Surrogate Gradient Learning in Spiking Neural Networks

Emre O. Neftci[†], *Member, IEEE,* Hesham Mostafa[†], *Member, IEEE,* Friedemann Zenke[†]

[†] All authors contributed equally. The order of authors is arbitrary.

Abstract

A growing number of neuromorphic spiking neural network processors that emulate biological neural networks create an imminent need for methods and tools to enable them to solve real-world signal processing problems. Like conventional neural networks, spiking neural networks are particularly efficient when trained on real, domain specific data. However, their training requires overcoming a number of challenges linked to their binary and dynamical nature. This tutorial elucidates step-by-step the problems typically encountered when training spiking neural networks, and guides the reader through the key concepts of synaptic plasticity and data-driven learning in the spiking setting. To that end, it gives an overview of existing approaches and provides an introduction to surrogate gradient methods, specifically, as a particularly flexible and efficient method to overcome the aforementioned challenges.

I. Introduction

Biological spiking neural networks (SNNs) are evolution's highly efficient solution to the problem of sensory 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 neural networks (RNNs), a class of stateful neural networks whose internal states evolve with time (Box. 1), have proven highly effective at solving real-time pattern recognition and noisy time series prediction problems [1]. RNNs and biological neural networks 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 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 non-stationary 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 non-differentiable dynamics implied by the binary nature of their outputs. While a considerable body of work has successfully demonstrated training of two-layer SNNs [6–8], *i.e.* networks without hidden units, the ability to train deeper SNNs with hidden layers has remained a major obstacle. Because hidden units and depth are crucial to efficiently solve many real-world problems, overcoming this obstacle is vital.

As network models grow larger and make their way into embedded and automotive applications, their power efficiency becomes increasingly important. Simplified neural network architectures that can run natively and efficiently on dedicated hardware are now being devised. This includes, for instance, networks of binary neurons which can dispense with energetically costly floating-point multiplications, or neuromorphic hardware that emulate the dynamics of SNNs [9].

These new hardware developments have created an imminent need for tools and strategies enabling 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.

II. Understanding SNNs as RNNs

We start with a brief walk-through on how to formally map a SNN to a RNN. Formulating SNNs as RNNs will allow us to directly transfer and apply existing methods to train RNNs and will serve as the conceptual framework for the rest of this article. To this end, we will first introduce the LIF neuron model with current-based synapses which has wide use in computational neuroscience [10]. Next, we will reformulate this model in discrete time and show its formal equivalence to a RNN with binary activation functions. Readers familiar with the LIF neuron model can skip the following steps up to Equation (8).

A LIF neuron with index i can formally be described in differential form as

$$\tau_{\mathrm{mem}} \frac{\mathrm{d}U_i}{\mathrm{d}t} = -(U_i - U_{\mathrm{rest}}) + RI_i \tag{4}$$

where $U_i(t)$ 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 [10]. Equation (4) shows that U_i acts as a leaky integrator of the input current I_i. Neurons emit spikes to communicate their

Box 1: Recurrent neural networks

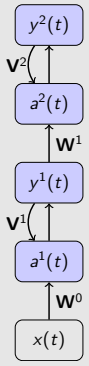
RNNs are networks of inter-connected units, or neurons, in which the network state at any point in time, $a[n]$ is a function of both external input $x[n]$ and the network's state at the previous time point $a[n-1]$. One popular RNN structure arranges neurons in multiple layers where every layer is recurrently connected and also receives input from the previous layer. More precisely, the dynamics of a network with L layers is given by:

$$y^l[n] = \sigma(\mathbf{a}^l[n]) \quad \text{for } l = 1, \ldots, L \tag{1}$$

$$a^l[n] = \mathbf{V}^l \mathbf{y}^l[n-1] + W^l \mathbf{y}^{l-1}[n-1] \quad \text{for } l = 1, \ldots, L \tag{2}$$

$$y^0[n] \equiv \mathbf{x}[n] \tag{3}$$

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



output to other neurons when their membrane voltage reaches the firing threshold ϑ. After each spike, the membrane voltage U_i is reset to the resting potential U_{rest} (Fig. 1). Due to this reset, Equation (4) only describes the subthreshold dynamics of a LIF neuron, *i.e.* the dynamics in absence of spiking output of the neuron.

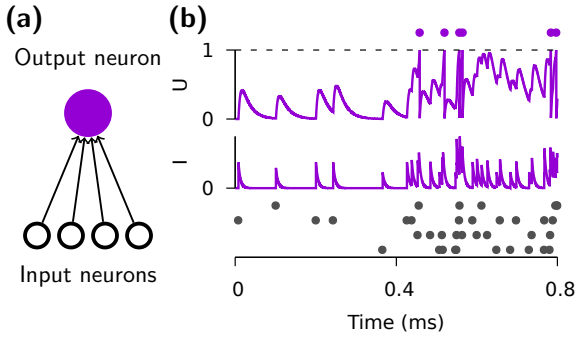


Fig. 1: Example LIF neuron dynamics. (**a**) Schematic of network setup. Four input neurons connect to one postsynaptic neuron. (**b**) Input and output activity over time. Bottom panel: Raster plot showing the activity of the four input neurons. Middle panel: The synaptic current I. Top panel: The membrane potential U of the output neuron as a function of time. Output spikes are shown as points at the top. During the first 0.4 s the dynamics are strictly "sub-threshold" and individual postsynaptic potentials (PSPs) are clearly discernible. Only when multiple PSPs start to sum up, the neuronal firing threshold (dashed) is reached and output spikes are generated.

In SNNs the input current is typically generated by synaptic currents triggered by the arrival of presynaptic spikes $S_j(t)$. When working with differential equations, it is convenient to denote a spike train $S_j(t)$ as a sum of Dirac delta functions $S_j(t) = \sum_k \delta(t - t_j^k)$ with the firing times t_j^k of neuron j.

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

given by

$$\frac{\mathrm{d}I_i}{\mathrm{d}t} = -\frac{I_i(t)}{\tau_{\mathrm{syn}}} + \sum_j W_{ij} S_j(t) \tag{5}$$

where the sum runs over all presynaptic neurons j and W_{ij} are the corresponding afferent weights. Neuron i has a self connection if the index i appears in this sum. 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.

We can incorporate the reset term in Equation (4) through an extra term that instantaneously pulls the membrane potential down by an amount $(\vartheta - U_{\mathrm{rest}})$ whenever the neuron emits a spike:

$$\frac{\mathrm{d}U_i}{\mathrm{d}t} = -\frac{1}{\tau_{\mathrm{mem}}}(U_i - U_{\mathrm{rest}}) + RI_i + S_i(t)(U_{\mathrm{rest}} - \vartheta) \tag{6}$$

It is customary to solve Equations (5) and (6) numerically in discrete time which allows us to interpret the spike train S_i of neuron i as a nonlinear function of the membrane voltage $S_i[n] \propto \Theta(U_i[n] - \vartheta)$ where Θ denotes the Heaviside step function. Without loss of generality, we set $U_{\mathrm{rest}} = 0$, $R = 1$, and $\vartheta = 1$. When using a small simulation time step $\Delta_t > 0$, Equation (5) is well approximated by

$$I_i[n+1] = \alpha I_i[n] + \sum_j W_{ij} S_j[n] \tag{7}$$

with the decay strength $\alpha \equiv \exp\left(-\frac{\Delta_t}{\tau_{\mathrm{syn}}}\right)$. Note that $0 < \alpha < 1$ for finite and positive τ_{syn}. Moreover, $S_j[n] \in \{0, 1\}$. We use n to denote the time step to emphasize the discrete dynamics. We can now express Equation (6) as

$$U_i[n+1] = \beta U_i[n] + I_i[n] - S_i[n] \tag{8}$$

with $\beta \equiv \exp\left(-\frac{\Delta_t}{\tau_{\mathrm{mem}}}\right)$. The output spikes of neuron i are simply given by $S_i[n] = \Theta(U_i[n])$.

Equations (7) and (8) characterize the dynamics of a RNN. Specifically, the state of neuron i is given by the instantaneous synaptic currents I_i and the membrane voltage U_i (Box. 1). The computations necessary to update the cell state can be unrolled in time as illustrated in the computational graph (Figure 2).

We have now seen that SNNs constitute a special case of RNNs. However, so far we have not explained how their parameters are set to implement a specific computational function. This is the focus of the rest of this tutorial, in which we present a variety of learning algorithms that systematically change the parameters towards implementing a specific function.

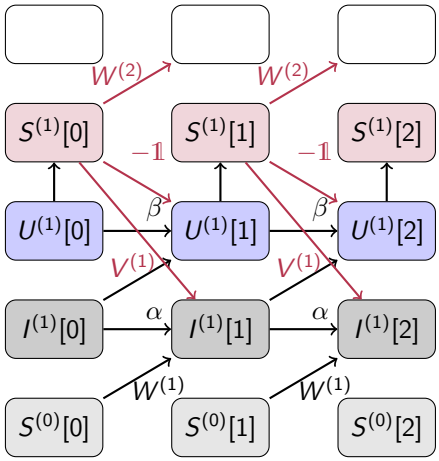


Fig. 2: Illustration of the computational graph of a SNN in discrete time. Time steps flow from left to right. Input spikes $S^{(0)}$ are fed into the network from the bottom and propagate upwards to higher layers. The synaptic currents I are decayed by α in each time step and fed into the membrane potentials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

III. 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. In the following, we discuss the most common methods before analyzing their applicability to SNNs.

There are several stereotypical ingredients that define the training process. 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, for example, this loss could be the squared difference between the predicted and the true value. The second ingredient is a mechanism that updates the network's weights to minimize the loss. One of the simplest and most powerful mechanisms 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 relating to the activity and weights of the units they project to (downstream units). Gradient-descent learning can solve this *credit assignment problem* by providing explicit expressions for these updates through the chain rule of derivatives. 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 multi-layer perceptrons (MLPs) and RNNs in the same way and the temporal credit assignment problem which only occurs in RNNs.

A. Spatial credit assignment

In MLPs, gradient descent can be implemented efficiently using the backpropagation of error algorithm (Box. 2). In its simplest form, this algorithm propagates errors "backwards" from the

output of the network to earlier (upstream) neurons.

Using backpropagation 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. While this theoretical guarantee is desirable, it comes at the cost of certain communication requirements — namely that gradients have to be communicated back through the network — and increased memory requirements as the neuron states need to be kept in memory until the errors become available.

Box 2: The Gradient Backpropagation Rule for Neural Networks

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

$$W_{ij} \leftarrow W_{ij} - \eta \Delta W_{ij}, \text{where } \Delta W_{ij} = \frac{\partial \mathcal{L}}{\partial W_{ij}} = \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial W_{ij}} \tag{9}$$

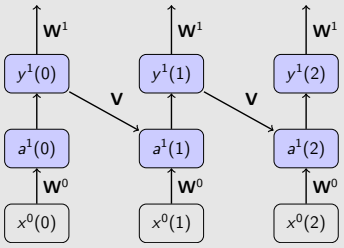
with $a_i = \sum_j W_{ij} x_j$ the total input to the neuron and η a small learning rate. The first term is the error of neuron i and the second term reflects the sensitivity of the neuron output to changes in the parameter. In multilayer networks, gradient descent is expressed as the backpropagation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

$$\frac{\partial}{\partial W_{ij}^l} \mathcal{L} = \delta_i^l y_j^{l-1}, \text{ where } \delta_i^l = \sigma'\left(a_i^l\right) \sum_k \delta_k^{l+1} W_{ik}^{\top,l}, \tag{10}$$

where the $\delta_i^L = \frac{\partial \mathcal{L}}{\partial y_i^L}$ is the error of output neuron i and $y_i^0 = x_i$ and $\top$ indicates the transpose.

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

The same rule can be applied to RNNs. In this case the recurrence is "unrolled" meaning that an auxiliary network is created by making copies of the network for each time step. The unrolled network is simply a deep network with shared feedforward weights W^l and recurrent weights V^l, on which the standard Backpropagation (BP) applies:



"Unrolled" RNN

$$\begin{aligned} \Delta W_{ij}^l &\propto \frac{\partial}{\partial W_{ij}^l} \mathcal{L}[n] = \sum_{m=0}^{t} \delta_i^l[m] y_j^{l-1}[m], \text{ and } \Delta V_{ij}^l \propto \frac{\partial}{\partial V_{ij}^l} \mathcal{L}[n] = \sum_{m=1}^{t} \delta_i^l[s] y_j^l[m-1] \\ \delta_i^l[n] &= \sigma'\left(a_i^l[n]\right) \left(\sum_k \delta_k^{l+1}[n] W_{ik}^{\top,l} + \sum_k \delta_k^l[n+1] V_{ik}^{\top,l} \right), \end{aligned} \tag{11}$$

Applying backpropagation to an unrolled network is referred to as backpropagation through time.

B. Temporal credit assignment

When training RNNs, we also have to consider temporal dependencies which requires solving a temporal credit assignment problem (Fig. 2). In the following we review two common methods to achieve this:

1) The "backward" method: This method applies the same strategies as with spatial credit assignment by "unrolling" the network in time (Box. 2). Backpropagation through time (BPTT) solves the temporal credit assignment problem by back propagating errors through the unrolled network. This method works backward through time after completing a forward update. The use of standard backpropagation on the unrolled network directly enables the use of autodifferentiation tools offered in modern machine learning toolkits [3, 11].
2) The forward method: In some situations it is beneficial to propagate all necessary information for gradient computation forward in time [12]. 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 feed-forward weight W becomes:

$$\Delta W_{ij}^m \propto \frac{\partial \mathcal{L}[n]}{\partial W_{ij}^m} = \sum_k \frac{\partial \mathcal{L}[n]}{\partial y_k^L[n]} P_{ijk}^L[n], \text{ with } P_{ijk}^l[n] = \frac{\partial}{\partial W_{ij}^m} y_k^l[n]$$
$$P_{ijk}^l[n] = \sigma'(a_k^l[n]) \left(\sum_{j'} V_{ij'}^l P_{ijj'}^l[n-1] + \sum_{j'} W_{ij'}^l P_{ijj'}^{l-1}[n-1] + \delta_{lm} y_i^{l-1}[n-1] \right). \tag{12}$$

Gradients with respect to recurrent weights V_{ij}^l can be computed in a similar fashion [12]. The backward optimization method is generally more efficient in terms of computation, but requires maintaining all the 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. On the other hand, the forward method requires maintaining variables P_{ijk}^l, resulting in a $O(N^3)$ space complexity per layer. This 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 section V-B. In summary, efficient algorithms to train RNNs exist. However, to apply them to SNNs we first need to overcome several key challenges which we will discuss next.

IV. Credit assignment with spiking neurons: Challenges and solutions

Before the solutions introduced above can be applied to SNNs, two key challenges need to be overcome. The first challenge is the non-differentiability of the spiking nonlinearity. Equations (11) and (12) reveal that the expressions for both the forward and the backward learning methods contain the derivative of the neural activation function $\sigma' \equiv \frac{\partial y_i^l}{\partial a_i^l}$ 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 (Fig. 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 here are inspired by the methods first developed in binary networks [13, 14].

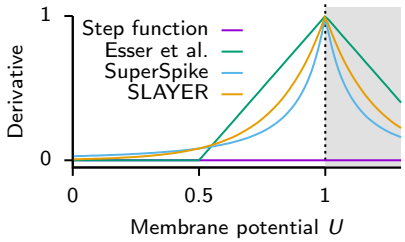


Fig. 3: *Commonly used surrogate derivatives*. The step function has zero derivative (violet) everywhere except at 0 where it is ill defined. Examples of surrogate derivatives which have been used to train SNNs. Green: Piece-wise linear [3, 15, 16]. Blue: Derivative of a fast sigmoid [2]. Yellow: Exponential [11]. Note that the axes have been rescaled on a per-function-basis for illustration purposes.

The second challenge concerns the implementation of the optimization algorithm itself. Standard backpropagation 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, a brain, or a neuromorphic device). As memory is ample in modern computers, the backward method is often preferred. On the other hand, processing in dedicated neuromorphic hardware and, more generally, non-von Neumann computers, may have specific locality requirements (Box. 3) that can complicate matters. On such hardware, the forward approach may therefore be preferable. In practice, however, the scaling of both methods has proven unsuitable for many SNN models and additional simplifying approximations are common. In the following sections, we describe solutions to these challenges and approximations that make learning in SNNs more tractable.

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

differentiable, or iv) defining a surrogate gradient as a continuous relaxation of the real gradients. Approaches pertaining biologically motivated local learning rules and network translation have been reviewed extensively in Tavanaei et al. [5]. In this tutorial we therefore focus on the latter two supervised approaches which we will refer to as the "smoothed" and the surrogate gradient (SG) approach. 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.

A. Smoothed spiking neural networks

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 (i) soft nonlinearity models, (ii) probabilistic models, for which gradients are only well defined in expectation, or models which either rely entirely on (iii) rate or (iv) single-spike temporal codes.

1) 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, for instance, the Hodgkin-Huxley, Morris-Lecar, and FitzHugh-Nagumo models [10]. In practice this approach has only been applied successfully by Huh and Sejnowski [17] using an augmented integrate-and-fire model in which the binary spiking nonlinearity was replaced by a continuous-valued gating function. The resulting network constitutes a RNN 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 SNN, namely the binary activation function.

2) Gradients in probabilistic models: Another example for smooth models are binary probabilistic models. In simple terms, stochasticity effectively smooths out the hard binary nonlinearity which makes it possible to define a gradient on expectation values. Binary probabilistic models have been objects of extensive study in the machine learning literature mainly in the context of (restricted) Boltzmann machines [18]. Similarly, the propagation of gradients has been studied for binary stochastic models [14].

Probabilistic models are practically useful because the log-likelihood of a spike train is a smooth quantity which can be optimized using gradient descent [19]. Although this insight was first discovered in networks without hidden units, the same ideas were later extended to multi-layer networks [20]. Similarly, Guerguiev et al. [21] used probabilistic neurons to study biologically

plausible ways of propagating error or target signals using segregated dendrites (see Section V-A). In a similar vein, variational learning approaches were shown to be capable of learning useful hidden layer representations in SNNs [22–24]. However, the injected noise necessary to smooth out the effect of binary nonlinearities often poses a challenge for optimization [23].

3) 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 supra-threshold 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 [25], Neftci et al. [26] used an effectively rate-coded input scheme to demonstrate competitive performance on standard machine learning benchmarks such as CIFAR10 and MNIST. Similarly Lee et al. [27] 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, 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 spatially averaging over large populations of spiking neurons, though this requires the use of more neurons.

4) 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 formulated SNNs as a set of firing times. In such a temporal coding setting, individual spikes could carry significantly more information than rate-based schemes.

The idea behind training temporal coding networks was pioneered in SpikeProp [28]. In this work the analytic expressions of firing times for hidden units were linearized, allowing to analytically compute approximate hidden layer gradients. More recently, a similar approach without the need of linearization was used in [29] where the author computed the spike timing gradients explicitly for non-leaky integrate-and-fire 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 [28] required each

hidden unit to emit exactly one spike per trial. Moreover, it is difficult to define firing time for quiescent units, although population sparseness is presumably crucial for power efficiency.

B. Surrogate gradients

SG methods provide an alternative approach for overcoming the difficulties associated with the "hard" nonlinearity when training SNNs. Their defining characteristic is that instead of changing the model definition as in the smoothed approaches, a SG is introduced as a continuous relaxation of the non-smooth spiking nonlinearity for purposes of numerical optimization (Fig. 4). Importantly, this approximation is only used in the parameter updates and does not affect the forward pass of the network directly. Moreover, the use of SGs allows to efficiently train SNNs end-to-end without the need to specify which coding scheme is to be used in the hidden layers.

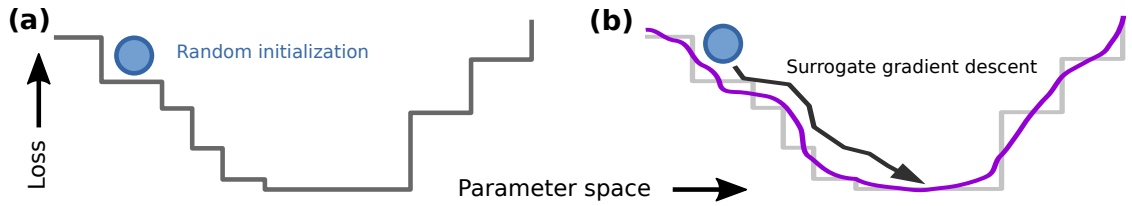


Fig. 4: *Illustration of surrogate gradient descent*. (**a**) Cartoon of a loss function with vanishing gradient almost everywhere. Consequently, also at the random initialization point (blue), the gradient is zero and gradient descent fails. (**b**) To rescue gradient descent it is often possible to find a surrogate loss (violet) and its corresponding SG which approximate the original problem but have favourable properties for optimization, i.e. smoothness and finite-valued.

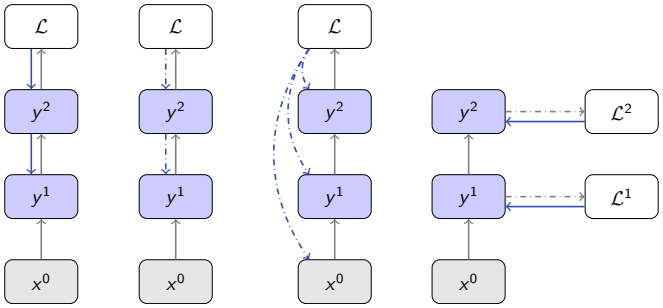
Like standard gradient-descent, SG learning can deal with the spatial and temporal credit assignment problem by either BPTT or by forward methods, e.g. through the use of eligibility traces (see Section III-B for details). Alternatively, additional approximations can be introduced which may offer advantages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

1) Surrogate derivatives for spiking nonlinearity: A set of works have used SG to specifically overcome the challenge of the "hard" nonlinearity. In these works typically a standard algorithm such as BPTT is used with one minor modification: within the algorithm each occurrence of the derivative of the spiking nonlinearity is replaced by the derivative of a smooth function. Implementing these approaches is possible in most auto-differentiation-enabled machine learning toolkits.

One of the first uses of such a SG is described in Bohte [16] where the derivative of a spiking neuron non-linearity was approximated by the derivative of a truncated quadratic function, thus resulting in a rectifying linear unit (ReLU) as surrogate derivative (Fig. 3). This is similar in flavor to the solution proposed to optimize binary neural networks [13]. The same idea underlies the training of large-scale convolutional networks with binary activations on classification problems using neuromorphic hardware [15]. Zenke and Ganguli [2] proposed a three factor online learning rule using the negative part of a fast sigmoid to construct a SG. Recently, Bellec et al. [3] successfully trained networks of neurons with slow temporal dynamics using the same surrogate derivative. Encouragingly, the authors found that such networks can perform on par with conventional long short-term memory (LSTM) networks. Finally, Shrestha and Orchard [11] used an exponential function and reported competitive performance on a range of neuromorphic benchmark problems.

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 increasing towards the firing threshold (Fig. 3). While a more systematic comparison of different surrogate nonlinearities is still pending, overall the diversity found in the present literature suggests that the success of the method is not crucially dependent on the details of the surrogate used to approximate the derivative.

2) Surrogate gradients affecting locality of the 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 which introduce more far-reaching modifications which may completely alter the way error signals 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 exact "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 biophysical constraints (Box. 3). Similarly, neuromorphic hardware may pose certain



(a) BP (b) FA (c) DFA (d) Local Errors

Fig. 5: *Strategies for relaxing gradient BP requirements*. 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) Feedback Alignment [30] replaces the transposed matrix with a random one. (c) Direct Feedback Alignment [31] directly propagates the errors from the top layer to the hidden layers. (d) Local errors [24] uses a fixed, random, auxiliary cost function at each layer.

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. In the following Applications Section, we will review several promising SG approaches which introduce far larger deviations from the "true gradients" and often allow learning at a greatly reduced complexity and computational cost.

V. Applications

In this section, we present a number of illustrative applications of smooth or surrogate gradients to SNNs which exploit a number of the unique features of SNNs, namely, their internal continuous-time dynamics that allow them to use time as a coding dimension; and their input-driven nature where the network stays quiescent until incoming inputs/spikes trigger activity in the network.

A. Feedback alignment and random error backpropagation

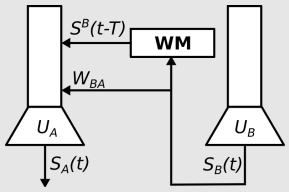
One family of algorithms that relaxes some of the requirements of BP are feedback alignment or, more generally, random BP algorithms [30–32]. These are approximations to the gradient BP rule that side-step the non-locality problem by replacing weights in the BP rule with random ones (Fig. 5b): $\delta_i^l = \sigma'\left(a_i^l\right)\sum_k \delta_k^{l+1} G_{ki}^l$, where G is a fixed, random matrix with the same dimensions as W_{ki}. The replacement of $W^{\top,l}$ with a random matrix G^l breaks the dependency of the backward phase on 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 (Fig. 5c) [31]: $\delta_i^l = \sigma'\left(a_i^l\right)\sum_k \delta_k^L H_{ki}^l$, where 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, extended simulations of linear networks show that, during learning, the network adjusts its feed-forward weights such that they partially align with the (random) feedback weights, thus permitting them to convey useful error information [30]. 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 [26]. To obtain the surrogate gradients, the authors approximated the derivative of the neural activation function using a boxcar function. Networks using this learning rule performed remarkably well, and were shown to operate continuously and asynchronously without the alternation between forward and backward passes that is necessary in BP.

Box 3: Local models of computation

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:

$$D = D_{loc} \cup D_{nloc}, \text{where } D_{loc} = \{W_{BA}, S_A(t), U_A(t)\}$$
$$\text{and } D_{nloc} = \{S_B(t-T), U_B\}.$$



Here, $S^B(t-T)$ refers to the output of neuron B T seconds ago, U_A, U_B are the respective membrane potentials, and W_{BA} is the synaptic weight from B to A. Variables under D_{loc} are directly available to Neuron A and are thus local to it.
On the other hand, variable $S^B(t-T)$ is temporally non-local and U_B is spatially non-local to neuron A. Non-local information can be transmitted through special structures, for example dedicated encoders and decoders for U_B and a form of working memory (WM) for $S_B(t-T)$. Although locality in a model of computation can make its use challenging, it enables massively parallel computations with dynamical interprocess communications.

B. Supervised learning with local three factor learning rules

SuperSpike is a biologically plausible three factor learning rule which makes use of several of the approximations introduced above [2]. Although the underlying motivation of the study is geared toward a deeper understanding of learning in biological neural networks, the learning rule may prove interesting for hardware implementations because it does not rely on BPTT. Specifically, the rule uses synaptic eligibility traces to solve the temporal credit assignment problem.

We now provide a short account on why SuperSpike can be seen as one of the forward-in-time optimization procedures. SuperSpike was derived for temporal supervised learning tasks in which a given output neuron learns to spike at predefined times. To that end, SuperSpike minimizes the van Rossum distance with kernel ϵ between a set of output spike train $S_k(t)$ and their corresponding target spike trains $S_k^*(t)$

$$\mathcal{L} = \frac{1}{2}\int_{-\infty}^{t} \mathcal{L}(s)\ ds = \frac{1}{2}\int_{-\infty}^{t} \left(\epsilon * \left(S_k(s) - S_k^*(s)\right)\right)^2 ds \approx \frac{1}{2}\sum_n \left(\epsilon * \left(S_k[n] - S_k^*[n]\right)\right)^2 \quad (13)$$

where the last approximation corresponds to transitioning to discrete time. To perform online gradient descent, we need to compute the gradients of $\mathcal{L}[n]$. Here we first encounter the derivative $\frac{\partial}{\partial W_{ij}}\epsilon * S_k[n]$. Because the (discrete) convolution is a linear operator, this expression simplifies to $\epsilon * \frac{\partial S_k[n]}{\partial W_{ij}}$. In SuperSpike ϵ is implemented as a dynamical system (see [2] for details). To compute derivatives of the neuron's output spiketrain of the form $\frac{\partial S_i[n]}{\partial W_{ij}}$ we differentiate the network dynamics (Equations (7) and (8)) and obtain

$$\frac{\partial S_k[n+1]}{\partial W_{ij}} = \Theta'(U_k[n+1] - \vartheta)\left[\frac{\partial U_k[n+1]}{\partial W_{ij}}\right] \quad (14)$$

$$\frac{\partial U_k[n+1]}{\partial W_{ij}} = \beta\frac{\partial U_k[n]}{\partial W_{ij}} + \frac{\partial I_k[n]}{\partial W_{ij}} - \frac{\partial S_k[n]}{\partial W_{ij}} \quad (15)$$

$$\frac{\partial I_k[n+1]}{\partial W_{ij}} = \alpha\frac{\partial I_k[n]}{\partial W_{ij}} + S_j[n] \quad (16)$$

The above equations define a dynamical system, which, given the starting conditions $S_k[0] = U_k[0] = I_k[0] = 0$, can be simulated online and forward in time to produce all relevant derivatives. Crucially, to arrive at useful surrogate gradients, SuperSpike makes two approximations. First, Θ' is replaced by a smooth surrogate derivative $\sigma'(U[n] - \vartheta)$ (cf. Fig. 3). Second, the reset term with the negative sign in Equation (15) is dropped, which empirically leads to better results. With these definitions in hand, the final weight updates are given by

$$\Delta W_{ij}[n] \propto e_i[n]\epsilon * \left[\sigma'(U_k[n])\frac{\partial U_k[n]}{\partial W_{ij}}\right] \quad (17)$$

where $e_i[n] \equiv \epsilon * (S_i - S_i^*)$. These weight updates depend only on local quantities (Box. 3).

Above, we have considered a simple two layer network (cf. Fig. 2). If we were to apply the same strategy to compute updates in a network with a hidden layer or with recurrent connections, the equations would become more complicated and non-local. To avoid introducing non-locality, SuperSpike propagates error signals from the output layer directly to the hidden

units like in random BP or *direct feedback alignment* (cf. Section V-A; Fig. 5c; [30–32]). With these approximations SuperSpike achieves temporal credit assignment by propagating all relevant quantities forward in time, while it relies on random BP to perform spatial credit assignment.

C. Encoding information in spike times

There are various ways of encoding information in spike streams. One particularly powerful encoding uses spike times as the information carrying quantities. 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. For this example, we use non-leaky integrate and fire neurons described by:

$$\frac{\mathrm{d}U_i}{\mathrm{d}t} = I_i \quad \text{with} \quad I_i = \sum_j W_{ij} \sum_r \Theta(t - t_i^r) \exp\left(-(t - t_i^r)\right) \tag{18}$$

where t_i^r is the time of the r^{th} spike from neuron j, and Θ is the Heaviside step function.

Consider the simple *exclusive or* (XOR) 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 has to 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 (Fig. 6a). When designing a 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 6b shows the network's response after training (see [29] 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.

D. Learning using local errors

Multi-layer neural networks are hierarchical feature extractors. Through successive linear projections and point-wise non-linearities, neurons become tuned (respond most strongly) to particular spatio-temporal features in the input. While 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 backpropagation do), high-quality features can also be obtained by more local methods. The non-local component of the weight update equation (Eq. (10)) is the

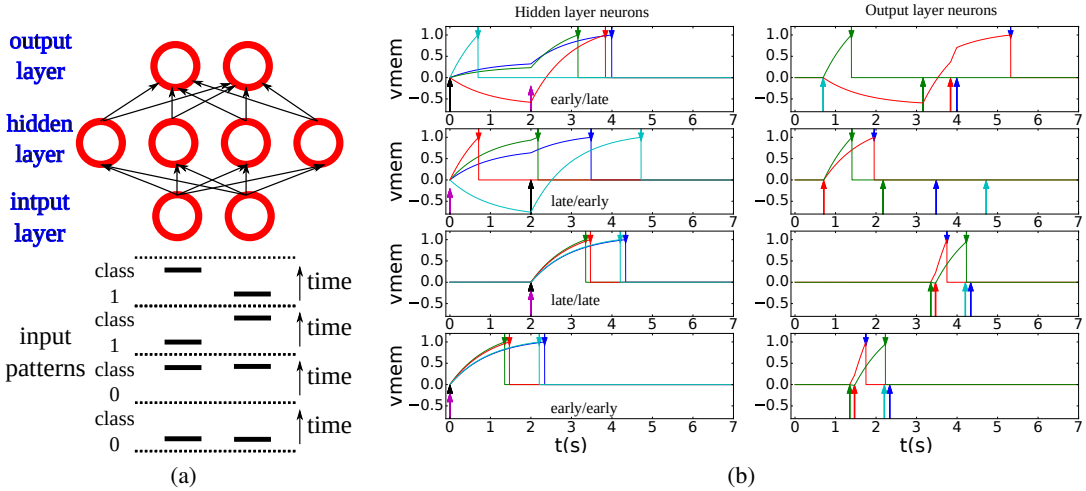


Fig. 6: *Temporal XOR problem*. (a) An SNN with one hidden layer. Each input neuron emits one spike which can either be late or early resulting in four possible input patterns that should be classified into two classes. (b) For the four input spike patterns (one per row), the right plots show the membrane 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.

.

error term $\delta_i^l[n]$. Instead of obtaining this error term through backpropagation, we require that it be generated using information local to the layer. One way of achieving this is to define a layer-wise 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 δ^l becomes:

$$\delta_i^l[n] = \sigma'\left(a_i^l[n]\right)\frac{\mathrm{d}}{\mathrm{d}y_i^l[n]}\mathcal{L}^l(y^l[n]) \quad where \quad \mathcal{L}^l(y^l[n]) \equiv \mathcal{L}(W_r^l y^l[n], \hat{y}^l[n]) \tag{19}$$

with $\hat{y}^l[n]$ a pseudo-target for layer l, and W_r^l a fixed random matrix that projects the activity vector at layer l to a vector having the same dimension as the pseudo-target. In essence, this formulation assumes that an auxiliary random layer is attached to layer l and the goal is to modify W^l so as to minimize the discrepancy between the auxiliary random layer's output and the pseudo-target. The simplest choice for the pseudo-target is to use the top-layer target. This forces each layer to learn a set of features that are able to 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 [33].

This approach was recently used in SNNs in combination with the SuperSpike (cf. Section V-B)

forward method to overcome the temporal credit assignment problem [4]. As in SuperSpike, the SNN model is simplified by using a feedforward structure, and 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 $O(N)$ (instead of $O(N^3)$ for the forward method 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 multi-layer SNNs. Furthermore, the simplifications enable the use of existing autodifferentation methods in machine learning frameworks to systematically derive synaptic plasticity rules from task-relevant cost functions and neural dynamics. This approach was benchmarked on the DVS Gestures dataset (Fig. 7), and performs on par with standard BP or BPTT rules.

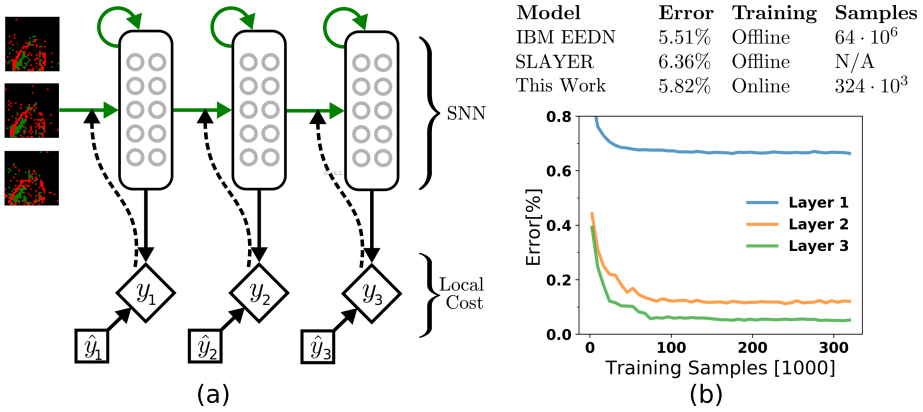


Fig. 7: Deep Continuous Local Learning (DCLL) with spikes [4], applied to the event-based DVSGestures dataset. (a) A three layer convolutional SNN is trained with SG using local errors generated using fixed random projections to a local classifier. Learning in DCLL scales linearly with the number of neurons thanks to local rate-based cost functions formed by spike-based basis functions. Thanks to the SG approach, the plasticity dynamics (dashed line) are synthesized with automatic differentiation under PyTorch. (b) This SNN outperforms BPTT methods [11], while requiring multiple orders of magnitude fewer training iterations (samples) compared to other approaches. (bottom) The network learns successively higher-quality features, allowing each layer to perform more accurate classification.

VI. Discussion

We have outlined how 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 unprecedented performance levels on a range of real-world problems. This transition marks the beginning of an exciting time in which SNNs will become increasingly interesting for applications which were previously dominated by

RNNs; SGs provide a framework that ties together ideas from machine learning, computational neurosciences, and neuromorphic computing. From the viewpoint of computational neuroscience, the approaches presented in this paper 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 custom low-power neuromorphic devices.

The spectacular successes of modern artificial neural networks (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 tutorial, 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 (EN); the National Science Foundation under grant 1640081 (EN); the swiss national science foundation early postdoc mobility grant P2ZHP2_164960 (HM) ; the Wellcome Trust [110124/Z/15/Z] (FZ).

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] F. Zenke and S. Ganguli, "SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks," *Neural Computation*, vol. 30, no. 6, pp. 1514–1541, Apr. 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. Curran Associates, Inc., 2018, pp. 795–805.

[4] J. Kaiser, H. Mostafa, and E. Neftci, "Synaptic plasticity for deep continuous local learning," *arXiv preprint arXiv:1812.10766*, 2018.

[5] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, Dec. 2018.

[6] R. Gütig, "To spike, or when to spike?" *Current Opinion in Neurobiology*, vol. 25, pp. 134–139, Apr. 2014.

[7] R.-M. Memmesheimer, R. Rubin, B. lveczky, and H. Sompolinsky, "Learning Precisely Timed Spikes," *Neuron*, vol. 82, no. 4, pp. 925–938, May 2014.

[8] N. Anwani and B. Rajendran, "NormAD-normalized approximate descent based supervised learning rule for spiking neurons," in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8.

[9] K. Boahen, "A neuromorph's prospectus," *Computing in Science Engineering*, vol. 19, no. 2, pp. 14–28, Mar. 2017.

[10] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[11] 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. Curran Associates, Inc., 2018, pp. 1419–1428.

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

[13] 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," *arXiv:1602.02830 [cs]*, Feb. 2016, arXiv: 1602.02830.

[14] Y. Bengio, N. Lonard, and A. Courville, "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation," *arXiv:1308.3432 [cs]*, Aug. 2013, arXiv: 1308.3432.

[15] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proc Natl Acad Sci U S A*, vol. 113, no. 41, pp. 11 441–11 446, Oct. 2016.

[16] S. M. Bohte, "Error-Backpropagation in Networks of Fractionally Predictive Spiking Neurons," in *Artificial Neural Networks and Machine Learning ICANN 2011*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jun. 2011, pp. 60–68.

[17] 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. Curran Associates, Inc., 2018, pp. 1440–1450.

[18] D. Ackley, G. Hinton, and T. Sejnowski, "A learning algorithm for Boltzmann machines," *Cognitive Science: A Multidisciplinary Journal*, vol. 9, no. 1, pp. 147–169, 1985.

[19] J.-P. Pfister, T. Toyoizumi, D. Barber, and W. Gerstner, "Optimal Spike-Timing-Dependent Plasticity for Precise Action Potential Firing in Supervised Learning," *Neural Computation*, vol. 18, no. 6, pp. 1318–1348, Apr. 2006.

[20] 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, Oct. 2015.

[21] J. Guerguiev, T. P. Lillicrap, and B. A. Richards, "Towards deep learning with segregated dendrites," *eLife Sciences*, vol. 6, p. e22901, Dec. 2017.

[22] 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, Jun. 2013.

[23] D. J. Rezende and W. Gerstner, "Stochastic variational learning in recurrent spiking networks," *Front. Comput. Neurosci*, vol. 8, p. 38, 2014.

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

[25] E. Hunsberger and C. Eliasmith, "Spiking deep networks with lif neurons," *arXiv preprint arXiv:1510.08829*, 2015.

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

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

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

[29] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 7, pp. 3227–3235, 2018.

[30] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature Communications*, vol. 7, 2016.

[31] 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. Curran Associates, Inc., 2016, pp. 1037–1045.

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

[33] H. Mostafa, V. Ramesh, and G. Cauwenberghs, "Deep supervised learning using local errors," *Frontiers in neuroscience*, vol. 12, p. 608, 2018.