Learning to Solve Real-World Problems with Spiking Neural Networks

Emre O. Neftci[†], *Member, IEEE,* Hesham Mostafa[†], *Member, IEEE,* Friedemann Zenke[†]
[†] All authors contributed equally. The order of authors is random.

Abstract

A growing number of neuromorphic spiking neural network (SNN) processors that emulate biological neural networks are creating an imminent need for methods and tools enabling them to solve real-world problems involving real-time signal processing. Like conventional neural networks, SNNs 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 challenges typically encountered when training SNNs, and guides the reader through the key concepts of synaptic plasticity and data-driven learning in the spiking setting. In so doing, it introduces surrogate gradient methods as a particularly flexible and efficient method to overcome the aforementioned challenges and discusses the “tricks of the trade”.

CONTENTS

* Shorten the introduction * Describe brain/spiking before the theoretical discussion on Turing machines.

I. Introduction

A plethora of signal processing problems are concerned with real-time pattern recognition or noisy time series prediction. RNNs prove highly effective [1] at solving such problems. RNNs are a class of stateful neural networks whose internal state evolves either in continuous time or through discrete time steps. RNNs are universal computing architectures that are able to reproduce the behavior of an arbitrary Turing machine [2]. The *program* implemented by an RNN, however, is described in a fundamentally different way from a conventional Turing machine. Instead of a state transition table or a sequence of instructions, the behavior of RNN is implicitly defined by the network weights and the equations governing the behavior of each neuron in the network. While this implicit definition is harder to parse and understand than conventional programs, it makes it possible to learn by tuning the network weights. Early on, Alan Turing recognized this potential of RNNs in a prescient paper [3] in which he described recurrent networks of logic gates where the connections between logic gates can be adapted to suit the task. Turing's paper was inspired by the most complex RNNs known to us: biological brains. While RNNs used for practical applications nowadays are different in many fundamental ways from biological networks, they both share the same general architecture and employ learning based on weight adjustment. The interest in RNNs as a brain-inspired methods stems from a growing body of work is establishing formal equivalences between recurrent neural networks, and a class of spiking neural networks that can be efficiently emulated in dedicated neuromorphic hardware [4–7]. It is thus natural to draw inspiration from biological networks when desigining RNNs, as the former were evolution's highly-optimized answer to the problem of sensory signal processing. With RNNs, two key challenges need to be addressed. First, before an RNN can be used to predict outcomes on new data, it needs to be trained. Training is the optimization procedure in which the the parameters or weights of the RNN are adjusted to approximate a specific function that the RNN is to perform. Training RNNs can be impeded by a variety of factors common for real world applications, e.g. noise and non-stationary of the data. Second, as models grow larger and make their way into embedded and automotive applications, their power efficiency is of increasing importance [8]. To tackle the latter problem, simplified neural network applications have been devised which are amenable to be run natively on dedicated hardware, such as neuromorphic hardware and neural network accelerators. For instance, one common approach are binary activation neural network implementations which can dispense with energetically costly floating-point multiplications. These form a special subset of RNNs with a binary activation functions and constitute an area of active study both due to their potential for power efficient computation and more fundamentally because SNNs are the dominant network model employed by the brain.

A body of work has focused on training spiking neural networks without hidden units [9–14], *i.e.* units which are neither directly connected to the input nor the output of the network. Properly training hidden units is necessary to solve a majority of complex, real-world problems. Their training however requires efficient solutions to the spatial and temporal credit assignment problems, and creates a set of specific challenges.

In this article we focus on these challenges and describe possible remedies by specifically considering the problem of supervised learning in SNNs.

II. Understanding SNNs as RNNs (FRIEDEMANN)

In this section we provide a brief walk-through how to formally map a SNN to RNN which will provide the basis to training a spiking neural network with surrogate gradients (SGs). To that end, we will first introduce the leaky integrate-and-fire (LIF) neuron model with current-based synapses as a standard neuron model which has wide use in computational neuroscience [16].

Next, we will reformulate this model in discrete time and show its formal equivalence to a discrete time RNN with binary activation function. Formulating SNNs as RNNs allows us to directly transfer and apply existing methods to train RNNs and will serve as the conceptional framework for the rest of this article. If you are not familiar with the LIF neuron model, you can skip most of the following derivations and up to Equation (6).

We begin by recalling that a single LIF neuron with index i can formally be described in differential form as

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

where $U_i(t)$ is the membrane potential which describes the state of the neuron, 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 [16]. It is easy to see that U_i acts as a leaky integrator of the input current I_i. Strictly speaking Equation (1) only describes the subthreshold dynamics of a LIF neuron, i.e. the dynamics in absence of spiking output of the neuron. Neurons emit spikes to communicate their output to other neurons when their membrane voltage hits the firing threshold ϑ. After each spike the membrane voltage U_i is reset to the resting potential $U_i \rightarrow U_{\mathrm{rest}}$.

In spiking neural networks the input current is typically generated by synaptic currents that are 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 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. Where sensible and if not mentioned otherwise, we use the index variable j for presynaptic indices, whereas i is reserved for postsynaptic neurons.

Synaptic currents are not static in time, but follow specific temporal dynamics themselves. A reasonable first approximation is to model their time course as an exponential current following each presynaptic spike. Moreover, we assume that synaptic currents sum linearly. Thus we can write:

$$\frac{\mathrm{d}I_i}{\mathrm{d}t} = -\frac{I_i(t)}{\tau_{\mathrm{syn}}} + \sum_j W_j S_j(t) \tag{2}$$

where the sum runs over all presynaptic neurons j and W_j are the afferent weights. Neuron i has an "autapse", ie. 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.

By combining the above definitions into Equation (1) we get

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

It is customary to solve Equations (2) and (3) numerically in discrete time. In discrete time, we can interpret the spike train S_i of neuron i as a nonlinear function of the membrane voltage

$$S_i(t) \propto \Theta(U_i(t) - \vartheta) \tag{4}$$

where Θ denotes the Heaviside step function. For the sake of brevity we now set $U_{\mathrm{rest}} = 0$, $R = 1$, and $\vartheta = 1$, which does not affect our arguments. Assuming a small simulation time step of $\Delta_t > 0$ Equation (2) is approximated by

$$I_i(t+1) = \alpha I_i(t) + \sum_j W_{ij} S_j(t) \tag{5}$$

with the decay strength α given by $\alpha = \exp\left(-\frac{\Delta_t}{\tau_{\mathrm{syn}}}\right)$. Note that $0 < \alpha < 1$ for finite and positive τ_{syn}. Moreover, $S_j(t) \in [0,1]\ \forall t$. We can now express Equation (3) as follows

$$U_i(t+1) = \beta U_i(t) + I_i(t) - S_i(t) \tag{6}$$

with $\beta = \exp\left(-\frac{\Delta_t}{\tau_{\mathrm{mem}}}\right)$. In discrete time, the output "spikes" of neuron i are simply given by $S_i(t) = \Theta(U_i(t))$.

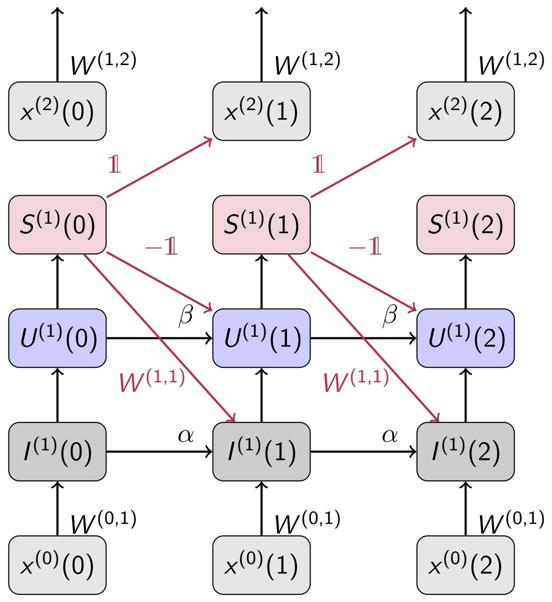


Fig. 1: Illustration of the computational graph of a spiking neural network in discrete time. Time steps flow from left to right. Inputs $x^{(1)}$ are fed into the network from the bottom and propagate upwards to higher layers. The synaptic currents I are decayed by α in each time step and feed into 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 $W^{(1,1)}$. Finally, it may also be communicated downstream via $W^{(1,2)}$ to another downstream network layer or, alternatively, a readout layer on which a supervised cost function is defined.

Equations (5) and (6) characterize the dynamics of a RNN (Figure 1). Specifically, the cell state of neuron i is characterized by the instantaneous synaptic currents I_i and the membrane voltage U_i. It is straight forward to evaluate these equations. Moreover, we can use standard machine learning libraries with auto-differentiation capabilities to do so.

We have how seen that SNNs constitute a special case of RNNs. However, so far we have not talked about how the weights in any RNN should be set to implement a specific computational function. Next we turn our attention to learning. In other words, we now focus on methods which permit us to systematically change the weights in a given network to implement a specific computational function.

III. The RNN learning problem (Hesham)

Because RNNs can be used in a variety of scenarios ranging from time series prediction, over language translation, to automatic speech recognition, powerful machine learning methods exist to train them. In the following we discuss the most common approaches before analyzing their applicability to SNNs.

The ingredients of the learning process are typically the same throughout. The first ingredient is a cost function, or a loss, which is minimized when the network's response corresponds to the desired behavior. In time series prediction for example, this loss could be the difference between the predicted and the true value. The second ingredient is a mechanism that updates the network's weights so as to minimize the loss. The simplest such mechanism is gradient descent. In neural networks, gradient descent can be implemented efficiently using the backpropagation algorithm (Box. 1). There, errors propagate backwards from the point and time of error generation to earlier time points and earlier neurons. More precisely, consider two neurons, u and v, connected through a weight w. **Description of non-locality and the problem with the non-differentiable spike generation non-linearity (Box. 2)**

Because of the structural similarly it seems only natural to assume that gradient-based learning can directly be applied to SNNs in the same way as this is done for vanilla RNNs. However, in the spiking neuron case several an important issue arises with regard to their differentiabilty. To better understand this issue we now consider the case of single LIF neuron using the framework we introduced above.

IV. Challenges of gradient-based learning with spiking neurons

As we have learned in the previous sections, gradient-based training is used commonly to train RNNs.

There are two key challenges to gradient-based learning in SNN. Both challenges relate to The first challenge concerns the lack of differentiability of the spiking architecture itself. The second challenge

By inspection of Equations (12) and (2) it is easy to appreciate the occurrence of the partial derivative ρ' of the neuronal activation function ρ. For our spiking neuron, however, we have $\rho(U(t)) = S(U(t)) = \Theta(U(t) - \vartheta)$ (cf. Eq. (4)) and thus

$$\rho'(U(t)) = \frac{\partial \rho(U(t))}{\partial U(t)} = \begin{cases} 0 & U \neq \vartheta \\ \infty & U = \vartheta \end{cases} \tag{7}$$

The nonlinearity ρ appears as a multiplicative factor in the Expressions (12) and (2), and consequently the gradients evaluate to either 0 or ∞ respectively which makes them unsuitable for gradient based optimization.

V. Alternative and approximate approaches to learning in spiking neurons

To overcome this limitation, several approaches have been devised to train SNNs with varying degrees of success. The most common approaches can be coarsely classified into the following categories: i) resorting to entirely unsupervised learning for the hidden units, ii) translating conventionally trained "rate-based" neural networks to spiking neural networks, 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 to unsupervised learning and network translation have been reviewed extensively in [15]. In this tutorial we focus on the latter two supervised approaches which we will refer to in the following as the "smoothed" and SG approach. We will first briefly review existing literature on common "smoothing" and SG approaches before turning to an in-depth discussion of how to build functional spiking neural networks.

A. Smoothed spiking neural networks (Friedemann)

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 soft nonlinearity models, probabilistic models, for which gradients technically are only well defined, or models which either rely rely on rate-codes or 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 [16]. However, in practice this approach has only been applied successfully by [17] using an augmented integrate-and-fire model in which the binary spiking nonlinearity was replaced by an continuous-valued gating function. The resulting network constitutes a RNN which can be optimized using standard methods of back-propagation through time (BPTT) or real-time recurrent learning (RTRL).

It remains unclear, however, how spike reset dynamics should be coupled into such models. More importantly, the soft threshold models compromise on one of the key features of SNN, namely the binary activation function which is essential for power efficiency.

2) Gradients in probabilistic models: Another example for smooth models are binary probabilistic models. Binary probabilistic models have been objects of extensive study in the machine learning literature mainly in the context of (restricted) Boltzmann machines for which efficient local learning rules exist [18]. Similarly, the propagation of gradients has been studied for binary stochastic models [19]. In simple terms, stochasticity effectively smooths out the hard binary nonlinearity which makes it possible to define a gradient on expectation values. SGs approaches, as we will elucidate below, are direct extensions of these ideas to the class of more domain specific spiking neuron models without the need for stochasticity.

In the neurosciences, the utility of probabilistic models has first been appreciated because it allows to formally define the smooth log-likelihood of a spike train which can be optimized using gradient descent [20]. Although this insight was first discovered in networks without hidden units, the same ideas were later extended to multi-layer networks with a single hidden layer [21]. In a similar vein, variational learning approaches have been proven capable in learning useful hidden layer representations in SNNs [22–24]. However, at the same time the injected noise necessary to smooth out the effect of binary nonlinearities has proven challenging to optimize [23].

3) Gradients for rate-based codes: Another common approach to obtain gradients in spiking neural networks 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] used an effectively rate-coded input scheme to demonstrate competitive performance on standard machine learning benchmarks such as CIFAR10 and MNIST. A similarly [26] demonstrated deep learning in spiking neural networks using ...

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 time because several repeats are needed to average out discretization noise. To some degree this problem can be overcome by spatial averaging over large populations of spiking neurons. However, this may comes at the expense of more neurons. Alternatively, temporal and resource limitations could be overcome by instead relying on spike timing as the information-carrying quantity. This is the idea behind spike timing based coding.

4) Gradients for single spike timing based codes: In an effort to optimize SNNs without potentially harmful noise injection and without reverting to a fully rate-based coding scheme, several studies have formulated spiking neural networks as a set of firing times. In such a temporal coding setting, individual spikes can carry significantly more information compared to rate-based schemes.

When using a purely firing time based code, the information carrying quantity is firing time which can depend smoothly on neuronal inputs as long as a hidden unit does not fall quiescent. The idea of single spike temporal coding was pioneered in SpikeProp [27]. In this work the analytic expressions of firing times for hidden units were linearized, allowing to compute approximate hidden layer gradients fully analytically. Several variations and extensions of this approach have been developed, to cope with, for instance, multiple hidden layers and spikes [28]. More recently, a similar approach without the need for 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 to yield competitive performance to conventional networks on MNIST.

Although the spike timing formulation does in certain cases yield well-defined gradients, it may suffer from certain limitations. For instance, the formulation of SpikeProp 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, as it is ubiquitously found in the brain, is presumably crucial for power efficiency.

B. *Surrogate gradients (Friedemann)*

To overcome some of the difficulties which are associated with smoothed SNNs, we now introduce the notion of SG methods. Their defining characteristic is that instead of changing the model definition, a SG is introduced as a continuous relaxation of the non-smooth spiking threshold for purposes of numerical optimization. Importantly, this approximation is only used in the parameter updates and does not affect the forward pass of the network directly. The use of SGs allows to efficiently train SNNs without specifying whether a temporal or spike-timing based coding scheme is to be used.

Like vanilla gradient-descent, SG learning can deal with the spatial and temporal credit assignment problem by either BPTT or, alternatively, by forward-propagation of all relevant information through eligibility traces (see Section VI-A for details). In doing so, SG learning can be reduced to local three factor rules which may be more amenable 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.

One of the first uses of this approach is described in Bohte [30] where the partial derivative of a spiking neuron was approximated by the derivative of a rectifying linear unit (ReLU) and similar path was taken to optimize binary neural networks [31]. Essentially the same idea underlies the training of large-scale convolutional networks with binary activations on neuromorphic hardware [32]. More recently, [33] successfully trained networks of neurons with slow temporal dynamics using BPTT. Encouragingly, the authors found that such networks can perform at par with conventional long short-term memory (LSTM) networks. Finally, [34] illustrated biologically plausible online learning using a similar SG approach to learn precisely timed spike train transformations.

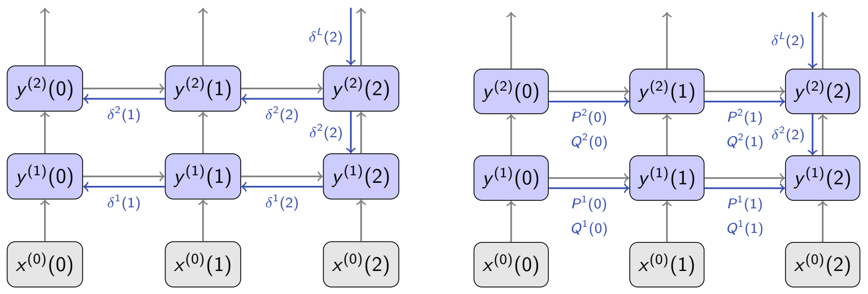
[35]

VI. Credit Assignment in Spiking Neural Networks

In gradient-based learning, the parameter updates contain a term that relates to the local error or, more generally, the derivative of the loss. In a neural network, this term is affected by the computations downstream from the node containing the trained parameter. This relationship is caused by the necessity to assign credit to that node, but its target value is not provided. In conventional neural networks, this *credit assignment problem* is solved by backpropagating the errors from the output layer using the chain rule (*i.e.* the gradient BP rule). The gradient BP rule relies on the immediate availability of backpropagated errors represented with high-precision memory. In conventional computers, Back-propagation (BP) requires storage that is shared across connected nodes in the graph. This type of shared memory does not exist in the brain, as states and parameters are local to the neurons and synapses (Box. 2). Non-locality can be spatial or temporal, or both. To solve spatial non-locality, a special communication channel that transmits the errors to the relevant nodes must be provisioned [36]. To solve temporal non-locality, some form of (working) memory is required to maintain the history of the relevant states. Non-localities, both temporal and spatial, are arguably the most challenging and constraining aspect of spiking neural networks and synaptic plasticity [6]. Although non-locality in a model of computation can be seen as a serious shortcoming, it enables a tremendous advantage compared to conventional computer: massively parallel computing with carefully controlled (and potentially dynamic) interprocess communication. Thus, solving the ability to learn and process using local information extends well beyond the neuroscientists' agenda: it can enable information processing systems that approach the reliability and energy-efficiency of the brain. In the following two sections, we describe solutions to the spatial and temporal credit assignment problems in spiking neural networks.

A. *Temporal Credit Assignment (Emre)*

Solving a temporal credit assignment problem can be achieved using two different methods [37], one backward method, one forward method (Fig. 2) For illustration purposes here, we assume a neural network with L layers and a cost function $\mathcal{L}[t] = \sum_i (U_i^L[t] - \hat{U_i^L}[t])^2$.



(a) Backward Method (b) Forward Method

Fig. 2: *Strategies for Temporal Credit Assignment.* (Left) At time $t = 2$ Errors are propagated backwards. To compute the parameter updates, the states of every neuron ($U^{(i)}$, $I^{(i)}$, and $S^{(i)}$, $i = 0, 1, 2$) and the inputs ($x^{(0)}$) for each time step are stored. (Right) At time t=2, all the information from previous time steps has been propagated forward. The nature of this information is determined by the network structure and neuron dynamics. Note that, in the forward method, the gradient step is backpropagated along the layers in time step $t = 2$, but in isolation of the other timesteps. This backward step is challenging in itself, and solutions to achieve this in a local fashion are discussed in Sec. VI-B. Nodes y represent the nodes S, U, I and x for illustration purposes in this figure.

.

1) The "backward" method. The recurrent neural network is "unrolled" meaning that an auxiliary network is created by making copies of the network for each time step (see

figure 1). The unrolling of the network clarifies the flow of information in the inference and training stages ((Fig. 2)). An important property of the unrolled network is that the parameters are shared across the time axis. Thus, the unrolled network is simply a deep network with shared weights on which one can apply the standard BP applies:

$$\begin{aligned}\Delta W_{ij}^{(l,l)} &\propto \frac{\partial}{\partial W_{ij}^{(l,l)}}\mathcal{L}[T] = \sum_{t=0}^{T}\delta_i^l[t]S_j^{l-1}[t],\\ \delta_i^l[t] &= \rho'\left(U_i^l[t]\right)\left(\sum_k \delta_k^{l+1}[t]W_{ki}^{(l,l+1)} + \sum_k \delta_k^l[t+1]W_{ki}^{(l,l)}\right),\end{aligned} \tag{8}$$

where T is the duration of the sequence, and using the notation $\delta_i^L[t] = U_i^L[t] - \hat{U}_i^L[t]$ This rule is known as BPTT, and the main difference compared to BP is the sum over time steps t. BPTT solves the temporal credit assignment problem by back propagating through the unrolled network. This method works backward through time after completing a forward update. In practice this is an expensive operation, as the entire history of activation must be buffered to computed the gradients. The use of standard backpropagation on the unrolled network directly enables the use of autodifferentiation tools offered in model machine learning toolkits [5, 7, 38].

2) The forward method: The information required for computing the errors is propagated forward. For a LIF neuron with first order dynamics, recall that the discrete-time membrane potential and synaptic current can be expressed as Eqs. (5) and (6). The gradient of a cost function $\mathcal{L}[t]$ with respect to weight W_{ij} is then:

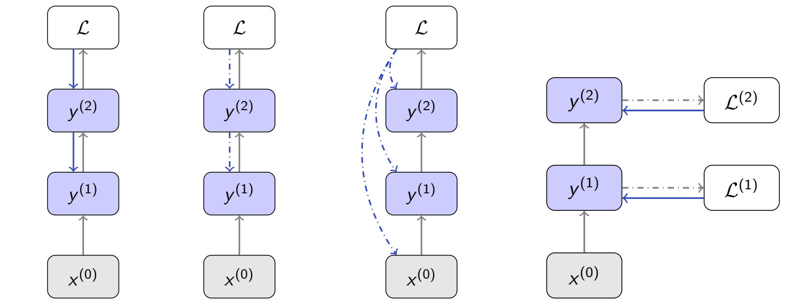
$$\begin{aligned}\Delta W_{ij} &\propto \frac{\partial}{\partial W_{ij}}\mathcal{L}[t+1] = \delta_i[t+1]P_{ij}[t+1]\\ P_{ij}[t+1] &= \frac{\partial}{\partial W_{ij}}U_i[t+1] = \beta P_{ij}[t] + Q_{ij}[t] + \sum_j W_{ij}\sigma'(U_j[t-1])P_{ij}[t-1]\\ Q_{ij}[t+1] &= \frac{\partial}{\partial W_{ij}}I_i[t+1] = \alpha Q_{ij}[t] + \sigma(U_i[t]) + \sum_j W_{ij}\sigma'(U_j[t-1])P_{ij}[t-1],\end{aligned}$$

Here, we replaced the spiking variable S_j with the smooth activation function $\sigma(U_j[t])$. where σ' is the derivative of σ.

In essence, both solutions transform the temporal credit assignment problem to a spatial one and applying the solutions for spatial credit assignment (see below). The backward approach is more efficient in terms of computation, but requires maintaining all the inputs and activations for each time step. Thus the space complexity of the backward approach is $O(NT)$, where N is the number of neurons.

On the other hand, the forward method requires maintaining variables P_{ij} and Q_{ij}. The space complexity is $O(N^2)$, where N is the number of neurons. In a single layer feed-forward architecture without refractory mechanisms, the third term above vanishes and P_{ij}, Q_{ij} no longer depend on j. In this case, the space complexity becomes $O(N)$. This simplification can lead to an efficient synaptic plasticity rule for deep local learning, as described later.

As memory is ample in modern von Neumann computers, the backward method is preferred for an implementation on a conventional computer. On the other hand, processing in dedicated neuromorphic hardware and, more generally, non-von Neumann computers may not have immediate access to such large memory. In this case, the forward approach is preferable. Furthermore, the forward approach is appealing from a biological point of view, since the learning rule is



(a) BP (b) Feedback Alignment (c) Random BP (d) Local Errors

Fig. 3: *Spatial error credit assignment and strategies for relaxing gradient BP requirements.* Dashed lines indicate random connections

consistent with synaptic plasticity in the brain and "three-factor" rules.

B. Spatial Credit Assignment (HM, EN, FZ)

Using backpropagation to adjust deep layer weights ensures that the weight update will reduce the output layer error for the current training example (if the learning rate is small enough). While this theoretical guarantee is highly desirable, it comes at the cost of more complex communication requirements, namely that errors have to be communicated back through the network, and increased memory requirements as the neuron states need to be buffered until the errors become available. These requirements can be relaxed by either: 1) Approximating the error backpropagation step through cheaper means that incur less memory, communication and computational overhead; 2) Formulating an alternative learning problem where the errors for each neuron are proximally generated, both in space and time. In the following, we describe examples of both type of solutions.

a) Gradient BPTT (FZ):

b) Random Error BP: One family of algorithms that relaxes the symmetry requirement of BP are feedback alignment or random BP algorithms [**?** **?**]. These are approximations to the gradient BP rule that side-step the non-locality problem by replacing weights in the learning channel with random ones, leading to remarkably little loss in classification performance on benchmark tasks (requirement (i) in (Box. 1)). 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 align with the (random) feedback weights, which are effective in communicating gradients [**?**]. 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 [**?**]. Extended experimentations with Event-driven random back-propagation (eRBP) show that the spiking nature of neuromorphic hardware and the lack of complex non-linear computations at the neuron do not prevent accurate learning on classification tasks, and can operate continuously and asynchronously without alternation of forward or backward passes. Up to moderate classification accuracies, a spiking implementation requires an equal or fewer number of synaptic operations (SynOp) compared to multiply accumulation (MAC) operation to reach a given accuracy for both networks [**?**]. While a standard computer remains the architecture of choice if classification accuracy on a stationary dataset is the target regardless of energy efficiency, neuromorphic hardware is a strong contender if low-power

Box 1: The Gradient Backpropagation Rule

The task of learning is to minimize this cost over the entire dataset. This can be done efficiently using the gradient descent rule, which modifies the network parameters $\mathbf{w}$ in the direction opposite to the gradient:

$$w_{ij} \leftarrow w_{ij} - \eta \Delta w_{ij}, \text{where } \Delta w_{ij} = \frac{\partial}{\partial w_{ij}} L = \rho' \left(\sum_j w_{ij} x_j \right) e_i x_j, \tag{11}$$

and where η is a small learning rate. In deep networks, the weights of the hidden layer neurons are modified by backpropagating the errors from the prediction layer using the chain rule. Using superscripts $l = 0, ..., N$ to denote the layer (0 is input, N is output):

$$\frac{\partial}{\partial W^l_{ij}} L = \delta^l_i S^l_j, \text{ where } \delta^l_i = \rho' \left(U^l_i \right) \sum_k \delta^{l+1}_k [t] W^{l+1}_{ki}, \tag{12}$$

where the $\delta^N_i = e_i$, as in Eq. 11 and $y^0_i = x_i$. This update rule is ubiquitous in deep learning [39] and known as the gradient backpropagation algorithm. Learning is typically carried out in forward passes (evaluation of the neural network activities) and backward passes (evaluation of δs).

learning on non-stationary data is the objective, since energy efficiency is improved at least by a factor equal to the achieved Joule/MAC to Joule/SynOp ratio and learning is on-going.

c) 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. 12) is the error term $\delta^l_i[t]$. 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 $E^l(\mathbf{y}^l[t])$ and use this local loss to obtain the errors. In such a local learning setting, the weight update equation becomes:

$$\frac{\partial}{\partial W^{(l-1,l)}_{ij}} \mathcal{L}^l = -\sum_{t=0}^{T} \delta^l_i[t] y^{l-1}_j[t], \text{ where } \delta^l_i[t] = \rho' \left(\sum_j W^{(l-1,l)}_{ij} y^{l-1}_j[t] \right) \frac{\mathrm{d}}{\mathrm{d}y^l_i[t]} \mathcal{L}^l(\mathbf{y}^l[t]) \tag{9}$$

One convenient form for the local loss that works well in practice is:

$$\mathcal{L}^l(\mathbf{y}^l[t]) \equiv \mathcal{L}(\mathbf{W}^l_r \mathbf{y}^l[t], \hat{\mathbf{y}}^l[t]) \tag{10}$$

where $\hat{\mathbf{y}}^l[t]$ is a pseudo-target for layer l, $\mathbf{W}^r_l$ is a fixed random matrix that projects the activity vector at layer l to a vector having the same dimension as the pseudo-target, and $\mathcal{L}$ is a function measuring the discrepancy between its two arguments. In essence, this formulation assumes an auxiliary random layer is attached to layer l and the goal is to modify $\mathbf{W}^{(l-1,l)}$ 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.

Box 2: Non-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. Information necessary for learning can be non-local (it is available elsewhere), or global (it is shared among all processing elements).
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. Spatially global signals are often required for learning on practical tasks, and even commonplace in the brain (*e.g.* neuromodulation). Non-local information can be transmitted through special structures, for example dedicated encoders and decoders for U_B and a form of working memory for $S^B(t-T)$. An important challenge is to map relevant computations on functions of D by minimizing the cost of communicating D_{nloc}.

VII. Implementation

VIII. Application: Learning Gesture Recognition

We demonstrate that deep learning algorithms that locally approximate the gradient backpropagation updates using locally synthesized gradients enable synaptic plasticity in spiking neural networks [**?**]. This approach uses the forward method to overcome temporal credit assignment and local errors to overcome spatial credit assignment. To enable an efficient implementation in GPUs, the neuron model is simplified by omitting recurrent weights, including refractory mechanisms, in the backward pass. This omission results in a forward method that scales as $O(N)$ (instead of $O(N^2)$ for the full forward method).

$$\text{Equations here or not?} \tag{13}$$

The result is a highly efficient synaptic plasticity rule for multi-layer spiking neural networks. Furthermore, our method utilizes existing autodifferentation methods in machine learning frameworks to systematically derive synaptic plasticity rules from task-relevant cost functions and neural dynamics. We benchmark our approach on the DVS Gestures dataset (Fig. 4).

IX. Discussion

References

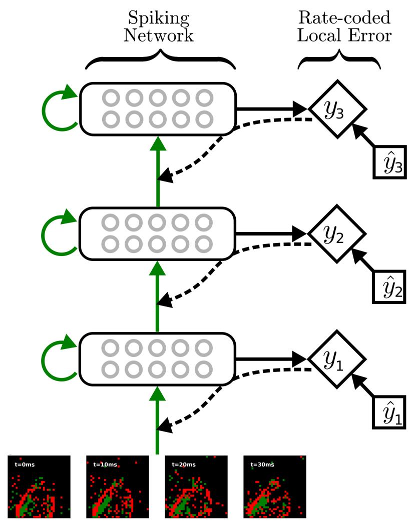
[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[2] H. Siegelmann and E. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[3] D. Ince, *Mechanical intelligence (collected works of AM Turing)*. North-Holland Publishing Co., 1992.

[4] F. Zenke and S. Ganguli, "Superspike: Supervised learning in multi-layer spiking neural networks," *arXiv preprint arXiv:1705.11146*, 2017.

[5] S. Woźniak, A. Pantazi, and E. Eleftheriou, "Deep networks incorporating spiking neural dynamics," *arXiv preprint arXiv:1812.07040*, 2018.



Model	Error	Training Samples
IBM EEDN	5.51%	Offline $64 \cdot 10^6$ samples
Slayer	6.36%	Offline # samples not reported
DCLL (This Work)	5.819%	Online $324 \cdot 10^3$ samples

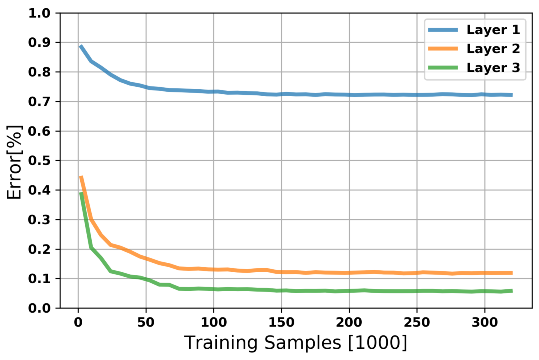


Fig. 4: Deep Continuous Local Learning (DCLL) with spikes, applied to the DVSGestures dataset [**?**]. A three layer convolutional spiking neural network is trained with SG using local errors generated using fixed random projections to a local classifier. Learning in DCLL scales linearly with the number 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. Results here are shown for the DVS Gestures dataset, 11 gestures recorded using a Dynamic Vision Sensor.

[6] E. O. Neftci, “Data and power efficient intelligence with neuromorphic learning machines,” *iScience*, vol. 5, pp. 52–68, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2589004218300865

[7] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, “Long short-term memory and learning-to-learn in networks of spiking neurons,” *arXiv preprint arXiv:1803.09574*, 2018.

[8] K. Boahen, “A neuromorph’s prospectus,” *Computing in Science Engineering*, vol. 19, no. 2, pp. 14–28, Mar. 2017.

[9] F. Ponulak and A. Kasiński, “Supervised Learning in Spiking Neural Networks with ReSuMe: Sequence Learning, Classification, and Spike Shifting,” *Neural Computation*, vol. 22, no. 2, pp. 467–510, Oct. 2009.

[10] R. Gütig and H. Sompolinsky, “The tempotron: a neuron that learns spike timing–based decisions,” *Nature Neuroscience*, vol. 9, pp. 420–428, 2006.

[11] R. V. Florian, “The Chronotron: A Neuron That Learns to Fire Temporally Precise Spike Patterns,” *PLOS ONE*, vol. 7, no. 8, p. e40233, Aug. 2012.

[12] A. Mohemmed, S. Schliebs, S. Matsuda, and N. Kasabov, “Span: spike pattern association neuron for learning spatio-temporal spike patterns,” *Int. J. Neur. Syst.*, vol. 22, no. 04, p. 1250012, Jun. 2012.

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

[14] 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.

[15] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. S. Maida, "Deep Learning in Spiking Neural Networks," *arXiv:1804.08150 [cs]*, Apr. 2018, arXiv: 1804.08150.

[16] 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.

[17] D. Huh and T. J. Sejnowski, "Gradient descent for spiking neural networks," *arXiv preprint arXiv:1706.04698*, 2017.

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

[20] 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.

[21] 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.

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

[27] 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.

[28] A. Banerjee, "Learning Precise Spike Train–to–Spike Train Transformations in Multilayer Feedforward Neuronal Networks," *Neural Computation*, vol. 28, no. 5, pp. 826–848, Mar. 2016.

[29] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *arXiv preprint arXiv:1606.08165*, 2016.

[30] 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.

[31] 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.

[32] 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.

[33] G. Bellec, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, "Long short-term memory and Learning-to-learn in networks of spiking neurons," *arXiv:1803.09574 [cs, q-bio]*, Mar.

2018, arXiv: 1803.09574.

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

[35] P. O'Connor and M. Welling, "Deep spiking networks," *arXiv preprint arXiv:1602.08323*, 2016.

[36] 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.

[37] 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.

[38] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," *arXiv preprint arXiv:1810.08646*, 2018.

[39] D. E. Rumelhart, J. L. McClelland, P. R. Group *et al.*, *Parallel distributed processing*. IEEE, 1988, vol. 1.

X. Author Information

Emre Neftci, UC Irvine (eneftci@uci.edu)
Dr. Neftci is an assistant professor in the department of Cognitive Sciences and Computer Science at UC Irvine. He received his masters degree in Physics at Ecole Polytechnique Federal de Lausanne (EPFL) and his PhD in Neuroinformatics from the Institute of Neuroinformatics at the university of Zurich and ETH Zurich in neuromorphic engineering. His current research explores the bridges between neuroscience and machine learning, with the focus of theoretical and computational modeling of learning algorithms that are best suited to neuromorphic hardware and non-von Neumann computing architectures.

Hesham Mostafa, UCSD (hmmostafa@ucsd.edu)
Dr. Mostafa obtained a master's degree in electrical engineering from the Technical University of Munich in 2010, and a PhD in Neuroinformatics from the Institute of Neuroinformatics at the university of Zurich and ETH Zurich. He is currently a post-doc at the integrated systems neuro-engineering lab at the Institute of Neural Computation at University of California San Diego. 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, University of Oxford (friedemann.zenke@cncb.ox.ac.uk)
Dr. Zenke is a Sir Henry Wellcome post-doctoral fellow at the University of Oxford. He studied physics at the University of Bonn, Germany and at the Australian National University in Canberra. He received his PhD in the laboratory of Wulfram Gerstner at the EPFL where he studied the interaction of synaptic and homeostatic plasticity in spiking neural network models. He then joined the laboratory of Surya Ganguli at Stanford University as a post-doctoral fellow where he used machine learning approaches to explore learning and memory formation in biologically inspired neural networks. He currently continues this line of research in Tim Vogels' Lab in Oxford.