hardware developers. In Section 4.3, we demonstrate how EONS can optimize within different hardware constraints for a single neuromorphic platform. We also note that previous version of EONS have been used to train networks for a variety of hardware implementations. We note that the type of results that are presented in Section 4.3 can be used as part of a neuromorphic hardware co-design process in developing new hardware implementations. Because EONS typically produces small networks, and can be easily expanded to include additional objectives such as network size (as shown in Section 4.5), we believe that EONS networks will be of use to neuromorphic hardware developers who are working with experimental devices and who may benefit from smaller networks that are easier to build. We intend to further pursue multi-objective optimization approaches for EONS to produce more energy efficient, area constrained, and resilient SNNs for neuromorphic hardware that are more well-suited to real-world deployment.

Our third user group that we hope to target with EONS is algorithms researchers. We demonstrate in Section 4.4 that EONS can be used to optimize a reservoir for a liquid state machine. In the future, we intend to use EONS to refine networks that have been pre-trained using other algorithms, including back-propagation-like algorithms. We also intend to use EONS to optimize learning rules for unsupervised and semi-supervised learning tasks.

EONS is not without its difficulties. Depending on the problem and the size of the network, EONS can take many epochs and large population sizes to converge. One direct consequence is that it can require a very large amount of CPU cycles. Fortunately, it parallelizes easily; however, if one is limited in CPU resources, EONS may require too much of a computing burden. A more subtle consequence is that EONS configures a network for each population member in each epoch, and each of these networks may be different in structure. Therefore, if a software simulator is not available, and the communication overhead to neuromorphic hardware is high, the hardware/software boundary becomes the limiting factor in performance, significantly hampering EONS. Utilizing EONS for Intel's Loihi processor [6] has proved challenging in this respect. It was also a bottleneck for leveraging GPUs when performing EONS on the DANNA neuromorphic simulator [10].

We would like to stress that EONS exists independently from the TENNLab Neuromorphic Software Framework [24] and from any particular neuromorphic processor or application. Its most recent design, to work on general graphs rather than SNNs, has been made to increase its applicability. It exists as a C++ package, with Python bindings for interoperability with more popular machine learning frameworks like scikit-learn and OpenAI Gym.

ACKNOWLEDGMENTS

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 by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

REFERENCES

[1] S.M. Bohte, J.N. Kok, and J.A. La Poutré. 2000. SpikeProp: backpropagation for networks of spiking neurons.. In *ESANN*. 419–424.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] S. Buckley, A. N. McCaughan, J. Chiles, R. P. Mirin, S. W. Nam, J. M. Shainline, G. Bruer, J. S. Plank, and C. D. Schuman. 2018. Design of superconducting optoelectronic networks for neuromorphic computing. In *IEEE International Conference on Rebooting Computing*. Tysons, VA, 36–42.

[4] S. Cawley, F. Morgan, B. McGinley, S. Pande, L. McDaid, S. Carrillo, and J. Harkin. 2011. Hardware spiking neural network prototyping and application. *Genetic Programming and Evolvable Machines* 12, 3 (2011), 257–280.

[5] G. Chakma, N. D. Skuda, C. D. Schuman, J. S. Plank, M. E. Dean, and G. S. Rose. 2018. Energy and Area Efficiency in Neuromorphic Computing for Resource Constrained Devices. In *Proceedings of ACM Great Lake Symposium on VLSI (GLSVLSI)*. Chicago, IL, 379–383.

[6] M. Davies *et al.* 2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* 38, 1 (2018), 82–99.

[7] Peter U Diehl, Guido Zarrella, Andrew Cassidy, Bruno U Pedroni, and Emre Neftci. 2016. 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, 1–8.

[8] Mihaela Dimovska, J. Travis Johnston, Catherine D. Schuman, J. Parker Mitchell, and Thomas E. Potok. 2019. Multi-Objective Optimization for Size and Resilience of Spiking Neural Networks. In *2019 IEEE Annual Ubiquitous Computing, Electronics, and Mobile Communication Conference*. IEEE, In press.

[9] A. Disney, J. Reynolds, C. D. Schuman, A. Klibisz, A. Young, and J. S. Plank. 2016. DANNA: A Neuromorphic Software Ecosystem. *Biologically Inspired Cognitive Architectures* 9 (July 2016), 49–56.

[10] A. W. Disney, J. S. Plank, and M. Dean. 2018. Four Simulators of the DANNA Neuromorphic Computing Architecture. In *International Conference on Neuromorphic Computing Systems*. ACM, Knoxville, TN.

[11] Chao Du, Fuxi Cai, Mohammed A Zidan, Wen Ma, Seung Hwan Lee, and Wei D Lu. 2017. Reservoir computing using dynamic memristors for temporal information processing. *Nature communications* 8, 1 (2017), 2204.

[12] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. 2015. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*. 1117–1125.

[13] P. Ferré, F. Mamalet, and S.J. Thorpe. [n.d.]. Unsupervised feature learning with winner-takes-all based STDP. *Frontiers in computational neuroscience* 12 ([n. d.]).

[14] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary intelligence* 1, 1 (2008), 47–62.

[15] F. Gomez, J. Schmidhuber, and R. Miikkulainen. 2008. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research* 9, May (2008), 937–965.

[16] M. S. Hasan, C. D. Schuman, J. S. Najem, R. Weiss, N. D. Skuda, A. Belianinov, C. P. Collier, S. A. Sarles, and G. S. Rose. 2018. Biomimetic, Soft-Material Synapse for Neuromorphic Computing: From Device to Network. In *IEEE 13th Dallas Circuits and Systems Conference (DCAS)*. https://doi.org/10.1109/DCAS.2018.8620187

[17] N. Kasabov, V. Feigin, Z. Hou, Y. Chen, L. Liang, R. Krishnamurthi, M. Othman, and P. Parmar. 2014. Evolving spiking neural networks for personalised modelling, classification and prediction of spatio-temporal patterns with a case study on stroke. *Neurocomputing* 134 (2014), 269–279.

[18] E. Kim, J. Yarnall, P. Shaha, and G. T. Kenyon. 2019. A Neuromorphic Sparse Coding Defense to Adversarial Images. , 8 pages.

[19] Dhireesha Kudithipudi, Qutaiba Saleh, Cory Merkel, James Thesing, and Bryant Wysocki. 2016. Design and analysis of a neuromemristive reservoir computing architecture for biosignal processing. *Frontiers in neuroscience* 9 (2016), 502.

[20] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. 2016. Training deep spiking neural networks using backpropagation. *Frontiers in neuroscience* 10 (2016), 508.

[21] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al. 2019. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 293–312.

[22] J. Parker Mitchell, Catherine D. Schuman, Robert M. Patton, and Thomas E. Potok. 2019. Caspian: A Neuromorphic Development Platform. In *Submitted*.

[23] 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. 2019. The TENNLab Suite of LIDAR-Based Control Applications for Recurrent, Spiking, Neuromorphic Systems. In *44th Annual GOMACTech Conference*. Albuquerque.

[24] James S Plank, Catherine D Schuman, Grant Bruer, Mark E Dean, and Garrett S Rose. 2018. The TENNLab exploratory neuromorphic computing framework. *IEEE Letters of the Computer Society* 1, 2 (2018), 17–20.

[25] Anvesh Polepalli, Nicholas Soures, and Dhireesha Kudithipudi. 2016. Digital neuromorphic design of a liquid state machine for real-time processing. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 1–8.

[26] Daniel Rasmussen. 2019. NengoDL: Combining deep learning and neuromorphic modelling methods. *Neuroinformatics* (2019), 1–18.

[27] John JM Reynolds, James S Plank, and Catherine D Schuman. 2019. Intelligent Reservoir Generation for Liquid State Machines using Evolutionary Optimization. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[28] B. Schrauwen, D. Verstraeten, and J. Van Campenhout. 2007. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th european symposium on artificial neural networks*. 471–482.

[29] C.D. Schuman, G. Bruer, A.R. Young, M. Dean, and J.S. Plank. 2018. Understanding Selection And Diversity For Evolution Of Spiking Recurrent Neural Networks. In *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[30] C.D. Schuman, A. Disney, S.P. Singh, G. Bruer, J.P. Mitchell, A. Klibisz, and J.S. Plank. 2016. Parallel evolutionary optimization for neuromorphic network training. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*. IEEE Press, 36–46.

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

[32] C. D. Schuman, T. E. Potok, S. Young, R. Patton, G. Perdue, G. Chakma, A. Wyer, and G. S. Rose. 2017. Neuromorphic computing for temporal scientific data classification. In *Neuromorphic Computing Symposium (NCS '17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3183584.3183612

[33] William Severa, Craig M Vineyard, Ryan Dellana, Stephen J Verzi, and James B Aimone. 2019. Training deep neural networks for binary communication with the Whetstone method. *Nature Machine Intelligence* 1, 2 (2019), 86.

[34] S.B. Shrestha and G. Orchard. 2018. SLAYER: Spike layer error reassignment in time. In *Advances in Neural Information Processing Systems*. 1412–1421.

[35] K.O. Stanley and R. Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[36] Johannes C Thiele, Olivier Bichler, and Antoine Dupret. 2018. Event-based, timescale invariant unsupervised online deep learning with STDP. *Frontiers in computational neuroscience* 12 (2018), 46.

[37] X. Yao. 1999. Evolving artificial neural networks. *Proc. IEEE* 87, 9 (1999), 1423–1447.

[38] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. 2015. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 4.