Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke

Surrogate Gradient Learning in Spiking Neural Networks

Bringing the power of gradient-based optimization to spiking neural networks

©ISTOCKPHOTO.COM/JUST_SUPER

Digital Object Identifier 10.1109/MSP.2019.2931595
Date of current version: 29 October 2019

Spiking neural networks (SNNs) are nature's versatile solution to fault-tolerant, energy-efficient signal processing. To translate these benefits into hardware, a growing number of neuromorphic spiking NN processors have attempted to emulate biological NNs. These developments have created an imminent need for methods and tools that enable such systems to solve real-world signal processing problems. Like conventional NNs, SNNs can be trained on real, domain-specific data; however, their training requires the overcoming of a number of challenges linked to their binary and dynamical nature. This article elucidates step-by-step the problems typically encountered when training SNNs and guides the reader through the key concepts of synaptic plasticity and data-driven learning in the spiking setting. Accordingly, it gives an overview of existing approaches and provides an introduction to surrogate gradient (SG) methods, specifically, as a particularly flexible and efficient method to overcome the aforementioned challenges.

Introduction

Biological SNNs are a highly efficient solution to the problem of signal processing. Therefore, taking inspiration from the brain is a natural approach to engineering more efficient computing architectures. In the area of machine learning, recurrent NNs (RNNs), a class of stateful NNs whose internal state evolves with time (see "Recurrent Neural Networks"), have proven highly effective at solving real-time pattern recognition and noisy time-series prediction problems [1]. RNNs and biological NNs share several properties, such as a similar general architecture, temporal dynamics, and learning through weight adjustments. Based on these similarities, a growing body of work is now establishing formal equivalences between RNNs and networks of spiking leaky integrate-and-fire (LIF) neurons, which are widely used in computational neuroscience [2]–[5].

RNNs are typically trained using an optimization procedure in which the parameters or weights are adjusted to minimize a given objective function. Efficiently training large-scale RNNs is challenging due to a variety of extrinsic factors, such as noise and nonstationarity of the data, but also due to the inherent difficulties

of optimizing functions with long-range temporal and spatial dependencies. In SNNs and binary RNNs, these difficulties are compounded by the nondifferentiable dynamics implied by the binary nature of their outputs. Although a considerable body of work has successfully demonstrated the training of two-layer SNNs [6]–[8] without hidden units as well as networks with recurrent synaptic connections [9], [10], the ability to train deeper SNNs with hidden layers has remained a major obstacle. Because hidden units and depth are crucial for efficiently solving many real-world problems, overcoming this obstacle is vital.

Recurrent Neural Networks

Recurrent neural networks (RNNs) are networks of interconnected units, i.e., neurons, in which their network state at any point in time is a function of both external input and the network's state at the previous time point, as shown in Figure S1. More precisely, the dynamics of a network with L layers is given by:

$$\mathbf{y}^{(l)}[n] = \sigma(\mathbf{a}^{(l)}[n]) \quad \text{for } l = 1, \ldots, L,$$
$$\mathbf{a}^{(l)}[n] = \mathbf{V}^{(l)}\mathbf{y}^{(l)}[n-1] + \mathbf{W}^{(l)}\mathbf{y}^{(l-1)}[n-1] \quad \text{for } l = 1, \ldots, L,$$
$$\mathbf{y}^{(0)}[n] \equiv \mathbf{x}[n],$$

where $\mathbf{a}^{(l)}[n]$ is the state vector of the neurons at layer l, σ is an activation function, and $\mathbf{V}^{(l)}$ and $\mathbf{W}^{(l)}$ are the recurrent and feedforward weight matrices of layer l, respectively. External inputs $\mathbf{x}[n]$ typically arrive at the first layer. Nonscalar quantities are typeset in boldface.

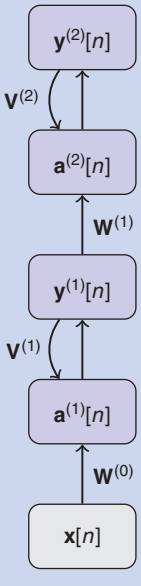

FIGURE S1. One popular RNN structure arranges neurons in multiple layers, where every layer is recurrently connected and also receives input from the previous layer.

As network models grow larger and make their way into embedded and automotive applications, their power efficiency becomes increasingly important. Simplified NN architectures that can run natively and efficiently on dedicated hardware are now being devised. This includes, e.g., networks of binary neurons or neuromorphic hardware that emulate the dynamics of SNNs [11]. Both types of networks dispense with energetically costly floating-point multiplications, making them particularly advantageous for low-power applications compared to NNs executed on conventional hardware.

These new hardware developments have created an imminent need for tools and strategies that enable efficient inference and learning in SNNs and binary RNNs. In this article, we discuss and address the inherent difficulties in training SNNs with hidden layers and introduce various strategies and approximations used to successfully implement them. (A repository containing tutorials for SG learning in SNNs can be found at: https://github.com/surrogate-gradient-learning.)

Understanding SNNs as RNNs

We begin by formally mapping SNNs to RNNs. Formulating SNNs as RNNs will allow us to directly transfer and apply existing training methods for RNNs and will serve as the conceptual framework for the rest of this article.

Before we proceed, we must make a note about terminology. We use the term *RNNs* in its widest sense to refer to networks whose state evolves in time according to a set of recurrent dynamical equations. Such dynamical recurrence can be because of the explicit presence of recurrent synaptic connections between neurons in the network. This is the common understanding of what an RNN is. But, importantly, dynamical recurrence can also arise in the absence of recurrent connections. This happens, e.g., when stateful neuron or synapse models, which have internal dynamics, are used. Because the network's state at a particular time step recurrently depends on its state in previous time steps, these dynamics are intrinsically recurrent. In this article, we use the term *RNN* for networks exhibiting either or both types of recurrence. Moreover, we introduce the term *recurrently connected NN* (*RCNN*) for the subset of networks that have explicit recurrent synaptic connections. We now describe the mathematical treatment of RCNNs, which closely resembles that of RNNs.

We first introduce an LIF neuron model with current-based synapses, which has wide use in computational neuroscience [12]. Next, we reformulate this model in discrete time and show its formal equivalence to an RNN with binary activation functions. Readers familiar with an LIF neuron model can skip the steps in "Recurrent Neural Networks" as well as (1)–(4), up to (5).

An LIF neuron in layer l with index i can formally be described in differential form as

$$\tau_{\text{mem}} \frac{\mathrm{d}U_i^{(l)}}{\mathrm{d}t} = -(U_i^{(l)} - U_{\text{rest}}) + RI_i^{(l)}, \tag{1}$$

where $U_i^{(l)}(t)$ is the membrane potential, U_{rest} is the resting potential, τ_{mem} is the membrane time constant, R is the input resistance, and $I_i(t)$ is the input current [12]. Equation

(1) shows that $U_i^{(l)}$ acts as a leaky integrator of the input current $I_i^{(l)}$. Neurons emit spikes to communicate their output to other neurons when their membrane voltage reaches the firing threshold ϑ. After each spike, the membrane voltage $U_i^{(l)}$ is reset to the resting potential U_{rest} (Figure 1). Due to this reset, (1) describes only the subthreshold dynamics of an LIF neuron, i.e., the dynamics in absence of spiking output of the neuron.

In SNNs, the input current is typically generated by synaptic currents triggered by the arrival of presynaptic spikes $S_j^{(l)}(t)$. When working with differential equations, it is convenient to denote a spike train $S_j^{(l)}(t)$ as a sum of Dirac delta functions $S_j^{(l)}(t) = \sum_{s \in C_j^{(l)}} \delta(t - s)$, where s runs over the firing times $C_j^{(l)}$ of neuron j in layer l.

Synaptic currents follow specific temporal dynamics themselves. A common first-order approximation is to model their time course as an exponentially decaying current following each presynaptic spike. Moreover, we assume that synaptic currents sum linearly. The dynamics of these operations are given by

$$\frac{\mathrm{d}I_i^{(l)}}{\mathrm{d}t} = \underbrace{-\frac{I_i^{(l)}(t)}{\tau_{\text{syn}}}}_{\text{exp. decay}} + \underbrace{\sum_j W_{ij}^{(l)} S_j^{(l-1)}(t)}_{\text{feedforward}} + \underbrace{\sum_j V_{ij}^{(l)} S_j^{(l)}(t)}_{\text{recurrent}}, \tag{2}$$

where the sum runs over all presynaptic neurons j and $W_{ij}^{(l)}$ are the corresponding afferent weights from the layer below. Further, the $V_{ij}^{(l)}$ corresponds to explicit recurrent connections within each layer. Because of this property, we can simulate a single LIF neuron with two linear differential equations whose initial conditions change instantaneously whenever a spike occurs. Through this property, we can incorporate the reset term in (1) through an extra term that instantaneously decreases the membrane potential by the amount $(\vartheta - U_{\text{rest}})$ whenever the neuron emits a spike:

$$\frac{\mathrm{d}U_i^{(l)}}{\mathrm{d}t} = -\frac{1}{\tau_{\text{mem}}}((U_i^{(l)} - U_{\text{rest}}) + RI_i^{(l)}) + S_i^{(l)}(t)(U_{\text{rest}} - \vartheta). \tag{3}$$

It is customary to approximate the solutions of (2) and (3) numerically in discrete time and to express the output spike train $S_i^{(l)}[n]$ of neuron i in layer l at time step n as a nonlinear function of the membrane voltage $S_i^{(l)}[n] \equiv \Theta(U_i^{(l)}[n] - \vartheta)$, where Θ denotes the Heaviside step function and ϑ corresponds to the firing threshold. Without loss of generality, we set $U_{\text{rest}} = 0$, $R = 1$, and $\vartheta = 1$. When using a small simulation time step $\Delta_t > 0$, (2) is well approximated by

$$I_i^{(l)}[n+1] = \alpha I_i^{(l)}[n] + \sum_j W_{ij}^{(l)} S_j^{(l-1)}[n] + \sum_j V_{ij}^{(l)} S_j^{(l)}[n], \tag{4}$$

with the decay strength $\alpha \equiv \exp(-(\Delta_t/\tau_{\text{syn}}))$. Note that $0 < \alpha < 1$ for finite and positive τ_{syn}. Moreover, $S_j^{(l)}[n] \in \{0, 1\}$. We use n to denote the time step to emphasize the discrete dynamics. We can now express (3) as

$$U_i^{(l)}[n+1] = \beta U_i^{(l)}[n] + I_i^{(l)}[n] - S_i^{(l)}[n] \tag{5}$$

with $\beta \equiv \exp(-(\Delta_t/\tau_{\text{mem}}))$.

Equations (4) and (5) characterize the dynamics of an RNN. Specifically, the state of neuron i is given by the instantaneous synaptic currents $I_i^{(l)}$ and the membrane voltage $U_i^{(l)}$ (see "Recurrent Neural Networks"). The computations necessary to update the cell state can be unrolled in time, as is best illustrated by the computational graph shown in Figure 2.

We have now seen that SNNs constitute a special case of RNNs; however, we have not yet explained how their parameters are set to implement a specific computational function. This is the focus of the rest of this article, in which we present a variety of learning algorithms that systematically change the parameters toward implementing specific functionalities.

Methods for training RNNs

Powerful machine-learning methods are able to train RNNs for a variety of tasks ranging from time-series prediction, to language translation, to automatic speech recognition [1]. In this section, we discuss the most common methods before analyzing their applicability to SNNs.

There are several common ingredients that define the training process in RNNs. The first ingredient is a cost or loss function, which is minimized when the network's response corresponds to the desired behavior. In time-series prediction, e.g., this loss could be the squared difference between the predicted and true values. The second ingredient is a mechanism that updates the network's weights to minimize the loss. One of the simplest and most powerful mechanisms used to achieve this is to perform gradient descent on the loss function. In network architectures with hidden units (i.e., units whose activity affect the loss indirectly through other units), the parameter updates contain terms that relate to the activity and weights of the downstream units they project to. Gradient-descent learning solves this credit assignment problem by providing explicit expressions for these updates through the chain rule of derivatives.

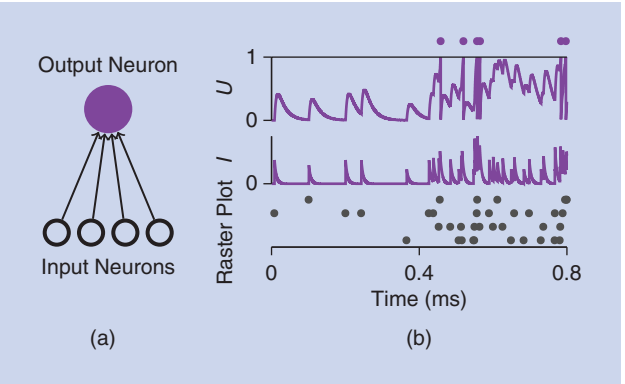

FIGURE 1. An example of LIF neuron dynamics. (a) The schematic of a network setup. Four input neurons connect to one postsynaptic neuron. (b) The input and output activity over time. At the bottom of (b) is the Raster plot, which shows activity of the four input neurons; in the middle is the synaptic current I; and at the top is the membrane potential U of the output neuron as a function of time, with output spikes shown as points. During the first 0.4 s, the dynamics are strictly "subthreshold," and individual postsynaptic potentials are clearly discernible. Only when multiple postsynaptic potentials start to sum up is the neuronal firing threshold (dashed) reached and output spikes generated.

As we will now see, the learning of hidden-unit parameters depends on an efficient method to compute these gradients. When discussing these methods, we distinguish between solving the spatial credit assignment problem, which affects multilayer perceptrons (MLPs) and RNNs in the same way, and the temporal credit assignment problem, which only occurs in RNNs. In the following section, we discuss the common algorithms that provide both types of credit assignment.

Spatial credit assignment

To train MLPs, credit or blame needs to be assigned spatially across the layers and their respective units. This spatial credit assignment problem is solved most commonly by the backpropagation (BP)-of-error algorithm (see "The Gradient Backpropagation Rule for Neural Networks"). In its simplest form, this algorithm propagates errors "backward" from the output of the network to upstream neurons. Using BP to adjust hidden-layer weights ensures that the weight update will reduce the cost function for the current training example, provided the learning rate is small enough. Although this theoretical guarantee is desirable, it comes at the cost of certain communication requirements, i.e., that gradients must be communicated back through the network, and increased memory requirements as the neuron states must be kept in memory until the errors become available.

Temporal credit assignment

When training RNNs, we also must consider the temporal interdependencies of network activity. This requires solving the temporal credit assignment problem shown in Figure 2. There are two common methods used to achieve this:

1) *The "backward" method*: This method applies the same strategies used for spatial credit assignment by "unrolling" the network in time (see "The Gradient Backpropagation Rule for Neural Networks"). BP through time (BPTT) solves the temporal credit assignment problem by backpropagating errors through the unrolled network. This method works backward through time after completing a forward pass. The use of standard BP on the unrolled network directly enables the use of autodifferentiation tools offered in modern machine-learning toolkits [3], [13].
2) *The forward method*: In some situations, it is beneficial to propagate all necessary information for gradient computation forward in time [14]. This formulation is achieved by computing the gradient of a cost function $\mathcal{L}[n]$ and maintaining the recursive structure of the RNN. For example, the "forward gradient" of the feedforward weight $\mathbf{W}$ becomes

$$\Delta W_{ij}^{[m]} \propto \frac{\partial \mathcal{L}[n]}{\partial W_{ij}^{[m]}} = \sum_k \frac{\partial \mathcal{L}[n]}{\partial y_k^{(L)}[n]} P_{ijk}^{[L],m}[n],$$

with

$$P_{ijk}^{(l,m)}[n] = \frac{\partial}{\partial W_{ij}^{[m]}} y_k^{(l)}[n]$$

$$P_{ijk}^{(l,m)}[n] = \sigma'(a_k^{(l)}[n])\Big(\sum_{j'} V_{ij'}^{(l)} P_{ij'j}^{(l,m)}[n-1] + \sum_{j'} W_{ij'}^{(l)} P_{ij'j}^{(l-1,m)}[n-1] + \delta_{lm} y_i^{(l-1)}[n-1]\Big). \quad (6)$$

Gradients, with respect to recurrent weights $V_{ij}^{(l)}$, can be computed in a similar fashion [14].

The backward optimization method is generally more efficient in terms of computation, but requires the maintaining of all inputs and activations for each time step. Thus, its space complexity for each layer is $O(NT)$, where N is the number of neurons per layer, and T is the number of time steps. Conversely, the forward method requires maintaining variables $P_{ijk}^{(l,m)}$, resulting in an $O(N^3)$ space complexity per layer. Although $O(N^3)$ is not a favorable scaling compared to $O(NT)$ for large N, simplifications of the computational graph can reduce the memory complexity of the forward method to $O(N^2)$ [2], [15], or even $O(N)$ [4]. These simplifications also reduce computational complexity, rendering the scaling of forward algorithms comparable to, or better than, BPTT. Such simplifications are at the core of several successful approaches, which we describe in the "Applications" section. Furthermore, the forward method is more appealing from a biological point of view, since the learning rule can be made consistent with synaptic plasticity in the brain and "three-factor" rules, as discussed in the "Supervised Learning With Local Three-Factor Learning Rules" section. In summary, efficient algorithms used to train RNNs exist. In the following section, we focus on training SNNs.

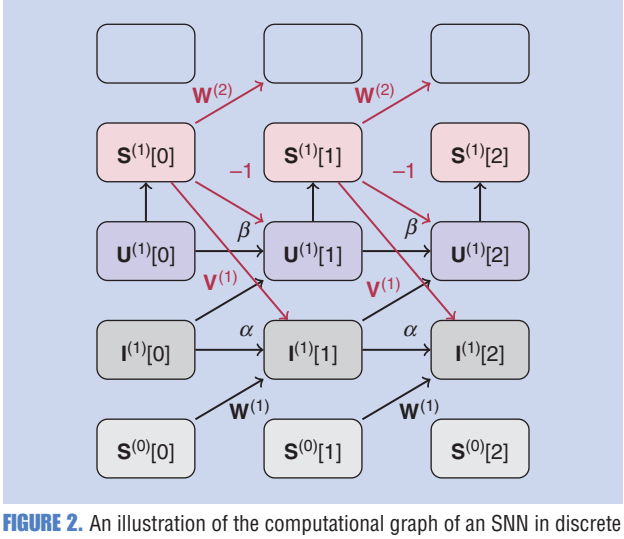

FIGURE 2. An illustration of the computational graph of an SNN in discrete time. The time steps flow from left to right. Input spikes $\mathbf{S}^{(0)}$ are fed into the network from the bottom and propagate upward to higher layers. The synaptic currents $\mathbf{I}^{(1)}$ are decayed by α in each time step and fed into the membrane potentials $\mathbf{U}^{(1)}$. The $\mathbf{U}^{(1)}$ are similarly decaying over time, as characterized by β. Spike trains $\mathbf{S}^{(1)}$ are generated by applying a threshold nonlinearity to the membrane potentials $\mathbf{U}^{(1)}$ in each time step. Spikes causally affect the network state (red connections). First, each spike causes the membrane potential of the neuron that emits the spike to be reset. Second, each spike may be communicated to the same neuronal population via recurrent connections $\mathbf{V}^{(1)}$. Finally, it may also be communicated via $\mathbf{W}^{(2)}$ to another downstream network layer or, alternatively, a readout layer on which a cost function is defined.

Credit assignment with spiking neurons: Challenges and solutions

Thus far, we have discussed common algorithmic solutions used for training RNNs. Before these solutions can be applied

to SNNs, however, two key challenges need to be overcome. The first challenge concerns the nondifferentiability of the spiking nonlinearity. Equations (S2) and (6) reveal that the expressions for both the forward- and backward-learning methods contain the derivative of the neural activation function $\sigma' \equiv \partial y_i^{(l)}/\partial a_i^{(l)}$ as a multiplicative factor. For a spiking neuron, however, we have $S(U(t)) = \Theta(U(t) - \vartheta)$, whose derivative is zero everywhere except at $U = \vartheta$, where it is ill defined (see Figure 3). This all-or-nothing behavior of the binary spiking nonlinearity stops gradients from "flowing" and makes LIF neurons unsuitable for gradient-based optimization. The same issue occurs in binary neurons, and some of the solutions proposed in this section are inspired by methods first developed in binary networks [16], [17].

The second challenge concerns the implementation of the optimization algorithm itself. Standard BP can be expensive in terms of computation, memory, and communication and may be poorly suited to the constraints dictated by the hardware that implements it (e.g., a computer, brain, or neuromorphic device). Processing in dedicated neuromorphic hardware and, more generally, non-von Neumann computers may have specific locality requirements (see "Local Models of Computation"), which can complicate matters. On such hardware, the forward approach may therefore be preferable. In practice, however, the scaling of both methods ($O(N^3)$ and $O(NT)$) has proven unsuitable for many SNN models. For example, the size of the convolutional SNN models trained with BPTT for gesture classification [20] are graphics processing unit (GPU) memory bounded. Additional simplifying approximations that reduce the complexity of the forward method will be discussed in greater detail. In the following sections, we describe approximate solutions to these challenges that make learning in SNNs more tractable.

To overcome the first challenge in training SNNs, which is concerned with the discontinuous spiking nonlinearity, several approaches have been devised with varying degrees of success. The most common approaches can be coarsely classified into the following categories: 1) resorting to entirely biologically inspired local learning rules for the hidden units; 2) translating conventionally trained "rate-based" NNs to SNNs; 3) smoothing the network model to be continuously differentiable; or

The Gradient Backpropagation Rule for Neural Networks

The task of learning is to minimize a cost function $\mathcal{L}$ over the entire data set. In a neural network (NN), this can be achieved by gradient descent, which modifies the network parameters W in the direction opposite to the gradient

$$W_{ij} \leftarrow W_{ij} - \eta \Delta W_{ij}, \text{ where } \Delta W_{ij} = \frac{\partial \mathcal{L}}{\partial W_{ij}} = \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial W_{ij}},$$

with $a_i = \sum_j W_{ij} x_j$ as the total input to the neuron, y_i as the output of neuron i, and η as a small learning rate. The first term is the error of neuron i, and the second term reflects the sensitivity of the neuron output to changes in the parameter. In multilayer networks, gradient descent is expressed as the backpropagation (BP) of the errors starting from the prediction (output) layer to the inputs. Using superscripts $l = 0, \ldots, L$ to denote the layer (0 is input, L is output)

$$\frac{\partial}{\partial W_{ij}^{(l)}} \mathcal{L} = \delta_i^{(l)} y_j^{(l-1)}, \text{ where } \delta_i^{(l)} = \sigma'(a_i^{(l)}) \sum_k \delta_k^{(l+1)} W_{ik}^{\top,(l)}, \quad \text{(S1)}$$

where σ' is the derivative of the activation function, $\delta_i^{(L)} = \partial \mathcal{L}/\partial y_i^{(L)}$ is the error of output neuron i, and $y_i^{(0)} = x_i$ and $\top$ indicate the transpose.

This update rule is ubiquitous in deep learning and known as the *gradient BP algorithm* [1]. Learning is typically carried out in forward passes (evaluation of the NN activities) and backward passes (evaluation of δs).

The same rule can be applied to recurrent NNs. In this case, the recurrence is "unrolled," meaning that an auxiliary network is created by making copies of the network for each time step, as depicted in Figure S2. The unrolled network is simply a deep network with shared feedforward weights $\mathbf{W}^{(l)}$ and recurrent weights $\mathbf{V}^{(l)}$, on which the standard BP applies

$$\Delta W_{ij}^{(l)} \propto \frac{\partial}{\partial W_{ij}^{(l)}} \mathcal{L}[n] = \sum_{m=0}^{n} \delta_i^{(l)}[m]\, y_j^{(l-1)}[m], \quad \text{and}$$

$$\Delta V_{ij}^{(l)} \propto \frac{\partial}{\partial V_{ij}^{(l)}} \mathcal{L}[n] = \sum_{m=1}^{n} \delta_i^{(l)}[m]\, y_j^{(l)}[m-1]$$

$$\delta_i^{(l)}[m] = \sigma'(a_i^{(l)}[m]) \left(\sum_k \delta_k^{(l+1)}[m]\, W_{ik}^{\top,(l)} + \sum_k \delta_k^{(l)}[m+1]\, V_{ik}^{\top,(l)} \right). \quad \text{(S2)}$$

Applying BP to an unrolled network is referred to as *BP through time*.

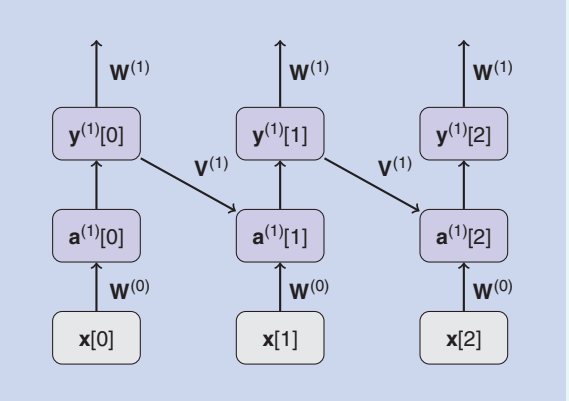

FIGURE S2. An "unrolled" recurrent NN.

4) defining an SG as a continuous relaxation of the real gradients. Approaches pertaining to biologically motivated local learning rules (i.e., category 1) and network translation (i.e., category 2) have been reviewed extensively [5], [21]. In this article, we therefore focus on the latter two supervised approaches (i.e., categories 3 and 4), which we will refer to as the *smoothed* and *SG approaches*, respectively. First, we review existing literature on common "smoothing" approaches before turning to an in-depth discussion of how to build functional SNNs using SG methods.

Smoothed SNNs

The defining characteristic of smoothed SNNs is that their formulation ensures well-behaved gradients, which are directly suitable for optimization. Smooth models can be further categorized into 1) soft nonlinearity models; 2) probabilistic models, for which gradients are well defined only in expectation, or models that either rely entirely on 3) rate; or (4) single-spike temporal codes.

Gradients in soft nonlinearity models

This approach can, in principle, be applied directly to all spiking neuron models, which explicitly include a smooth spike-generating process. This includes, e.g., the Hodgkin–Huxley, Morris–Lecar, and FitzHugh–Nagumo models [12]. In practice, this approach has been applied successfully only by Huh and Sejnowski [22], using an augmented IF model in which the binary spiking nonlinearity was replaced by a continuous-valued gating function. The resulting network constitutes an RCNN, which can be optimized using standard methods of BPTT or real-time recurrent learning (RTRL). Importantly, the soft threshold models compromise on one of the key features of SNNs, i.e., the binary spike propagation.

Gradients in probabilistic models

Another example of smooth models is binary probabilistic models. In simple terms, stochasticity effectively smooths out discontinuous binary nonlinearity, which makes it possible to define a gradient on expectation values. Binary probabilistic models have been objects of extensive study in machine-learning literature, mainly in the context of (restricted) Boltzmann machines [23]. Similarly, the propagation of gradients has been studied for binary stochastic models [17]. Probabilistic models are practically useful because the log-likelihood of a spike train is a smooth quantity, which can be optimized using gradient descent [24]. Although this insight was first discovered in networks without hidden units, the same ideas were later extended to multilayer networks [25]. Similarly, Guerguiev et al. [26] used probabilistic neurons to study biologically plausible ways of propagating error or target signals using segregated dendrites (see the "Feedback Alignment and Random Error BP" section). In a similar vein, variational learning approaches were shown to be capable of learning useful hidden-layer representations in SNNs [27]–[29]. However, the injected noise needed for smoothing out the effect of binary nonlinearities often poses a challenge for optimization [28]. How noise, which is found ubiquitously in neurobiology, influences learning in the brain remains an open question.

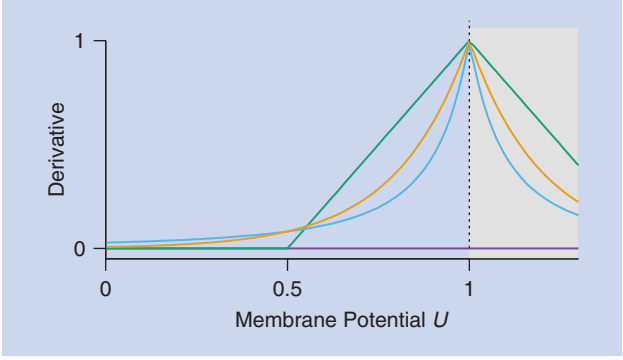

FIGURE 3. Commonly used surrogate derivatives. The step function has a zero derivative (violet) everywhere except at 0, where it is ill defined. The green (piecewise linear) [3], [18], [19], blue (derivative of a fast sigmoid) [2], and yellow (exponential) [13] lines are examples of surrogate derivatives that have been used to train SNNs. The gray shaded area is inaccessible due to the spike generation. Note that we have plotted absolute values and rescaled the axes on a per-function-basis for illustration purposes.

Local Models of Computation

The locality of computations is characterized by the set variables available to the physical processing elements and depends on the computational substrate. To illustrate the concept of locality, we assume two neurons, A and B, and would like neuron A to implement a function on domain D, defined as: $D = D_{\text{loc}} \cup D_{\text{nloc}}$, where $D_{\text{loc}} = \{W_{BA}, S_A(t), U_A(t)\}$.

Here, $S^B(t-T)$ refers to the output of neuron B, T seconds ago, U_A and U_B are the respective membrane potentials, and W_{BA} is the synaptic weight from B to A, as shown in Figure S3. Variables under D_{loc} are directly available to neuron A and are thus local to it.

Conversely, variable $S^B(t-T)$ is temporally nonlocal and U_B is spatially nonlocal to neuron A. Although locality in a model of computation can make its use challenging, it enables massively parallel computations with dynamical interprocess communications.

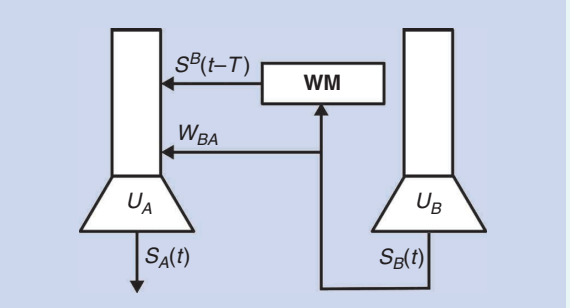

FIGURE S3. Nonlocal information can be transmitted through special structures, e.g., dedicated encoders and decoders for U_B and a form of working memory (WM) for $S^B(t-T)$.

Gradients in rate-coding networks

Another common approach to obtain gradients in SNNs is to assume a rate-based coding scheme. The main idea is that spike rate is the underlying information-carrying quantity. For many plausible neuron models, the suprathreshold firing rate depends smoothly on the neuron input. This input-output dependence is captured by the so-called f-I curve of a neuron. In such cases, the derivative of the f-I curves is suitable for gradient-based optimization.

There are several examples of this approach. For instance, Hunsberger and Eliasmith [30] as well as Neftci et al. [31] used an effectively rate-coded input scheme to demonstrate competitive performance on standard machine-learning benchmarks, such as CIFAR-10 and MNIST. Similarly, Lee et al. [32] demonstrated deep learning in SNNs by defining partial derivatives on low-pass filtered spike trains.

Rate-based approaches can offer good performance, but they may be inefficient. On the one hand, the precise estimation of firing rates requires averaging over a number of spikes. Such averaging requires either relatively high firing rates or long averaging times because several repeats are needed to average out discretization noise. This problem can be partially addressed by spatial averaging over large populations of spiking neurons. However, this may require the use of larger neuron numbers.

Finally, the distinction between rate coding and probabilistic networks can be blurry because many probabilistic network implementations use rate coding at the output level. Both types of models are differentiable, but for different reasons: Probabilistic models are based on a firing probability densities [24]. Importantly, the firing probability of a neuron is a continuous function. Although measuring probability changes requires "trial averaging" over several samples, it is the underlying continuity of the probability density that formally allows for defining differential improvements and thus, for deriving gradients. By exploiting this feature, probabilistic models have been used to learn precise output spike timing [24], [25]. In contrast, deterministic networks always emit a fixed-integer number of spikes for a given input. To nevertheless get at a notion of differential improvement, one may consider the number of spikes over a given time interval within single trials. When averaging over sufficiently large intervals, the resulting firing rates behave as a quasi-continuous function of the input current. This smooth input-output relationship is captured by the neuronal f-I curve, which can be used for optimization [30], [31]. Operating at the level of rates, however, comes at the expense of temporal precision.

Gradients in single-spike timing-coding networks

In an effort to optimize SNNs without potentially harmful noise injection and without reverting to a rate-based coding scheme, several studies have considered the outputs of neurons in SNNs to be a set of firing times. In such a temporal coding setting, individual spikes could carry significantly more information than rate-based schemes that consider only the total number of spikes in an interval.

The idea behind training temporal coding networks was pioneered in SpikeProp [33]. For this article, the analytic expressions of firing times for hidden units were linearized, allowing for the analytical computing of approximate hidden-layer gradients. More recently a similar approach, devoid of the need for linearization, was used in [34], where the author computed the spike-timing gradients explicitly for non-LIF neurons. Intriguingly, the work showed competitive performance on conventional networks and benchmarks.

Although the spike-timing formulation does, in some cases, yield well-defined gradients, it may suffer from certain limitations. For instance, the formulation of SpikeProp [33] required each hidden unit to emit exactly one spike per trial because it is impossible to define firing time for quiescent units. Ultimately, such a nonquiescence requirement could be in conflict with power efficiency, for which it is conceivably beneficial to, e.g., have only a subset of neurons active for any given task.

Surrogate gradients

SG methods provide an alternative approach for overcoming the difficulties associated with the discontinuous nonlinearity. Moreover, they offer opportunities to reduce the potentially high algorithmic complexity associated with training SNNs. Their defining characteristic is that, instead of changing the model definition as in the smoothed approaches, an SG is introduced. In this section, we make two distinctions. We first consider SGs, which constitute a continuous relaxation of the nonsmooth spiking nonlinearity for purposes of numerical optimization (Figure 4). Such SGs do not explicitly change the optimization algorithm itself and can be used, e.g., in combination with BPTT. Further, we also consider SGs with more profound changes that explicitly affect locality of the underlying optimization algorithms themselves to improve the computational and/or memory access overhead of the learning process. One example of this approach that we will discuss involves replacing the global loss by a number of local loss functions. Finally, the use of SGs allows for the efficient end-to-end training of SNNs without needing to specify which coding scheme is to be used in the hidden layers.

Similar to standard gradient-descent learning, SG learning can deal with the spatial and temporal credit assignment problem by either BPTT or forward methods, e.g., through the use of eligibility traces (see the "Methods for Training RNNs" section for details). Alternatively, additional approximations, which may offer advantages specifically for hardware implementations, can be introduced. In the following section, we briefly review existing work that relies on SG methods before focusing on a more in-depth treatment of the underlying principles and capabilities.

In the example in Figure 4(a), we linearly interpolated between the random initial and final (postoptimization) weight matrices of the hidden-layer inputs $\mathbf{W}^{(1)}$ (network details: two input, two hidden, and two output units trained on a binary classification task). Note that the loss function [gray in Figure 4(a)] displays characteristic plateaus with a zero gradient, which is detrimental for numerical optimization.

As shown in Figure 4(b), to perform numerical optimization in this network, we constructed an SG (violet) which, in contrast

to the true gradient (gray), is nonzero. Note that we obtained the "true gradient" via the finite differences method, which, in itself, is an approximation. Importantly, the SG approximates the true gradient but retains favorable properties for optimization, i.e., continuity and finiteness. The SG can be thought of as the gradient of a virtual surrogate loss function [the violet curve in (a) obtained by numerical integration of the SG and scaled to match loss at the initial and final points]. This surrogate loss remains virtual because it is generally not computed explicitly. In practice, suitable SGs are obtained directly from the gradients of the original network through sensible approximations. This is a key difference with respect to some other approaches [22], in which the entire network is replaced explicitly by a surrogate network on which gradient descent can be performed using its true gradients.

Surrogate derivatives for the spiking nonlinearity

A set of works have used SG to specifically overcome the challenge of discontinuous spiking nonlinearity. In these works, typically, a standard algorithm such as BPTT is used with one minor modification: within the algorithm, each occurrence of the spiking nonlinearity derivative is replaced by the derivative of a continuously differentiable function. Implementing these approaches is straightforward in most autodifferentiation-enabled machine-learning toolkits.

One of the first uses of such an SG is described by Bohte in [19], where the derivative of a spiking neuron nonlinearity was approximated by the derivative of a truncated quadratic function, thus resulting in a rectifying linear unit as the surrogate derivative, as shown in Figure 3. This is similar in spirit to the solution proposed to optimize binary NNs [16]. The same idea underlies the training of large-scale convolutional networks with binary activations on classification problems using neuromorphic hardware [18]. Zenke and Ganguli [2] proposed a three-factor online learning rule using a fast sigmoid to construct an SG. Shrestha and Orchard [13] used an exponential function and reported competitive performance on a range of neuromorphic benchmark problems. Additionally, O'Connor et al. [35] described a spike-based encoding method inspired by sigma-delta modulators. They used their method to approximately encode both the activations and errors in standard feedforward artificial NNs (ANNs), and apply standard BP on these sparse-approximate encodings.

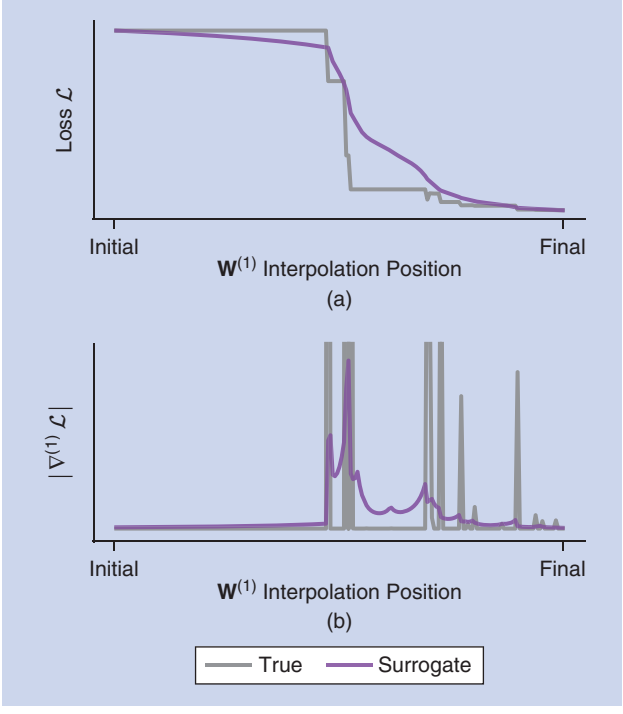

FIGURE 4. An empirical comparison of gradients and SG in an SNN. (a) The value of the loss function (gray) of an SNN classifier along an interpolation path over the hidden-layer parameters $\mathbf{W}^{(1)}$. (b) The absolute value of the hidden-layer SG along the interpolation path.

Surrogate derivatives have also been used to train spiking RCNNs, where dynamical recurrence arises due to the use of LIF neurons as well as recurrent synaptic connections. Recently, Bellec et al. [3] successfully trained RCNNs with slow temporal neuronal dynamics using a piecewise linear surrogate derivative. Encouragingly, the authors found that such networks can perform on par with conventional long short-term memory networks. Similarly, Woźniak et al. [36] reported comparable performance on a series of temporal benchmark data sets.

In summary, a plethora of studies have constructed SG using different nonlinearities and trained a diversity of SNN architectures. These nonlinearties, however, have a common underlying theme: All functions are nonlinear and monotonically increase toward the firing threshold, as shown in Figure 3. Although a more systematic comparison of different surrogate nonlinearities is still pending, overall, the diversity found in current literature suggests that the success of the method is not crucially dependent on the details of the surrogate used to approximate the derivative.

Surrogate gradients that affect the locality of update rules

The majority of studies discussed in the previous section introduced a surrogate nonlinearity to prevent gradients from vanishing (or exploding), but, by relying on methods such as BPTT, they did not explicitly affect the structural properties of the learning rules. There are, however, training approaches for SNNs that introduce more far-reaching modifications, which may completely alter the way error or target signals are propagated (or generated) within the network. Such approaches are typically used in conjunction with the aforementioned surrogate derivatives. There are two main motivations for such modifications, which are typically linked to physical constraints that make it impossible to implement the "correct" gradient-descent algorithm. For instance, in neurobiology, biophysical constraints make it impossible to implement BPTT without further approximations. Studies interested in how the brain could solve the credit assignment problem focus on how simplified "local" algorithms could achieve similar performance while adhering to the constraints of the underlying biological wetware (see "Local Models of Computation"). Similarly, neuromorphic hardware may pose certain constraints with regard to memory or communications, which impede the use of BPTT and call for simpler and often more local methods for training on such devices.

As training SNNs using SGs advances to deeper architectures, it is foreseeable that additional problems, similar to the ones encountered in ANNs, will arise. For example, several approaches currently rely on SGs derived from sigmoidal activation functions, as shown in Figure 3. The use of sigmoidal activation functions, however, is associated with vanishing gradient problems. Another set of challenges, which may need addressing in the future, could be linked to the bias that SGs introduce into the learning dynamics.

In the following section, we review a selection of promising SG approaches, which introduce far larger deviations from the "true gradients" and still allow for learning at a greatly reduced complexity and computational cost.

Applications

In this section, we present a selection of illustrative applications of smooth or SGs to SNNs, which exploit both the internal continuous-time dynamics of the neurons and their event-driven nature. The latter allows a network to remain quiescent until incoming spikes trigger activity.

Feedback alignment and random error BP

One family of algorithms that relaxes some of the requirements of BP is feedback alignment or, more generally, random BP algorithms [Figure 5(b) and (c)] [37]–[39]. These are approximations to the gradient BP rule that sidestep the nonlocality problem by replacing weights in the BP rule with random ones, as shown in Figure 5(b): $\delta_i^{(l)} = \sigma'(a_i^{(l)}) \sum_k \delta_k^{(l+1)} G_{ki}^{(l)}$, where $\mathbf{G}^{(l)}$ is a fixed, random matrix with the same dimensions as $\mathbf{W}$. The replacement of $\mathbf{W}^{\mathrm{T},(l)}$ with a random matrix $\mathbf{G}^{(l)}$ breaks the dependency of the backward phase on $\mathbf{W}^{(l)}$, enabling the rule to be more local. One common variation is to replace the entire backward propagation by a random propagation of the errors to each layer, as depicted in Figure 5(c) [38]: $\delta_i^{(l)} = \sigma'(a_i^{(l)}) \sum_k \delta_k^{(L)} H_{ki}^{(l)}$, where $\mathbf{H}^{(l)}$ is a fixed, random matrix with appropriate dimensions.

Random BP approaches lead to remarkably little loss in classification performance on some benchmark tasks. Although a general theoretical understanding of random BP is still a subject of intense research, simulation studies have shown that, during learning, the network adjusts its feedforward weights such that they partially align with the (random) feedback weights, thus permitting them to convey useful error information [37]. Building on these findings, an asynchronous spike-driven adaptation of random BP using local synaptic plasticity rules with the dynamics of spiking neurons was demonstrated in [31]. To obtain SGs, the authors approximated the derivative of the neural activation function using a symmetric function that is zero everywhere except in the vicinity of zero, where it is constant. Networks using this learning rule performed remarkably well, and were shown to operate continuously and asynchronously without the alternation between forward and backward passes, which is necessary in BP. One important limitation with random BP applied to SNNs was that the temporal dynamics of the neurons and synapses was not taken into account in the gradients. SuperSpike solves this problem.

Supervised learning with local three-factor learning rules

SuperSpike is a biologically plausible three-factor learning rule. In contrast to many existing three-factor rules that fall into the category of "smoothed approaches" [24]–[29], SuperSpike is an SG approach that combines several approximations to render it more biologically plausible [2]. Although the underlying motivation of the study is geared toward a deeper understanding of learning in biological NNs, the learning rule may prove interesting for hardware implementations because it is an online rule that does not require backpropagating error information through time. Specifically, the rule uses synaptic eligibility traces to solve the temporal credit assignment problem.

The SuperSpike learning rule is a forward-in-time optimization procedure that was derived for temporal supervised learning tasks in which a given output neuron learns to spike at predefined times. To that end, it minimizes the van Rossum distance with kernel λ between a set of output spike trains $S_i(t)$ and their corresponding target spike trains $S_i^*(t)$

$$\mathcal{L} = \frac{1}{2} \int_{-\infty}^{t} \sum_i (\lambda * (S_i[s] - S_i^*[s]))^2 ds \approx \frac{1}{2} \sum_{n,k} \underbrace{(\lambda * (S_i[n] - S_i^*[n]))^2}_{\equiv e_i^2[n]}, \tag{7}$$

where the last approximation corresponds to transitioning to discrete time. To avoid nonlocality, SuperSpike relies on a form of random BP to propagate error signals directly from the output layer to the hidden units. In deep networks, we expect this coarse approximation to cause problems for learning. In such cases, it may be important to compensate for layer-specific delays or to use entirely different approaches for credit assignment (compare the "Learning Using Local Errors" section). Because hidden layers use the same learning rule as the output layer, in the following section, we focus on a network without hidden layers to illustrate the online character of the rule.

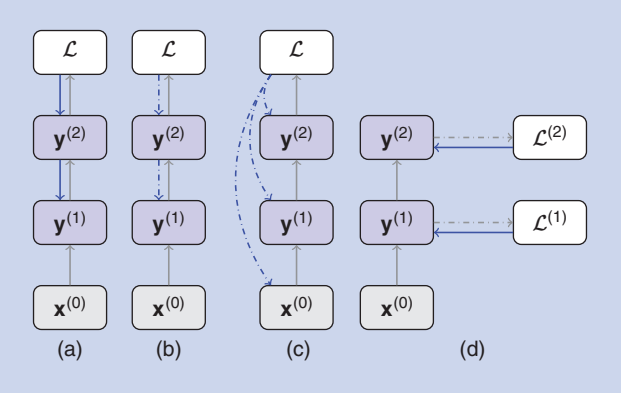

FIGURE 5. The strategies for relaxing gradient BP requirements. The dashed lines indicate fixed, random connections. (a) BP propagates errors through each layer using the transpose of the forward weights by alternating forward and backward passes. (b) FA [37] replaces the transposed matrix with a random one. (c) DFA [38] directly propagates the errors from the top layer to the hidden layers. (d) Local errors [29] uses a fixed, random, auxiliary cost function at each layer.

To perform online gradient descent on $\mathcal{L}$, we compute the gradients of the squared output error signals $e_i^2[n]$ at each time step n. Here, we first encounter the derivative $(\partial/\partial W_{ij})\lambda * S_i[n]$. Because the (discrete) convolution is a linear operator, this expression simplifies to $\lambda * (\partial S_i[n]/\partial W_{ij})$. To compute derivatives of the neuron's output spike train of the form $\partial S_i[n]/\partial W_{ij}$, we differentiate the network dynamics, i.e., (4) and (5), and obtain

$$\frac{\partial S_i[n+1]}{\partial W_{ij}} = \Theta'(U_i[n+1]-\vartheta)\left[\frac{\partial U_i[n+1]}{\partial W_{ij}}\right], \tag{8}$$

$$\frac{\partial U_i[n+1]}{\partial W_{ij}} = \beta\frac{\partial U_i[n]}{\partial W_{ij}} + \frac{\partial I_i[n]}{\partial W_{ij}} - \frac{\partial S_i[n]}{\partial W_{ij}}, \tag{9}$$

$$\frac{\partial I_i[n+1]}{\partial W_{ij}} = \alpha\frac{\partial I_i[n]}{\partial W_{ij}} + S_j[n]. \tag{10}$$

Equations (8)–(11) define a dynamical system which, given the starting conditions $S_i[0] = U_i[0] = I_i[0] = 0$, can be simulated online and forward in time to produce all relevant derivatives. Importantly, the convolution with λ is implemented similarly to (9) and (10) as a double integrator (see [2]). These equations are conceptually similar to those derived under RTRL, i.e., (6). Crucially, to arrive at useful SGs, SuperSpike makes two approximations. First, Θ' is replaced by a smooth surrogate derivative $\sigma'(U[n]-\vartheta)$ (compare Figure 3). Second, the reset term with the negative sign in (9) is dropped, which empirically leads to better results. With these definitions in hand, the final weight updates are given by

$$\Delta W_{ij}[n] \propto e_i[n]\lambda * \left[\sigma'(U_i[n]-\vartheta)\frac{\partial U_i[n]}{\partial W_{ij}}\right], \tag{11}$$

where $e_i[n] \equiv \lambda * (S_i - S_i^*)$. These weight updates depend only on local quantities and error signals (see "Local Models of Computation").

So far, in this section, we have considered a simple two-layer network (compare Figure 2) without recurrent connections. If we were to apply the same strategy to compute updates in an RCNN or a network with an additional hidden layer, the equations would become more complicated and nonlocal. SuperSpike, when applied to multilayer networks, sidesteps this issue by propagating error signals from the output layer directly to the hidden units, as in random BP (compare the "Feedback Alignment and Random Error BP" section) Figure 5(c), [37]–[39]. For networks with additional hidden layers, the output errors are simply broadcast through either random or structured weights A:

$$\Delta W_{ij}^{(l)}[n] \propto \left[\sum_k A_{ik}^{(l)} e_k[n]\right] \lambda * \left[\sigma'(U_i^{(l)}[n]-\vartheta)\frac{\partial U_i^{(l)}[n]}{\partial W_{ij}^{(l)}}\right]. \tag{12}$$

Thus, SuperSpike achieves temporal credit assignment by propagating all relevant quantities forward in time through eligibility traces defined by the neuronal dynamics [(9) and (10)], while it relies on random BP to perform spatial credit assignment.

Although the work by Zenke and Ganguli [2] was centered around feedforward networks, Bellec et al. [15] show that similar biologically plausible three factors rule can also be used to train RCNNs efficiently.

Learning using local errors

In practice, the performance of SuperSpike does not scale favorably for large multilayer networks. The scalability of SuperSpike can be improved by introducing local errors, as described in this section.

Multilayer NNs are hierarchical feature extractors. Through successive linear projections and pointwise nonlinearities, neurons become tuned (i.e., respond most strongly) to particular spatiotemporal features in the input. Although the best features are those that take into account the subsequent processing stages, and which, are learned to minimize the final error (as the features learned using BP do), high-quality features can also be obtained by more local methods. The nonlocal component of the weight update, i.e., (S1), is the error term $\delta_i^{(l)}[n]$. Rather than obtaining this error term through BP, it can be generated using information local to the layer. One way of achieving this is to define a layerwise loss $\mathcal{L}^{(l)}(y^{(l)}[n])$ and use this local loss to obtain the errors. In such a local learning setting, the local errors $\delta^{(l)}$ become

$$\delta_i^{(l)}[n] = \sigma'(a_i^{(l)}[n])\frac{d}{dy_i^{(l)}[n]}\mathcal{L}^{(l)}(\mathbf{y}^{(l)}[n]),$$

where

$$\mathcal{L}^{(l)}(\mathbf{y}^{(l)}[n]) \equiv \mathcal{L}(\mathbf{G}^{(l)}\mathbf{y}^{(l)}[n], \hat{\mathbf{y}}^{(l)}[n]), \tag{13}$$

with $\hat{\mathbf{y}}^{(l)}[n]$ a pseudotarget for layer l, and $\mathbf{G}^{(l)}$ a fixed random matrix that projects the activity vector at layer l to a vector having the same dimension as the pseudotarget. In essence, this formulation assumes that an auxiliary random layer is attached to layer l, with the goal of modifying $\mathbf{W}^{(l)}$ so as to minimize the discrepancy between the auxiliary random layer's output and the pseudotarget. The simplest choice for the pseudotarget is to use the top-layer target. This forces each layer to learn a set of features that can match the top-layer target after undergoing a fixed random linear projection. Each layer builds on the features learned by the layer below it, and we empirically observe that higher layers are able to learn higher-quality features that allow their random and fixed auxiliary layers to better match the target [40].

A related approach was explored with SNNs [41], where separate networks provided high-dimensional temporal signals that improve learning. Local errors were recently used in SNNs in combination with the SuperSpike (compare the "Supervised Learning With Local Three-Factor Learning Rules" section) forward method to overcome the temporal credit assignment problem [4]. As in SuperSpike, the SNN model is simplified by using a feedforward structure and by omitting the refractory dynamics in the optimization; however, the cost function was defined to operate locally on the instantaneous rates of each layer. This simplification results in a forward method whose space complexity scales as $O(N)$ [rather than $O(N^3)$

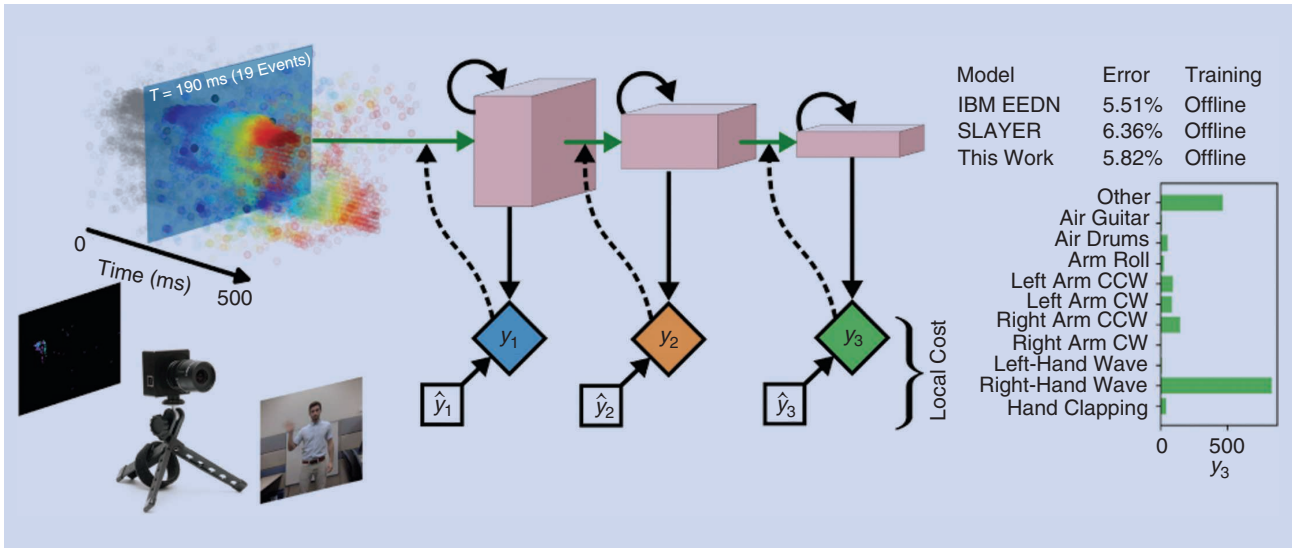

FIGURE 6. DECOLLE with spikes [4] applied to the event-based DVS Gestures data set. The feedforward weights (green) of a three-layer convolutional SNN are trained with SG; local errors are generated using fixed random projections onto a local classifier. Learning in DECOLLE scales linearly with the number of neurons, thanks to local rate-based cost functions formed by spike-based basis functions. The circular arrows indicate recurrence due to the statefulness of the LIF dynamics (no recurrent synaptic connections were used here) and are not trained. This SNN outperforms BPTT methods [13] and requires fewer training iterations [4] compared to other approaches. SLAYER: Spike Layer Error Reassignment [20]; EEDN: energy-efficient deep neuromorphic networks.

for the forward method, $O(N^2)$ for SuperSpike, or $O(NT)$ for the backward method], while still making use of spiking neural dynamics. Thus, the method constitutes a highly efficient synaptic plasticity rule for multilayer SNNs. Furthermore, the simplifications enable the use of existing automatic differentiation methods in machine-learning frameworks to systematically derive synaptic plasticity rules from task-relevant cost functions and neural dynamics (see [4] and included tutorials), thereby making deep continuous local learning (DECOLLE) easy to implement. This approach was benchmarked on the dynamic vision sensor (DVS) Gestures data set (Figure 6), and performs on par with standard BP or BPTT rules.

Learning using gradients of spike times

Difficulties in training SNNs stem from the discrete nature of the quantities of interest, such as the number of spikes in a particular interval. The derivatives of these discrete quantities are zero almost everywhere, which necessitates the use of SG methods. Alternatively, we can choose to use spike-based quantities that have well-defined, smooth derivatives. One such quantity is spike times. This capitalizes on the continuous-time nature of SNNs and results in highly sparse network activity, as the emission time of even a single spike can encode significant information. Just as importantly, spike times are continuous quantities that can be made to depend smoothly on the neuron's input. Working with spike times is thus a complementary approach to SG and achieves the same goal: obtaining a smooth chain of derivatives between the network's outputs and inputs. For this example, we use nonleaky neurons described by

$$\frac{\mathrm{d}U_i}{\mathrm{d}t} = I_i \quad \text{with} \quad I_i = \sum_j W_{ij} \sum_r \Theta(t - t_j^r) \exp(-(t - t_j^r)), \quad (14)$$

where t_j^r is the time of the rth spike from neuron j and Θ is the Heaviside step function.

Consider the simple exclusive or problem in the temporal domain: a network receives two spikes, one from each of two different sources. Each spike can either be "early" or "late." The network must learn to distinguish between the case in which the spikes are either both early or both late, and the case where one spike is early and the other is late, as shown in Figure 7(a). When designing an SNN, there is significant freedom in how the network input and output are encoded. In this case, we use a first-to-spike code in which we have two output neurons, and the binary classification result is represented by the output neuron that spikes first. Figure 7(b) shows the network's response after training (see [34] for details on the training process). For the first input class (early/late or late/early), one output neuron spikes first, and for the other class (early/early or late/late), the other output neuron spikes first.

Conclusions

We have outlined how discrete-time SNNs can be studied within the framework of RNNs and discussed successful approaches for training them. We have specifically focused on SG approaches for two reasons: SG approaches are able to train SNNs to perform at unprecedented performance levels on a range of real-world problems. This transition marks the beginning of an exciting time in which SNNs will garner increasing interest for applications that were previously dominated by nonspiking RNNs; SGs provide a framework that ties together ideas from machine learning, computational neurosciences, and neuromorphic computing. We emphasize that, although SGs are well defined in the discrete-time

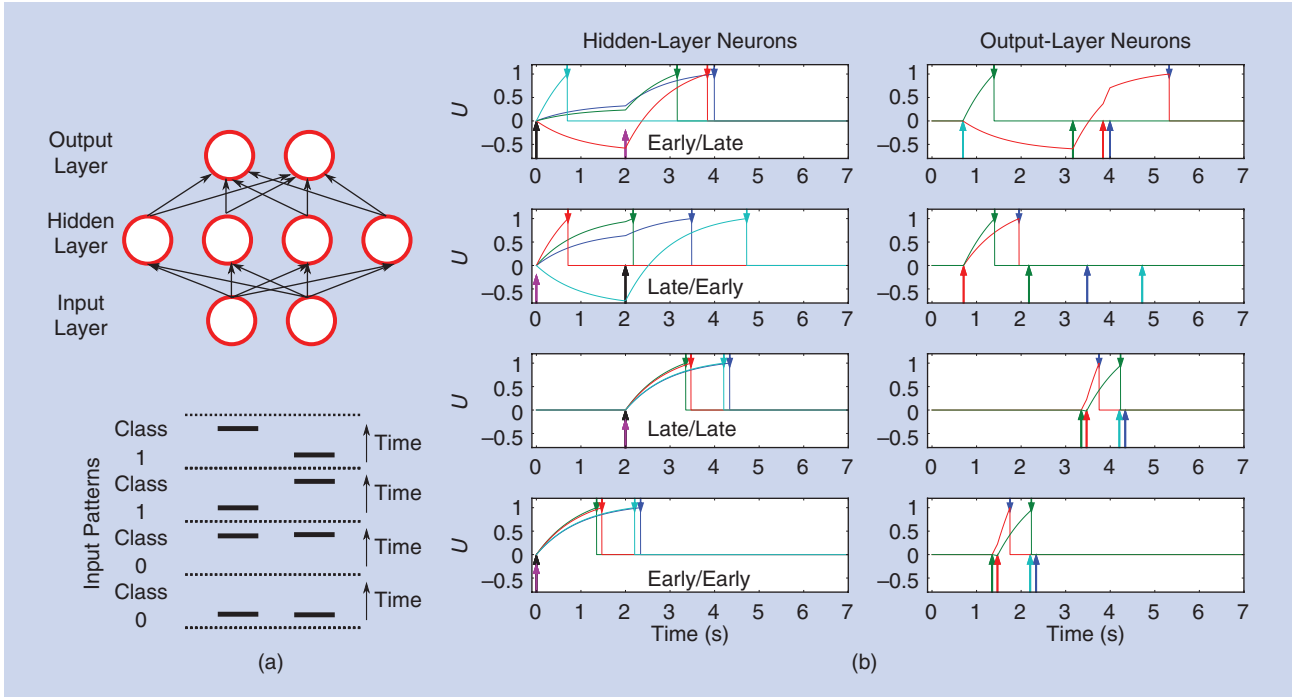

FIGURE 7. The temporal exclusive or problem. (a) An SNN with one hidden layer. Each input neuron emits one spike, which can either be late or early, resulting in four possible input patterns that should be classified into two classes. (b) For the four input spike patterns (one per row), the right plots show the membrane potentials of the two output neurons, while the left plots show the membrane potentials of the four hidden neurons. Arrows at the top of the plot indicate output spikes from the layer, while arrows at the bottom indicate input spikes. The output spikes of the hidden layer are the input spikes of the output layer. The classification result is encoded in the identity of the output neuron that spikes first.

framework studied in this article, the theoretical foundations of SGs for SNNs remain an open problem, including the generalization of spike-based BPTT to continuous-time dynamics and the optimal choice of smooth activation functions. From the viewpoint of computational neuroscience, the approaches presented in this article are appealing because several of them are related to "three-factor" plasticity rules, which are an important class of rules believed to underlie synaptic plasticity in the brain. Finally, for the neuromorphic community, SG methods provide a way to learn under various constraints on communication and storage, which makes SG methods highly relevant for learning on customized, low-power neuromorphic devices.

The spectacular successes of modern ANNs were enabled by algorithmic and hardware advances that made it possible to efficiently train large ANNs on vast amounts of data. With temporal coding, SNNs are universal function approximators that are potentially far more powerful than ANNs with sigmoidal nonlinearities. Unlike large-scale ANNs, which had to wait for several decades until the necessary computational resources were available for training them, we currently have the necessary resources, whether in the form of mainstream compute devices such as CPUs or GPUs, or custom neuromorphic devices, to train and deploy large SNNs. The fact that SNNs are less widely used than ANNs is thus primarily due to the algorithmic issue of trainability. In this article, we provided an overview of various exciting developments that are gradually addressing the issues encountered when training SNNs. Fully addressing these issues would have immediate and wide-ranging implications, both technologically and in relation to learning in biological brains.

Acknowledgments

This work was supported by the Intel Corporation (to Emre Neftci), the National Science Foundation under grant 1640081 (to Emre Neftci), the Swiss National Science Foundation Early Postdoc Mobility Grant P2ZHP2_164960 (to Hesham Mostafa), and the Wellcome Trust [110124/Z/15/Z] (to Friedemann Zenke).

Authors

Emre O. Neftci (eneftci@uci.edu) received his M.Sc. degree in physics from École Polytechnique Fédérale de Lausanne, Switzerland, and his Ph.D. degree in neuroinformatics from the Institute of Neuroinformatics at the University of Zürich and ETH Zürich, in 2010. Currently, he is an assistant professor in the Department of Cognitive Sciences and Computer Science at the University of California, Irvine. His current research explores the bridges between neuroscience and machine learning, with a focus on the theoretical and computational modeling of learning algorithms that are best suited to neuromorphic hardware and non-von Neumann computing architectures. He is a Member of the IEEE.

Hesham Mostafa (hesham.mostafa@intel.com) received his M.Sc. degree in electrical engineering from the Technical

University of Munich, Germany, in 2010 and his Ph.D. degree in neuroinformatics from the Institute of Neuroinformatics at the University of Zürich and ETH Zürich in 2016. Currently, he is a research scientist in the office of the CTO with Intel's Artificial Intelligence Products Group. His research interests include combining ideas from machine learning and computational neuroscience for developing biologically inspired and hardware-efficient learning and optimization algorithms, and physically implementing these algorithms using CMOS and novel device technologies.

Friedemann Zenke (friedemann.zenke@fmi.ch) received his diploma in physics from the University of Bonn, Germany, and the Australian National University, Canberra, in 2009 and received his Ph.D. degree from the École Polytechnique Fédérale de Lausanne, Switzerland, on the interaction of synaptic and homeostatic plasticity in spiking neural network models, in 2014. Currently, he is a junior group leader at the Friedrich Miescher Institute for Biomedical Research, Basel, Switzerland. His research interests include studying learning in biologically inspired network models with a focus on deep credit assignment and unsupervised learning.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[2] F. Zenke and S. Ganguli, "SuperSpike: Supervised learning in multilayer spiking neural networks," *Neural Comput.*, vol. 30, no. 6, pp. 1514–1541, 2018.

[3] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, "Long short-term memory and learning-to-learn in networks of spiking neurons," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Red Hook, New York: Curran Associates, 2018, pp. 795–805.

[4] J. Kaiser, H. Mostafa, and E. Neftci, Synaptic plasticity for deep continuous local learning. 2018. [Online]. Available: arxiv:1811.10766

[5] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Netw.*, vol. 111, pp. 47–63, Mar. 2019.

[6] R. Gütig, "To spike, or when to spike?" *Curr. Opin. Neurobiol.*, vol. 25, pp. 134–139, Apr. 2014.

[7] R.-M. Memmesheimer, R. Rubin, B. Ölveczky, and H. Sompolinsky, "Learning precisely timed spikes," *Neuron*, vol. 82, no. 4, pp. 925–938, 2014.

[8] N. Anwani and B. Rajendran, "NormAD-normalized approximate descent based supervised learning rule for spiking neurons," in *Proc. IEEE Int. Joint Conf. Neural Networks (IJCNN)*, 2015, pp. 1–8. doi: 10.1109/IJCNN.2015.7280618.

[9] A. Gilra and W. Gerstner, "Predicting non-linear dynamics by stable local learning in a recurrent spiking neural network," *eLife*, vol. 6, Nov. 2017. doi: 10.7554/eLife.28295, [Online]. Available: https://elifesciences.org/articles/28295

[10] W. Nicola and C. Clopath, "Supervised learning in spiking neural networks with FORCE training," *Nat. Commun.*, vol. 8, Dec. 2017. doi: 10.1038/s41467-017-01827-3, [Online]. Available: https://www.nature.com/articles/s41467-017-01827-3

[11] K. Boahen, "A neuromorph's prospectus," *Comput. Sci. Eng.*, vol. 19, no. 2, pp. 14–28, 2017.

[12] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[13] S. B. Shrestha and G. Orchard, "SLAYER: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Red Hook, New York: Curran Associates, 2018, pp. 1419–1428.

[14] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, no. 2, pp. 270–280, 1989.

[15] G. Bellec, F. Scherr, E. Hajek, D. Salaj, R. Legenstein, and W. Maass, Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. 2019. [Online]. Available: arxiv:1901.09049

[16] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1. 2016. [Online]. Available: arxiv:1602.02830

[17] Y. Bengio, N. Léonard, and A. Courville, Estimating or propagating gradients through stochastic neurons for conditional computation. 2013. [Online]. Available: arxiv:1308.3432

[18] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry et al., "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proc. Nat. Acad. Sci.*, vol. 113, no. 41, pp. 11,441–11,446, 2016.

[19] S. M. Bohte, "Error-backpropagation in networks of fractionally predictive spiking neurons," in *Proc. Int. Conf. Artificial Neural Networks (ICANN)*, 2011, 60–68.

[20] S. B. Shrestha and G. Orchard, "SLAYER: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems*, vol. 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Red Hook, New York: Curran Associates, 2018, pp. 1412–1421. [Online]. Available: http://papers.nips.cc/paper/7415-slayer-spike-layer-error-reassignment-in-time.pdf

[21] L. F. Abbott, B. DePasquale, and R.-M. Memmesheimer, "Building functional networks of spiking model neurons," *Nat. Neurosci.*, vol. 19, no. 3, pp. 350–355, 2016.

[22] D. Huh and T. J. Sejnowski, "Gradient descent for spiking neural networks," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Red Hook, New York: Curran Associates, 2018, pp. 1440–1450.

[23] D. Ackley, G. Hinton, and T. Sejnowski, "A learning algorithm for Boltzmann machines," *Cogn. Sci.: A Multidisciplinary J.*, vol. 9, no. 1, pp. 147–169, 1985.

[24] J.-P. Pfister, T. Toyoizumi, D. Barber, and W. Gerstner, "Optimal spike-timing-dependent plasticity for precise action potential firing in supervised learning," *Neural Comput.*, vol. 18, no. 6, pp. 1318–1348, 2006.

[25] B. Gardner, I. Sporea, and A. Grüning, "Learning spatiotemporally encoded pattern transformations in structured spiking neural networks," *Neural Comput.*, vol. 27, no. 12, pp. 2548–2586, 2015.

[26] J. Guerguiev, T. P. Lillicrap, and B. A. Richards, "Towards deep learning with segregated dendrites," *eLife*, vol. 6, Dec. 2017. doi: 10.7554/eLife.22901, [Online]. Available: https://elifesciences.org/articles/22901

[27] J. Brea, W. Senn, and J.-P. Pfister, "Matching recall and storage in sequence learning with spiking neural networks," *J. Neurosci.*, vol. 33, no. 23, pp. 9565–9575, 2013.

[28] D. J. Rezende and W. Gerstner, "Stochastic variational learning in recurrent spiking networks," *Front. Comput. Neurosci.*, vol. 8, p. 38, April 2014. doi: 10.3389/fncom.2014.00038.

[29] H. Mostafa and G. Cauwenberghs, "A learning framework for winner-take-all networks with stochastic synapses," *Neural Comput.*, vol. 30, no. 6, pp. 1542–1572, 2018.

[30] E. Hunsberger and C. Eliasmith, Spiking deep networks with LIF neurons. 2015. [Online]. Available: arxiv:1510.08829

[31] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines," *Front. Neurosci.*, vol. 11, p. 324, June 2017.

[32] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Front. Neurosci.*, vol. 10, Nov. 2016. doi: 10.3389/fnins.2016.00508, [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnins.2016.00508/full

[33] S. M. Bohte, J. N. Kok, and H. La Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1–4, pp. 17–37, 2002.

[34] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 7, pp. 3227–3235, 2018.

[35] P. O'Connor, E. Gavves, and M. Welling, Temporally efficient deep learning with spikes. 2017. [Online]. Available: arxiv:1706.04159

[36] S. Woźniak, A. Pantazi, and E. Eleftheriou, Deep networks incorporating spiking neural dynamics. 2018. [Online]. Available: arxiv:1812.07040

[37] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nat. Commun.*, vol. 7, no. 1, 2016. doi: /10.1038/ncomms13276, [Online]. Available: https://www.nature.com/articles/ncomms13276

[38] A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Red Hook, New York: Curran Associates, 2016, pp. 1037–1045.

[39] P. Baldi and P. Sadowski, "A theory of local learning, the learning channel, and the optimality of backpropagation," *Neural Netw.*, vol. 83, pp. 51–74, Nov. 2016.

[40] H. Mostafa, V. Ramesh, and G. Cauwenberghs, "Deep supervised learning using local errors," *Front. Neurosci.*, vol. 12, p. 608, Aug. 2018.

[41] W. Nicola and C. Clopath, "Supervised learning in spiking neural networks with force training," *Nat. Commun.*, vol. 8, no. 1, 2017. doi: 10.1038/s41467, [Online]. Available: https://www.nature.com/articles/s41467-017-01827-3

SP