SPAIC: A Spike-Based Artificial Intelligence Computing Framework

Chaofei Hong, Mengwen Yuan, Mengxiao Zhang, Xiao Wang, and Chengjun Zhang
Zhejiang Lab, CHINA

Jiaxin Wang, Gang Pan, and Huajin Tang
Zhejiang University, CHINA

Abstract—Neuromorphic computing is an emerging research field that aims to develop new intelligent systems by integrating theories and technologies from multiple disciplines, such as neuroscience, deep learning and microelectronics. Various software frameworks have been developed for related fields, but an efficient framework dedicated to spike-based computing models and algorithms is lacking. In this work, we present a Python-based spiking neural network (SNN) simulation and training framework, named SPAIC, that aims to support brain-inspired model and algorithm research integrated with features from both deep learning and neuroscience. To integrate different methodologies from multiple disciplines and balance flexibility and efficiency, SPAIC is designed with a neuroscience-style frontend and a deep learning-based backend. Various types of examples are provided to demonstrate the wide usability of the framework, including neural circuit simulation, deep SNN learning and neuromorphic applications. As a user-friendly, flexible, and high-performance software tool, it will help accelerate the rapid growth and wide applicability of neuromorphic computing methodologies.

Digital Object Identifier 10.1109/MCI.2023.3327842
Date of current version: 8 January 2024

This paper was recommended for publication by Associate Editor Kai Qin.
Corresponding author: Huajin Tang (e-mail: htang@zju.edu.cn).
I. Introduction

Recent advances in multiple research fields have brought us closer to understanding and mimicking the brain. In neuroscience, models and theories accumulated at different scales have been developed to explore and understand neural systems. In deep learning, tremendous progress has been made to mimic brain functions with simplified models [1]. Even though these two fields set sail with a similar goal, they are developing in parallel and arriving at very different answers. Hence, the idea of integrating the neural mechanisms with machine learning techniques to develop brain-inspired computing systems with more intelligent functions or better computational efficiency has attracted increasing attention [2], [3], [4].

Efficient and easily accessible software tools have largely accelerated the development of deep learning [5], [6]. Similarly, the development of computational neuroscience has also been accompanied by a series of specialized simulation tools. However, the current software tools in both deep learning and neuroscience are not designed for this new interdisciplinary research. These two fields have developed separate approaches to modeling and studying neural models. Computational neuroscience frameworks should be able to build complex neural network structures, precisely model various biologically realistic features and flexibly customize neural dynamics with different scales. Deep learning frameworks, on the other hand, work on deep networks with simple units and emphasize large-scale parallel computing, gradient calculation, and optimization. The lack of a generally compatible training and simulation framework poses considerable obstacles for researchers to integrate theories from multiple disciplines and quickly validate new ideas. To meet this demand, we present spike-based artificial intelligence computing (SPAIC), a Python-based spiking neural network (SNN) training and simulation framework that blends programming styles and techniques from both deep learning and neuroscience and provides a platform for easily testing brain-inspired mechanisms and theories with neuromorphic models [7].

SNNs are common tools in computational neuroscience for modeling neural systems in a bottom-up manner. Compared to artificial neural networks (ANNs), SNNs capture the key computational properties of the biological neural system with temporal dynamics. Researchers have used SNNs to study the mechanisms underlying various neural dynamic behaviors, explore the functional roles of different neuronal arrangements and connectivity patterns, and simulate brain activities at a high level of realism [8]. With improving understanding of the brain, it is expected that such a biologically realistic model can better mimic the brain’s remarkable intelligence abilities than more abstract models. Several SNN simulation tools such as Neuron [9], Brian2 [10] and Nest [11] have been developed and can help researchers construct network models with customized neural dynamics and simulate neural activities with high precision. However, these software tools are not optimized for building complex intelligent models, especially those based on big data and training algorithms, and their computational efficiency is lower than that of current deep learning frameworks. One key objective of our framework is to support the construction of biologically realistic SNN models with high flexibility and efficiency. To achieve this goal, the SPAIC framework is designed with a frontend-backend architecture that decouples neural model creation and simulation. In the frontend, SPAIC provides user-friendly interfaces that can hierarchically compose complex network structures with neuron assemblies and various connection policies and can flexibly define neural dynamics with customizable neuron models, synapses, and learning rules. In the backend, the frontend network models are built into an optimized computation graph and run by a simulator engine based on popular deep learning frameworks, such as PyTorch [6] and TensorFlow [5], which provide efficient CPU/GPU parallel computing and autograd techniques. Moreover, neuroscience also examines brain functions from a top-down perspective, often focusing on brain states and information transfer at larger scales with higher-level models [12], [13], [14]. Hence, our framework also support higher-level models such as ANNs and mean-field neural models, and these different-level models can be built separately or merged into one hybrid network model. In this way, researchers can easily build and test their ideas with more abstract models, and then those results can be used as guidance for detailed SNN modeling.

In addition to modeling the biological characteristics and functional organization of neural systems, the use of optimization techniques is essential to achieve complex functionalities. In recent years, deep learning has provided techniques and theories for optimizing complex network structures. Another key objective of our framework is to naturally blend the understanding and techniques in the fields of deep learning, cognitive neuroscience and computational neuroscience into one unified modeling procedure. However, incorporating deep learning methodologies, such as gradient descent, into SNN models remains challenging, both in theory and software engineering. Moreover, biological neural systems utilize a diverse range of learning mechanisms to achieve their outstanding learning abilities, including timing-based or rate-based local learning rules [15], dopamine-driven learning [16], and wiring/rewiring mechanisms [17], which are not in line with the gradient descent optimization methodology. Although numerous works have demonstrated that SNNs can be efficiently trained using surrogate gradient algorithms [18], the inclusion of more complex structures and neural dynamics to achieve superior performance remains to be explored. Therefore, it is important to develop new learning rules that combine the effectiveness of deep learning algorithms with the efficiency and robustness characteristic of biological learning systems. To help researchers with this endeavor, SPAIC designs a learner class that defines learning algorithms in a general training framework, which supports both gradient-based learning and local (plasticity-based) learning rules. Some conventional learning algorithms are implemented in the framework, but users are encouraged to develop their own algorithms through this framework.

The SPAIC framework is an open-source project that is under intensive development. The source code of the framework is available at https://github.com/ZhejianglabNCRC/SPAIC, the documentation of the framework can be found at https://spaic.readthedocs.io. The paper is structured as follows.
In Section II, we review the existing software tools for neural network modeling in the fields of deep learning, neuroscience, and neuromorphic computing and discuss why a new framework is needed. In Section III, the structure and usage procedure of our framework are described in detail, where the motivation and functionality of each software component are explained. Example case studies are given in Section IV to demonstrate the potential usage situations of SPAIC. Then, the future developments of SPAIC are discussed in Section V.

### II. Related Work

Various software tools have been developed to model neural systems. Each is tailored toward specific application domains. According to their characteristics, these software tools can be roughly divided into three categories: deep learning, computational neuroscience, and neuromorphic computing frameworks. Computational neuroscience frameworks focus on simulating the details of physiological structures and neurons, while deep learning frameworks focus on achieving brain functions using simplified models and learning algorithms. Neuromorphic computing frameworks must combine the above two types of frameworks to construct models with brain-like functions and physiological details. In this section, we describe the relevant software tools and their challenges. Some popular frameworks are compared in Table I.

#### A. Deep Learning Frameworks

In recent years, many deep learning frameworks have emerged, such as Caffe [19], Theano [20], PyTorch [6], and TensorFlow [5]. They focus on the development, training, and inference of deep neural networks. Among them, PyTorch and TensorFlow are the two most popular open-source libraries for machine learning to date. The first version of TensorFlow (TensorFlow 1.x) only supported static computation graphs, whereas TensorFlow 2.0 supports dynamic models and simplifies the process of building machine learning frameworks. It is well suited for large-scale distributed training and can run on CPUs, GPUs, or large-scale distributed systems. PyTorch provides a Pythonic programming style and supports dynamic tensor computations with automatic differentiation and strong GPU acceleration. Its fast performance is achieved because it is written mostly in C++. PyTorch makes it easy for users to develop, debug, and run neural network models. In other words, these frameworks were primarily designed to optimize deep ANNs based on simple mathematical models, such as multilayer perceptions (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). They provide flexible APIs to perform general-purpose computations, but they are not optimized for building SNNs and do not natively support SNN-specific algorithms and models. Directly using these frameworks to develop SNN algorithms is possible but generally not optimal due to the time and effort needed.

#### B. Computational Neuroscience Frameworks

According to the needs of theoretical neuroscience simulations, several simulation software tools have emerged, mainly including NEURON [21], GENESIS [22], CARLsim [23], NEST [24], Brian2 [10], and BrainPy [25], which achieve...
different levels of biological realism. NEURON and GENESIS focus on simulating detailed realistic biological neurons with properties that include complex branching morphologies and multiple channel types. CARLsim, NEST, Brian2, and BrainPy focus on the dynamics and structures of neural systems rather than on the detailed morphological and biophysical properties of individual neurons. They can be used for simulating large heterogeneous networks. A major advantage of NEURON, NEST, Brian2, and BrainPy is that, in addition to their built-in neurons and connection objects, users can employ low-level language (such as C++) or mathematical model descriptions to design the dynamics of neurons and connections. This provides convenience for investigating new mechanisms. However, esoteric syntax may lead to a steep learning curve for new users. In addition, the lack of automatic differentiation support makes these tools unsuitable for training SNNs for machine learning tasks.

C. Neuromorphic Computing Frameworks

Frameworks such as Nengo [26], BindsNet [27], LAVA [28], sntTorch [29], and SpikingJelly [30] focus on the behaviors of SNNs and can be applied in the field of machine learning. They are built on deep learning frameworks (such as PyTorch and TensorFlow) to facilitate the fast simulation of SNNs on CPU- and GPU-based computational platforms, and they utilize deep learning training procedures to optimize their model parameters. Nengo is regarded as a cognitive modeling tool. It can be used to build large-scale models based on the neural engineering framework (NEF) to simulate advanced functions of the brain or brain regions. An extended version named NengoDL is also provided, which uses TensorFlow to improve the simulation speeds of Nengo models and implement automatic conversion from Keras models to Nengo networks. LAVA is a software framework used to develop neuro-inspired applications for Intel’s Loihi chips. To date, it has released a deep learning high-level library based on PyTorch for offline backpropagation-based training. SpikingJelly and BindsNet are SNN libraries developed on top of PyTorch, enabling rapid prototyping and concise feature syntax. In general, they are closer to deep learning procedures and do not effectively support neuroscience, e.g., simulating the anatomical and biophysical properties of neurons and neural circuits.

These frameworks focus on either neuroscience or deep learning, rather than encompassing both. SPAIC uses PyTorch as its computation backend, which is efficient and suitable for machine learning tasks. In addition, popular neuroscience-based neuron types, synapse types and connection types are provided for users to choose from. In this way, SPAIC can be seen as a bridge between the AI computing and neuroscience domains, enabling researchers to easily integrate the neural mechanisms found in neuroscience with machine learning techniques to develop better AI approaches.

III. Package Structure

The main structure of SPAIC is shown in Figure 1. The SPAIC platform consists of three functional blocks:

1) The network components contain all the components of the model that provide a frontend for setting up the network.
2) Within the simulation procedure, the simulation process of the network complies with the computation graph in the Backend.
3) The training and analysis tools provide features such as I/O interfaces, plot functions, training log functions, etc.
SPAIC provides a Network object to contain all the network components. Users can build complex structures using basic components, such as Node, NeuronGroup, Connection, and Assembly. Auxiliary components, such as the Learner, Optimizer, and Monitor, can be added according to the users’ requirements. The Backend should also be attached to the Network to compile the frontend network model.

A. Network Components

1) Network

The Network is the top level of the model in which all other components should be included. The network model can be defined by inheriting the Network class and adding network components to the __init__ function. The run function of the Network class starts the simulation of all network components in the Backend. The following code shows how to construct and run a network model.

```python
# Construct a TestNet
class TestNet(spaic.Network):
    def __init__(self):
        super(TestNet, self).__init__()
        ...
    
    net = TestNet()
    ...
    net.run(run_time)
```

2) Assembly

The Assembly is one of the most important components of SPAIC. It is an abstract neural population class from which the Network, NeuronGroup, Node, and Module are all inherited. It defines the basic network structure and attributes. Users can build large-scale and complex networks by stacking blocks that are defined by the Assembly objects. An example code is shown as follows:

```python
# Construct an assembly object as a subregion
class SubRegionA(spaic.Assembly):
    def __init__(self):
        super(SubRegionA, self).__init__()
        self.l1 = spaic.NeuronGroup(num=10, model='lif')
        self.l2 = spaic.NeuronGroup(num=20, model='lif')
        self.connection = spaic.Connection(pre=self.l1, post=self.l2)

class TestNet(spaic.Network):
    def __init__(self):
        super(TestNet, self).__init__()
        ...
    
    self.regionA = SubRegionA()
    ...
```

3) NeuronGroup

A NeuronGroup is a group of neurons with the same neuron model and connection pattern. Additionally, a NeuronGroup should contain all the details of its neurons, such as the initial voltage and spiking threshold. Users should provide the neuron model and the number of neurons when creating a NeuronGroup. SPAIC provides a series of built-in neuron models, such as the leaky integrate-and-fire (LIF) model [31], adaptive exponential integrate-and-fire (aEIF) model [32], Izhikevich (IZH) model [33], and Hodgkin–Huxley (HH) model [34]. The following code shows how to create a group with 50 LIF neurons.

```python
# NeuronGroup
self.l1 = spaic.NeuronGroup(num=50, model='lif')
```

The parameters of the neuron model, such as the time constant $\tau_m$, can be modified as keyword arguments of the NeuronGroup initialization function. If the parameter values are not given, SPAIC uses the default parameter values of these neuron models. In addition, users can define auxiliary attributes, such as the spatial position and neuron types in the NeuronGroup, which will be useful for constructing complex networks. Notably, a NeuronGroup also has a parameter_variables attribute, which allows users to specify a list of trainable parameters by passing their names to the attribute.

4) Node

Nodes are the input and output conversion units of the neural network, and they contain a coding mechanism that encodes inputs into spikes or decodes spikes into outputs. Nodes have five different subclasses:

- **Encoder**: Compared with the static numerical inputs of ANNs, SNNs use spike trains, which is consistent with the way that the brain transmits information. The Encoder implements a function for converting the input data into input spikes. SPAIC provides some common encoding methods, such as PoissonEncoder and LatencyEncoder. As an example, the following code defines a PoissonEncoder object that transforms the input into Poisson spike trains.

```python
# Encode input
self.input = spaic.Encoder(num_node_num,
    coding_method='poisson')
```

- **Decoder**: The main purpose of the Decoder is to convert the output spikes or voltages to numerical signals. SPAIC provides some common decoding methods, such as SpikeCounts and FirstSpike. For example, the following code defines a SpikeCounts object to obtain the number of spikes of each output neuron.

```python
# Decode the output spikes of layer 2
self.output = spaic.Decoder(num=10,
    coding_method='spike_counts',
    coding_var_name='p')
```

- **Generator**: This is a special encoder that generates spike trains or currents with a specified pattern. For example, in some computational neuroscience studies, users need special inputs, such as Poisson spikes, to model background cortical activities. To meet such requirements, some common pattern generators, such as PoissonGenerator or ConstantCurrentGenerator, are provided in SPAIC.

- **Reward**: During the execution of the reinforcement learning task, a Reward object is used to evaluate the performance of the agent based on the output neuron activities and the task objectives. For example, the GlobalReward for the classification task determines the predicted result according to the number of spikes or the maximum membrane potential. If the predicted result is
the same as the expected result, a positive reward is returned. Otherwise, a negative reward is returned.

**Action:** An *Action* is also a special decoder that transforms output activities into an action. The main usage of the *Action* is to choose the next action according to the action selection mechanism of the target object during reinforcement learning tasks. For example, *PopulationRateAction* takes the label of the neuron with the largest firing rate as its action.

5) **Topology**

The **Topology** component is used to specify the interactions between *Assembly* objects and consists of *Connection*, *Synapse*, *Projection*, and *ConnectPolicy* objects. *Connection* is the most generic topology implementation that connects elementary *Assembly* objects, such as *Node* and *NeuronGroup*. *Connection* can specify *Synapse* to define how presynaptic neurons affect postsynaptic neurons. A *Projection* is an abstract topology representing the communication between high-level *Assembly* objects that contain multiple *Connection* objects. *ConnectPolicy* defines a rule to generate specific *Connection* objects in a *Projection*.

*Connection* and *Synapse*: *Connection* objects link presynaptic neurons to postsynaptic neurons with certain connectivity patterns and synaptic weights. SPAIC supports many different connection forms, such as *FullConnection*, *RandomConnection*, *SparseConnection*, and *ConvolutionConnection*. In addition, *Synapse* objects are critical neural connection structures that usually transmit information from the source neurons to the target neurons. *Synapse* objects can be specified when creating a *Connection* object, and the input is filtered by the *Synapse* kernel. If there is no specific *Synapse* in a *Connection* object, the default synapse type is used, which directly adds the weighted sum of presynaptic spikes to the postsynaptic neurons’ membrane potential. For example, the following code shows how to build a full connection that connects the input layer and layer 1:

```
# Define an assembly with different types of neurons
class SubRegion(SpaiC.Assembly):
    def __init__(self):
        super(SubRegionA, self).__init__()
        self.11 = SpaiC.NeuronGroup(num=10, model='lif', type='excitory')
        self.12 = SpaiC.NeuronGroup(num=20, model='lif', type='inhibitory')

layer1 = SubRegion()
layer2 = SubRegion()

# Define specific connections using a projection object
ei_policy = SpaiC.IncludeTypePolicy(
    pre_types=['excitory'], post_types=['inhibitory'])
ei_project = SpaiC.Projection(pre=layer1, post=layer2, policies=[ei_policy])
```

6) **Module**

A *Module* is a special *Assembly* subclass that is directly inherited from *torch.nn.Module*. Therefore, SPAIC can implement functions supported by PyTorch and has the capability to design hybrid networks by combing ANNs and SNNs. The following example shows how to use a *Module* to add a convolution layer and a batch normalization layer to a network.

```
# Module with a convolution layer and a batch normalization layer
class conv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size):
        super(conv, self).__init__()
        self.conv2d = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size),
            nn.BatchNorm2d(out_channels))
        self.x = self.conv2d(x)

    def forward(self, x):
        return x

self.layer1_layer2_conv = conv.Module(
    module=conv(in_channels=3, out_channels=64, kernel_size=3),
    input_targets=self.layer1, input_var_names='O',
    output_targets=self.layer2, output_var_names='layr')
```

7) **Monitor**

The *Monitor* records the state variables of connections and neurons during the whole simulation period. At present, two types of monitors are available in SPAIC: *SpikeMonitor* and *StateMonitor*. *SpikeMonitor* records the spike trains, and *StateMonitor* records all other states. The following example code shows how to build a *StateMonitor* and a *SpikeMonitor* to record the voltage and output of layer 1, respectively.

```
# Monitor the voltage and output of layer 1
self.mon_v = SpaiC.StateMonitor(self.layer1, 'V')
self.mon_spike = SpaiC.SpikeMonitor(self.layer1, 'O')
```

8) **Learner**

The *Learner* is a base class for all learning and optimization algorithms of the network. The learning algorithm is the main
part of the Learner that can update any specified network parameters in the trainable scope. Here, two common types of SNN algorithms are provided as examples. The first type of algorithm is based on gradient backpropagation, such as STCA [35] and STBP [36], which are mainly realized via surrogate gradients. SPAIC provides an interface for setting any surrogate gradient function to the neuron’s spike generation function. Another type is based on synaptic plasticity, such as spike timing-dependent plasticity (STDP) [37], which is more biologically plausible. Furthermore, SPAIC supports algorithms for reinforcement learning, such as RSTDP [38], which is based on the STDP learning rule with a reward mechanism. These STDP-style learning rules are realized in the framework by tracing the relevant local variables in each time step, and the weights are updated using the trace variables when a target event is evoked (e.g., presynaptic or postsynaptic spikes). Users may customize their local or nonlocal learning algorithms by subclassing the Learner base class, and predefined functions are available to assist with the coding process.

The Optimizer is another part of the learner. It contains many optimization algorithms, such as adaptive moment estimation (Adam) and stochastic gradient descent (SGD), which can be used to optimize the network parameters of gradient-based algorithms. The usage of the gradient-based STCA learning algorithm and the Adam optimization algorithm is as follows:

```python
# Learner
self.learner = spaic.Learner(trainable=self,
                              algorithm='STCA')
self.learner.set_optimizer('Adam', 0.001)
... learner.optim_zero_grad()
learner.optim_step()
```

B. The Simulation Procedure

SPAIC decouples the neural model creation and simulation processes into the frontend and backend, respectively. The network components described above provide the frontend interfaces for creating a static symbolic description of the network, in which neural dynamics are described by finite difference equations. Then, the Backend can take the model description in the frontend, transform it into backend operations, and build an optimal computation graph that is suitable for implementing the simulation. SPAIC simulates SNNs using a clock-driven mode such that the network’s state variables in each time step are generally updated using the states from the last step, and the update cycle is repeated until it reaches the end of the runtime. In this mode, presynaptic neurons pass outputs to postsynaptic neurons every time step using tensor values (1 stands for a spike, and 0 stands for a nonspike). When the simulation procedure starts, the Backend fetches data from the Encoder and performs operations on the entire computation graph. Furthermore, the Decoder or Monitor can obtain data from the Backend. The whole process is shown in Figure 2.

Building a computation graph is a key procedural step from frontend network representation to backend simulation, where the operation devices, orders, and dependencies are determined. Even though most network states are updated using the last step’s values, the presynaptic spikes in this step can be directly passed to postsynaptic neurons if the network is a directed acyclic graph (DAG). However, biological neural circuits contain many cyclic graphs, such as lateral connections and feedback connections. Hence, a brain-inspired model usually cannot be directly built into a DAG due to its circular dependence on the calculations. SPAIC provides two construction strategies that build a model into a parallel or serial computation graph. First, the parallel computation graph is the most common method in neuroscience SNN simulators. In this computation graph, all connections transmit the presynaptic spike values of the last step so that all components can run in parallel. Second, for consistency with the serial computation process in deep neural networks (DNNs), SPAIC provides another construction strategy in which its connections try to use presynaptic spikes of the current step. If the network contains cyclic graphs, it uses topological sorting [39] to search the network and decompose its structure into feedforward and cyclic parts. In the cyclic part, one of the connections is selected to utilize the spikes from the previous step, addressing the circular dependency problem. Consequently, the network model can be transformed into a DAG.

C. Training and Analysis Tools

Training and analysis are the most time-consuming tasks in neural network research. SPAIC supports numerous useful tools and I/O interfaces to provide a more user-friendly environment, such as Plots and TrainingLog. The Plot object provides functions for generating diagrams of the results and process analyses, and TrainingLog records the parameter variations to help users analyze the role of algorithms.
The I/O interface component provides four tools: Dataset, DataLoader, Dataset, and Environment, each serving as a distinct component for data management.

- **Dataset**: organizes and formats data, e.g., generating an index from data and batching according to the batch size.
- **DataPort**: a data transmission interface that can receive data in real time.
- **Environment**: used for reinforcement learning.

Finally, SPAIC contains the NetworkSave component in its library tools, which allows users to save the parameters or the whole trained model for further use.

### D. Custom Extension

SPAIC can also be extended by custom-defined neuron models, connections, and learning rules following the customization guidance, which is discussed below.

Network components can be customized by incorporating additional variables and operations. Two types of coding methods are available to customize operations, as outlined below.

1. **String-based commands**: SPAIC objects (e.g., NeuronGroup, Connection, etc.) provide a variable dictionary and an operation list. When users want to implement an operation, such as A=B+C, the first step is to add the variables used in the operation into the variable dictionary as follows.

```python
self._variables['A'] = 0.0
self._variables['B'] = 0.0
self._variables['C'] = 0.0
```

The second step is to append the string of the operation into the operation list. The main operation format is that the first variable represents the result, and the second item contains the names of the basic operations supported in the Backend. The remaining variables from the third term represent the input parameters of the calculation formula.

```python
self._operations.append(('A', '+', 'B', 'C'))
```

2. **Standalone function**: When the calculation formula is difficult to implement using the basic operations provided in the Backend, users can employ the standalone function approach. First of all, the desired operation needs to be implemented in a Python function that is compatible with the chosen backend. Then, the string of the operation should be appended to the operation list, where the second item is the callable function. An example is shown below.

```python
def add_func(a, b):
    return a + b
self._operations.append(('A', 'add_func', 'B', 'C'))
```

### 1) Custom Neuron Model

The abstract class `NeuronModel` implements functionality that is common to all neuron types. Users need to define the calculation formula for the neurons themselves in the body of the `__init__` function, and register the customized model using the `NeuronModel.register` function.

### 2) Connection

Users can define their own connection types by creating a class that inherits from `Connection`, and override the `__init__` function within the new class. Many different synapses exist in the neural systems, and SPAIC allows its users to add their own synapse models. SPAIC provides an abstract `SynapseModel` class that is similar to `NeuronModel`.

### 3) Learning Rule

The `Learner` is an abstract class for all learning and optimization algorithms of the network. The new learning algorithm class should override the `custom_rule` function of the `Learner` to access and update any specified network parameters. The `trainable` and `pathway` lists are utilized to specify the components associated with training. The trainable list contains the network components whose parameters can be updated by the learning rule. The pathway list contains the network components that are useful to the learning rule but its parameters should not be updated. For example, in a gradient backpropagation learning rule, if only the last few layers of the network are trained, the remaining layers should be added to the pathway list so that the loss gradients can be passed through the whole network.

Two families of learning rules are supported in SPAIC: gradient-based and plasticity-based rules. Users can follow the STCA or STDP format in SPAIC to customize a gradient-based algorithm. If users want to add plasticity-based algorithms, they can follow the STDP format.

### IV. Examples

Here, we provide six exemplary applications, showcasing the versatility and capabilities of the SPAIC framework. The source code for all the examples in this article are located at https://github.com/ZhejianglabNCRC/SPAIC_paper_examples.

#### A. Computational Neuroscience Modeling

The central motivation of SPAIC is to assist the development of intelligent computing models inspired by the biological brain. Hence, it is fundamental that SPAIC can support the models and methods used in computational neuroscience. Then, those features can be introduced to a brain-inspired AI system.

### Cortical microcircuit model

We first demonstrate the implementation of a spiking cortical microcircuit model using SPAIC and employ a mean-field counterpart model to show the multiscale modeling process in SPAIC. As shown in Figure 3(a), the Potjans-Diesmann model contains two types of neurons distributed in four layers [L2/3, L4, L5, L6], which represents the cortical microcircuit network below a 1-mm surface [40]. In each layer, the subnetwork can be viewed as an internally connected excitation and inhibition balanced network [41], which can be built as a prototype submodule using the `assembly` object. Then, the cortical microcircuit can be constructed by stacking and connecting the four layers with different parameters. In this case study, neurons are constructed using the LIF model. The neurons are randomly connected with
a given probability using SparseConnection, while the external noise input into each neural population is modeled using Poisson-Generator. As shown in Figure 3(c), the model exhibits irregular and stationary spiking activities that are similar to the results in [40]. Additionally, the LIF spiking model can be substituted by a neural mass rate model with trainable coupling efficiency; then, this abstract model can be trained to achieve comparable results (Figure 3(b)(d)).

C. Elegans thermotaxis circuit

The second example is a spiking model of the C. elegans thermotaxis circuit, which is adapted from earlier studies [42], [43]. This network has a small number of well-characterized neuron types (Figure 4(a)) and is known to generate a stereotypical triphasic motor pattern (clockwise turns, counterclockwise turns, and random walks) when the sensor neurons experience temperature changes. The microcircuit encompassing interneurons AIY, AIZ, and RIA performs a derivative operation, which enables C. elegans to achieve diversion and track favorable temperature contours. The SPAIC network simulation is shown in Figure 4(c). We use the aEIF model to model the dynamics of the neurons and adopt FullConnection to link individual neurons. As the input temperature shifts above or below the threshold, the output neurons turn clockwise or counterclockwise, respectively, and are alternately active. Thus, the model can control the temperature steering of C. elegans to track along the optimal temperature region, as demonstrated in Figure 4(b).

B. Training Deep SNNs

SPAIC is suitable for training deep SNNs to solve problems in the machine learning domain. Here, we present examples to show how to build deep SNNs using SPAIC for implementing machine learning tasks, such as speech recognition and image recognition.

Speech recognition

As shown in Figure 5, a four-layer SNN is trained to perform supervised learning on the TIDIGITS speech dataset [44], which consists of digit utterances from ‘zero’ to ‘nine’ and ‘oh’. The dataset is split into 3465 training samples and 1485 testing samples. Notably, when directly reading speech files, tens of thousands of data are contained in one second of audio along with considerable redundant information. Therefore, before performing the speech recognition task, it is necessary to preprocess the raw speech files to obtain the dataset features. In SPAIC, two popular preprocessing methods are made available, namely, Mel-frequency cepstral coefficients (MFCCs) [45] and keypoint extraction (KP) [46]. Here, the MFCC preprocessing method is employed to extract features, and the STCA learning algorithm is used to train the network.

The results in Figure 6 show that competitive performance is achieved within five training epochs. These results demonstrate the effectiveness of the network training methods, making SPAIC suitable for training deep SNNs to solve machine learning tasks.

Image recognition

In addition to constructing an SNN with fewer layers, SPAIC is convenient for implementing a deeper SNN. In recent years, the deep residual network (ResNet) has replaced visual geometry group (VGG) as the basic feature extraction network in computer vision [47]. Here, we implement a spiking ResNet-18 model ($64k3(1 - BN - SpikingNeuron - spiking residual block(a(i \in 64, 128, 256, 512)(j \in 1, 2) * 4 - AP - FC))$ on SPAIC. The spiking residual block is the key structure for implementing machine learning tasks, such as speech recognition and image recognition.
implementing ResNet. When the residual block is built, ResNets with arbitrary depth can be realized through structure repetition. In the spiking residual block, the rectified linear unit (ReLU) activation layers of the standard residual blocks are substituted with spiking neurons. The basic structure of the spiking residual block is shown in Figure 7(a). The input and output of the basic block have the same dimensions, and thus, there is no convolution operation on the shortcut path. When the input and output have different dimensions or stride $>1$, there is a convolution operation on the shortcut, and the structure of the residual block is shown in Figure 7(b). The Assembly and Module classes of SPAIC facilitate the convenient implementation of residual blocks or any complex deep model structure.

Without using any training tricks, we train the spiking ResNet-18 model from scratch with surrogate gradient learning to validate the ability of SPAIC to implement deep SNNs. The experimental results shown in Figure 8 demonstrate that the spiking ResNet-18 model implemented on SPAIC achieves comparable performance (90.24%) on the CIFAR-10 dataset compared to that of other networks.

C. Neuromorphic Computing Applications

SPAIC is able to support neuromorphic computing applications that will eventually operate on robots or terminal systems. These tasks require the platform to 1) run fast enough to be able to respond in real-time and 2) provide data interfaces to communicate with the other system during the network simulation. Here, we present a simulated robot implementing a reinforcement learning task to demonstrate the usability of SPAIC in those applications.

**SNN and ANN based hybrid reinforcement learning**

Reinforcement learning is another important machine learning algorithm that is concerned with learning to control agents (such as robots) to maximize their performance with respect to a long-term objective. For example, an end-to-end SNN network for the mapless navigation task of a mobile robot is constructed by adopting the reinforcement learning method (Spiking-DDPG), which incorporates a spiking actor network (SAN) and a deep critic network.

![FIGURE 5](image)  
**FIGURE 5** The network structure for TIDIGITS speech recognition. It is a four-layer SNN with full connections trained by the STCA algorithm.

![FIGURE 6](image)  
**FIGURE 6** The network can develop suitable representations, as demonstrated by the increasing average test accuracy over five runs.

![FIGURE 7](image)  
**FIGURE 7** The structure of the residual block. (a) Basic block. (b) Residual block with downsampling. If the input and output have different dimensions or stride $>1$, convolution is performed on the shortcut.
The Gazebo Robot simulator is used as middleware for both training and validation. PoissonEncoder is used to encode the input state obtained from the Gazebo simulator, which is a $1 \times 24$ array composed of the relative distance and direction from the robot to the goal, the linear and angular velocities of the robot, and the distance observations from the laser range scanner. During training, the SAN generates an action based on the input state given by the robot simulator, which is then used to control the left and right wheel speeds of the robot. Additionally, the action is fed to the deep critic network to predict the associated Q-value for SAN training.

Based on the action commands, the Gazebo environment generates a reward to update the Q-value for training the deep critic network. The SPAIC platform supports a wide range of neurons and training policies that can be applied by users to modify the SNN actor network to perform any end-to-end robot control task. Figure 9 depicts the training and validation environment instance in the Gazebo simulator and the details of the implementation of the Spiking-DDPG training process.

D. Integration of Neuroscience and Deep SNN Learning

The most prominent advantage of the SPAIC framework is its ability to integrate theories and techniques from both neuroscience and deep learning into brain-inspired SNN models. The learning backend of SPAIC is designed to ensure that biological network features, such as top-down connections, delays, and synaptic plasticity, are compatible with gradient backpropagation algorithms. In this way, researchers can easily test neuroscience theories by training biological models to perform real AI tasks and as a result, find new possibilities for brain-inspired AI systems.

Hybrid learning of the CANN

In the above case studies, we demonstrate the training process of a deep SNN using surrogate gradient learning rules. To enhance the SNN’s learning capability, a promising direction is to introduce biological learning mechanisms and network features. The continuous attractor neural network (CANN) is a canonical model used in neuroscience studies to describe a wide range of functions in neural systems, such as orientation and spatial location representations [48], and memory replay in the hippocampus [49]. However, CANN implementations for deep learning tasks have not been proposed in previous studies. In this example, we demonstrate how to build a brain-inspired network model with a CANN structure and synaptic plasticity and then train it on the MNIST image recognition task using gradient-based learning rules. As shown in Figure 10(a), a two-layer feedforward model is constructed, which consists of 400 and 10 neurons. The hidden layer neurons are interconnected with lateral inhibition to form continuous attractor dynamics. The hidden layer neurons are modeled with an adaptive threshold so that each neuron has an equal firing probability after training. The feedforward connections of the hidden layer are updated with both a gradient-based optimizer and STDP. Since the STDP rule continuously updates the weights in the forward pass while the gradient-based optimizer updates after the backward pass, a meta-learning approach is employed to ensure that these two weight updates are compatible. We compare the performance of the CANN trained with and without STDP and a feedforward network (FFN) trained...
with the same neuron size and surrogate gradient algorithm. Furthermore, the robustness of these networks is tested by evaluating their performance in processing images corrupted by salt-and-pepper noise of varying intensities. Figure 10(b) shows that STDP does not help increase the CANN’s classification accuracy but improves the network’s robustness to noise. As shown in Figure 10(c), the hidden layer weights trained with STDP are more focused on global features, which may not be as informative as the fragment features obtained from purely gradient-based learning, but the redundant features are more resistant to noise. In addition, both CANNs perform better than the FFN in terms of their classification accuracy and robustness to noise. This result implies that continuous attractor dynamics may help with image classification by enforcing feature integration within the hidden layer.

E. Performance Benchmarks

To provide an impartial comparison with the prominent existing platforms, three experiments are designed to compare their memory and time costs. The first experiment involves simulating a simple network without training. The second experiment focuses on comparing the platforms that support backpropagation on the MNIST dataset recognition task. Finally, the third experiment involves comparing neuroscience simulation platforms on networks equipped with the STDP rule. All benchmark experiments are run on a workstation with Ubuntu 18.04 LTS, an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10 GHz, 128 GB of RAM, and an Nvidia Quadra GV100 GPU with 32 GB of memory. To record the memory usages of the platforms, PyTorch.cuda.max_memory_allocated is used for all GPU experiments, and Memory_Profiler is used for all CPU experiments. All GPU experiments only record the graphics card memory usages.

For the first simulation experiment, SPAIC, SpikingJelly, BindsNet, and BrainPy are tested with both the CPU and GPU, while Brian2 and NEST are tested with only the CPU. The test network consists of an input layer containing 100 neurons using a 30-Hz Poisson input and a test layer with a varying size. The output layer of this network has \( n \) (from 10 thousand to 10 million) LIF neurons forevolving speed tests. Each model is executed for a duration of 100 ms with \( dt = 1.0 \text{ ms} \) time steps. The model is run for 100 cycles, and the entire running time is calculated. Brian2 and NEST simulate spike transmission in an event-driven mode, and their performance varies with different spike rates. In this case, a moderate spike input rate is established for the benchmark test. Noteworthy, NEST is exclusively tested in a small-scale setting (\( n/C^2 < 10^5 \)) because its performance is not competitive at larger scales. The performance results can be seen in Figure 11. As shown in Figure 11(a), CPU-only SPAIC performs best in small-scale networks with \( n < 10^6 \). CPU-only SpikingJelly achieves the best performance on larger-scale networks with \( n \geq 10^6 \). Figure 11(b) depicts that GPU-based SPAIC performs best for networks with \( n \leq 10^6 \), and it performs comparably well on SpikingJelly at larger network scales.
The BrainPy platform simulations are terminated for \( n \geq 10^7 \) due to out-of-memory errors.

The second benchmark assesses a series of platforms that focus on deep learning, such as SPAIC, SpikingJelly, BindsNet, and PyTorch, using the GPU. A network with 784 Poisson input nodes, a hidden layer with \( n \) (from 100 to 10000) LIF neurons and an output layer with 10 nodes is trained by a surrogate gradient. The other simulation configurations are the same as those in the first experiment. As shown in Table II, the advantage of SPAIC becomes increasingly obvious with increasing hidden layer scale. In conclusion, SPAIC is suitable for deep learning tasks. Noteworthy, SPAIC exhibits better time performance than SpikingJelly and PyTorch, despite the fact that they all utilize the same PyTorch backend. It could be due to SPAIC’s utilization of low-level APIs (e.g., functions and tensors) for computation, in contrast to both SpikingJelly and PyTorch, which employ high-level APIs like nn.Module to build and run their networks.

The third benchmark tests the STDP learning task. In this benchmark test, the network consists of 100 Poisson input nodes, a hidden layer with either 100 or 10000 neurons and an output layer of 10000 neurons, and the STDP rule is implemented in the connections. Notably, when tested with the minimum network scale, LAVA exhibited significantly longer simulation times, exceeding 700 seconds. This extended duration can be attributed to the fact that LAVA was specifically designed to accurately simulate the behavior of the Loihi hardware. Here, we do not further compare LAVA with other simulation platforms. The STDP learning rule is used on the connection between layer 1 and layer 2. According to the characteristics of Brian2 and NEST, their connection weights are adjusted to control the spike rates of layer 1 and layer 2 to 30 Hz and 100 Hz, respectively. As shown in Table II, SPAIC running on a GPU achieves satisfactory performance in all situations. In conclusion, SPAIC is suitable for computational neuroscience tasks.

Although our benchmark is not indicative of the model performance in all circumstances, it still shows that SPAIC can achieve good performance compared with the existing platforms. Furthermore, the rich functionality shown in Table I is one of the highlights of SPAIC. Additionally, flexibility is a significant aspect for evaluating platforms. We define flexibility based on the ability of a framework to develop learning algorithms. As shown in Table II, SPAIC supports both types of learning algorithms. As shown in Figure 12, the typical platforms can be characterized based on their performance, flexibility, and functionality, demonstrating the superiority of SPAIC over its counterparts.

### V. Further Development

The SPAIC framework is currently under continuous development, and there is still much room for improvement. Below, we list several ongoing developments and prospective trajectories for our framework.
1) Greater algorithm and model support: We are continuously extending the support of SPAIC for SNN learning algorithms, encoding/decoding methods and neuronal/synaptic models. We plan to integrate more learning types, such as genetic algorithms and network architecture searches. In the future, new models will also be added and algorithms will be developed by our research group based on SPAIC. Even though users tend to use their own customized models and algorithms in the applications, the ample supports for current models provide an entry point for working with the framework.

2) Performance optimization: Optimizing the speed and memory usage of the training and simulation processes will improve the efficiency of SNN experiments. The frontend-backend structure facilitates in-depth optimization of performance but also requires considerable time to refine detailed computations. Several attempts are ongoing to achieve improved performance. 1) In the torch backend, dynamic computation graphs can be converted into static graphs, thereby enhancing computational efficiency through fusion and transformation operations. 2) JAX will be added to the backend engine, and its JIT techniques will be used to improve performance. 3) The sparse matrix representation and computation processes will be customized to improve their efficiency in large sparse networks.

3) ODE solver support: Currently, SPAIC uses discrete-time dynamical models to iteratively compute the models with the Euler or exponential Euler method, which is easy for back-propagation through time (BPTT) based learning algorithms. However, to support simulations with various precision levels and more explicit dynamical model representations, we plan to support model representation with the ODE form and add ODE solvers such as Runge-Kutta methods. To support gradient-based learning in more complex ODE solvers, adjoint methods should be imported into the framework.

4) Extension tools and graphical user interface (GUI): In this primary version of SPAIC, the core SNN simulation and training functions are provided. Future plans involve expanding the framework by incorporating tools that facilitate model building and analysis. For example, the framework is scheduled to incorporate support for visualization tools such as Tensorboard, which can help analyze the model and training process.

**VI. Discussion and Conclusion**

The intention of developing the SPAIC framework is to assist new brain-inspired modeling studies and various applications in the neuromorphic community, including SNN algorithms, neural system models, and robotic systems.

Neuromorphic computing is still an emerging multidisciplinary research field, where multiple theories and methodologies are complementary to each other, and the boundary of this field is relatively ambiguous. Deciding what features should be emphasized for such a computing framework to support both training and simulation is a considerable challenge. Currently, the mainstream research on neuromorphic computing approximates the performance of state-of-the-art deep learning methods with equivalent SNN models. One important topic involves optimizing gradient propagation in SNNs to ensure desirable gradient distributions in deep SNNs. Various surrogate gradient functions have been proposed for this task [35], [36], [50], and recently, Li et al. [51] proposed a method to dynamically adapt the surrogate gradient functions to match each layer’s state. Moreover, regulation techniques for SNNs have been proposed, such as the threshold-dependent batch normalization (tdBN) method [52] and the membrane potential rectification method [53]. With recent developments, various deep SNN structures, such as the spiking ResNet, can be trained efficiently and achieve comparable performance to that of their deep learning counterparts in image classification tasks [54]. In the above works, SNNs were viewed as special RNNs, and in such configurations, deep learning techniques can be adapted to SNN algorithms.

However, this deep learning viewpoint may also be limited or even misleading for brain-inspired computing research, as there are fundamental differences between SNNs and RNNs. For example, RNN populations are designed to compute outputs at every time step; information processing occurs at the population level (tensor), while spiking neurons only emit spikes very sparsely, and computation mainly occurs within single-neuron dynamics [55]. Current SNN algorithms typically use simple neuron models and ultrashort time steps to efficiently perform AI tasks [56]. This kind of simplified SNN model can hardly contain long-term memory, and hence, the information processing capability of its neural dynamics is greatly limited. On the other hand, some studies have begun to introduce more complex neural dynamics and structural features to develop neuromorphic computing models. For example, Chen et al. employed resonate neuron and oscillation dynamics to facilitate network information processing [57]. Backward residual connections were introduced to deep SNNs to fully utilize each layer’s information processing capability [58]. To facilitate brain-inspired computation studies, we design the SPAIC network construction interface with a neuroscience style such that biologically feasible features can be easily incorporated into brain-inspired models. Furthermore, deep learning methodologies are incorporated into our framework as powerful tools for optimizing network functions. The Learner is utilized to integrate learning theories from both fields into a unified procedure. This hybrid coding style is most suitable for current brain-inspired modeling studies.

To meet the needs of brain-inspired modeling methodologies, we must seek a balance between the neuroscience principles and the deep learning techniques to provide considerable flexibilities in the computing framework. In SPAIC, a guideline is offered for constructing a network model by combining Assembly, Connection, and Learner objects, with each object specialized in one aspect of neural computation. On the other hand, each of these network components provides interfaces to directly customize their backend computations. Hence, users can easily build their models. Moreover, while the framework offers functions to model biological features such as synaptic dynamics, sparse connections, and conduction delays, users are
provided with the flexibility to implement their own realizations to better suit their usage cases. In summary, SPAIC is a highly customizable framework that can help researchers easily and efficiently build and test brain-inspired models and develop various AI applications.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2020AAA0105900, in part by the National Natural Science Foundation of China under Grants 62236007, 62106234, and 61925603, and in part by the Key Research Project of Zhejiang Lab under Grant 2021KCOAC01.

References

[1] L. C. Yan, B. Yoshua, and H. Geoffrey, “Deep learning,” Nature, vol. 521, no. 7553, pp. 436–444, 2015.
[2] L. F. Abbott, “Theoretical neuroscience rising,” Neuron, vol. 60, no. 3, pp. 489–495, 2008.
[3] H.-J. Park and K. Friston, “Structural and functional brain networks: From connections to cognition,” Strum, vol. 342, no. 6158, 2013, Art. no. 1238411.
[4] A. H. Marblestone, G. Wayne, and K. P. Kording, “Toward an integration of deep learning and neuroscience,” Front. Comput. Neurosci., vol. 10, 2016, Art. no. 94.
[5] M. Abshoff et al., “Towards a system for large-scale machine learning,” in Proc. 12th USENIX Symp. Operating Syst. Des. Implementation, 2016, pp. 265–283.
[6] A. Pasuke et al., “Pytorch: An imperative style, high-performance deep learning library,” in Proc. Adv. Neural Inf. Process. Syst., 2019, pp. 8026–8037.
[7] C. D. Schuman et al., “Opportunities for neuromorphic computing algorithms and applications,” Nature Comput., Sci., vol. 2, no. 1, pp. 10–19, 2022.
[8] X. Fan and H. Markram, “A brief history of simulation neuroscience,” Front. Neuroinform., vol. 13, 2019, Art. no. 32.
[9] N. T. Carnevale and J. L. M. Hines, The NEURON Book: Cambridge, U.K.: Cambridge Univ. Press, 2006.
[10] M. Stümbirg, R. Brette, and F. D. Goodman, “Brian 2, an intuitive and efficient neural simulator,” Elife, vol. 8, 2019, Art. no. e47314.
[11] J. M. Eppel, M. Helus, E. Muller, M. Diesmann, and M.-O. Gewaltig, “Pyne: A convenient interface to the next simulator,” Front. Neuroinform., vol. 2, 2009, Art. no. 363.
[12] M. S. Gazagnini, The Cognitive Neurosciences: Cambridge, MA, USA MIT Press, 2009.
[13] N. N. Kriegeskorte and P. K. Douglas, “Artificial neural networks: Bringing the power of gradient-based optimization to spiking neural networks,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2021, pp. 1016–1014, 2023.
[14] W. Fang et al., “Spikeinglidy,” 2020. Accessed: May 21, 2022. [Online]. Available: https://github.com/tangwei23456/spikeinglidy
[15] R. B. Stein, “Some models of neuronal variability,” J. Physiol., vol. 117, no. 1, pp. 35–40, 1940.
[16] N. Theobald and A. F. Fluxey, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” J. Physiol., vol. 117, no. 4, 1952, Art. no. 500.
[17] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, “Spatio-temporal backpropagation for training high-performance spiking neural networks,” Front. Neuroni., vol. 12, 2018, Art. no. 331.
[18] S. Song, K. D. Müller, and L. F. Abbott, “Competitive Hebbian learning through spike-time-dependent synaptic plasticity,” Nature, Neurosci., vol. 3, no. 9, pp. 79–86, 2000.
[19] R. V. Florian, “Reinforcement learning through modulation of spike-time-dependent synaptic plasticity,” Neural Comput., vol. 19, no. 6, pp. 1468–1502, 2007.
[20] A. B. Kahn, “Topological sorting of large networks,” Commun. ACM, vol. 5, no. 3, pp. 1568–1562, 1962.
[21] T. C. Portis and M. Diesmann, “The cell-type specific cortical microlcircuit: Relating structure and activity in a full-scale spiking network model,” Cereb. Cortex, vol. 24, no. 3, pp. 785–806, 2014.
[22] N. Brunel, “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons,” J. Comput. Neurosci., vol. 8, no. 3, 2000.
[23] T. Komata, H.asaki, N. Nishida, and I. Mom, “Thermotaxis of C. elegans as a model for temperature perception, neuronal information processing and neural plasticity,” in Proc. IEEE Int. Joint Conf. Neural Net., 2014, pp. 2079–2086.
[24] R. G. Leonard and G. Doddington, “Tadgits speech corpus,” Texas Instru., 1993.
[25] M. Usman, Z. Ahmad, and M. Wajid, “Dataset of raw and pre-processed speech signals, me frequency cepstral coefficients of speech and heart rate measurements,” in Proc. IEEE 5th Int. Conf. Signal Process., Comput. Control, 2019, pp. 376–379.
[26] M. B. Bhat, H. Tang, P. Gu, and X. Xu, “Spike-based encoding and learning of spatio-temporal features for robust sound recognition,” Neurocomputing, vol. 313, pp. 65–73, 2018.
[27] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. IEEE Comput. Conf. Vis. Pattern Recognit., 2016, pp. 770–778.
[28] M. Bora, A. Rao, and R. Bajcsy, “Mimicking the worm—an adaptive spiking neural circuit for sound tracking inspired by C. elegans thermotaxis,” in Proc. IEEE Int. Joint Conf. Neural Net., 2014, pp. 2079–2086.
[29] A. H. Azizi, L. Wiskott, and S. Cheng, “A computational model for prey in the lappanvus,” Front. Comput. Neurosci., vol. 7, 2013, Art. no. 161.
[30] C. G. S. S. Sarwar, P. Panda, G. Smirnov, and K. Roy, “Enabling spike-based backpropagation for training deep neural network architectures,” Front. Neuroni., vol. 14, 2020, Art. no. 119.
[31] Y. Li, Y. Guo, S. Zhang, S. Deng, Y. Hai, and S. Gu, “Differentiable spike: Rethinking gradient-descent for training spiking neural networks,” Adv. Neural Inf. Process. Syst., vol. 34, pp. 2342–2343, 2021.
[32] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, “Going deeper with directly-trained larger spiking neural networks,” in Proc. AAAI Conf. Artif. Intell., 2021, vol. 35, no. 12, pp. 11062–11070.
[33] Y. Guo et al., “Recals-SNN: Rectifying membrane potential distribution for directly training spiking neural networks,” in Proc. IEEE/CVF Conf. Visual Comput. Pattern Recognit., 2022, pp. 326–335.
[34] W. Fang, Z. Yu, Y. Chen, T. Huang, T. Masquelier, and Y. Tian, “Deep residual learning in spiking neural networks,” Adv. Neural Inf. Process. Syst., vol. 34, pp. 21056–21069, 2021.
[35] W. Gerstner and W. M. Kistler, Spiking Neuron Models: Single Neurons, Populations, Plasticity, Cambridge, U.K.: Cambridge Univ. Press, 2002.
[36] G. Datta and P. A. Beerdl, “Can deep neural networks be converted to ultra low-latency spiking neural networks?,” in Proc. IEEE Des., Automat. Test Eur. Conf. EchC., 2022, pp. 718–723.
[37] Y. Cheng, H. Qi, M. Zhang, and Y. Wang, “Deep-spike neural network with neural oscillation and spike-phase information,” in Proc. AAAI Conf. Artif. Intell., 2021, vol. 35, no. 8, pp. 7073–7080.
[38] P. Panda, S. A. Aken, and K. Roy, “Towards scalable, efficient and accurate deep spiking neural networks with backward residual connections, stochastic soft-max, and hybridization,” Front. Neuroni., vol. 14, 2020, Art. no. 653.