An Evolutionary Optimization Framework for Neural Networks and Neuromorphic Architectures

Catherine D. Schuman
Computational Data Analytics
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6085
Email: schumancd@ornl.gov

James S. Plank, Adam Disney, and John Reynolds
EECS Department
University of Tennessee
Knoxville, Tennessee 37996
Email: jplank@utk.edu
{adisney1, jreyno40}@vols.utk.edu

***Abstract*—As new neural network and neuromorphic architectures are being developed, new training methods that operate within the constraints of the new architectures are required. Evolutionary optimization (EO) is a convenient training method for new architectures. In this work, we review a spiking neural network architecture and a neuromorphic architecture, and we describe an EO training framework for these architectures. We present the results of this training framework on four classification data sets and compare those results to other neural network and neuromorphic implementations. We also discuss how this EO framework may be extended to other architectures.**

I. Introduction

New neural network (NN) architectures in both software and hardware are continually being developed by the computing community. One of the major issues that must be addressed with new NN architectures is determining how that architecture will be trained or designed to perform particular tasks. Especially for hardware NN implementations (or neuromorphic implementations), it is essential that an initial training method for designing the architecture be determined early in the process in order to gauge the usefulness and applicability of the implementation. However, the designers of neuromorphic implementations often fail to define a training method specifically for their architectures and instead attempt to map existing NNs onto their architecture. These NN architectures may not map exactly to the neuromorphic architecture, which can result in a loss of performance, and the NN may not take advantage of all of the characteristics of the hardware implementation. We propose instead that it is worthwhile to develop a training method for each individual hardware implementation that will operate within the characteristics and limitations of that implementation. Using training for each specific hardware implementation allows for the full capabilities of the system to be determined, rather than testing only a subset of its characteristics that are available when mapping another architecture to the implementation.

In this work, we present an evolutionary optimization (EO) framework for determining both the parameters and structure of two NN architectures. One of the two is currently implemented in software; the other architecture is designed specifically for hardware. A very similar EO implementation is used for both architectures, and in the future directions section, we discuss how a similar method may be implemented for other neuromorphic networks (or other neural network types in general). This training method gives a convenient way to test the capabilities of a NN implementation to determine whether it is worthwhile to further explore the implementation.

In the following section, we discuss previous work using EO and other methods for a variety of NN types. We review the characteristics of a spiking NN model called Neuroscience-Inspired Dynamic Architectures (NIDA) [1]–[4] and a spiking NN model for a hardware implementation called Dynamic Adaptive Neural Network Array (DANNA) [5], [6]. We then present an EO training method for both NIDA and DANNA that determines the parameters and structure for both NN architectures. We present results for four classification benchmark tasks from the UCI Machine Learning Repository [7], and compare the results for NIDA and DANNA with other published NN implementations. Finally, we discuss some future directions for both NIDA and DANNA, as well as the ability to apply a similar EO method to other NN types.

II. Related Work

Evolutionary optimization (EO), and more specifically genetic algorithms (GA), have been used to design various types of neural networks (NNs) for several decades. Though gradient-based optimization methods such as back-propagation have been and continue to be the dominant method for training NNs, there are several compelling reasons for using other EO/GA methods for some aspects of NN training. Yao notes several major reasons for using EO methods instead of gradient-based methods in [8], but the primary advantage that we are exploiting is that they can be applied to any NN architecture.

978-1-5090-0620-5/16/$31.00 ©2016 IEEE

For feed-forward neural networks (FFNNs), EO has been used to determine the weights of synapses in lieu of another optimization method such as back-propagation [9]–[13], to determine the number of hidden layers and/or the number of neurons per hidden layer of the network while using gradient-based optimizations to determine weights [14]–[21], and to determine both structure and the weights of the network [22]–[29]. Though there has been some success in using EO or GAs for weight-based optimization in FFNNs, gradient-based optimization usually produces sufficient results. However, GAs and EO have been very successful in optimizing the structure of FFNNs; the primary alternative approach for determining the structure of the network is hand-tooling by the user. EO and GAs have also been successfully applied in developing ensembles of NNs [30].

EO and GAs have been used to train recurrent neural networks (RNNs). The weight values in RNNs can be trained using variations on back-propagation or other gradient-based methods, but because of the recurrent connections in the network, these methods often are much more computationally expensive than the corresponding methods in FFNNs, and usually require significant alterations to accommodate the change in network structure [31]. Again, EO and GAs have been used in RNNs to train weight values alone [12], [32]–[34] and weight and structure simultaneously [35]–[38].

With spiking neural networks (SNNs), there are even more complications with gradient-based methods. In addition to the inclusion of recurrent connections, SNNs also have a dynamic component (or a time component) that is not present in FFNNs or RNNs. SNNs are used in a variety of ways, and there are methods for training them using unsupervised techniques [39], [40], gradient-based techniques [41]–[45], and cuckoo-search [46]. EO and GAs have also been used to determine weights for fixed-structure SNNs [47]–[53] and to determine weights and structure for SNNs [54]–[56].

In this work, we apply EO to design networks for DANNA, a neuromorphic computing platform. Most major neuromorphic computing projects either hand-tool their networks based on biological systems or use traditional algorithms to design a network type and then map that network type to their architecture [57]–[60]. There are other neuromorphic hardware projects in the community that implement a range of neural network types and training methods. These include SNN implementations on field-programmable gate arrays (FPGAs), one that is trained using a mixture of unsupervised/supervised learning [61] and another that is trained using a GA [62]. There are also implementations that make use of memristors in neural network implementations trained using back-propagation [63], [64]. Other implementations suitable for hardware that have been tested in simulation include a FFNN model trained using back-propagation that is tolerant to faulty transistors [65] and a probabilistic SNN trained using gradient descent learning and bio-inspired learning methods [66]. A variety of other hardware implementations that are based on neural networks have also been developed, such as a self-organizing map [67] and an extreme learning machine [68].

III. NIDA

Neuroscience-Inspired Dynamic Architecture (NIDA) networks are spiking neural networks that are embedded in a bounded three-dimensional (3D) box [1]–[4]. They are composed of neurons and synapses, where neurons may be located throughout the bounded box and any two neurons may be connected by a synapse. Activity in NIDA networks is simulated using a discrete-event simulation. A simulated time step in NIDA is defined as the amount of time it takes for charge to propagate one distance unit along a synapse.

NIDA neurons are accumulate-and-fire neurons with two associated parameters, threshold and refractory period. NIDA neurons accumulate charge and fire when their threshold is reached. Upon firing, NIDA neurons enter a refractory period, during which they may still accumulate charge, but they may not fire. The refractory period in NIDA neurons is used to bound the total amount of activity in the network in any given simulated time unit. In all of our results, we have kept the refractory period fixed at one simulated time unit, but this value is programmable and can be changed for each individual neuron. The charge and threshold value of a neuron are floating-point numbers in the range $[-1, 1]$. NIDA neurons may be input neurons, output neurons, or hidden neurons. Input neurons are the only neurons that can receive charge from the environment/user. Output neurons are monitored specially by the application; their output usually determines a decision for a particular task or application.

NIDA synapses are directed connections between neurons, connecting a pre-synaptic neuron to a post-synaptic neuron. NIDA synapses have a delay and a weight. The delay is the amount of simulated time it takes for a fire from the pre-synaptic neuron to affect the charge on the post-synaptic neuron. In the current implementation of NIDA networks, the delay value of the synapse is directly related to the length of the synapse (or the Euclidean distance between the pre-synaptic and post-synaptic neurons); thus, longer synapses correspond to longer delays. NIDA synapses also have an associated weight, which is a floating-point number in the range $[-1, 1]$. The weight of the synapse affects how a fire from the pre-synaptic neuron affects the charge level of the post-synaptic neuron; that is, the weight of the synapse is added to the charge level of the post-synaptic neuron. During the simulation of a NIDA network, the weights of the synapses are adjusted by processes inspired by long-term potentiation (LTP) and long-term depression (LTD) in the brain [69]. If a particular event causes a neuron to fire, then the weight of the corresponding synapse will be increased by a small amount. If an event arrives at the post-synaptic neuron during that neuron's refractory period, then the weight of the corresponding synapse will be decreased by a small amount. Each synapse also has a "learning rate," which is how often LTP/LTD can occur on that synapse.

IV. DANNA

Dynamic Adaptive Neural Network Arrays (DANNAs) are a neuromorphic hardware implementation of NIDA networks

[5], [6]. A DANNA is an array of neuromorphic elements, where each element is programmable as either a neuron or a synapse. DANNAs have an associated clock on which all of the behaviors of the network occur. All of the parameter values in DANNA are discrete values rather than continuous values. Each element in the array can connect to up to 16 of its near-neighbors (its eight immediate neighbors, and eight additional neighbors one row or column away). The restricted connectivity in DANNA is the primary difference in the types of networks that can be built between NIDA and DANNA.

DANNA neurons have threshold values, which are integers in the range $[-128, 127]$. Unlike NIDA neurons, DANNA neurons do not have a programmable refractory period. Instead, DANNA neurons have a fixed refractory period in that they may only fire once during a particular clock cycle. When a neuron fires, it broadcasts the event to its 16 neighbors, each of which receives the event only if it is actively listening for events from that neuron.

DANNA synapses have two programmable parameters: delay and weight. Unlike NIDA synapses, the delay value is not tied to a Euclidean distance; instead it is programmed to be an integer in the range $[0, 127]$, where the units of the delay are network clock cycles. DANNA synapses also have programmable weight values, which are integers in the range $[-128, 127]$. DANNA synapses have an associated LTP/LTD process, which operates similarly to NIDA. Like neurons, synapses broadcast their firing events to their 16 neighbors, each of which receives the event only if it is actively listening for it. For that reason, synapses may be treated as one-to-many connections, and synapses can chain together without intervening neurons. These allow for more complex network structures. Synapses specify one output direction for LTP/LTD.

There are significant enough differences in the operation of NIDA and DANNA that it does not make sense to build a NIDA network and convert the NIDA network to a DANNA network. Instead, we develop a DANNA-specific evolutionary optimization in order to produce DANNA networks. An initial implementation of DANNA on field programmable gate arrays (FPGAs) has been completed. A VLSI implementation of DANNA is also in progress.

V. Training: Evolutionary Optimization

We use evolutionary optimization (EO) methods to design both NIDA and DANNA networks. There are many reasons that we have chosen to use EO as the design method for NIDA and DANNA, many of which are discussed in [4]. In general, using EO allows for both discrete and continuous optimization to happen simultaneously. For neural networks, the inclusion of a time component in the operation of the network is known to make gradient optimization more difficult. For NIDA and DANNA, because variable delays are used throughout the network, adapting a gradient-based optimization method to optimize synapse weight values would be extremely difficult. EO also allows for a variety of applications to be implemented without changing the characteristics of the underlying method; in particular, the only major component of the EO that changes as the application changes is the fitness function. It is relevant to note that the fitness function is a major component of the EO and its performance drastically affects the performance of the EO. However, most of the EO method itself stays static across all application types.

There are a few characteristics of the type of EO method that we use that are especially useful when employing them to design neuromorphic systems. First, no prior knowledge about either the parameters or the structure of the networks is required. Without an optimization method that determines the structure, a user would need to determine a structure *a priori*. Thus, even though the determining both structure and parameters complicates the design method, it ensures that there is no user bias in the selection of the structure. Additionally, different neuromorphic architectures employ different structure types and various types of neuron and synapse models. Small adaptations to EO methods are required to accommodate for different architectures and neuron/synapse models, but the EO methods as a whole may still be applied. Moreover, as many or as few restrictions on the optimization may be employed; since neuromorphic hardware architectures often have more restrictions than other neural network models, this is an important feature of EO. We describe the EO methods for NIDA and DANNA and how they differ. Very similar EO methods may be employed for other neuromorphic architectures as a viable initial implementation for general neuromorphic architecture optimization in order to determine the characteristics and capabilities of those architectures. Of course, it is likely worthwhile to explore architecture-specific optimizations that may be more computationally efficient than EO, but EO methods can provide a tremendous amount of information about an architecture's characteristics that may be useful in determining more specific optimizations.

In our EO method, there are two task-specific functions that the user must implement for each individual application. The first task-specific method returns a randomly initialized network. This method is used at the beginning of the EO to create a population of networks, and it is also used over the course of the EO method to randomly generate networks to add to the population. At minimum, this function specifies the inputs and outputs of a network that are consistent across all networks in the population; thus, every network in the population of networks will have the same interface to the environment. This initialization method can then randomly initialize hidden neurons and synapses in the network, or it can include any knowledge about the type of structure or parameters that the user may know about task *a priori*. The other task-specific function is the fitness function, which takes a network as input and returns a numerical score for that network for the particular task.

The overall framework of the EO method is consistent for both NIDA and DANNA. We start by creating an initial parent population of networks using the task-specific network initialization function. Then, each of the networks in the population is evaluated using the fitness function. Once a score for each network has been determined, a selection

method can be used to preferentially select better performing networks to serve as parents. We utilize tournament selection, but other selection algorithms such as roulette wheel selection or stochastic universal selection, may be used. We repeatedly select pairs of networks from the parent population using tournament selection. With each pair of parents, we probabilistically apply crossover and mutation based on the crossover and mutation rates, respectively. If neither crossover nor mutation are applied, then the parent networks are simply duplicated to produce children. The crossover and mutation operations are specific to NIDA and DANNA and will be discussed in more detail in Sections V-A and V-B, respectively, but very similar reproduction operators may be developed for other architectures. The resulting children networks are saved in a child population, and the selection and reproduction processes are repeated until the child population has the same number of networks as the parent population. At this point, the generation or epoch comes to an end; the child population replaces the parent population, and selection and reproduction is repeated. This process continues until a predetermined fitness value or epoch/generation number is reached. The primary differences between the EO for NIDA and DANNA are in the way the networks are represented in the population and how the reproduction operators operate on those representations. The details for these implementations are discussed in the following sections.

A. NIDA Representation and Reproduction

NIDA networks are stored as a graph representation, where the nodes are the neurons and the edges are the synapses. Each neuron stores its location in the 3D bounded box, along with all of its associated parameters and all of its incoming and outgoing synapses. Each synapse stores its own associated weight value and delay value.

The crossover operation operates directly on the graph representation of the network. Crossover takes two parent networks and produces two child networks. For each crossover that occurs, a plane through the bounded box is constructed by randomly selecting two neurons in one of the parent networks. One of those neurons is selected to be on the plane and the vector between the two neurons is used as the normal vector to the plane. This plane in the bounded box will split each of the parent networks into two parts, a part above the plane and a part below the plane. One child is constructed by taking the neurons and synapses from the part above the plane from parent 1 and the neurons and synapses from the part below the plane in parent 2, and the other child is constructed from the part above the plane in parent 2 and the part below the plane in parent 1. For synapses that span the plane, the synapse is placed in the child network that contains its pre-synaptic neuron and the neuron in the child network that is closest to the location of the synapse's original post-synaptic neuron is selected to be the post-synaptic neuron of the new synapse in the child.

Mutations in the NIDA networks make both parameter and structural changes. The mutation operation takes one network and mutates that network in place. The three parameter mutations are: changing the sign of the weight value of a randomly selected (RS) synapse, changing the weight value of a RS synapse to a RS value, and changing the threshold value of a RS neuron to a RS value. The structural mutations are: adding a synapse between two RS neurons that are not already connected by a synapse, deleting an existing RS synapse, adding a neuron at a RS location in the 3D bounded box and two associated synapses between the newly created neuron and two RS existing neurons in the network, and deleting a RS neuron and all of its associated synapses.

B. DANNA Representation and Reproduction

DANNA networks are stored as two-dimensional arrays, where each element in the array stores its identity (neuron/synapse) and the appropriate parameters. Connections between elements are stored as "enabled inputs" in both neurons and synpases. Each synapse also stores its output for LTP/LTD. The parameters that are programmable beyond connectivity in the network are the thresholds of neurons and the weights and delays of synapses.

The crossover operation for DANNA operates directly on the array representation of the network. For each crossover, the arrays are split either on a row or a column; both options are equally likely. A randomly selected row/column number is selected, avoiding the first and last row/column, which we will call the split line. Similar to what occurs in NIDA, both of the parents are then split into two parts (part A and part B) along this split line. All of the neurons from part A of parent 1 are added to the first child, and all of neurons from part B of parent 2 are also added to the first child. The synapses in part A of parent 1 and part B of parent 2 that are not affected by the split line are also added to the first child. For synapses that are affected by the split, there are special rules for adding that synapse to the child. If both of the source and destination elements are on the other side of the split for that synapse, then the synapse is omitted. If only one of the source or destination is on the other side of the split, the crossover operation looks for another neuron in the child that is in the same location or within the connection ring of the synapse. If no neuron exists, the synapse is omitted. Otherwise, the synapse is connected to the new neuron. The second child is formed similarly by combining part B of parent 1 and part A of parent 2.

Mutations in DANNA also make both parameter and structural changes. The three parameter mutations are changing the threshold of 25 percent of the neurons in the array to RS values, changing the weight of 25 percent of the synapses in the array to RS values, and changing the delay of 25 percent of the synapses in the array to RS values. The four structural mutations for DANNA are adding a path between two RS neurons, where the path alternates neurons and synapses; adding a cycle between two RS neurons by creating two paths of alternating neurons and synapses; deleting a RS neuron from the array and all of its associated synapses; and deleting a RS synapse from the array.

TABLE I
DATA SET CHARACTERISTICS

Data Set	Instances	Inputs	Output Classes
Iris	150	4	3
Wisconsin Breast Cancer	699	10	2
Pima Indian Diabetes	768	8	2
Wine	178	12	3

VI. RESULTS

We trained NIDA and DANNA with the EO methods described above for four classification tasks from the UCI Machine Learning Repository [7] that have been examined widely in the literature: Iris, Wisconsin Breast Cancer, Pima Indian Diabetes, and Wine. Table I gives the characteristics of these data sets. In Section VI-A, we give results for NIDA and DANNA on these tasks for varying parameter values to show the effect of parameters on the final result. In Section VI-B, we discuss various implementations for a fitness function and show how the different implementations affect the results for NIDA and DANNA. In Section VI-C, we compare our best results with results reported in the literature for each of the data sets using a variety of neural network methods, both in software and hardware.

For these data sets, we encode the input values as integers between 0 and 10, inclusive, by scaling the raw attributes as needed. Each input attribute has an associated input neuron in the NIDA or DANNA network. If the scaled input value is x, then x pulses are applied to the associated input neuron in the network. There is an interval of 5 time steps for NIDA or clock cycles for DANNA between pulses. Each network has a single output neuron; the number of fires for the output neuron indicates the selected output class. Our DANNA EO implementation and simulator have been recently developed and have not been optimized, so their performance is slower than that of NIDA's simulator and EO. As a result, we have fewer results for DANNA.

A. EO Parameter Effects

We explored the effect of the starting numbers of neurons/synapses for NIDA and neurons/paths for DANNA, and the crossover and mutation rates. Fig. 1 shows the variation when changing the initial number of neurons and synapse/synapse paths. Although the averages vary across the different initial configurations, the behavior is fairly consistent. Both NIDA and DANNA allow for structural modification and the numbers of neurons and synapses change over the course of evolution. These results are consistent with what we would expect, which is that, in general, the EO will increase or decrease the number of neurons and synapses as needed for the application.

Fig. 2 shows how the crossover and mutation rates affect performance for both NIDA and DANNA. We explored less values for crossover and mutation for DANNA because of time constraints imposed by DANNA's simulator and EO. We found that for both NIDA and DANNA, the crossover rates had much less impact on the overall performance of the EO than the mutation rates. In general, higher mutation rates also achieved better performance. For NIDA, higher crossover rates also tended to lead to improved performance. For DANNA, however, there was not one clear trend for how the crossover rate affected performance. Overall, we recommend exploring crossover and mutation rates when applying the EO framework to a new architecture or application. It may also be worthwhile to explore adapting mutation/crossover rates as part of the EO framework.

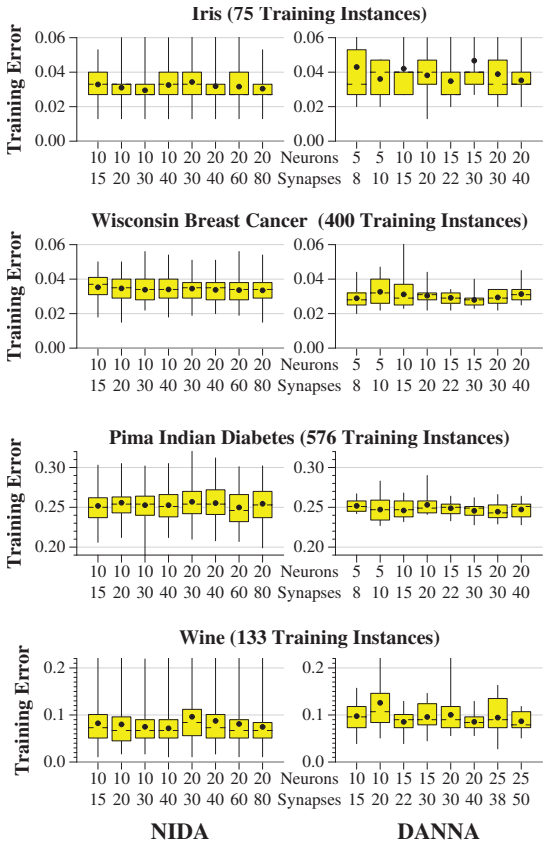

Fig. 1. Tukey plots showing how the performance varies for 100 different runs of NIDA and for 20 different runs of DANNA on each of the four data sets. The top number on the x-axis labels is the starting number of neurons and the bottom number is the starting number of synapse paths (DANNA) or synapses (NIDA).

B. Fitness Function Implementations

The implementation of the fitness function can have a dramatic effect on the performance of the EO framework. We explored several implementations of the fitness function for classification tasks. For every fitness implementation, we look at the number of fires for the output neuron in some time window of the simulation (for example, during the last 50 time steps of simulation). For the data sets with two possible classes (Breast Cancer and Diabetes), if the output neuron fires in the time window the network classifies the instance as type A and if the output neuron does not fire, it is classified as type B. For the data sets with three possible classes (Iris and

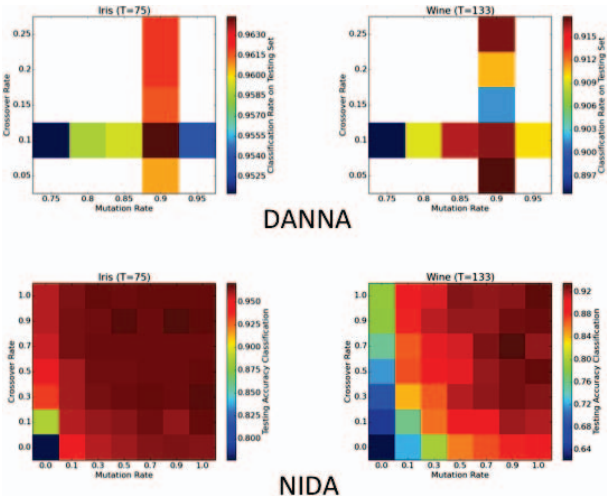

Fig. 2. Heatmaps showing how the performance varies for 20 different runs of DANNA and for 100 different runs of NIDA for crossover and mutation rates for the Iris and Wine data sets. White boxes in the DANNA heatmap indicated untested combinations of crossover and mutation rates.

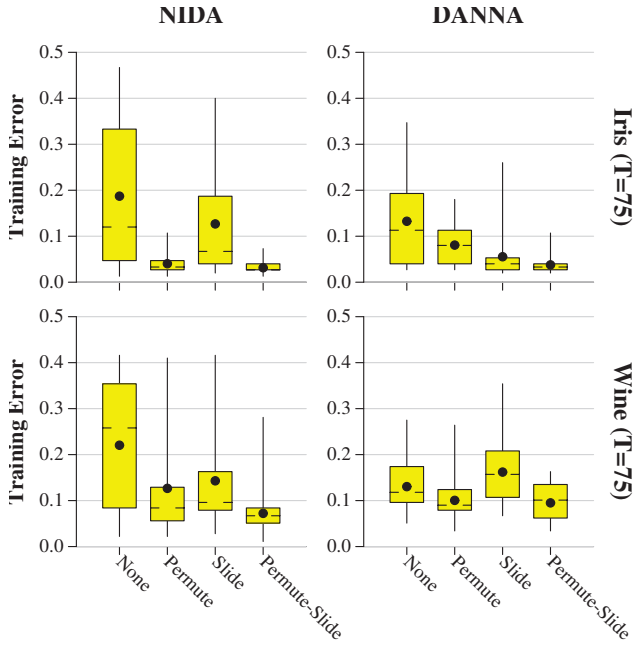

Fig. 3. Tukey plots showing how the performance varies for 100 different runs of NIDA and for 20 different runs of DANNA for various fitness implementations for the Iris and Wine data sets.

Wine), type A corresponds to no fires in the time window, type B corresponds to between 1 and 9, inclusive, fires in the time window, and type C is 10 or more fires.

In a variation on the original implementation called "slide," we vary the time window in which we measure output fires by changing both the duration of the window and the starting and stopping time of the window. The returned fitness value is the best fitness value over all examined time windows. In a variation of the original implementation called "permute," we consider the number of fires in the specified window and permute the classification labels for each type. For example, for the Breast Cancer set, we evaluate the fitness when assuming 0 fires corresponds to benign and one or more fires corresponds to malignant. We also evaluate the fitness assuming 0 fires corresponds to malignant and one or more fires corresponds to benign. Similarly, for the three-class data sets there are six possible permutations of output class and number of fires. Again, all possible permutations are examined and the best fitness value across all permutations is the fitness value used in the EO. We also combine these implementations for the "slide-permute" variation.

Fig. 3 shows the results for varying the fitness implementation. One advantage of these fitness implementations is that we are only required to simulate the NIDA or DANNA network once. By storing all of the fires of the output neuron during the simulation, we can complete all of the computations for slide or permute without repeating the simulation in order to determine the best result. These types of fitness functions can also result in two networks with the same fitness evaluation that look and behave entirely differently. We found that this improved the overall performance of the EO, and we speculate that this occurs because it encourages diversity in the population, even for a set of high performing networks. In general, the permute variation improved the performance more than the slide variation, but the combination slide-permute variation gave the best results.

C. Comparisons

We divided the comparison results into 11 categories:

1) BP/LM: Artificial neural networks (ANNs) trained with back-propagation (BP) or Levenberg-Marquardt (LM).
2) EA: ANNs trained with evolutionary algorithms (EA).
3) Hybrid EA/Local: ANNs trained with a combination of EAs and local optimization methods such as BP or LM.
4) Ensemble NN: Ensembles of ANNs.
5) Ensemble NN EA: Ensembles of ANNs that are trained in part with EAs.
6) Spiking: Spiking neural networks (SNNs) trained with biologically-inspired methods.
7) Spiking BP: SNNs trained with a form of BP.
8) Spiking EA: SNNs trained with EAs (including NIDA).
9) Hardware: ANNs and SNNs implemented in hardware (including DANNA).
10) Memristor BP: Neural networks implemented with memristors that are trained using forms of BP.
11) Other NN: Other types of neural networks or neural network training methods.

Each of the results by other methods were reported in the cited papers. When multiple papers cite the same method but used different numbers of training instances, both sets of results are reported. Some of the reported results are given as an average, others as the best result, and still others do not report whether the average or best result is given. The best results for NIDA and DANNA are given. Tables II through V

give the results for each of the four classification tasks. In each of these tables, the results are grouped by category or type; within those groups, the methods are sorted by classification error on the testing set. The tables also give the number of training instances used; if the number of training instances used for that method were not clear, the number column is left blank. For NIDA and DANNA, different numbers of training instances were used for three of the four data sets so that their results can be compared with other techniques utilizing the same number of training instances. The results for NIDA and DANNA are highlighted in each of the tables.

Table II gives the results for the Iris data set. Overall, the methods programmed using EO/EAs perform well, especially for SNNs. The results for DANNA are also the best results for neuromorphic hardware. The best results overall were achieved by a SNN trained using EO (FeaSTAP [53]), but both NIDA and DANNA outperform all of the remaining methods, indicating that EO for training of both SNNs and neuromorphic hardware is a good approach for simple tasks.

Table III gives the results for the Wisconsin Breast Cancer data set. For this task, there was good performance across all of the different categories of methods, with the best performance achieved by a traditional neural network trained with a hybrid of GA and the LM method (GALM [15]). DANNA, once again, achieves very good performance with respect to other hardware implementations, but it is outperformed by memristor implementations that are trained with back-propagation. For this data set in particular, gradient-based methods seem to be sufficient for solving the task. However, the methods that utilize EO techniques achieve comparable performance with the gradient-based techniques.

Table IV gives the results for the Pima Indian Diabetes data set. The best performance for this task was achieved by ensemble NN methods that utilize EO techniques. DANNA achieves slightly better performance than the other hardware implementation, and NIDA achieves better performance than all of the other SNN methods and NNs trained with BP/LM techniques or EO techniques. We did not explore ensemble methods for NIDA or DANNA, but we can explore this in the future to improve their overall classification performance.

Table V gives the results for the Wine data set. The techniques that utilize EO once again achieve the best performance on this task, with NIDA and DANNA achieving the best performance overall. The only comparable methods are traditional neural networks trained with EO.

Overall, NIDA and DANNA perform well on all four classification tasks with respect to reported results in the literature. DANNA in particular performs very well amongst other hardware implementations, demonstrating that utilizing EO as a training method for neuromorphic architectures is a valid approach for exploring the capabilities of those architectures.

TABLE II
IRIS RESULTS

Type	Method	Training Instance	Testing Error
BP/LM	BP [49]		2.66
	BPrule [53]		3.8
	MBP [49]		4
	SMBP [49]		4
	MatlabLM [61]	120	4.1
	MatlabBP [61]	120	4.2
	BP [52]	75	15
	BP [51]		16.17
	LM [52]		19.8
	LM [51]		23.37
EA	Vazquez [52]	75	4.4
	Strat-SSP-ep [22]		4.68
	ANNT [53]		4.7
	FeaSANNT [53]		5.3
	Ep [22]	111	6.17
	SSP200 [22]	111	6.5
	SSP100 [22]	111	7.17
	Rank [22]	111	7.28
	Strat-SSP-roul [22]		7.35
	Strat-SSP-rank [22]		7.46
	SSP5 [22]	111	8.22
	Roul [22]	111	8.43
Hardware	DANNA	75	0.7
	DANNA	120	0.7
	DANNA	111	1.3
	Hardware PSNN [66]		3.3
	Streamcluster+SOM [67]		6.67
	Streamcluster [67]		11.33
	SNN Bako [61]	120	16.6
Hybrid EA/Local	GaX [14]		2.84
	SaX [14]		2.95
	StdX [14]		3.24
Memristor BP	Circuit [64]		2.8
	Algorithm [64]		2.9
	Circuit with Noise [64]		4.7
Spiking	SRM SNN (1-D) [50]		2.7
	SRM SNN (sparse) [50]		4
	SWAT [66]		4.7
	CS [46]	120	5.33
Spiking BP	Theta Neuron BP [45]	100	2
	WeightLimit [66]		3.4
	SpikeProp [50]		3.8
	SpikeProp [45]	100	3.9
	BP A [45]	100	4.5
	SpikeProp [61]	120	4.7
	BP B [45]	100	10
Spiking EA	FeaSTAP [53]		0
	NIDA	75	0.7
	NIDA	111	0.7
	NIDA	120	0.7
	DE [46]	120	1.67
	Dynamic synapse SNN [50]		2.7
	SNN with Parde [49]		2.7
	Vazquez [51]		5.42

VII. DISCUSSION AND FUTURE WORK

The purpose of this work is to demonstrate that the use of EO is a viable method for designing neural network (NN) architectures in general and for neuromorphic architectures in particular. Although NIDA is currently implemented only in software, its implementation and embedding in a three-dimensional space lends it to hardware implementations with analog components, such as memristors. The results presented in Section VI-B also indicate that EO has been a good method for other NN architectures and neuromorphic hardware implementations beyond NIDA and DANNA.

TABLE III
WISCONSIN BREAST CANCER RESULTS

Type	Method	Training Instance	Testing Error
BP/LM	BP [15]	525	0.91
	MatlabBP [50]	341	1.5
	MatlabLM [50]	341	2
	LM [15]	525	3.17
	MatlabLM [61]	444	3.5
	MatlabBP [61]	444	3.8
EA	Fogel [13]	400	1.95
	Roul [22]		3.05
	SSP100 [22]		3.14
	Strat-SSP-rank [22]		3.22
	SSP200 [22]		3.2
	Strat-SSP-rank [22]		3.22
	Strat-SSP-roul [22]		3.23
	Ep [22]		3.28
	Rank [22]		3.33
	SSP5 [22]		3.33
	GA [15]	525	16.76
Ensemble NN	CNNE-15,2 [30]		1.2
	CNNE-10,2 [30]		1.3
	CNNE-10,4 [30]		1.5
Hardware	DANNA	444	1.9
	DANNA	525	1.9
	DANNA	400	2
	Hardware PSNN [66]		2.8
	Embrace-FPGA [62]	444	3.2
	SNN Bako [61]	444	10.5
Hybrid EA/Local	GALM [15]	525	0.02
	GaX [14]		0.46
	GABP [15]	525	1.43
	EPNet [17]	525	1.719
	MPANN [20]	400	1.9
	SaX [14]		2.48
	StdX [14]		6.19
Memristor BP	Algorithm [64]		1.3
	Circuit [64]		1.5
	Circuit with Noise [64]		1.5
	Hasan [63]	499	7.5
Spiking	SWAT [66]		2.1
	SRM-based SNN [50]	341	2.8
Spiking BP	BP B [45]	599	1
	Theta Neuron BP [45]	599	1
	Weight Limit [66]		2.1
	SpikeProp [50]	341	2.2
	SpikeProp [61]	444	2.7
	SpikeProp [45]	599	3
	BP A [45]	599	3.7
Spiking EA	Jin [54]	525	1.2
	NIDA	525	1.4
	NIDA	400	1.5
	NIDA	444	1.5
	Dynamic-synapse SNN [50]	341	2.7

TABLE IV
PIMA INDIAN DIABETES RESULTS

Type	Method	Training Instance	Testing Error
BP/LM	BP [15]	576	21.76
	SMBP [49]		21.88
	MBP [49]		23.43
	LM [15]	576	25.77
	BP [49]		36.45
EA	GA [15]	576	36.46
Ensemble NN	CNNE-15,2 [30]		19.6
	CNNE-10,2 [30]		19.8
	CNNE-10,4 [30]		20.1
Ensemble NN EA	DIVACE-majority [30]		12.5
	DIVACE-average [30]		14.1
	EENCL-WTA [30]		15.6
	DIVACE-WTA [30]		15.7
	EPNet-rank [30]		17.2
	EPNet-error-rate-majority [30]		17.2
	EENCL-average [30]		17.2
	EENCL-majority [30]		17.2
	EPNet-OptimalSubset [30]		18.2
	EPNet-RLS [30]		19.3
	EPNet-best [30]		19.8
Hardware	DANNA	576	22
	ELM [68]	576	22.3
Hybrid EA/Local	GaX [14]		18.4
	SaX [14]		19.39
	StdX [14]		20.82
	EpNet [17]	576	21.875
	GALM [15]	576	28.29
	GABP [15]	576	36.46
Spiking	CS [46]	614	74.77
Spiking EA	NIDA	576	19
	DE [46]	614	26.29
	SNN with PARDE [49]		37.69

We are encouraged by the preliminary results for both NIDA and DANNA on the classification tasks presented in this work. We intend to continue improving the EO method by exploring parallel implementations for use on high-performance computers. We are also exploring parallelizations of the DANNA simulator that will improve its performance, and we intend to explore incorporating the DANNA hardware (currently on FPGA) directly in the EO as well.

We describe how the EO differs for the NIDA and DANNA architectures in the way the networks are represented and how new networks are created in the EO using reproduction operators. Custom EO implementations for other NN architectures and neuromorphic hardware implementations can be created by utilizing similar representations and reproduction operators. By using EO methods as a preliminary training technique for neuromorphic architectures, especially an EO method that evolves structure along with parameters, neuromorphic architects may explore their architectures' capabilities with a relatively simple EO implementation. We are currently developing an arbitrary network EO framework that will be applicable to more general neuromorphic networks.

VIII. CONCLUSION

In this work, we proposed an evolutionary optimization (EO) framework for designing neural network architectures, both for software and hardware. We reviewed a spiking neural network model called NIDA and a neuromorphic hardware implementation called DANNA and discussed how the EO framework applies to both. We showed how parameters and the fitness function implementation affected the performance of the EO framework for both NIDA and DANNA; in particular, we saw that structural parameters were not as important as reproduction parameters, and that fitness functions that en-

TABLE V
WINE RESULTS

Type	Method	Training Instance	Testing Error
BP/LM	LM [51]		13.84
	LM [52]	89	35.67
	BP [52]	89	40.45
	BP [51]		63.63
EA	MPENSGA2E [21]	125	2.64
	MPENSGA2S [21]	125	2.64
	Strat-SSP-ep [22]		4.68
	Strat-SSP-rank [22]		4.91
	Ep [22]		5.12
	SSP100 [22]		6.14
	SSP200 [22]		6.18
	Rank [22]		6.23
	SSP5 [22]		7.49
	Roul [22]		8.46
	Strat-SSP-roul [22]		8.96
	Vazquez [52]	89	18.99
Hardware	DANNA	133	2.2
	DANNA	142	2.8
	Streamcluster+SOM [67]		6.74
	Streamcluster [67]		33.15
Other NN	MO-RSM [21]	125	0.69
	MO-RBFNN [21]	125	1.79
	WD [21]	125	1.95
	SO-RSM [21]	125	2.33
	MIN-MSE [21]	125	4.53
	SNRF [21]	125	7.8
Spiking	CS [46]	142	9.22
Spiking EA	NIDA	142	0.6
	NIDA	133	1.1
	DE [46]	142	12.56
	Vazquez [51]		22.2

courage diversity in the population were favored over simpler functions. We also compared the performance of our EO framework with NIDA and DANNA to other neural network and neuromorphic methods. Overall, we found that our framework performed very well, and that EO methods in general tended to outperform other methods. We are highly encouraged by these results and plan to continue our development of both NIDA and DANNA, as well as exploring the application of the EO framework to other neural network and graph types.

ACKNOWLEDGMENT

Prepared in part by Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, Tennessee 37831-6285; managed by UT- Battelle, LLC, for the U.S. Department of Energy under contract DE-AC05- 00OR22725. This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-16-1-0065. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements , either expressed or implied, of the Air Force Research Laboratory or the U.S. Government. The authors thank Michael Peek and Chris Welsh for their generosity with equipment for running simulations.

REFERENCES

[1] C. D. Schuman and J. D. Birdwell, "Variable structure dynamic artificial neural networks," *Biologically Inspired Cognitive Architectures*, vol. 6, pp. 126–130, 2013.

[2] ——, "Dynamic artificial neural networks with affective systems," *PloS one*, vol. 8, no. 11, p. e80455, 2013.

[3] C. D. Schuman, J. D. Birdwell, and M. E. Dean, "Spatiotemporal classification using neuroscience-inspired dynamic architectures," *Procedia Computer Science*, vol. 41, pp. 89–97, 2014.

[4] C. D. Schuman, J. D. Birdwell, and M. Dean, "Neuroscience-inspired dynamic architectures," in *Biomedical Science and Engineering Center Conference (BSEC), 2014 Annual Oak Ridge National Laboratory*. IEEE, 2014, pp. 1–4.

[5] M. E. Dean, C. D. Schuman, and J. D. Birdwell, "Dynamic adaptive neural network array," in *Unconventional Computation and Natural Computation*. Springer, 2014, pp. 129–141.

[6] C. D. Schuman, A. Disney, and J. Reynolds, "Dynamic adaptive neural network arrays: a neuromorphic architecture," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015, p. 3.

[7] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[8] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[9] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms." in *IJCAI*, vol. 89, 1989, pp. 762–767.

[10] D. B. Fogel, L. J. Fogel, and V. Porto, "Evolving neural networks," *Biological cybernetics*, vol. 63, no. 6, pp. 487–493, 1990.

[11] N. Saravanan and D. B. Fogel, "Evolving neural control systems," *IEEE Intelligent Systems*, no. 3, pp. 23–27, 1995.

[12] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *The Journal of Machine Learning Research*, vol. 9, pp. 937–965, 2008.

[13] D. B. Fogel, E. C. Wasson, and E. M. Boughton, "Evolving neural networks for detecting breast cancer," *Cancer letters*, vol. 96, no. 1, pp. 49–53, 1995.

[14] N. García-Pedrajas, D. Ortiz-Boyer, and C. Hervás-Martínez, "An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization," *Neural Networks*, vol. 19, no. 4, pp. 514–528, 2006.

[15] E. Alba and J. F. Chicano, "Training neural networks with ga hybrid algorithms," in *Genetic and Evolutionary Computation–GECCO 2004*. Springer, 2004, pp. 852–863.

[16] M. M. Islam and X. Yao, "Evolving artificial neural network ensembles," in *Computational intelligence: a compendium*. Springer, 2008, pp. 851–880.

[17] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *Neural Networks, IEEE Transactions on*, vol. 8, no. 3, pp. 694–713, 1997.

[18] Y. Liu and X. Yao, "A population-based learning algorithm which learns both architectures and weights of neural networks," *Chinese Journal of Advanced Software Research*, vol. 3, pp. 54–65, 1996.

[19] D. White and P. Ligomenides, "Gannet: A genetic algorithm for optimizing topology and weights in neural network design," in *New Trends in Neural Computation*. Springer, 1993, pp. 322–327.

[20] H. A. Abbass, "An evolutionary artificial neural networks approach for breast cancer diagnosis," *Artificial Intelligence in Medicine*, vol. 25, no. 3, pp. 265–281, 2002.

[21] D. Yeung, J.-C. Li, W. Ng, and P. Chan, "Mlpnn training via a multiobjective optimization of training error and stochastic sensitivity," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[22] P. P. Palmes, T. Hayasaka, and S. Usui, "Mutation-based genetic neural network," *Neural Networks, IEEE Transactions on*, vol. 16, no. 3, pp. 587–600, 2005.

[23] J. E. Fieldsend and S. Singh, "Pareto evolutionary neural networks," *Neural Networks, IEEE Transactions on*, vol. 16, no. 2, pp. 338–354, 2005.

[24] F. H. Leung, H.-K. Lam, S.-H. Ling, and P. K. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *Neural Networks, IEEE Transactions on*, vol. 14, no. 1, pp. 79–88, 2003.

[25] J. C. F. Pujol and R. Poli, "Evolving the topology and the weights of neural networks using a dual representation," *Applied Intelligence*, vol. 8, no. 1, pp. 73–84, 1998.

[26] D. E. Moriarty and R. Mikkulainen, "Efficient reinforcement learning through symbiotic evolution," *Machine learning*, vol. 22, no. 1-3, pp. 11–32, 1996.

[27] K. Tang, C. Chan, K. Man, and S. Kwong, "Genetic structure for nn topology and weights optimization," 1995.

[28] V. Maniezzo, "Genetic evolution of the topology and weight distribution of neural networks," *Neural Networks, IEEE Transactions on*, vol. 5, no. 1, pp. 39–53, 1994.

[29] D. Dasgupta and D. R. McGregor, "Designing application-specific neural networks using the structured genetic algorithm," in *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*. IEEE, 1992, pp. 87–96.

[30] M. M. Islam and X. Yao, "Evolving artificial neural network ensembles," in *Computational intelligence: a compendium*. Springer, 2008, pp. 851–880.

[31] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Neural Networks, IEEE Transactions on*, vol. 5, no. 2, pp. 157–166, 1994.

[32] A. P. Wieland, "Evolving neural network controllers for unstable systems," in *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, vol. 2. IEEE, 1991, pp. 667–673.

[33] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 4. IEEE, 2003, pp. 2588–2595.

[34] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Efficient non-linear control through neuroevolution," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 654–662.

[35] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *Neural Networks, IEEE Transactions on*, vol. 5, no. 1, pp. 54–65, 1994.

[36] F. Gomez and R. Miikkulainen, "2-d pole balancing with recurrent evolutionary networks," in *ICANN 98*. Springer, 1998, pp. 425–430.

[37] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[38] N. T. Siebel and G. Sommer, "Evolutionary reinforcement learning of artificial neural networks," *International Journal of Hybrid Intelligent Systems*, vol. 4, no. 3, p. 171, 2007.

[39] T. Natschläger, B. Ruf, and M. Schmitt, "Unsupervised learning and self-organization in networks of spiking neurons," in *Self-Organizing neural networks*. Springer, 2002, pp. 45–73.

[40] S. G. Wysoski, L. Benuskova, and N. Kasabov, "Fast and adaptive network of spiking neurons for multi-view visual pattern recognition," *Neurocomputing*, vol. 71, no. 13, pp. 2563–2575, 2008.

[41] S. M. Bohte, J. N. Kok, and H. La Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1, pp. 17–37, 2002.

[42] O. Booij and H. tat Nguyen, "A gradient descent rule for spiking neurons emitting multiple spikes," *Information Processing Letters*, vol. 95, no. 6, pp. 552–558, 2005.

[43] S. Ghosh-Dastidar and H. Adeli, "A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection," *Neural Networks*, vol. 22, no. 10, pp. 1419–1431, 2009.

[44] J. Xin and M. J. Embrechts, "Supervised learning with spiking neural networks," in *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 3. IEEE, 2001, pp. 1772–1777.

[45] S. McKennoch, T. Voegtlin, and L. Bushnell, "Spike-timing error backpropagation in theta neuron networks," *Neural computation*, vol. 21, no. 1, pp. 9–45, 2009.

[46] R. Vazquez *et al.*, "Training spiking neural models using cuckoo search algorithm," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE, 2011, pp. 679–686.

[47] D. Floreano and C. Mattiussi, "Evolution of spiking neural controllers for autonomous vision-based robots," in *Evolutionary Robotics. From Intelligent Robotics to Artificial Life*. Springer, 2001, pp. 38–61.

[48] H. Hagras, A. Pounds-Cornish, M. Colley, V. Callaghan, and G. Clarke, "Evolving spiking neural network controllers for autonomous robots," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 5. IEEE, 2004, pp. 4620–4626.

[49] N. Pavlidis, D. Tasoulis, V. P. Plagianakos, G. Nikiforidis, and M. Vrahatis, "Spiking neural network training using evolutionary algorithms," in *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 4. IEEE, 2005, pp. 2190–2194.

[50] A. Belatreche, L. P. Maguire, and M. McGinnity, "Advances in design and application of spiking neural networks," *Soft Computing*, vol. 11, no. 3, pp. 239–248, 2007.

[51] R. A. Vázquez, "Pattern recognition using spiking neurons and firing rates," in *Advances in Artificial Intelligence–IBERAMIA 2010*. Springer, 2010, pp. 423–432.

[52] R. Vazquez, A. Cachón *et al.*, "Integrate and fire neurons and their application in pattern recognition," in *Electrical Engineering Computing Science and Automatic Control (CCE), 2010 7th International Conference on*. IEEE, 2010, pp. 424–428.

[53] M. Valko, N. C. Marques, and M. Castellani, "Evolutionary feature selection for spiking neural network pattern classifiers," in *Artificial intelligence, 2005. epia 2005. portuguese conference on*. IEEE, 2005, pp. 181–187.

[54] Y. Jin, R. Wen, and B. Sendhoff, "Evolutionary multi-objective optimization of spiking neural networks," in *Artificial Neural Networks–ICANN 2007*. Springer, 2007, pp. 370–379.

[55] R. Batllori, C. B. Laramee, W. Land, and J. D. Schaffer, "Evolving spiking neural networks for robot control," *Procedia Computer Science*, vol. 6, pp. 329–334, 2011.

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

[57] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. Merolla, K. Boahen *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[58] S. B. Furber, D. R. Lester, L. Plana, J. D. Garside, E. Painkras, S. Temple, A. D. Brown *et al.*, "Overview of the spinnaker system architecture," *Computers, IEEE Transactions on*, vol. 62, no. 12, pp. 2454–2467, 2013.

[59] T. Pfeil, A. Grübl, S. Jeltsch, E. Müller, P. Müller, M. A. Petrovici, M. Schmuker, D. Brüderle, J. Schemmel, and K. Meier, "Six networks on a universal neuromorphic computing substrate," *Frontiers in neuroscience*, vol. 7, 2013.

[60] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[61] L. Bako, "Real-time classification of datasets with hardware embedded neuromorphic neural networks," *Briefings in bioinformatics*, p. bbp066, 2010.

[62] S. Cawley, F. Morgan, B. McGinley, S. Pande, L. McDaid, S. Carrillo, and J. Harkin, "Hardware spiking neural network prototyping and application," *Genetic Programming and Evolvable Machines*, vol. 12, no. 3, pp. 257–280, 2011.

[63] R. Hasan and T. M. Taha, "Enabling back propagation training of memristor crossbar neuromorphic processors," in *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE, 2014, pp. 21–28.

[64] D. Soudry, D. Di Castro, A. Gal, A. Kolodny, and S. Kvatinsky, "Memristor-based multilayer neural networks with online gradient descent training," 2015.

[65] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 356–367.

[66] H.-Y. Hsieh and K.-T. Tang, "Hardware friendly probabilistic spiking neural network with long-term and short-term plasticity," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 24, no. 12, pp. 2063–2074, 2013.

[67] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 36–45.

[68] M. Suri and V. Parmar, "Exploiting intrinsic variability of filamentary resistive memory for extreme learning machine architectures," *Nanotechnology, IEEE Transactions on*, vol. 14, no. 6, pp. 963–968, Nov 2015.

[69] M. I. Rabinovich, P. Varona, A. I. Selverston, and H. D. Abarbanel, "Dynamical principles in neuroscience," *Reviews of modern physics*, vol. 78, no. 4, p. 1213, 2006.