# ELUCIDATING THE THEORETICAL UNDERPINNINGS OF SURROGATE GRADIENT LEARNING IN SPIKING NEURAL NETWORKS

**Julia Gygax & Friedemann Zenke**
Friedrich Miescher Institute for Biomedical Research
Faculty of Science, University of Basel
Basel, Switzerland
firstname.lastname@fmi.ch

## ABSTRACT

Training spiking neural networks to approximate complex functions is essential for studying information processing in the brain and neuromorphic computing. Yet, the binary nature of spikes constitutes a challenge for direct gradient-based training. To sidestep this problem, surrogate gradients have proven empirically successful, but their theoretical foundation remains elusive. Here, we investigate the relation of surrogate gradients to two theoretically well-founded approaches. On the one hand, we consider smoothed probabilistic models, which, due to lack of support for automatic differentiation, are impractical for training deep spiking neural networks, yet provide gradients equivalent to surrogate gradients in single neurons. On the other hand, we examine stochastic automatic differentiation, which is compatible with discrete randomness but has never been applied to spiking neural network training. We find that the latter provides the missing theoretical basis for surrogate gradients in stochastic spiking neural networks. We further show that surrogate gradients in deterministic networks correspond to a particular asymptotic case and numerically confirm the effectiveness of surrogate gradients in stochastic multi-layer spiking neural networks. Finally, we illustrate that surrogate gradients are not conservative fields and, thus, not gradients of a surrogate loss. Our work provides the missing theoretical foundation for surrogate gradients and an analytically well-founded solution for end-to-end training of stochastic spiking neural networks.

***Keywords*** Spiking neural networks · surrogate gradients · stochastic automatic differentiation · stochastic spiking neural networks

## 1 Introduction

Our brains efficiently process information in spiking neural networks (SNNs) that communicate through short stereotypical electrical pulses called spikes. For understanding information processing in the brain and instantiating similar capabilities *in silico*, SNNs are an indispensable tool. Like conventional artificial neural networks (ANNs), SNNs require training to implement specific functions. However, typical SNN models are not differentiable due to the binary nature of the spike, which precludes the use of standard gradient-based training techniques based on backpropagation (BP) [1]. To overcome this problem, there are multiple options. One can dispense with hidden layers altogether [2, 3], but this option has its obvious limitations. Alternatively, one makes the neuron model differentiable [4] or works in a reduced solution space, for instance, by only considering the timing of existing spikes for which gradients exist [5–8]. Finally, one can replace the gradient with a suitable surrogate [9–15]. In this article, we focus on the latter. Surrogate gradient (SG) approaches are empirically successful and not limited to changing the timing of existing spikes while working with non-differentiable neuron models.

However, SGs lack a solid theoretical foundation. We address this gap in our understanding by analyzing the commonalities and differences of SGs with theoretically well-founded approaches based on stochastic networks. Specifically, we focus on smoothed probabilistic models (SPMs) [14, 16] and the more recently proposed stochastic automatic differentiation (stochAD) framework [17]. SPMs typically rely on stochasticity to smooth out the optimization landscape in expectation to enable gradient computation on this smoothed loss. In this article, we particularly focus on neuron models with escape noise [18] which are commonly used to smooth the non-differentiable spikes in expectation and for which exact gradients are computable [19, 20]. However, extending SPMs to multi-layer and, in particular, deep neural networks has been difficult because they preclude using BP, which usually requires additional approximations [21]. In contrast, stochAD is a recently developed framework for automatic differentiation (AD) in programs with discrete randomness, i.e. with discrete random variables. While this opens up the door for BP or other recursive gradient computation schemes, the framework has not yet been applied to SNNs.

Here, we jointly analyze the above methods, elucidate their relations, and provide a rigorous theoretical foundation for SG-descent in stochastic SNNs. We first provide some background information on each of the above methods before starting our analysis with the case of a single Perceptron. From there, we move to the more complex case of multi-layer Perceptrons (MLPs), where we elaborate the theoretical connection between SGs and stochAD, and end with deep networks of leaky integrate-and-fire (LIF) neurons. Finally, we substantiate the theoretical results with empirical simulations.

## 2 Background on SGs, SPMs, and stochAD

Before analyzing the relation between SGs, gradients of the log-likelihood in SPMs, and stochAD, we briefly review these methods. The first two approaches have been developed for training SNNs. Their goal is to compute reasonable gradient approximations, but they use different methods for smoothing spikes. In contrast, the third approach is not tailored to SNNs and aims to compute unbiased gradients of arbitrary functions with discrete randomness by introducing the concept of stochastic derivatives.

**Smoothed probabilistic models.** SPMs are based on stochastic networks in which the gradients are well-defined in expectation [14, 16]. Typically such models consist of a noisy neuron model, such as the LIF neuron with escape noise [18] and a probabilistic loss function. On the one hand, this includes models that require optimizing the log-likelihood of the target spike train being generated by the current model [19]. However, this method only works effectively without hidden units. On the other hand, SPMs also comprises models that follow a variational strategy [20], which makes them applicable to networks with a single hidden layer. In general, the gradients that are computed within the SPM framework are given by

$$\frac{\partial}{\partial w}\mathbb{E}\left[\mathcal{L}\right] \ ,$$

where $\mathcal{L}$ is the loss and $w$ is an arbitrary model parameter. The computational cost associated with evaluating $\mathbb{E}\left[\mathcal{L}\right]$ precludes training deeper networks in practice. This is because SPMs lack support for AD, as we will explain in detail in Section 3.2.1.

**Surrogate gradients.** SGs are a heuristic that constitute a continuous relaxation of the non-differentiable spiking activation function that occurs when computing gradients in SNNs [14]. To that end, one systematically replaces the derivative of the hard threshold by a surrogate derivative (SD), also called pseudo-derivative [13], when applying the chain rule. The result then serves as a *surrogate* for the gradient.

For example, when computing the derivative of the loss with respect to a weight, $\frac{\partial \mathcal{L}}{\partial w}$, the problematic derivative of a Heaviside $\frac{\partial \Theta(u)}{\partial u}$ is replaced by an SD

$$\frac{\partial \mathcal{L}}{\partial w} \leftarrow \frac{\partial \mathcal{L}}{\partial \Theta(u)} \underbrace{\frac{\partial \sigma(\beta_{\text{SG}} \cdot u)}{\partial u}}_{\text{SD}} \frac{\partial u}{\partial w} \ .$$

Different SD functions are used in practice. For instance, rectified linear functions have been used successfully as SD [5, 11, 13], whereas SuperSpike [12] used a scaled derivative of a fast sigmoid $h(x) = \frac{1}{(\beta|x|+1)^2}$. However, SNN training is robust to the choice of SDs [22, 23]. In binarized neural networks, the notion of SGs is known as the straight-through estimator (STE) [10, 24–26].

**Stochastic automatic differentiation.** AD in general aims at calculating the derivative, i.e., infinitesimal changes, of a function at a given location in parameter space together with a primal evaluation of the function at the same location. The chain rule is used to calculate the derivatives, and the individual derivative terms can be calculated in forward or backward mode, with the latter being called BP [27]. To extend this to exact AD in programs with discrete randomness, the stochAD framework Arya et al. [17] introduces stochastic derivatives. To deal with discrete randomness, stochastic derivatives consider not only infinitesimally small continuous changes but also finite changes with infinitesimally small probability. Following Arya et al. [17], a stochastic derivative of a random variable $X(p)$ consists of a triple $(\delta, w, y)$, where $\delta$ is the "almost sure" derivative, $w$, the derivative of the probability of a finite change, and $Y$, the alternate value, i.e., the value of $X(p)$ in case of a finite jump. Although stochastic derivatives are suitable for forward mode AD, they cannot be used directly for BP as of now. This is because they application of the chain rule would require the derivatives to be scalars, whereas they consist of a triple. The stochAD framework outlines a way to convert them to a single scalar value, which makes them compatible with BP albeit at the cost of introducing bias. Given a stochastic derivative, its smoothed stochastic derivative is

$$\tilde{\delta} = \mathbb{E}[\delta + w(Y - X(p))|X(p)]. \tag{1}$$

for one realization of the random variable $X(p)$ [17].

## 3 Analysis of the relation between SGs, SPMs, and stochAD for direct training of SNNs

To fathom the theoretical foundation of SG learning, we focused on the theoretically well-grounded SPMs and the recently proposed stochAD framework. Because both approaches assume stochasticity whereas SGs are typically applied in deterministic settings, we start our analysis with focus on stochastic networks. We will later discuss deterministic networks as a special case of the stochastic setting.

To keep our analysis general and independent of the choice of the loss function, we consider the Jacobian $\nabla_w y$ defined at the network's output $y$ where $w$ are the trainable parameters or weights. For simplicity and without loss of generality, we further consider networks with only one output such that the above is equivalent to studying

$$\nabla y = \left( \frac{\partial}{\partial w_1} y, \ldots, \frac{\partial}{\partial w_n} y \right) \, .$$

To further ease the analysis, we start with binary Perceptrons, thereby neglecting all temporal dynamics and the reset of conventional spiking neuron models, while retaining the essential binary spike generation process (see Fig. 1A, Methods). We begin our comparison by examining a single neuron before moving on to MLPs. We defer the discussion of LIF neurons to Section 5.

### 3.1 Analysis of a single binary Perceptron

The deterministic Perceptron is defined as

$$\begin{aligned} u &= W^T x + b \\ y &= \Theta(u - \theta) \, , \end{aligned} \tag{2}$$

where $\Theta(\cdot)$ is the Heaviside step function and $u$ is the membrane potential, which depends on the weights $W$, the bias $b$, the input $x$, and the firing threshold $\theta$. The Jacobian is related to the gradient via

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial w_i} \, ,$$

where the problematic derivative of the non-differentiable Heaviside function appears in $\frac{\partial y}{\partial w_i}$. When computing SGs the derivative of the Heaviside function is replaced with the corresponding SD

$$\frac{\partial}{\partial w_i} y(u - \theta) := \frac{\partial}{\partial w_i} \sigma_\beta(u - \theta) \, . \tag{3}$$

To see how the above expression compares to the derivative of the corresponding SPM, we consider the stochastic Perceptron

$$\begin{aligned} u &= W^T x + b \\ p &= f(u - \theta) = \sigma_\beta(u - \theta) \\ y &\sim \text{Ber}(p) \, . \end{aligned} \tag{4}$$

To model stochasticity, we model escape noise with the probability of firing $f(\cdot)$ given by the sigmoid function $\sigma_\beta(u - \theta) = \frac{1}{1+\exp(-\beta(u-\theta))}$, where $\beta$ controls the steepness. Importantly, there is some degree of freedom as to which escape noise function we chose. Other common choices in the realm of spiking neurons are the exponential or error function [18]. For our current choice, the sigmoid, we find that it approaches the step function in the limit $\beta \to \infty$ and

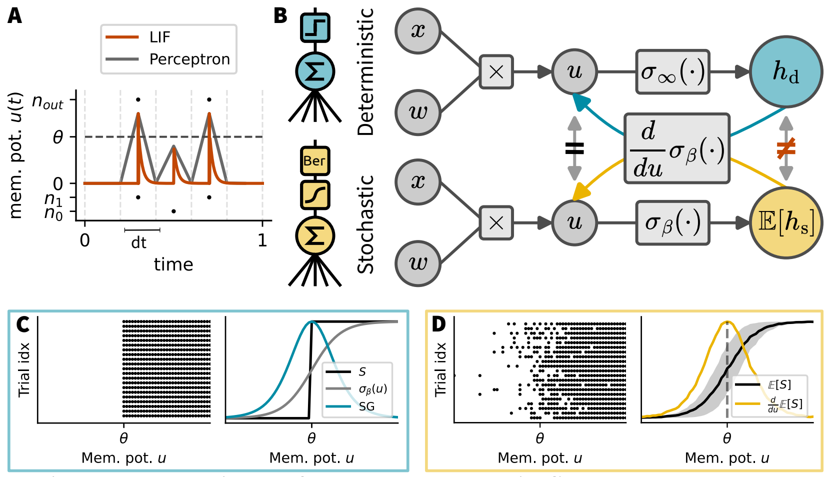


**Figure 1. SGs are equivalent to gradients of expected outputs in SPMs and smoothed stochastic derivatives in single binary Perceptrons. (A)** Membrane potential dynamics of an LIF neuron (red) in comparison with the Perceptron. When input spikes, $n_0$ and $n_1$, are received, they excite the LIF neuron which causes the membrane potential to increase. Once it reaches the threshold, output spikes, $n_{out}$, are emitted. In the limit of a large simulation time step ($dt \gg \tau_{mem}$) and appropriate scaling of the input currents the LIF neuron approximates a Perceptron receiving time-locked input (gray lines). **(B)** Left: The simplified computational graph of a deterministic (blue) and a stochastic (yellow) Perceptron. Right: Forward pass in a deterministic (top) or stochastic (bottom) shallow network. The colored arrows indicate that both use the gradient of $\sigma_\beta(\cdot)$ on the backward pass, which is the gradient of the expected output of the stochastic neuron. In the case of the deterministic neuron, this constitutes the SG used instead of the non-existing gradient of the step function. **(C)** Network output over multiple trials (left) and the derivative in the deterministic Perceptron (right). For SG-descent, the derivative of a sigmoid (gray) is used to approximate the non-existing derivative of the step function (black). **(D)** Same as (C) but for the stochastic Perceptron. Escape noise leads to variability in the spike trains over trials (left). The expected output follows a sigmoid, and we can compute the derivative (yellow curve) of the expected output (right).

the stochastic Perceptron becomes deterministic. Finally, for ease of comparison, we use the same $\beta$ as above for the SD.

In SPMs, the idea is to compute the derivative of the expected loss at the output to smooth out the discrete spiking non-linearity [19]. In our setup, this amounts to computing the expected output of the stochastic Perceptron,

$$\mathbb{E}[y] = \sigma_\beta(u-\theta),$$

and taking its derivative

$$\frac{\partial}{\partial w_i}\mathbb{E}[y] = \frac{\partial}{\partial w_i}\sigma_\beta(u-\theta)\,. \tag{5}$$

We note that Expressions (5) and (3) are the same. Thus, the derivative of the expected output of the stochastic Perceptron is equivalent to the SD of the output of the deterministic Perceptron for suitable choices for the escape noise and SD functions (see Fig. 1B). For equivalent expressions, the choice of the SD should be matched to the derivative of the escape noise in the corresponding stochastic neuron.

Next, we compare these findings with the derivative obtained from the stochAD framework. To that end, we first apply the chain rule

$$\frac{\partial y}{\partial w_i} \;=\; \frac{\partial}{\partial p}\mathrm{Ber}(p)\frac{\partial}{\partial w_i}p \tag{6}$$

and use the smoothed stochastic derivative of a Bernoulli random variable following the steps of Arya et al. [17]. The right stochastic derivative for a Bernoulli random variable is given by $(\delta_R, w_R, Y_R) = (0, \frac{1}{1-p}, 1)$ if the outcome of the random variable was zero ($\mathrm{Ber}(p) = 0$) and zero otherwise. The left stochastic derivative is given by $(\delta_L, w_L, Y_L) = (0, \frac{-1}{p}, 0)$ if $\mathrm{Ber}(p) = 1$ and also zero otherwise. According to Eq. (1), the corresponding smoothed versions are $\widetilde{\delta}_R = \frac{1}{1-p}\cdot \mathbf{1}_{X(p)=0}$ and $\widetilde{\delta}_L = \frac{1}{p}\cdot \mathbf{1}_{X(p)=1}$. Since every affine combination of the left and right derivatives is a valid derivative, we can use $\frac{\partial}{\partial p}\mathrm{Ber}(p) = (1-p)\widetilde{\delta}_R + p\widetilde{\delta}_L = 1$ as the smoothed stochastic derivative of the Bernoulli random

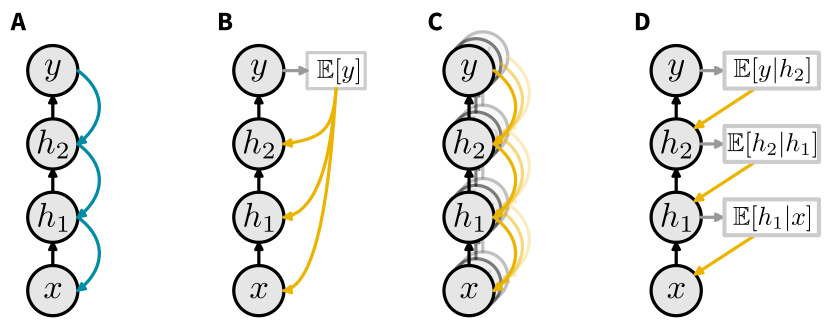


**Figure 2. Derivative computation in MLPs.** Schematic of an example network for which (surrogate) derivatives are computed according to different methods. The colored arrows indicate where partial derivatives are calculated. **(A):** SG descent relies on the chain rule for efficient gradient computation in a deterministic MLP. Thus, the derivative of the output with respect to a given weight is factorized into its primitives, which are indicated by the colored arrows. **(B)** SPMs approach the problem of non-differentiable spike trains by adding noise and then smoothing the output based on its expected value. Since this method does not allow the use of the chain rule, the derivative for each weight must be computed directly. **(C)** The derivative and the expected value are not interchangeable, which makes this option mathematically invalid. Furthermore, it is not possible to achieve the necessary smoothing using the expected value after such an interchange. **(D)** Smoothed stochastic derivatives in stochAD use the expected value of each node to compute the derivative; however, the method relies on expectation values conditioned on the activity of a specific forward pass.

variable. This results in

$$\frac{\partial y}{\partial w_i} = 1 \cdot \frac{\partial}{\partial w_i}\sigma_\beta(u-\theta)$$

when inserted into Eq. (6). Yet again, we obtain the same result as above.

Thus, for a single binary Perceptron, the results obtained from the different approaches are identical, given a sensible choice of SD, which should match the derivative of the escape noise function. However, it is worth noting that despite the identical derivatives, the resulting overall gradients, are different in the deterministic and stochastic case. This difference is due to distinct outputs $y$, which directly affect $\frac{\partial \mathcal{L}}{\partial y}$. For instance, there is typically no equivalence in the expected value since a binary output generally does not correspond to the expected value of a stochastic Perceptron, which can take any value between zero and one. We will see in the next section that these and other notable differences break the equivalence of the different approaches in MLPs.

## 3.2 Analysis of the multi-layer Perceptron

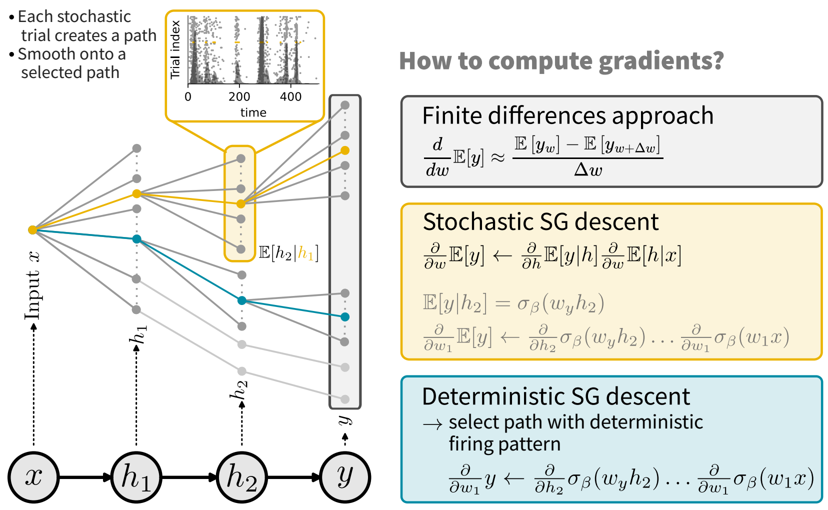
To analyze the relation of the different gradient approximation methods in the multi-layer setting, we begin by examining SPMs, which lack support for AD and thus an efficient algorithm to compute gradients. We then further discuss how stochAD provides smooth stochastic derivatives equivalent to SGs in multi-layer networks.

### 3.2.1 Output smoothing in multi-layer SPMs precludes efficient gradient computation

SPMs lack support for AD, because they smooth the expected loss landscape through stochasticity and therefore require the calculation of expected values at the network output. While output smoothing allows the implementation of the finite difference algorithm, this algorithm does not scale to large models and is therefore of little practical use for training ANNs [28, 29]. The application of AD, however, requires differentiable models, like standard ANNs, so that the chain rule can be used to decompose the gradient computation into simple primitives. This composability is the basis for efficient recursive algorithms like BP and real-time recurrent learning (RTRL) [27].

To see why SPMs do not support AD, let us consider a simple example network: Let $y$ be the output of a binary neural network with two hidden layers $h_1$, $h_2$ (Fig. 2). The output $y$ has the firing probability $p_y = \sigma_\beta(w_y^T h_2)$, and the hidden layers have the firing probabilities $p_1 = \sigma_\beta(w_1^T x)$ and $p_2 = \sigma_\beta(w_2^T h_1)$. We are looking for a closed-form expression of the derivative of the expected output for each parameter, e.g., $\frac{\partial}{\partial w_1}\mathbb{E}[y]$ for weight $w_1$ (Fig. 2B). In a deterministic and differentiable network, one can use the chain rule to split the expression into a product of partial derivatives as $\frac{\partial}{\partial w_1} y = \frac{\partial y}{\partial p_y}\frac{\partial p_y}{\partial h_2}\frac{\partial h_2}{\partial p_2}\dots\frac{\partial p_1}{\partial w_1}$ (see Fig. 2A). However, this is not possible for SPMs because

$$\frac{\partial}{\partial w_1}\mathbb{E}[y] \neq \mathbb{E}\left[\frac{\partial}{\partial w_1} y\right] .$$



• Each stochastic trial creates a path
• Smooth onto a selected path
Trial index
0
200
400
time
$\mathbb{E}[h_2|h_1]$
Input $x$
$h_1$
$h_2$
$y$
$x$
$h_1$
$h_2$
$y$
How to compute gradients?
Finite differences approach
$\frac{d}{dw}\mathbb{E}[y] \approx \frac{\mathbb{E}[y_w] - \mathbb{E}[y_{w+\Delta w}]}{\Delta w}$
Stochastic SG descent
$\frac{\partial}{\partial w}\mathbb{E}[y] \leftarrow \frac{\partial}{\partial h}\mathbb{E}[y|h]\frac{\partial}{\partial w}\mathbb{E}[h|x]$
$\mathbb{E}[y|h_2] = \sigma_\beta(w_y h_2)$
$\frac{\partial}{\partial w_1}\mathbb{E}[y] \leftarrow \frac{\partial}{\partial h_2}\sigma_\beta(w_y h_2)\dots\frac{\partial}{\partial w_1}\sigma_\beta(w_1 x)$
Deterministic SG descent
$\rightarrow$ select path with deterministic firing pattern
$\frac{\partial}{\partial w_1}y \leftarrow \frac{\partial}{\partial h_2}\sigma_\beta(w_y h_2)\dots\frac{\partial}{\partial w_1}\sigma_\beta(w_1 x)$

**Figure 3. SGs correspond to smoothed stochastic derivatives in stochastic SNNs.** The tree illustrates the discrete decisions associated with the binary spike generation process at different points over different layers of a stochastic MLP. A single forward pass in the network corresponds to a specific path through the tree which yields a specific set of spike trains. Another forward pass will result in a different path and spike patterns. Computing gradients using **finite differences** requires randomly sampling paths from the network and evaluating their averaged loss before and after a given weight perturbation. Although this approach is unbiased for small perturbations, the random path selection results in high variance. Furthermore, that approach is not scalable to large networks. **Stochastic SG-descent** is equivalent to smoothed stochastic derivatives in the stochAD framework. To compute the gradient, we roll out the network once and sample a random path in the tree which we now keep fixed (yellow). At each node, we then compute the expected output given the fixed activation of the previous layer $\mathbb{E}[h_i|h_{i-1}]$, which yields a low-variance estimate (see inset: spike raster, selected trial shown in yellow, spike trains of other trials in gray, expectation shown as shaded overlay). By choosing a surrogate function that matches the escape noise process, both methods give the same derivative for a spike with respect to the membrane potential. **Deterministic SG-descent** can be seen as a special case in which the random sampling of the path is replaced by a point estimate given by the deterministic roll out (blue).

While this inequality generally holds for any nonlinear derivative, in the case of binary networks and SNNs there is another potential problem with the above expression. Some of the factors involve the derivative of the binary output and are thus ill-defined. Smoothing them with stochasticity was the entire point of working with expectation values in the first place.

Finally, even if we could exchange the expectation value and the derivative, we would still be faced with the fact that the expectation of a product is usually not equal to the product of expectation values, unless the factors are independent (Fig. 2C), hence

$$\mathbb{E}\left[\frac{\partial}{\partial p_y}y\frac{\partial}{\partial h_2}p_y\frac{\partial}{\partial p_2}h_2\dots\frac{\partial}{\partial w_1}p_1\right] \neq \mathbb{E}\left[\frac{\partial}{\partial p_y}y\right]\mathbb{E}\left[\frac{\partial}{\partial h_2}p_y\right]\mathbb{E}\left[\frac{\partial}{\partial p_2}h_2\right]\dots\mathbb{E}\left[\frac{\partial}{\partial w_1}p_1\right] .$$

Clearly, in neural networks the factors are not independent, because the activity of downstream neurons depends on the activity of their upstream partners. Thus, it is not obvious how to compute gradients in multi-layer SPMs networks. We will see that stochAD suggests sensible solutions to the aforementioned problems which ultimately justify why we *can* in fact do some of the above operations. Consequently, in the following, we will only consider SGs and stochAD, which support BP.

### 3.2.2 stochAD constitutes the missing theoretical basis for surrogate gradients

Here we show how smoothed stochastic derivatives for stochastic binary MLPs relate to SGs. The smoothed stochastic derivative is defined by Eq. (1) for one realization of the random variable $X(p)$, where the discrete random variables are usually Bernoulli random variables in stochastic binary MLPs.

To gain an intuitive understanding of the essence of smoothed stochastic derivatives, we consider a single realization of a stochastic program, which, in our case, is a stochastic MLP. In other words, we run a single stochastic forward pass in the MLP and condition on its activity. At each point in time, each neuron either emits a one or a zero, i.e., a spike or none. Thus, we can think of all these binary decisions as edges in a tree, with each node representing the spike pattern of all neurons in a given layer. Any roll-out of a forward pass then corresponds to a particular randomly chosen path through this tree (Fig. 3). These paths originate from the same root for a given input $x$, yet the stochasticity in all of them is independent. In this tree of possible spike patterns, it is easy to understand how different methods for gradient approximation work.

Let us first consider the finite differences method. In this approach, the loss is evaluated once using the parameters $w$ and once using $w + \Delta w$, either considering a single trial or averaging over several independent trials. Hence, one randomly samples paths in the tree of spike patterns and compares their averaged output before and after weight perturbation. Since the randomness of the sampled paths is uncoupled, the finite difference approach results in high variance (cf. Fig. 3, gray), which scales with the inverse of the squared perturbation $\Delta w$. However, smaller perturbations allow a more accurate gradient approximation [30, 31]. A joint, i.e. coupled, randomness could clearly reduce the variance.

A better way of sampling, which is at the heart of stochAD, is to condition on the previous layer's output $h_{i-1}$ and then consider all possible spike patterns for the current layer $\mathbb{E}[h_i|h_{i-1}]$. Thus, we can consider the expected layer activity, given a randomly chosen path, e.g., the yellow path in Fig. 3. For this path the derivatives now need to be smoothed at each node. To do so, we compute the expectation in each layer conditioned on the activity of the previous layer along the selected path. After smoothing, it is possible to compute the path-wise derivative along a selected path with activity $h_1^*, h_2^*, y^*$:

$$\frac{\partial}{\partial w_1}\mathbb{E}[y] = \frac{1}{n_{\text{paths}}} \sum_{\text{paths}} \frac{\partial}{\partial p_y}\mathbb{E}[y|h_2^*] \frac{\partial}{\partial h_2}\mathbb{E}[p_y|h_2^*] \frac{\partial}{\partial p_2}\mathbb{E}[h_2|h_1^*] \cdots \frac{\partial}{\partial w_1}\mathbb{E}[p_1|x] \; .$$

Here we averaged over all possible paths, i.e., all possible combinations of the activities $h_1^*, h_2^*$. In practice, it is rarely possible to average over all possible combinations and one instead uses a Monte Carlo estimate, which still yields significantly lower variance than other schemes and can be computed efficiently using BP for a single path per update.

Given the above method for computing smoothed stochastic derivatives, we are now in the position to understand their relationship to SGs. Since we condition on a specific path in the tree of all possible spike patterns, we only compute derivatives of the expected output $h_i$ conditional on the output of the previous layer $h_{i-1}$ according to the chosen path at each node. Such an approach exactly corresponds to treating each node as a unit in the single layer case above. As we saw above, the derivative of the expected output of a single unit is equivalent to the corresponding SD *and* the corresponding smoothed stochastic derivative. Furthermore, when using SGs, there is no difference in how the method is applied in single versus multi-layer cases and there is always a well-defined path to condition on. Thus, SGs can also be understood as treating all units at each layer as single units. Thus, the stochAD framework provides the missing theoretical foundation for SGs when applied to stochastic networks. In conclusion, we find that smoothed stochastic derivatives in the stochAD framework and stochastic SGs are the same for reasonable choices of SD and the escape noise model.

**A word on SGs in deterministic networks.** In practice, SGs are often used in deterministic networks, which are a limit case of stochastic networks. While a stochastic network typically selects a different path on each trial, in the deterministic case there is only one path per input and parameter set. This difference introduces an additional selection bias into the estimation of the stochastic implementation which is generally small in practice (Supplementary Fig. S1). In the next section, we will examine some of the consequences of bias in deterministic networks in more detail.

## 4 Analysis of the bias effects of surrogate gradients

By design SGs are biased because they provide a non-vanishing estimate in situations in which the actual gradient is the zero. It is not clear a-priori what consequences such added bias has and whether SGs are gradients at all, which can be obtained by differentiating a surrogate loss. However, it is difficult to get a quantitative understanding of bias when comparing to the zero vector.

Thus, to take a closer look at biases due to SGs we now move to differentiable network models that have a well-defined gradient. While SG training is not required in such networks, it allows us to develop a quantitative understanding of the commonalities and differences we should expect. This comparison also allows us to check whether SGs satisfy the formal criteria of gradients.

### 4.1 Surrogate gradients introduce bias in differentiable MLPs

We first sought to understand whether SGs point in a "similar direction" as the actual gradient. Specifically, we asked whether SGs ensure sign concordance, i.e., whether they preserve the sign of the gradient components. To investigate

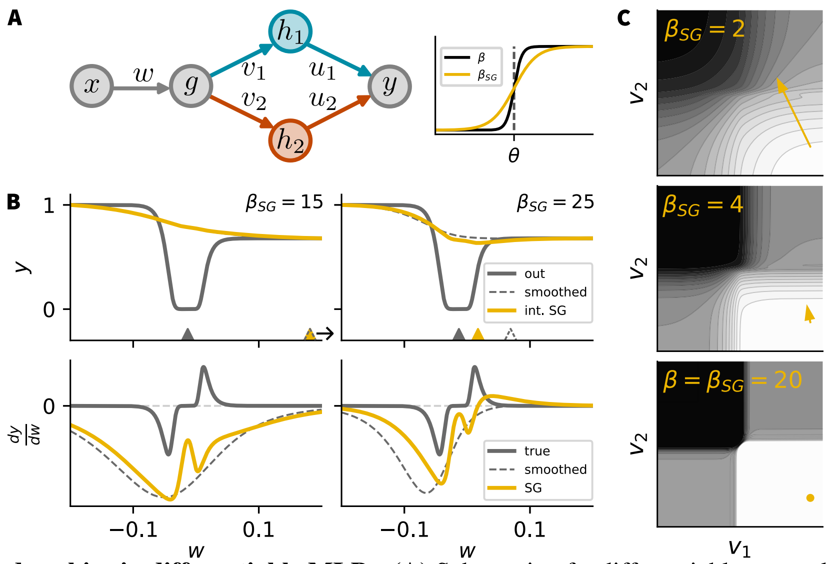


**Figure 4. SGs introduce bias in differentiable MLPs. (A)** Schematic of a differentiable network (left) with sigmoid activations (right) for which we compute an SD using the derivative of a flatter sigmoid (yellow) in contrast to the actual activation (black). **(B)** Top row: Network output (solid gray), smoothed network output (dashed), and integrated SD (yellow) as a function of $w$. The triangles on the x-axis indicate the minimum of the corresponding curves. Bottom row: Derivatives of the top row. Left and right correspond to a flatter ($\beta_{\text{SG}} = 15$) and a steeper ($\beta_{\text{SG}} = 25$) SD (see Table 1 for network parameters). Bottom: Gradient of the curves in the upper row. Note that the actual derivative and the surrogate can have opposite signs. **(C)** Heatmap of the optimization landscape along $v_1$ and $v_2$ for different $\beta_{\text{SG}}$ values (top to bottom). While the actual gradient can be asymptotically zero (see yellow dot, bottom), the SD provides a descent direction, thereby enabling learning (top and middle).

**Table 1. Parameter values for sign flip example.** Parameter values for the network in Fig. 4A, which serve as an example, that SGs can have the opposite sign of the actual gradient and thus point towards the opposite direction. Therefore, we cannot guarantee the SG to align with the actual gradient.

| **Parameter** | $w$ | $v_1$ | $v_2$ | $u_1$ | $u_2$ | $x$ | $\beta$ | $\beta_{\text{SG}}$ |
|---|---|---|---|---|---|---|---|---|
| **Value** | 0 | 0.05 | 0.1 | 1 | -1 | 1 | 100 | 25 |

this question, we consider a small network (Fig. 4A) defined by

$$\begin{aligned} g &= \sigma_\beta(wx) \\ h_1 &= \sigma_\beta(v_1 g) \\ h_2 &= \sigma_\beta(v_2 g) \\ y &= \sigma_\beta(u_1 h_1 + u_2 h2)\,, \end{aligned} \tag{7}$$

which has a well-defined gradient and provides a minimal working example. To study the effect of computing a SD, we replace the derivative of a sigmoid parameterized with $\beta$ by a surrogate function with $\beta_{\text{SG}}$, which is used to compute the SDs during the backward pass

$$\frac{\tilde{\partial}}{\partial u}\sigma(\beta \cdot u) = \beta_{\text{SG}}\sigma(\beta_{\text{SG}} \cdot u) \cdot (1 - \sigma(\beta_{\text{SG}} \cdot u)) \tag{8}$$

with $\beta > \beta_{\text{SG}}$. As before, the deterministic binary Perceptron corresponds to the limiting case $\beta \to \infty$.

To investigate the differences between the SG and the actual gradient, we are particularly interested in the derivative of the output with respect to the hidden layer weight $w$ (cf. Fig. 4A). The partial derivative of the output with respect to $w$ is given as

$$\frac{\partial}{\partial w} y \approx \underbrace{\frac{\tilde{\partial} y}{\partial h_1}\frac{\tilde{\partial} h_1}{\partial g}\frac{\tilde{\partial} g}{\partial w}}_{\text{blue path}} + \underbrace{\frac{\tilde{\partial} y}{\partial h_2}\frac{\tilde{\partial} h_2}{\partial g}\frac{\tilde{\partial} g}{\partial w}}_{\text{maroon path}}\,.$$

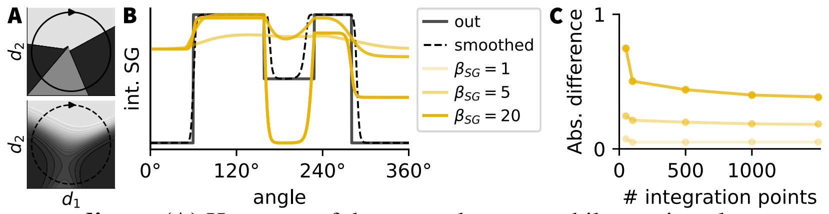


**Figure 5. SGs are not gradients. (A)** Heat map of the network output while moving along two random directions in the parameter space of the example network in a network with step activation (top) and sigmoid activation (bottom) (see Fig. 4 A). The circles indicate a closed integration path through parameter space, starting at the arrow. **(B)** Integral values of SDs as a function of the angle along the closed circular path shown in (A). Different shades of yellow correspond to different values of $\beta_{\text{SG}}$. The black line corresponds to the actual output of the network with step activation function (solid) or sigmoid activation (dashed). The integrated actual derivative of the network with sigmoid activation matches the output (dashed line) and is thus not visible in the plot. **(C)** Absolute difference between actual loss value and integrated SD as function of the number of integration steps. The numerical integrals converge to finite values. Thus the observed difference is not an artifact of the numerical integration.

Now inserting the SDs using (8) leads to

$$\begin{aligned}\frac{\partial}{\partial w}y &\approx \left(\frac{\widetilde{\partial} y}{\partial h_1}\frac{\widetilde{\partial} h_1}{\partial g} + \frac{\widetilde{\partial} y}{\partial h_2}\frac{\widetilde{\partial} h_2}{\partial g}\right)\frac{\widetilde{\partial} g}{\partial w} \\ &= \underbrace{\left(u_1 v_1 \cdot \sigma'_{\beta_{\text{SG}}}(v_1 g) + u_2 v_2 \cdot \sigma'_{\beta_{\text{SG}}}(v_2 g)\right)}_{\text{responsible for sign flip}} \cdot \underbrace{\beta_{\text{SG}}^3 \cdot \sigma'_{\beta_{\text{SG}}}(u_1 h_1 + u_2 h_2) \cdot \sigma'_{\beta_{\text{SG}}}(wx)}_{\text{positive factor}} \cdot x\,, \end{aligned} \tag{9}$$

where only the first factor in Eq. (9), which consists of two terms, determines the relative sign of the SD with respect to the actual derivative. This is because in the second factor, $\beta$, as well as $\beta_{\text{SG}}$, are always positive. Furthermore, the derivative of the sigmoid is always positive independently of the choice of $\beta$ or $\beta_{\text{SG}}$. Finally, $x$, the input data, does not change its sign dependent on $\beta_{\text{SG}}$. However, the first factor can change its sign, since it is a summation of two nonlinear functions with different hyperparameters $\beta$ or $\beta_{\text{SG}}$ and different weights, which may be negative. For instance, when we use specific parameter values (Table 1) in Eq. (9), the SD has the opposite sign of the actual derivative (Fig. 4B). Thus, already in this simple example there is no guarantee that the sign of the SG is preserved with respect to the actual gradient. As a consequence following the SG will not necessarily find parameter combinations that correspond to a minimum of the loss. We expect that this conclusion also holds for the case of binary MLPs or SNNs.

### 4.2 Surrogate gradients are not gradients of a surrogate loss

Given the above realization we wondered whether an SG can be understood as the gradient of a surrogate loss that is not explicitly defined. To answer this question, we note that if, and only if, the SG is the gradient of a scalar function, i.e. corresponds to a conservative field, then the integrating over any closed path must yield a zero integral. To check this, we considered the approximate Jacobian obtained using the SDs in the above example and numerically computed the integral over a closed circular path parameterized by the angle $\alpha$ in the two-dimensional parameter space for different values of $\beta$

$$I_{\text{SG}} = \int_{0^\circ}^{360^\circ} \frac{\widetilde{d} y(\theta_\alpha)}{d\theta_\alpha}\frac{d\theta_\alpha}{d\alpha}d\alpha\,,$$

where the tilde indicates, that we are using SDs (Fig. 5A,B; Methods). We found that integrating the SD did not yield a zero integral, whereas using the actual derivatives resulted in a zero integral as expected. Importantly, this difference was not explained by numerical integration errors due to the finite step size (Fig. 5C). Thus SGs cannot be understood as gradients of a surrogate loss.

## 5 From Perceptrons to LIF neurons

In our above treatment we focused on binary Perceptrons for ease of analysis. In the following we show that our findings readily generalize to networks of LIF neurons. To that end, we consider LIF neurons in discrete time which share many commonalities with the binary Perceptron (see also Fig. 1A). To illustrate these similarities let us consider a single LIF neuron with index $i$ described by the following discrete-time dynamics:

$$
\begin{aligned}
I_i[n+1] &= \lambda_\mathrm{s} I_i[n] + \sum_j w_{ij} S_j[n] && (10)\\
U_i[n+1] &= \left(\lambda_\mathrm{m} U_i[n] + (1-\lambda_\mathrm{m})\, I_i[n]\right)(1 - S_i[n]) && (11)\\
S_i[n] &= \Theta(U_i[n] - \theta)\,, && (12)
\end{aligned}
$$

with $\lambda_\mathrm{s} = \exp\left(-\frac{\Delta t}{\tau_\mathrm{s}}\right)$ and $\lambda_\mathrm{m} = \exp\left(-\frac{\Delta t}{\tau_\mathrm{m}}\right)$, where $\Delta t$ is the time step, $\tau_\mathrm{s}$ is the synaptic time constant, and $\tau_\mathrm{m}$ is the membrane time constant. While the first two equations characterize the linear temporal dynamics, the last equation captures the non-linearity of the neuron, the spiking threshold. Thus in this formulation, we can think of a LIF neuron as a binary Perceptron whose inputs are first processed through a linear filter cascade, i.e., the neuronal dynamics and additionally have a reset mechanism.

In the stochastic case, this filter does not change. In fact we keep the same equations for synaptic current Eq. (10) and membrane potential Eq. (11). However, instead of a deterministic Heaviside function as in Eq. (12), we use a stochastic spike generation mechanism with escape noise

$$
\begin{aligned}
p_i[n] &= \sigma_\beta(U_i[n] - \theta) && (13)\\
S_i[n] &\sim \mathrm{Ber}(p_i[n])\,. && (14)
\end{aligned}
$$

Again, this mechanism is in direct equivalence with the stochastic Perceptron case (cf. Eq. (4)) and permissive of computing smoothed stochastic derivatives.

Finally, the derivative of the current and the membrane potential with respect to the weights induce their own dynamics:

$$
\frac{d}{dw_{ij}} I_i[n+1] = \lambda_\mathrm{s} \frac{d}{dw_{ij}} I_i[n] + S_j[n]
$$

$$
\begin{aligned}
\frac{d}{dw_{ij}} U_i[n+1] &= \lambda_\mathrm{m}(1 - S_i[n]) \frac{d}{dw_{ij}} U_i[n]\\
&\quad + (1-\lambda_\mathrm{m})(1 - S_i[n]) \frac{d}{dw_{ij}} I_i[n]\\
&\quad - \left(\lambda_\mathrm{m} U_i[n] + (1-\lambda_\mathrm{m}) I_i[n]\right) \frac{d}{dw_{ij}} S_i[n]\,, \qquad (15)
\end{aligned}
$$

where we used the product rule to include the derivative of the reset term. To compute the smoothed stochastic derivative of the spike train, we use the affine combination of the left and right smoothed stochastic derivatives of a Bernoulli random variable according to Arya et al. [17] to get

$$
\begin{aligned}
\frac{d}{dw_{ij}} S_i[n] &= \underbrace{\frac{d}{dp_i[n]} \mathrm{Ber}(p_i[n])}_{=1} \frac{d}{dU_i[n]} \sigma_\beta(U_i[n]) \frac{d}{dw_{ij}} U_i[n]\\
&= \underbrace{\frac{d}{dU_i[n]} \sigma_\beta(U_i[n])}_{\mathrm{SD}} \frac{d}{dw_{ij}} U_i[n]\,. \qquad (16)
\end{aligned}
$$

Again, we find that this exactly recovers SGs as defined in Zenke et al. [12] when conditioning on the deterministically spiking path and hence confirms the equality between SGs and smoothed stochastic derivatives.

Going further, we can use Eq. (16) to write Eq. (15) such that it only depends on current and membrane potential derivatives

$$
\begin{aligned}
\frac{d}{dw_{ij}} U_i[n+1] &= \left(\lambda_\mathrm{m}(1 - S_i[n]) - \underbrace{\sigma'_\beta(U_i[n]) \cdot \left(\lambda_\mathrm{m} U_i[n] + (1-\lambda_\mathrm{m}) I_i[n]\right)}_{\text{derivative through reset}}\right) \frac{dU_i[n]}{dw_{ij}} \qquad (17)\\
&\quad + (1-\lambda_\mathrm{m})(1 - S_i[n]) \frac{dI_i[n]}{dw_{ij}}
\end{aligned}
$$

and can be computed forward in time. Most SNN simulators avoid BP through the reset term [12, 15] as this empirically improves performance for poorly scaled SGs [22]. Therefore, it is common practice to set the right-hand term in the

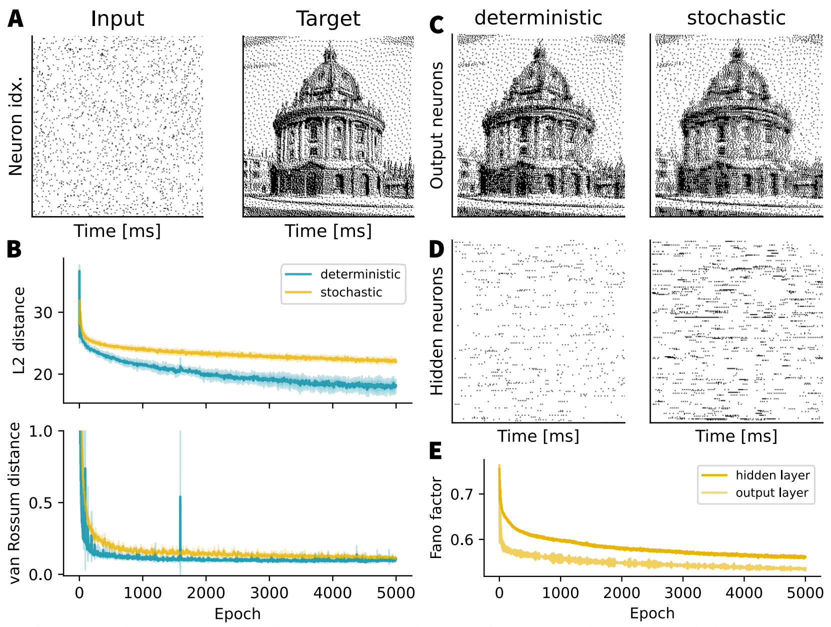


**Figure 6. SGs successfully train stochastic SNNs on a spike train matching task. (A)** Spike raster plots of the input and target spike trains of the spike train matching task. Time is shown on the $x$-axis, the neuron indices are on the $y$-axis and each dot is a spike. The task is to convert the given frozen Poisson input spike pattern into a structured output spike pattern depicting the Radcliffe Camera in Oxford. **(B)** $L_2$ loss (top) and van Rossum distance (bottom) over the course of training for the two network models. While the deterministic network outperforms the stochastic one in terms of $L_2$ distance, the difference is negligible for the van Rossum distance. **(C)** Output spike raster plots after training of the deterministic (left) and the stochastic SNNs (right). Although both methods faithfully reproduce the overall target structure, the deterministic network is slightly better at matching the exact timing. **(D)** Hidden layer spike raster plots. Despite the similar output (see C), the two networks show visibly different hidden layer activity patterns. **(E)** Average Fano factor over the course of training in the hidden and output layer of the stochastic network. Although the stochastic network reduces its variability during training to match the deterministic target, its hidden layer still displays substantial variability at the end of training.

parenthesis in Eq. (17), i.e., the derivative through the reset term, to zero. Conversely, strict adherence to stochAD would suggest keeping the reset term when back-propagating. However, similar to Zenke et al. [22], we find no difference in performance whether we back-propagate through the reset or not when the SG is scaled with $\frac{1}{\beta}$ (see supplementary Fig. S2), while without this scaling, BP through the reset negatively affects performance. Overall, we have shown that our results obtained from the analysis of Perceptrons are transferable to SNNs and hence confirm again the equality between SGs and smoothed stochastic derivatives in the stochAD framework.

## 6 Surrogate gradients are ideally suited for training SNNs

Above we have seen that SG-descent is theoretically justified by stochAD albeit only for stochastic spiking. This finding also suggests that SG-descent is suitable for training stochastic SNNs, with deterministic SNNs being only a special case (cf. Fig. 3). Next, we wanted to confirm this insight numerically. To that end, we trained deterministic and stochastic SNNs with SG-descent on a deterministic spike train matching task and a classification task.

For the spike train matching task, we assumed 200 input neurons with frozen Poisson spike trains. We then set up a feed-forward SNN with one hidden layer, initialized in the fluctuation-driven regime [32]. We used supervised SG training of a deterministic and stochastic version of the same network to match 200 target spike trains which were given by a dithered picture of the Radcliffe Camera (Fig. 6A; Methods). Both networks learned the task (Fig. 6B&C), albeit with visibly different hidden layer activity (Fig. 6D). The deterministic version outperformed the stochastic SNN in terms of $L_2$-distance at 1 ms, i.e. the temporal resolution of the simulation. However, when comparing their outputs according to the van Rossum distance [33] with an alpha-shaped filter kernel equivalent to the $\epsilon$-kernel of the LIF neuron ($\tau_{\text{mem}} = 10$ ms, $\tau_{\text{syn}} = 5$ ms), we found no difference in loss between the stochastic and deterministic networks (Fig. 6B). Finally, we observed that the Fano factor, a measure of stochasticity, of the stochastic SNN dropped during

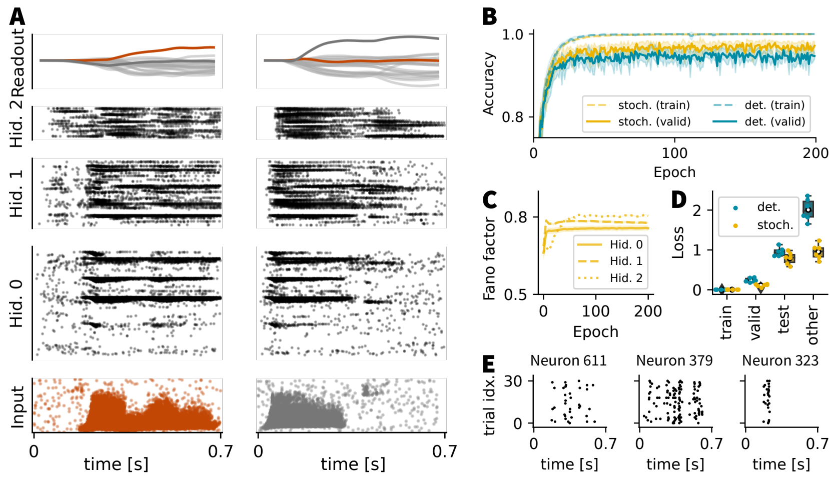


**Figure 7. SGs can successfully train stochastic CSNNs. (A)** Snapshot of network activity for two correctly classified sample inputs from the SHD dataset. Top: readout unit activity over time, the colored line indicates the activity of the unit that corresponds to the correct class. Below: Spike raster plots of the three convolutional hidden layers. The spike raster plots of the inputs are shown at the bottom in red and gray. Time is on the x-axis, the neuron index is on the y-axis and each dot represents a spike. **(B)** Learning curves for training (dashed) and validation (solid) accuracy for the stochastic and the deterministic case (average over $n = 6$ trials $\pm$ std). **(C)** The mean Fano factor of the different layers in the stochastic network over the course of training ($\pm$ std, $n = 3$). **(D)** The first three pairs of boxes show train, validation, and test loss of the CSNN as in (A) for the stochastic and the deterministic case for $n = 6$ random initializations. The rightmost boxes show test loss for the opposite activation function. This means the network trained deterministically is tested with stochastic activation and vice versa. **(E)** Raster plots over trials of the spiking activity of three randomly chosen units from the second hidden layer. The units show clear trial-to-trial variability reminiscent of cortical activity.

training to better accommodate the deterministic target (Fig. 6E). In summary, we find that the stochastic network exhibits comparable performance to the deterministic network. Thus SGs are suitable for training stochastic SNNs.

To verify that this finding generalizes to a more complex task, we trained a convolutional spiking neural network (CSNN) with three hidden layers on the Spiking Heidelberg Digits (SHD) dataset [34] with maximum-over-time readout (Methods). Like above, the stochastic network learned to solve the task with comparable training and validation accuracy to the deterministic network (Fig. 7B). Specifically, we found that the stochastic CSNN achieved a validation accuracy of $96 \pm 2$ %, (test $86 \pm 1$ %), compared to $96 \pm 2$ % (test $84 \pm$ 1%) for the deterministic CSNN. Furthermore, in the case of the stochastic SNN, we left the escape noise active during validation and testing. However, in Fig. 7D, one can see that the stochastic network performs equally well when tested in a deterministic environment (Fig. 7D, others). Conversely, the deterministically trained network does not perform well when evaluated under stochastic spike generation. In contrast to the previous task, we found that the Fano factor remained high during training (Fig. 7C), i.e. stochasticity is preserved. This is also reflected in a high trial-to-trial variability as shown in Fig. 7E. One can see that the spiking activity for three example neurons shows a high variability across trials for the same input. Thus, even for a more complex task, SG-descent is well suited for training stochastic SNNs, and stochasticity is preserved after training.

## 7 Discussion

We investigated the theoretical basis of the widely used SG descent method and showed that SGs can be formally derived for stochastic networks from the stochAD framework [17]. While SPMs rely on smoothing the loss function and are, therefore, ill-suited for AD, stochAD sidesteps this problem by defining smoothed stochastic derivatives that can be combined with the chain rule. We saw that SGs can be understood as the deterministic limiting case of stochAD. Moreover, our analysis uncovered two important insights: First, SGs provide biased gradient approximations in deterministic networks with well-defined gradients. Second, SGs do not correspond to gradients of a surrogate loss. Finally, we showed in simulations that SGs can directly train stochastic SNNs that exhibit comparable or better performance while preserving trial-to-trial variability reminiscent of cortical neurons in neurobiology.

**Related work.** The issue of non-existing derivatives is not unique to SNNs but is a well-known challenge when dealing with discrete random variables and their derivatives. It hence arises in various contexts, such as in general binary neural networks [10], sigmoid belief nets [35], or when dealing with categorical distributions [36, 37]. To address this challenge in arbitrary functions including discrete random variables, often the score function estimator, also called REINFORCE [38] is applied, as it provides unbiased estimates. This can be computed through stochastic computation graphs [39]. While Arya et al. [17] use coupled randomness to reduce variance, there are also methods building on reinforcement learning for low variance gradient estimation [40]. Conversely, to address the challenge of BP through non-differential functions in neural networks instead, a commonly used solution is the STE [24, 25], which replaces the derivative of a threshold function with 1, also called the identity STE in Yin et al. [26]. For SNNs the term SG is commonly used in deterministic networks where it refers to an STE with a nonlinear SD [14]. While both are successful in practice, we still lack a complete theoretical understanding. Bengio et al. [25] introduced the identity STE as a biased estimator, which guarantees the correct sign only in shallow networks. Yin et al. [26] studied coarse gradients, including the STE. They analyzed training stability and investigated which choice of STEs leads to useful weight updates. They concluded that the identity STE does not, while the rectified linear unit (ReLU) and clipped ReLU versions do. They further found that the identity STE might be repelled from certain minima. However, they studied this either in a network with only one nonlinearity or in an activation quantized network with a quantized ReLU as opposed to a Heaviside activation function, as we have in the case of SNNs. Liu et al. [41] showed that the identity STE provides a first-order approximation to the gradient, which was previously shown by Tokui et al. [42] specifically for Bernoulli random variables. However, other STEs were not considered, such as e.g. the sigmoidal STE, which would correspond to our SG. Another approach to dealing with discrete distributions was pursued by Maddison et al. [36] and Jang et al. [37]. They proposed a solution similar to the reparametrization trick but combined with a smooth relaxation of the categorical distribution. This amounts to training a network with continuous representations by sampling from a Gumbel-Softmax, while after training, the temperature of the Gumbel-Softmax distribution is set to zero to obtain a discrete output. Hence, while providing discrete output after training, such a network is trained with continuous neuronal outputs rather than spikes.

Interestingly, most of these solutions deal with discrete functions in the stochastic case, such as stochastic binary Perceptrons or a Bernoulli random variable, where noise is used for smoothing. But as we saw above, probabilistic approaches in the context of SPMs cannot be applied to deep SNNs without approximations. Therefore, stochastic SNN models have been implemented in the past mainly for theoretical and biological plausibility reasons ([19, 20]). Nevertheless, there were also approaches that made it possible to train multi-layer SNNs with AD, such as the MultilayerSpiker proposed by Gardner et al. [21], which uses the spike train itself as the SD of a spike train. Today, SNNs are mostly implemented as networks of deterministic LIF neurons with a Heaviside function as spiking non-linearity, as they tend to have higher performance. Recently, however, Ma et al. [43] found empirically that stochastic SNNs with different types of escape noise can be trained with SG descent to high performance. While they showed a connection to deterministic SG descent, they did not discuss the implications for deep networks with stateful LIF neurons. Again, the effectiveness of SG descent in stochastic SNNs was confirmed, as suggested by the connection to the stochAD framework. Thus, our work not only provides empirical confirmation of the effectiveness of SGs in training stochastic SNNs but also provides a comprehensive theoretical explanation for their success beyond shallow networks.

**Limitations.** To gain insight into the theory behind SGs, we had to make some simplifying assumptions to allow an analytical investigation of the problem. Consequently, we performed our theoretical analysis on Perceptrons, which can be considered as a special limiting case of LIF neurons without memory. Thanks to this analogy, we were able to transfer analytical implications to the training of SNN with LIF neurons. For continuous-time simulations this is not as straightforward, as there is no well-defined function for how the spikes are derived from the membrane potential. For discretized LIF neurons, however, we were able to translate the results from binary perceptrons directly, as such networks can be understood as a special type of binary recurrent neural networks (RNNs) [14] where the output is given by a Heaviside step function.

Another point is that in LIF models, unlike Perceptrons, the question of how to deal with the reset is not decided. Smoothed stochastic derivatives allow and recommend running BP through the reset. However, previous work has recommended not running BP through the reset, as this leads to more robust performance, especially when the SG is not properly scaled [12, 15, 22]. Both options are possible when using SG-descent, and we have not noticed much difference in performance when scaling correctly. However, omitting the scaling by $\frac{1}{\beta}$ (as in the asymptotic SuperSpike in Zenke et al. [12]) slowed down learning (see Supplementary Fig. S2C). While omitting this scaling can potentially be counteracted by an optimizer using per-parameter learning rates, the prefactor due to $\beta$ in the unscaled case could become quite large across layers and thus still affect performance.

Next, while the bias of a gradient estimator is well defined for differentiable neural networks for which a gradient exists, this is not the case for deterministic SNNs. Therefore, in this article we have had to content ourselves to discussing the bias with respect to the gradient of the expected output in a corresponding stochastic SNN. However, smoothed stochastic derivatives are themselves biased with respect to the gradient of the expected output of the stochastic SNNs. This bias results from the dependence of the update step on the selection of a particular path for which the path-wise derivative is computed. While this bias can be reduced by averaging over many paths, such averaging does not work for deterministic SNNs where there is only one path per input and parameter set. This shows that deterministic SGs are biased with respect to the gradient of the corresponding stochastic SNNs. Furthermore, when we empirically analyzed the effects of the bias in deterministic SGs, we were again restricted to an environment with an existing gradient. Therefore, we have instead shown that SGs induce bias in deterministic networks with well-defined gradients. In the limiting case $\beta \to \infty$, these networks are equivalent to Perceptron networks with step activation functions. Thus, although we could not extrapolate our results to the case of SNN, we consider the above approaches to be a good approximation as they preserve the main properties of SGs.

Historically, stochastic SNNs have been evaluated on tasks that require exact deterministic spike trains at readout such as in the spike train matching task ([19–21]. However, despite being performed with stochastic SNNs such tasks actually punish stochasticity, at least in the readout. Therefore, stochastic SNNs respond by becoming more deterministic during training as we have observed when monitoring the Fano factor (cf. Fig. 6). In our setting, neurons have the possibility to change the effective steepness of the escape noise function by increasing their weights, as it is dependent on $\beta \cdot |w|$, and thus get more deterministic. Therefore, we believe that tasks which do not punish stochasticity, such as classification tasks, are much more suitable for evaluating the performance of stochastic SNNs as it allows training of well-performing SNNs that still show substantial variability in their responses.

**Future research.** We saw that SGs generally decrease the loss but they do not necessarily find a local minimum of the loss. This discrepancy is due to bias in the gradient approximation. However, learning in SNNs is only possible due to bias, because the actual gradient is almost always zero. This dichotomy extends even to networks with well-defined gradients, but a rough loss landscape, in which mollifying is successful in practice [44]. In conventional ANN training, bias is similarly often desirable. For instance, many optimizers rely on momentum to speed up training. This raises questions about the impact of bias on optimization effectiveness. Thus, an interesting line of future work is to study what constitutes a good local minimum in the case of SNNs, which types of bias help finding them, and how this bias relates to SGs.

We linked SG-descent to smoothed stochastic derivatives, which are biased with respect to standard stochastic derivatives. However, the latter can can only be computed in forward-mode AD, which is impractical for neural network training, as it comes with an increased computational cost. Hence, an interesting direction of future research is to investigate whether training improvements due to the bias reduction would justify the added computational load and whether the load could be reduced to make it applicable to SNN training, for instance, by using mixed-mode AD, such as in DECOLLE [45] or online spatio-temporal learning [46] (see [47] for a review).

In this article, we found an effective way of training stochastic SNNs. This ability is not only relevant to study the brain, where biological neurons exhibit trial-to-trial variability, but also for neuromorphic ultra-low-power applications in which circuits may be similarly noisy. The role of variability in biological systems is not fully understood and functional stochastic SNN models will help to further our understanding of the role of variability in the brain and may prove as a potential sources of representational drift [48]. Thus, it would be interesting to study how learning influences representational changes in plastic functional stochastic SNNs and whether we can identify specific dynamic signatures that allow drawing conclusions about the underlying learning mechanisms in the brain.

In conclusion, SGs are a valuable tool for training both deterministic and stochastic SNNs. In this article, we saw that stochAD provides a theoretical backbone to SGs learning which naturally extends to stochastic SNNs. Training such stochastic SNNs is becoming increasingly relevant for applications with noisy data or noisy hardware substrates and is essential for theoretical neuroscience as it opens the door for studying functional SNNs with biologically realistic levels of trial-to-trial variability.

## 8 Methods

Our Perceptron and SNN models were written in Python 3.10.4 and extended either on the stork library [32] which is based on Pytorch [49], or Jax [50]. The code repository can be found at `https://github.com/fmi-basel/surrogate-gradient-theory`. All models were implemented in discrete time. The following sections give more details on the different neuron models before providing all necessary details to the two learning tasks, including architecture and parameters used for a given task.

### 8.1 Neuron models

**Perceptron.** As mentioned in the theoretical results 3.1, the Perceptron is a simplified version of the LIF neuron model, where there is no memory and no reset. The membrane dynamics $U_i^l$ for Perceptron $i$ in layer $l$ are given by for a special case by Eq. (2) in the theoretical results section, and in general by

$$U_i^l \quad = \quad \sum_j w_{ij} S_j + b_i \ ,$$

where $w_{ij}$ are the feed-forward weights, $S_j$ the binary outputs (spikes) of the previous layer and $b_i$ is an optional bias term.

**Leaky integrate-and-fire neuron.** We used an LIF neuron model with exponential current-based synapses [18]. In discrete time, the membrane potential $U_i^l[n]$ of neuron $i$ in layer $l$ at time step $n$ is given by

$$U_i^l[n+1] = \left(\lambda_{mem} \cdot U_i^l[n] + (1 - \lambda_{mem}) \cdot I_i^l[n]\right) \cdot (1 - S_i^l[n]) \ , \tag{18}$$

where $I_i^l[n]$ is the input current and $S_i^l[n]$ output spike of the neuron itself, which governs reset dynamics. With multiplicative reset as above, the neuron stays silent for one time step after each spike. The membrane decay variable $\lambda_{mem} = \exp\left(-\frac{\Delta t}{\tau_{\text{mem}}}\right)$ is defined through the membrane time constant $\tau_{\text{mem}}$ and the chosen time step $\Delta t$. The input current $I_i^l[n]$ is given by

$$I_i^l[n+1] = \lambda_{syn} \cdot I_i^l[n] + \underbrace{\sum_j w_{ij}^l \cdot S_j^{l-1}[n]}_{\text{feedforward}} + \underbrace{\sum_k v_{ik}^l \cdot S_k^l[n]}_{\text{recurrent}} \ , \tag{19}$$

where $w_{ij}$ and $v_{ik}$ are the feedforward and recurrent synaptic weights, respectively, corresponding to the previous layers' spikes $S_j^{l-1}$ and the same layers' spikes $S_k^l$, respectively. The synaptic decay variable is again given as $\lambda_{syn} = \exp\left(-\frac{\Delta t}{\tau_{\text{syn}}}\right)$, defined through the synaptic time constant $\tau_{\text{syn}}$. At the beginning of each minibatch, the initial membrane potential value of all neurons was set to their resting potential $U_i^l[0] = U_{rest} = 0$ and the initial value for the synaptic current was zero as well, $I_i^l[0] = 0$.

### 8.2 Spike generation

The generation of a spike follows the same mechanism in both, the LIF and the Perceptron neuron model. Depending on the membrane potential value, the activation function generates a spike or not. The following paragraphs highlight the differences between the deterministic and the stochastic cases.

**Deterministic spike generation.** A spike is generated in every time step, in which the membrane potential crossed a threshold $\theta$. Hence, we were using the Heaviside step function $\Theta$ to generate a spike deterministically at time step $n$:

$$S_i^l[n] = \Theta\left(U_i^l[n] - \theta\right) \ . \tag{20}$$

**Stochastic spike generation.** A stochastic neuron model with escape noise [18] may spike, even if the membrane potential is below the threshold or vice versa not spike, even if the membrane potential is above the threshold. Therefore, in discrete time, the probability of spiking in time step $n$ is

$$p_i^l[n] = f(U_i^l[n] - \theta) = \sigma_\beta\left(U_i^l[n] - \theta\right) \tag{21}$$

for each neuron. We used $\sigma_\beta(x) = \frac{1}{1+\exp(-\beta \cdot x)}$, if not stated otherwise. The hyperparameter $\beta$ defines the steepness of the sigmoid, with $\beta \to \infty$ leading to deterministic spiking. Spikes were then drawn from a Bernoulli distribution with probability $p_i^l[n]$, hence

$$S_i^l[n] \sim \text{Ber}(p_i^l[n]) \ . \tag{22}$$

The expected value of spiking equals the spike probability, so $\mathbb{E}\left[S_i^l[n]\right] = p_i^l[n]$.

### 8.3 Differentiable example network

The minimal example network (Fig. 4A) was simulated using Jax [50]. To implement the main essence of SG descent in a differentiable network, we constructed a network of Perceptrons with sigmoid activation functions (see Eq. (7)). The SG was implemented by a less steep sigmoid, which was used instead of the actual activation function on the backward pass.

**Integrated (surrogate) gradients.** In general, the integral of the derivative of the loss should equal again the loss $\int \frac{\partial \mathcal{L}}{\partial w} dw = \mathcal{L}$. The same holds true for the network output. In Fig. 4, we computed this quantity for $w \in [-0.2, 0.2]$, as well as in two dimensions for the parameters $v_1$ and $v_2$. In Fig. 5, we did not compute this along a line but instead chose to compute this along a closed path in parameter space, i.e. a circle. To do so, we integrated the SG along a circle in a two-dimensional hyperplane which is spanned by two randomly chosen orthonormal vectors $d_1$ and $d_2$ in parameter space. The position along the circle is defined by an angle $\alpha$ and the circle is parametrized by $a$ and $b$ such that $a = r \cdot \sin(\alpha)$ and $b = r \cdot \cos(\alpha)$ with $r$ the radius of the circle. Hence, when integrating along the circle, the weights $\theta$ in the network changed according to the angle $\alpha$

$$\theta_\alpha = d_1 \cdot a(\alpha) + d_2 \cdot b(\alpha) \; .$$

### 8.4 Learning tasks

We trained SNNs to evaluate the differences in learning between the stochastic and deterministic SNN models trained with SG-descent. For this purpose, we chose a spike train matching and a classification task to cover different output modalities.

#### 8.4.1 Spike train matching

If an SNN can learn a precise timing of spikes, it must be able to match its output spike times with some target output spike times. Therefore in the spike train matching task, we ask, whether both, the stochastic and the deterministic network can learn deterministic output spike trains. For the training, no optimizer is used and since we use a minibatch size of one, this means we apply batch gradient descent. The details about the used architecture, neuronal, and training parameters are given in the following paragraphs as well as in Tables 2 and 3 for both, the stochastic and the deterministic versions of the network. The code was written in Jax.

**Task.** The target spike trains to match were generated from a picture of the Radcliffe Camera in Oxford using dithering to create a binary image. We set this image as our target spike raster, where the x-axis corresponds to time steps and the y-axis is the neuron index. Hence, the image provides a binary spike train for each readout neuron. As an input, we used frozen Poisson-distributed spike trains with a per-neuron firing rate of 50 Hz. This created a one-image dataset that requires 200 input neurons, and 200 output neurons and has a duration of 198 time steps, i.e. 198 ms when using $\Delta t = 1$ ms.

**Network architecture.** The above described spike train matching task is an easy task, that could also be solved by an SNN without a hidden layer. However, since we knew that in the shallow case without a hidden layer, the only difference between our stochastic and our deterministic versions would lie in the loss, we decided to use a network with one hidden layer. Hence the used network architecture was a feed-forward network with 200 input units, 200 hidden units, and 200 readout units, run with $\Delta t = 1$ ms for a total of 198 time steps (see also Table 3). Weights were initialized in the fluctuation-driven regime [32] with a mean of zero and target membrane fluctuations $\sigma_U = 1$. For the stochastic networks, we averaged the update over ten trials, i.e., we sampled ten paths in the tree of spike patterns, used each of them to approximate a gradient and took their average to perform the weight update before sampling another path for the next weight update (see Fig. 3).

**Reset at same time step.** Matching a target binary image without further constraints with a spike train might require a neuron to spike in two adjacent time steps. However, this is not possible with the membrane dynamics as in Eq. (18), where after every spike the neuron stays silent for one time step. Hence for this task, we used slightly modified membrane dynamics

$$U_i^l[n+1] = \lambda_{mem} \cdot U_i^l[n] \cdot (1 - S_i^l[n]) + (1 - \lambda_{mem}) \cdot I_i^l[n] \; , \tag{23}$$

where the reset was applied at the same time step as the spike and thus high enough input in the next time step could make the neuron fire again.

**Loss functions.** We trained the network using an $L_2$ loss function

$$L_2 = \frac{1}{N} \sum_{i=1}^{M^L} \sum_{n=1}^{T} \left( S_i^L[n] - \hat{S}_i[n] \right)^2 \tag{24}$$

to compute the distance between target spike train $\hat{S}_i$ for readout neuron $i$ and output spike train $S_i^L[n]$ at the readout layer $L$, where $M^L$ is the number of readout neurons and $T$ is the number of time steps. Furthermore, we also monitor the van Rossum distance [33]

$$L_{vR} = \frac{1}{2} \int_{-\infty}^{t} \left( \left( \alpha \hat{S}_i - \alpha S_i \right)(s) \right)^2 ds \tag{25}$$

between the target spike train and the output spike train, which might be a better distance for spike trains, since it also takes temporal structure into account by punishing a spike that is five time steps off more than a spike that is only one time step off. It does so, by convolving the output and target spike train first with a temporal kernel $\alpha$, before computing the squared distance. We chose $\alpha$ to be equivalent to the $\epsilon$-kernel of the LIF neuron

$$\alpha(t) = \frac{1}{1 - \frac{\tau_{\text{mem}}}{\tau_{\text{syn}}}} \left( \exp\left(-\frac{t}{\tau_{\text{syn}}}\right) - \exp\left(-\frac{t}{\tau_{\text{mem}}}\right) \right)$$

with $\tau_{\text{mem}} = 10$ ms, $\tau_{\text{syn}} = 5$ ms.

**Fano factor.** The Fano factor $F$ was calculated on the hidden layer and output spike trains $S$ as $F = \frac{\sigma_S^2}{\mu_S}$.

### 8.4.2 Classification on SHD

To evaluate performance differences, we also evaluated both the stochastic and the deterministic version on a classification task using a deep recurrent CSNN (as in Rossbroich et al. [32]), hence also increasing the difficulty for the stochastic model, which now had to cope with multiple noisy layers. The details about the used architecture, neuronal, and training parameters are given in the following paragraphs as well as in Tables 2 and 3 for both, the stochastic and the deterministic versions of the network.

**Task.** For the classification task, we used the SHD dataset [34], which is a real-world auditory dataset containing spoken digits in German and English from different speakers, hence it has 20 classes. The dataset can be downloaded from `https://ieee-dataport.org/open-access/heidelberg-spiking-datasets`. Cramer et al. [34] preprocessed the recordings using a biologically inspired cochlea model to create input spike trains for $n = 700$ neurons. The duration of the samples varies, hence we decided to consider only the first $T_{\text{SHD}} = 700$ ms of each sample, which covers $> 98\%$ of the original spikes. For our numerical simulations, we binned the spike trains using a $\Delta t = 2$ ms and hence we had $\frac{\Delta t}{T_{\text{SHD}}} = 350$ time steps in this task. 10% of the training data was used as a validation set, and for testing we used the officially provided test set, which contains only speakers that did not appear in the training set.

**Network architecture.** We used a deep recurrent CSNN for the SHD classification task (see also Table 3). There were 700 input neurons, that directly get fed the input spikes from the SHD dataset. The network had three recurrently connected hidden layers and used one-dimensional convolution kernels. Weights were initialized in the fluctuation-driven regime with a mean of zero, target membrane fluctuations $\sigma_U = 1$, and the proportion of fluctuations due to the feed-forward input was $\alpha = 0.9$. The network size as well as the parameters for the feed-forward and recurrent convolutional operations are summarized in Table 3. Again, we performed only one trial per update for the stochastic case.

**Readout units.** As opposed to the previous task, a classification task requires special readout units. The readout units did have the same membrane and current dynamics as a normal LIF neuron (Eqns. (18) and (19)), but they did not spike. Furthermore, we used a different membrane time constant $\tau_{\text{mem}}^{\text{RO}} = T_{\text{SHD}} = 700$ ms for the readout units. This allowed us to read out their membrane potential for classification (see paragraph on the loss function).

**Activity regularization.** To prevent tonic firing, we used activity regularization to constrain the upper bound of firing activity. To this end, we constructed an additional loss term as a soft upper bound on the average firing activity of each feature in our convolutional layers. Hence for every layer, we computed a term

$$g_{upper}^{l,k} = \left( \left[ \frac{1}{M^l} \sum_{i}^{M^l} \zeta_i^{l,k} - \vartheta_{upper} \right]_+ \right)^2 , \tag{26}$$

where $M^l = n_{features} \times n_{channels}$ is the number of neurons in layer $l$ and $\zeta_i^{l,k} = \sum_n^T S_i^{l,k}[n]$ is the spike count of neuron $i$ in layer $l$ given input $k$. We chose the parameter $\vartheta_{upper} = 7$ to constrain the average firing rate to 10 Hz. Hence, the total upper bound activity regularization loss is given by

$$L_{upper} = -\lambda_{upper} \sum_{l=1}^{L} \sum_{f=1}^{F^l} g_{upper}^{l,k} , \tag{27}$$

where $\lambda_{upper} = 0.01$ was the regularization strength. Those parameters can also be found in Table 3.

**Loss function.** As this is a classification task, we used a standard cross-entropy loss

$$L_{CE} = -\frac{1}{K}\sum_{i=1}^{k}\sum_{c=1}^{C} y_c^k \log\left(p_c^k\right) \tag{28}$$

to sum over all samples $K$ and all classes $C$. The correct class is encoded in $y_c^k$ as a one-hot encoded target for the input $k$. To compute the single probabilities $p_c^k$ for each class $c$, we first read out the maximum membrane potential values of each readout neuron over simulation time to get the activities

$$a_c^k = \max_n(U_c^L[n])\ . \tag{29}$$

From those activities, we computed the probabilities using a Softmax function $p_c^k = \frac{\exp(a_c^k)}{\sum_{c'}^{C}\exp(a_{c'}^k)}$.

**Optimizer.** We used the squared mean over root mean squared cubed (SMORMS3) optimizer [51], which chooses a learning rate based on how noisy the gradient is. The SMORMS3 optimizer keeps track of the three values $g_1$, $g_2$ and $m$, which were initialized to $g = g_2 = 0$ and $m = 1$. They are updated after every minibatch as follows:

$$\begin{aligned} r &= \frac{1}{m+1} \\ g &= (1-r)\cdot g + r\cdot\left(\frac{\partial\mathcal{L}}{\partial\theta}\right) \\ g_2 &= (1-r)\cdot g_2 + r\cdot\left(\frac{\partial\mathcal{L}}{\partial\theta}\right)^2 \\ m &= 1 + m\cdot\frac{1-g^2}{g_2+\epsilon}\ . \end{aligned}$$

Given this, SMORMS3 computes the current effective learning rate as $\eta_{\text{current}} = \frac{\min\left(\eta, \frac{g^2}{g_2+\epsilon}\right)}{\sqrt{g_2+\epsilon}}$, where $\eta$ is the initially chosen learning rate and $\epsilon$ is a small constant to avoid division by zero. Therefore the parameter update is $\Delta\theta = -\eta_{\text{current}}\cdot\frac{\partial\mathcal{L}}{\partial\theta}$.

**Fano factor.** The Fano factor was calculated after taking a moving average over the spike train $S$ with a window of 10 timesteps (20 ms) to get $S_{\text{bin}}$. Subsequently, the Fano factor $F$ was computed as

$$F = \frac{\sigma^2_{S_{\text{bin}}}}{\mu_{S_{\text{bin}}}}\ .$$

**Table 2. Noise and surrogate function parameters.** Parameters used in our numerical simulations for feed-forward networks on the spike-train matching task and deep recurrent CSNNs on the SHD classification task. We selected the learning rate based on best validation accuracy (cf. Supplementary Fig. S2B). The SuperSpike non-linearity $h(x)$ is the derivative of a fast sigmoid scaled by $\frac{1}{\beta}$: $h(x) = \frac{1}{(\beta|x|+1)^2}$.

| | **Spike train matching task** | | **Classification task** | |
|---|---|---|---|---|
| | stochastic | deterministic | stochastic | deterministic |
| Number of trials | 10 | 1 | 1 | 1 |
| Learning rate | $\eta_{\text{hid}} = 10^{-05}$ $\eta_{\text{out}} = 10^{-05}$ | $\eta_{\text{hid}} = 10^{-05}$ $\eta_{\text{out}} = 10^{-04}$ | 0.01 | 0.01 |
| *Escape noise* | | | | |
| Function | $\sigma(\cdot)$ | step | $\sigma(\cdot)$ | step |
| Parameter | $\beta_{\text{hid}} = 10$ $\beta_{\text{out}} = 100$ | - | $\beta = 10$ | - |
| *Surrogate gradient* | | | | |
| Function | $\sigma'(\cdot)$ | SuperSpike | SuperSpike | SuperSpike |
| Parameter | $\beta_{\text{hid}} = 10$ | $\beta = 10$ | $\beta = 10$ | $\beta = 10$ |

**Table 3. Network and training parameters.** Parameters used in numerical simulations for feed-forward SNNs on the spike-train matching task and deep recurrent CSNNs on the SHD classification task.

| | **Spike train matching task** | **Classification task** |
|---|---|---|
| Dataset | Radcliffe Camera | SHD |
| No. input neurons | 200 | 700 |
| No. hidden neurons | 200 | 16-32-64 |
| No. output neurons | 200 | 20 |
| No. training epochs | 5000 | 200 |
| Time step | 1 ms | 2 ms |
| Duration | 198 ms | 700 ms |
| Mini-batch size | 1 | 400 |
| Kernel size (ff) | - | 21-7-7 |
| Stride (ff) | - | 10-3-3 |
| Padding (ff) | - | 0-0-0 |
| Kernel size (rec) | - | 5 |
| Stride (rec) | - | 1 |
| Padding (rec) | - | 2 |
| Loss | $L_2$ or van Rossum distance [33] | Maximum over time |
| Optimizer | None (gradient descent) | SMORMS3[51] |
| *Neuronal parameters* | | |
| Spike threshold | 1 | 1 |
| Resting potential | 0 | 0 |
| $\tau_{\text{mem}}$ | 10 ms | 20 ms |
| $\tau_{\text{syn}}$ | 5 ms | 10 ms |
| $\tau_{\text{mem}}^{\text{RO}}$ | - | 700 ms |
| Reset | at same time step | at next time step |
| *Activity regularizer* | | |
| $\vartheta_{\text{upper}}$ | - | 7 |
| $\lambda_{\text{upper}}$ | - | 0.01 |

## Acknowledgments

The authors thank Frank Schäfer, Guillaume Bellec, and Wulfram Gerstner for stimulating discussions. This work was supported by the Swiss National Science Foundation (Grant Number PCEFP3_202981) and the Novartis Research Foundation.

## References

[1] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836. DOI: 10.1038/323533a0.

[2] Gütig, R. and Sompolinsky, H. “The Tempotron: A Neuron That Learns Spike Timing–Based Decisions”. In: *Nature Neuroscience* 9.3 (Mar. 2006), pp. 420–428. ISSN: 1546-1726. DOI: 10.1038/nn1643.

[3] Memmesheimer, R.-M., Rubin, R., Ölveczky, B. P., and Sompolinsky, H. "Learning Precisely Timed Spikes". In: *Neuron* 82.4 (May 2014), pp. 925–938. ISSN: 0896-6273. DOI: 10.1016/j.neuron.2014.03.026.

[4] Huh, D. and Sejnowski, T. J. "Gradient Descent for Spiking Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018.

[5] Bohte, S. M., Kok, J. N., and La Poutré, H. "Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons". In: *Neurocomputing* 48.1-4 (Oct. 2002), pp. 17–37. ISSN: 09252312. DOI: 10.1016/S0925-2312(01)00658-0.

[6] Mostafa, H. "Supervised Learning Based on Temporal Coding in Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2017), pp. 1–9. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2017.2726060.

[7] Wunderlich, T. C. and Pehle, C. "Event-Based Backpropagation Can Compute Exact Gradients for Spiking Neural Networks". In: *Scientific Reports* 11.1 (June 2021), p. 12829. ISSN: 2045-2322. DOI: 10.1038/s41598-021-91786-z.

[8] Klos, C. and Memmesheimer, R.-M. *Smooth Exact Gradient Descent Learning in Spiking Neural Networks*. Sept. 2023. arXiv: 2309.14523 [cs, q-bio].

[9] Bohte, S. M. "Error-Backpropagation in Networks of Fractionally Predictive Spiking Neurons". In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Ed. by Honkela, T., Duch, W., Girolami, M., and Kaski, S. Berlin, Heidelberg: Springer, 2011, pp. 60–68. ISBN: 978-3-642-21735-7. DOI: 10.1007/978-3-642-21735-7_8.

[10] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *arXiv:1602.02830 [cs]* (Feb. 2016). arXiv: 1602.02830.

[11] Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., Andreopoulos, A., Berg, D. J., McKinstry, J. L., Melano, T., Barch, D. R., Nolfo, C. di, Datta, P., Amir, A., Taba, B., Flickner, M. D., and Modha, D. S. "Convolutional networks for fast, energy-efficient neuromorphic computing". In: *Proceedings of the National Academy of Sciences of the United States of America* 113.41 (Oct. 2016), pp. 11441–11446. ISSN: 0027-8424. DOI: 10.1073/pnas.1604850113.

[12] Zenke, F. and Ganguli, S. "SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks". In: *Neural Computation* 30.6 (May 2018), pp. 1514–1541. ISSN: 1530888X. DOI: 10.1162/neco_a_01086. arXiv: 1705.11146.

[13] Bellec, G., Salaj, D., Subramoney, A., Legenstein, R., and Maass, W. "Long Short-Term Memory and Learning-to-learn in Networks of Spiking Neurons". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018.

[14] Neftci, E. O., Mostafa, H., and Zenke, F. "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-based Optimization to Spiking Neural Networks". In: *IEEE Signal Processing Magazine* 36.6 (Nov. 2019), pp. 51–63. ISSN: 15580792. DOI: 10.1109/MSP.2019.2931595.

[15] Eshraghian, J. K., Ward, M., Neftci, E., Wang, X., Lenz, G., Dwivedi, G., Bennamoun, M., Jeong, D. S., and Lu, W. D. "Training Spiking Neural Networks Using Lessons From Deep Learning". In: *Proceedings of the IEEE* 111.9 (Sept. 2023), pp. 1016–1054. ISSN: 1558-2256. DOI: 10.1109/JPROC.2023.3308088.

[16] Jang, H., Simeone, O., Gardner, B., and Gruning, A. "An Introduction to Probabilistic Spiking Neural Networks: Probabilistic Models, Learning Rules, and Applications". In: *IEEE Signal Processing Magazine* 36.6 (Nov. 2019), pp. 64–77. ISSN: 1558-0792. DOI: 10.1109/MSP.2019.2935234.

[17] Arya, G., Schauer, M., Schäfer, F., and Rackauckas, C. "Automatic Differentiation of Programs with Discrete Randomness". In: *Advances in Neural Information Processing Systems* 35 (Dec. 2022), pp. 10435–10447.

[18] Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. *Neuronal Dynamics -From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014. ISBN: 978-1-107-63519-7.

[19] Pfister, J.-P., Toyoizumi, T., Barber, D., and Gerstner, W. "Optimal Spike-Timing-Dependent Plasticity for Precise Action Potential Firing in Supervised Learning". In: *Neural Computation* 18.6 (June 2006), pp. 1318–1348. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.6.1318.

[20] Brea, J., Senn, W., and Pfister, J.-P. "Matching Recall and Storage in Sequence Learning with Spiking Neural Networks". In: *Journal of Neuroscience* 33.23 (June 2013), pp. 9565–9575. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.4098-12.2013.

[21] Gardner, B., Sporea, I., and Grüning, A. "Learning Spatiotemporally Encoded Pattern Transformations in Structured Spiking Neural Networks." In: *Neural computation* 27.12 (Dec. 2015), pp. 2548–86. ISSN: 1530-888X. DOI: 10.1162/NECO_a_00790.

[22] Zenke, F. and Vogels, T. P. "The Remarkable Robustness of Surrogate Gradient Learning for Instilling Complex Function in Spiking Neural Networks". In: *Neural Computation* 33.4 (Mar. 2021), pp. 899–925. ISSN: 0899-7667. DOI: `10.1162/neco_a_01367`.

[23] Herranz-Celotti, L. and Rouat, J. *Stabilizing Spiking Neuron Training*. 2024. arXiv: `2202.00282 [cs.NE]`.

[24] Hinton, G. E. *Lectures from the 2012 Coursera Course: Neural Networks for Machine Learning*. https://www.cs.toronto.edu/~hinton/coursera_lectures.html. 2012.

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

[26] Yin, P., Lyu, J., Zhang, S., Osher, S., Qi, Y., and Xin, J. "Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets". In: *International Conference on Learning Representations*. Sept. 2018.

[27] Marschall, O., Cho, K., and Savin, C. "A Unified Framework of Online Learning Algorithms for Training Recurrent Neural Networks". In: *arXiv:1907.02649 [cs, q-bio, stat]* (July 2019). arXiv: 1907.02649.

[28] Werfel, J., Xie, X., and Seung, H. S. "Learning curves for stochastic gradient descent in linear feedforward networks". In: *Advances in neural information processing systems*. 2004, pp. 1197–1204.

[29] Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., and Hinton, G. "Backpropagation and the brain". en. In: *Nature Reviews Neuroscience* (Apr. 2020). Publisher: Nature Publishing Group, pp. 1–12. ISSN: 1471-0048. DOI: `10.1038/s41583-020-0277-3`.

[30] Glasserman, P. "Estimating Sensitivities". In: *Monte Carlo Methods in Financial Engineering*. Ed. by Glasserman, P. New York, NY: Springer, 2003, pp. 377–420. ISBN: 978-0-387-21617-1. DOI: `10.1007/978-0-387-21617-1_7`.

[31] Fu, M. C. "Chapter 19 Gradient Estimation". In: *Handbooks in Operations Research and Management Science*. Ed. by Henderson, S. G. and Nelson, B. L. Vol. 13. Simulation. Elsevier, Jan. 2006, pp. 575–616. DOI: `10.1016/S0927-0507(06)13019-4`.

[32] Rossbroich, J., Gygax, J., and Zenke, F. "Fluctuation-Driven Initialization for Spiking Neural Network Training". In: *Neuromorphic Computing and Engineering* 2.4 (Dec. 2022), p. 044016. ISSN: 2634-4386. DOI: `10.1088/2634-4386/ac97bb`.

[33] Rossum, M. C. W. van. "A Novel Spike Distance". In: *Neural Computation* 13.4 (Apr. 2001), pp. 751–763. ISSN: 0899-7667. DOI: `10.1162/089976601300014321`.

[34] Cramer, B., Stradmann, Y., Schemmel, J., and Zenke, F. "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (July 2022), pp. 2744–2757. ISSN: 2162-2388. DOI: `10.1109/TNNLS.2020.3044364`.

[35] Neal, R. M. "Connectionist Learning of Belief Networks". In: *Artificial Intelligence* 56.1 (July 1992), pp. 71–113. ISSN: 0004-3702. DOI: `10.1016/0004-3702(92)90065-6`.

[36] Maddison, C. J., Mnih, A., and Teh, Y. W. *The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables*. Mar. 2017. DOI: `10.48550/arXiv.1611.00712`. arXiv: `1611.00712 [cs, stat]`.

[37] Jang, E., Gu, S., and Poole, B. "Categorical Reparameterization with Gumbel-Softmax". In: *International Conference on Learning Representations*. Nov. 2016.

[38] Williams, R. J. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8.3-4 (May 1992), pp. 229–256. ISSN: 0885-6125, 1573-0565. DOI: `10.1007/BF00992696`.

[39] Schulman, J., Heess, N., Weber, T., and Abbeel, P. "Gradient Estimation Using Stochastic Computation Graphs". In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R. 2015, pp. 3528–3536.

[40] Weber, T., Heess, N., Buesing, L., and Silver, D. "Credit Assignment Techniques in Stochastic Computation Graphs". In: *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*. PMLR, Apr. 2019, pp. 2650–2660.

[41] Liu, L., Dong, C., Liu, X., Yu, B., and Gao, J. *Bridging Discrete and Backpropagation: Straight-Through and Beyond*. Oct. 2023. arXiv: `2304.08612 [cs]`.

[42] Tokui, S. and Sato, I. "Evaluating the Variance of Likelihood-Ratio Gradient Estimators". In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, July 2017, pp. 3414–3423.

[43] Ma, G., Yan, R., and Tang, H. *Exploiting Noise as a Resource for Computation and Learning in Spiking Neural Networks*. May 2023. DOI: `10.48550/arXiv.2305.16044`. arXiv: `2305.16044 [cs]`.

[44] Gulcehre, C., Moczulski, M., Visin, F., and Bengio, Y. "Mollifying Networks". In: *International Conference on Learning Representations*. 2017.

[45] Kaiser, J., Mostafa, H., and Neftci, E. "Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE)". In: *Frontiers in Neuroscience* 14 (2020). ISSN: 1662-453X. DOI: `10.3389/fnins.2020.00424`.

[46] Bohnstingl, T., Woźniak, S., Pantazi, A., and Eleftheriou, E. "Online Spatio-Temporal Learning in Deep Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.11 (2023), pp. 8894–8908. DOI: 10.1109/TNNLS.2022.3153985.

[47] Zenke, F. and Neftci, E. O. "Brain-Inspired Learning on Neuromorphic Substrates". In: *Proceedings of the IEEE* 109.5 (May 2021), pp. 935–950. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.3045625.

[48] Micou, C. and O'Leary, T. "Representational Drift as a Window into Neural and Behavioural Plasticity". In: *Current Opinion in Neurobiology* 81 (Aug. 2023), p. 102746. ISSN: 0959-4388. DOI: 10.1016/j.conb.2023.102746.

[49] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[50] Bradbury, J., Frosting, R., Hawkings, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. *JAX: Composable Transformation of Python+NumPy Programs*. 2018.

[51] Funk, S. *RMSprop Loses to SMORMS3 - Beware the Epsilon!* https://sifter.org/simon/journal/20150420.html. Apr. 2015.

## A Supplementary figures

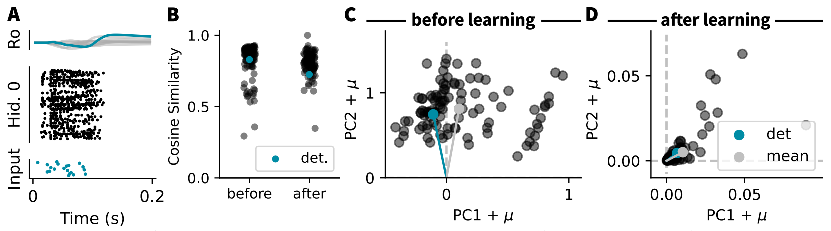


**Figure S1. The bias induced in SNN training due to deterministic SGs is small.** **(A)** Spike raster plot of a network trained on the random manifolds (Randman) dataset [22]; top: membrane potential of the readout units, middle: spike raster plot of the 128 hidden units, bottom: input data. Time is on the x-axis, the y-axis is the neuron index and each dot is a spike. **(B)** Cosine similarity of gradients obtained in 100 single trials in a stochastic SNN performing the Randman task in (A) w.r.t the mean gradient over 100 trials before and after learning. Teal shows the cosine similarity between the deterministic and the mean stochastic gradient. **(C)** First two principal components of the gradients obtained in different trials with the stochastic network before training. Teal is the deterministic, and silver is the mean gradient. **(D)** Same two principal components as in (C), but with the mean added back. One can see that there is a slight bias in the direction chosen by the deterministic update with respect to the average direction of an update in a stochastic network. **(D) & (E)** Same as (C) & (D) but after training on the Randman task for 200 epochs.

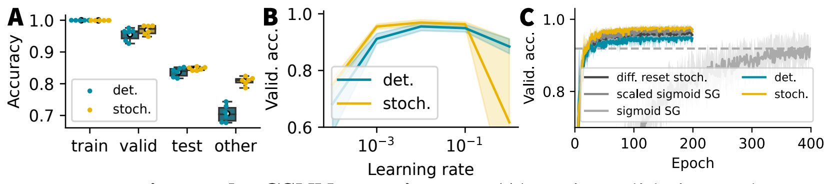


**Figure S2. Supplementary metrics on the CSNN experiments.** **(A)** Train, validation and test accuracy for the three-layer CSNN trained on SHD for the stochastic and deterministic case. In the case of the stochastic SNN, escape noise is applied during, training, validation and testing. The box labeled "other" shows the performance, if the stochastic network is evaluated on the test set without any escape noise being present and vice versa for the deterministic. **(B)** Validation accuracy for the stochastic and deterministic networks after training for different learning rates. **(C)** Validation accuracy for different setups, that applied specific changes to the setup from the main text (compare Fig. 7): *diff. reset stoch.*: stochastic network trained with BP also through the reset term. *Scaled sigmoid SG*: stochastic network trained with sigmoid instead of fast sigmoid SG, but scaled by $\frac{1}{\beta_{SG}}$, as done in Zenke et al. [12] for the fast sigmoid. *Sigmoid SG*: same as (B), but without scaling and $\beta_{SG} = 1$.

## B Example: stochastic derivative of a Perceptron as in stochAD [17]

Let us consider a stochastic binary Perceptron as in Eq. (4). Let us first consider only the derivative of the Bernoulli w.r.t the probability of firing $p$, namely $\frac{dy}{dp} = \frac{d}{dp}Ber(p)$, as we can later apply the chain rule after smoothing. For the right stochastic derivative (which takes into account jumps from 0 to 1), we assume $\epsilon > 0$, thus the differential $dy(\epsilon) = y(p+\epsilon) - y(p)$ will be

$$dy(\epsilon)(\omega) = \begin{cases} 1 & \text{if } 1-p-\epsilon \leq \omega < 1-p \\ 0 & \text{otherwise} \end{cases}$$

where $\omega$ is the actual sample drawn from the Bernoulli and we use this fixed randomness to compute the differential. Given a sample $y(p)(\omega) = 1$, there can be no jump, but if we start with $y(p)(\omega) = 0$, there is a probability of $\frac{1}{1-p}$ that the output of $y(p+\epsilon)(\omega)$ will jump from zero to one. A stochastic derivative is written as triple, where the first number is the "almost sure" part $\delta$ of the derivative, the second is the weight (probability) $w$ of a finite jump in the derivative, and finally, we have the alternate value $Y$, which is the new value of the output if the jump occurred. Hence, in our case, the correct stochastic derivative would be

$$(\delta_R, w_R, Y_R) = \begin{cases} (0, \frac{1}{1-p}, 1) & \text{if } y(p)(\omega) = 0 \\ (0, 0, 0) & \text{if } y(p)(\omega) = 1 \end{cases}.$$

For the left stochastic derivative, we would only consider jumps from one to zero. So the left stochastic derivative is

$$(\delta_L, w_L, Y_L) = \begin{cases} (0,0,0) & \text{if } y(p)(\omega) = 0 \\ (0, -\frac{1}{p}, 0) & \text{if } y(p)(\omega) = 1 \end{cases} .$$

To use stochastic derivatives with BP, one needs to smooth them first. This, however, is no longer an unbiased solution. The smoothed stochastic derivative is defined as $\widetilde{\delta} = \mathbb{E}\left[\delta + w(Y - y(p))|y(p)\right]$ (see also Eq. (1)). Hence in our case, we have $\widetilde{\delta}_R = \frac{1}{1-p} \cdot \mathbf{1}_{y(p)=0}$ and $\widetilde{\delta}_L = \frac{1}{p} \cdot \mathbf{1}_{y(p)=1}$. We know from (4), that $p = \sigma_\beta(u)$ and we can compute the continuous part of the derivative, e.g. $\frac{d}{du}p = \beta\sigma_\beta(u) \cdot (1 - \sigma_\beta(u))$ by applying the chain rule. Put together, we end up with

$$\begin{aligned} \widetilde{\delta}_R^{tot} &= \frac{1}{1-p} \cdot \mathbf{1}_{y(p)=0} \cdot \frac{dp}{du} \\ &= \frac{1}{1-\sigma_\beta(u)} \cdot \mathbf{1}_{y(p)=0} \cdot \beta\sigma_\beta(u)(1-\sigma_\beta(u)) \\ &= \beta\sigma_\beta(u) \cdot \mathbf{1}_{y(p)=0} \end{aligned}$$

for the smoothed right stochastic derivative, and

$$\begin{aligned} \widetilde{\delta}_L^{tot} &= \frac{1}{p} \cdot \mathbf{1}_{y(p)=1} \cdot \frac{dp}{du} \\ &= \frac{1}{\sigma_\beta(u)} \cdot \mathbf{1}_{y(p)=1} \cdot \beta\sigma_\beta(u)(1-\sigma_\beta(u)) \\ &= \beta(1-\sigma_\beta(u)) \cdot \mathbf{1}_{y(p)=1} \end{aligned}$$

for the smoothed left stochastic derivative. Now since every affine combination of the left and the right smoothed

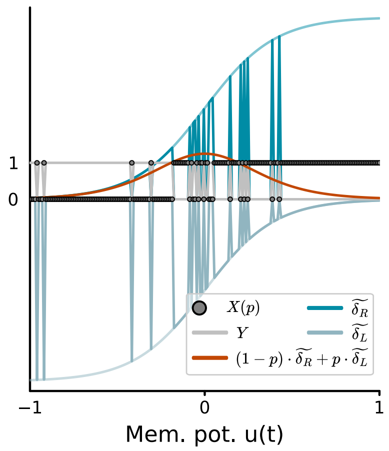


**Figure S3. Smoothed stochastic derivatives:** Any affine combination of a smoothed left (light blue) and a smoothed right (teal) stochastic derivative is a valid stochastic derivative (red) given a specific realization $X(p)$.

stochastic derivative is a valid smoothed stochastic derivative, we can choose our smoothed stochastic derivative to be $(1-p) \cdot \widetilde{\delta}_R^{tot} + p \cdot \widetilde{\delta}_L^{tot}$ which evaluates to

$$\begin{aligned} \widetilde{\delta}^{tot} &= (1-\sigma_\beta(u)) \cdot \beta\sigma_\beta(u) \cdot \mathbf{1}_{y(p)=0} + \sigma_\beta(u) \cdot \beta(1-\sigma_\beta(u)) \cdot \mathbf{1}_{y(p)=1} \\ &= \beta \cdot \sigma_\beta(u) \cdot (1-\sigma_\beta(u)) \end{aligned}$$

Therefore, when using a specific affine combination of the left and the right smoothed stochastic derivatives, we can write $\frac{dy}{du} = \beta \cdot \sigma_\beta(u) \cdot (1-\sigma_\beta(u)) = \beta \cdot \sigma'_\beta(u)$ thereby exactly recovering SDs in a stochastic network. Furthermore, we find the same expression for the SD independent of the outcome of the Bernoulli random variable. The SD and the smoothed left and right stochastic for one sample of a stochastic Perceptron are shown in Fig. S3.