

# Easy and efficient spike-based Machine Learning with mlGeNN

James C Knight  
J.C.Knight@sussex.ac.uk

University of Sussex  
School of Engineering and Informatics  
Brighton, United Kingdom

Thomas Nowotny  
T.Nowotny@sussex.ac.uk

University of Sussex  
School of Engineering and Informatics  
Brighton, United Kingdom

## ABSTRACT

Intuitive and easy to use application programming interfaces such as Keras have contributed majorly to the rapid acceleration of machine learning with artificial neural networks. Building on our recent works on translating ANNs to SNNs and training classifiers with eProp, we here present the mlGeNN interface as an easy way to define, train and test spiking neural networks on our efficient GPU based GeNN framework. We illustrate the use of mlGeNN by investigating the performance of a number of shallow spiking neural networks trained with the e-prop learning rule to recognise hand gestures from the DVS gesture dataset. We find that not only is mlGeNN vastly more convenient to use than the lower level PyGeNN interface, the new freedom to effortlessly and rapidly prototype different network architectures also gave us an unprecedented overview over how e-prop compares to other recently published results on the DVS gesture dataset across architectural details.

## CCS CONCEPTS

• **Computing methodologies** → *Bio-inspired approaches; Supervised learning; Vector / streaming algorithms.*

## KEYWORDS

spiking neural networks, efficient simulation, GPU

### ACM Reference Format:

James C Knight and Thomas Nowotny. 2022. Easy and efficient spike-based Machine Learning with mlGeNN. In *Neuro-Inspired Computational Elements Conference (NICE 2023)*, April 11-April 14, 2023, San Antonio, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 INTRODUCTION

The development of efficient spiking neural network (SNN) simulators has been a key area of computational neuroscience research for several decades [1, 5, 9, 10, 24]. However, the prevalent SNN simulators are not well-suited to the types of models and the workflows required for spike-based machine learning (ML). Consequently, many ML researchers have chosen to build libraries [7, 8, 11, 20, 26] on top of frameworks such as PyTorch [19] and TensorFlow [6] which allow defining SNNs in an environment that is familiar to ML researchers. In these libraries, the activity of a population of neurons is typically represented as a vector of activations and, for

an SNN, this vector is populated with ones for neurons that are spiking and zeros for quiescent neurons. This representation allows using the existing infrastructure of the underlying ML library for SNNs but, as real neurons often spike at comparatively low rates, propagating the activity of inactive neurons through the network leads to many unnecessary computations. Additionally, the connections between populations of neurons can be sparse, in particular in biologically inspired SNNs. In standard ML libraries, sparse connections are typically implemented as a weight matrix containing many zeros which leads to further computational inefficiencies.

To avoid these inefficiencies we have developed mlGeNN – a new library for spike-based ML built on the GPU-optimised sparse data structures and algorithms provided by our GeNN simulator [13, 14, 24]. We previously presented an initial version of mlGeNN [22] which provided workflows for converting ANNs trained using TensorFlow [6] into SNNs that could be simulated using GeNN. However, while we found that performing inference using our converted SNNs was faster than with competing libraries, SNNs are inherently at a disadvantage for static (image) classification tasks as by their nature they turn a single inference step in an ANN library into SNN dynamics across several, often numerous, timesteps. In this paper we introduce a second module of mlGeNN focused on training SNNs from scratch. This allows tackling ML problems that are inherently more suitable for SNNs, for instance classification problems that contain a temporal dimension such as speech or gesture recognition.

## 2 MLGENN

While fully describing the functionality of mlGeNN is beyond the scope of this paper, in the following sections we present an overview of some of our design decisions and the current feature set of mlGeNN. For more information about mlGeNN, we invite readers to visit [github](#) and explore our online documentation (**TODO: cite**).

### 2.1 Describing models

Machine learning frameworks such as PyTorch [19] and TensorFlow [6] are designed to efficiently process directed and acyclical computational graphs. In order to express recurrent connectivity within an otherwise directed acyclical graph, special recurrent layers are required to ‘hide’ the cyclical nature of the recurrent connectivity. However, when beginning to consider more brain-like models with highly recurrent connectivity, the whole model is likely to become one big recurrent layer which is not likely to be an efficient representation.

By contrast, mlGeNN allows Spiking Neural Networks (SNNs) with arbitrary topologies to be defined in terms of homogenous

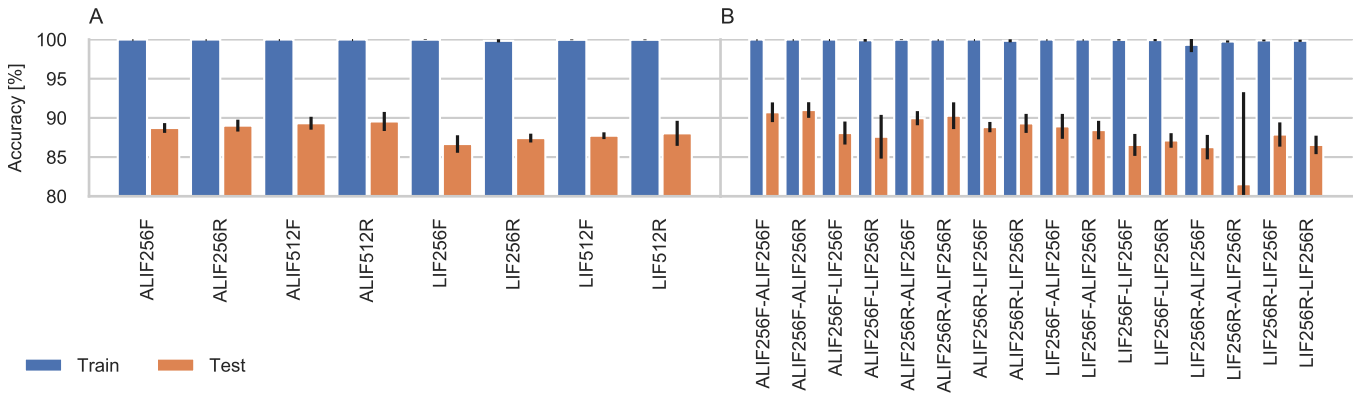
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

NICE 2023, April 11-April 14, 2023, San Antonio, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-XXXX-X/18/06.

<https://doi.org/XXXXXXX.XXXXXXX>



**Figure 1: Accuracy on the testing and training set of the DVS gesture dataset [2] on one (A) and two (B) layer classifiers trained with e-prop [4]. All models were trained for 100 epochs with a batch size of 512. Bars signify the mean and error bars the standard deviation of 5 models trained with different seeds.**

groups of neurons described by **Population** objects connected together with **Connection** objects. While this type of model description is common amongst SNN simulators used for computational neuroscience, the key difference in mlGeNN is that – like in ML frameworks – the model description is agnostic to how it eventually is going to be trained. **Connection** objects don’t need to provide a model of how it’s weights will be trained and additional learning-rule specific structures such as feedback connections do not need to be added by hand.

All populations and connections are owned by a **Network** object which acts as a context manager. A network with two populations of neurons could be simply created like:

```
network = Network()
with network:
    a = Population("poisson_input", 100)
    b = Population("integrate_fire", 100,
                  readout="spike_count")
```

```
Connection(a, b, Dense(1.0))
```

For simplicity, in this example, built-in neuron models with default parameters are specified using strings. However, if some of the default model parameters need to be altered, a model is required that does not have default parameters or a model is needed that is not built into mlGeNN, bespoke neuron parametrisations and models can be specified using a **Neuron** class instance. For example, if the Poisson population in the example needed to emit positive and negative spikes for positive and negative input values and the integrate-and-fire neuron should have a higher firing threshold, a **PoissonInput** and **IntegrateFire** object can be instantiated by the user like:

```
network = Network()
with network:
    a = Population(
        PoissonInput(signed_spikes=True), 100)
    b = Population(
        IntegrateFire(v_thresh=2.0), 100,
        readout="spike_count")
```

```
Connection(a, b, Dense(1.0))
```

By default, **Connection** objects use a ‘delta’ synapse model where the accumulated weight of incoming spikes is directly injected into neurons. However, if a somewhat more realistic model is required, e.g. where inputs are *shaped* to mimic the dynamics of real ion channels, this can be swapped out:

**(TODO: CONNECTION EXAMPLE MISSING?)**

While the flexibility to create networks with any topology is very useful, mlGeNN also provides a **SequentialNetwork** wrapper class – inspired by **Sequential** models in Keras – for specifying common feedforward models more tersely:

```
network = SequentialNetwork()
with network:
    a = InputLayer("poisson_input", 100)
    b = Layer(Dense(1.0), "integrate_fire", 100,
              readout="spike_count")
```

Finally, in the same way that Keras can easily be extended by subclassing built-in classes and implementing new functionality using TensorFlow constructs, mlGeNN can be extended by subclassing built-in classes and providing PyGeNN model descriptions. A full overview over the model description syntax is beyond the scope of this work but is described in more detail in our documentation **(TODO: CITE)** and previous work [13]. Nonetheless, the following example illustrates how a minimal Integrate-and-Fire neuron model could be defined for use in mlGeNN:

```
genn_model = {
    "var_name_types": [("V", "scalar")],
    "sim_code": "$ (V) += $(Isyn);",
    "threshold_condition_code": "$ (V) >= 1.0",
    "reset_code": "$ (V) = 0.0;",
    "is_auto_refractory_required": False}

class IntegrateFire(Neuron):
    v = ValueDescriptor("V")

    def __init__(self, v = 0.0, softmax = False,
```

```

        readout=None):
    super(IntegrateFire, self).__init__(
        softmax, readout)
    self.v = v

    def get_model(self, population, dt):
        return NeuronModel.from_val_descriptors(
            genm_model, "v", self, dt)

```

## 2.2 Using models for training and inference

Once a network structure has been defined, mlGeNN provides a range of *compilers* to produce GeNN models which can then be used for training or inference. The simplest compiler is the **InferenceCompiler** which builds a GeNN model with static weights and parallel batching support for efficiently performing inference on an SNN model:

```

compiler = InferenceCompiler(
    evaluate_timesteps=1500, batch_size=512)

```

```

compiled_net = compiler.compile(network)

```

The resulting **compiled\_net** object can then be used to evaluate the network on a dataset:

```

metrics, _ = compiled_net.evaluate(
    {input: inputs}, {output: labels})

```

By default, the evaluation method calculates sparse categorical accuracy against the provided labels but mlGeNN also provides alternative metrics for regression tasks and custom metrics can easily be implemented. As well as supporting the conversion of trained ANN models to SNNs as described in our previous work [22], mlGeNN now provides an e-prop compiler which can be used to train models with the e-prop learning rule [4]. This compiler checks the validity of the model and automatically adds the appropriate learning rules to its connections as well as feedback connections and error signal calculation logic:

```

compiler = EPropCompiler(
    example_timesteps=1500,
    losses="sparse_categorical_crossentropy", \textbf{
    optimiser="adam", batch_size=512)

```

```

compiled_net = compiler.compile(network)

```

Note that here again, loss functions and optimisers with default parameters are specified using strings but these are also fully customisable. After compiling with the **EPropCompiler**, the **compiled\_net** object can then be used to train the network on a dataset:

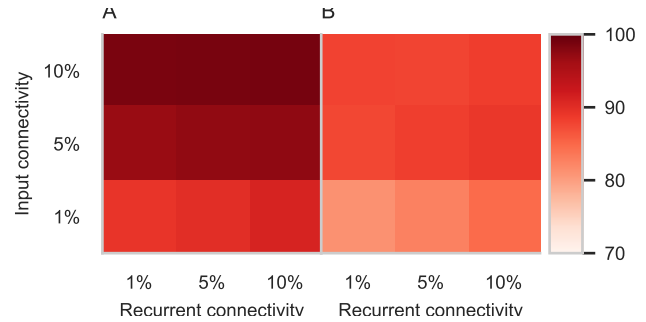
```

metrics, _ = compiled_net.train(
    {input: inputs}, {output: labels},
    num_epochs=args.num_epochs, shuffle=True)

```

## 2.3 Recording and debugging

One of the challenges of working with SNNs, compared to standard ANNs, is that neurons are stateful so the ability to efficiently record and visualise model state is vital for an efficient workflow. Inspired by Keras, mlGeNN has a callback system which – as well as being used internally to implement progress bars, trigger processes such



**Figure 2: Accuracy on the testing and training set of the DVS gesture dataset [2] on sparse models with ALIF512R architectures and varying levels of sparsity trained with e-prop [4]. All models were trained for 100 epochs with a batch size of 512. Connectivity percentages describe fixed probability of there being a synapse between any two neurons (with no self connections in the recurrent connectivity). “Input connectivity” describes connections from input to hidden layer. “Recurrent connectivity” describes recurrent connections in hidden layer.**

as weight updates during model training etc – can be used by the user to configure the recording of state variables and spikes.

**VarRecorder** can be used to record a population state variable over time and **SpikeRecorder** can be used to efficiently record spikes (using the efficient system described in Knight et al. [13]). For example here, we record the spikes emitted by the first layer **a** and the membrane voltages of the second layer **b** of the previous example:

```

callbacks = [SpikeRecorder(a, key="a_spikes"),
              VarRecorder(b, "v", key="b_v")]
metrics, cb_data = compiled_net.evaluate(
    {input: inputs}, {output: labels},
    callbacks=callbacks)

```

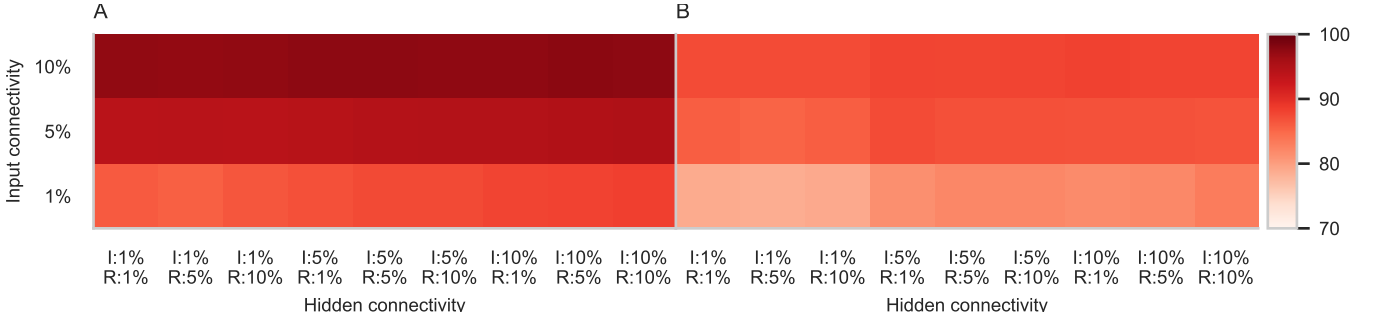
The **key** strings are used to uniquely identify recorded data produced by callbacks and, after the simulation has completed, allow the user to retrieve the recorded variables from the **cb\_data** dictionary which is returned when evaluating (or training) the model. For example, the membrane voltages emitted by all neurons during the presentation of the first input pattern could be accessed using **cb\_data["b\_v"][0]**.

In machine learning workflows, where models may be trained for millions of timesteps, it is also important to be able to *filter* what data is recorded to particular areas/stages of training to save memory and reduce overheads. Both **SpikeRecorder** and **VarRecorder** support filtering by neuron or example, e.g.

**SpikeRecorder(a, neuron=np.s\_[0:2])** could be used to record spikes from every other neuron and **SpikeRecorder(a, example\_filter=1000)** could be used to only record during the 1000th example.

## 3 RESULTS

In order to demonstrate the value of mlGeNN for spike-based ML research, we here present the results of a small exploration



**Figure 3: Accuracy on the training (A) and testing (B) set of the DVS gesture dataset [2] on sparse models with ALIF256F-ALIF256R architectures and varying levels of sparsity trained with e-prop [4]. All models were trained for 100 epochs with a batch size of 512. Connectivity percentages describe fixed probability of there being a synapse between any two neurons (with no self connections in the recurrent connectivity). “Input connectivity” describes connections from input to first hidden layer. “Hidden connectivity” describes *internal* connection from first to second hidden layers (“I”) and recurrent connections in second hidden layer (“R”).**

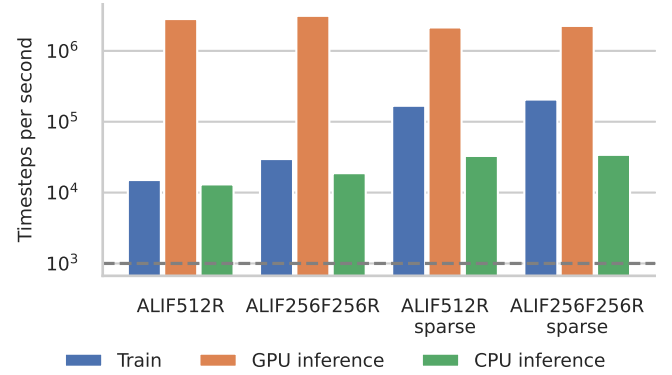
of SNN architectures, trained using e-prop [4] on the DVS gesture dataset [2], which has recently been used to evaluate the EGRU [21] and FPTT [25] learning rules. This dataset consists of 1342 recordings of 11 different hand and arm gestures, collected from 29 subjects under three different lighting condition using a DVS 128 event-based camera [17]. We used the Tonic library [16] to access the dataset and spatially downsample the events to  $32 \times 32$ . While mlGeNN does not depend on Tonic, it includes a `preprocess_tonic_spikes` helper function which converts spike trains from Tonic datasets into mlGeNN’s internal format.

### 3.1 Accuracy

Figure 1 shows the accuracy of a wide selection of one and two layer classifier models trained with e-prop. These include configurations with and without recurrent connections and using both Adaptive and standard Leaky Integrate-and-Fire (ALIF and LIF) neuron models. Of the single layer classifiers, the variant with a single layer of 512 recurrently connected ALIF neurons achieves the highest accuracy of  $(89.55 \pm 1.22)\%$  on the test set. Although this model only has a single hidden layer and around  $1.3 \times 10^6$  parameters, it out-performs a two layer EGRU model [21] with over  $4\times$  as many parameters which achieved 88.02 % accuracy.

Models using ALIF neurons also achieve higher accuracies than those using simpler LIF neurons in all of the two layer configurations. Interestingly, similar to the architecture employed by Yin et al. [25], the best performing model – with an accuracy of  $(91.00 \pm 1.00)\%$  on the test set – has a feedforward first layer followed by a recurrent second layer rather than two recurrently connected layers. While this performance is slightly lower than the  $(91.89 \pm 0.16)\%$  reported by Yin et al. [25], our model has approximately a third the number of parameters ( $0.66 \times 10^6$  vs  $1.8 \times 10^6$ ).

One of the significant advantages of mlGeNN is that it can exploit sparse connectivity to reduce training and inference time so, in the final part of our architectural exploration, we explore the effect of sparsity on the performance of our two best performing architectures (ALIF512R and ALIF256F-ALIF256R). The results of this exploration are shown in figures 2 and 3 and, while increasing sparsity



**Figure 4: Throughput of inference and training best four models on the DVS gesture dataset [2]. Horizontal dashed line indicates performance required to operate in real-time. GPU training was performed with a batch size of 512 and inference with a batch size of 264 (the full size of the test set). GPU training and inference were performed on an NVIDIA RTX A5000 GPU with 24 GB of memory. CPU inference was performed on an Intel Xeon Gold 6134 CPU.**

clearly impacts performance, sparse configurations of both architectures are still highly competitive. In fact, the best performing sparse versions of both architectures out-perform the two layer EGRU model [21], even though they have a small fraction of the number of parameters. The best-performing ALIF512 model – with 5 % connectivity in the input connections and 10 % connectivity in the recurrent connections – achieves an accuracy of  $(89.17 \pm 0.57)\%$  with  $60\times$  fewer parameters and the best-performing ALIF256F-ALIF256R model – with 10 % connectivity in both feed-forward connections and 1 % in the recurrent connectivity – achieves  $(88.17 \pm 0.57)\%$  with  $88\times$  fewer parameters.

### 3.2 Performance

- Training time - compare to FPTT 400 ms per frame with batch size 64 on some sort of 24 GB GPU
- Inference time CPU and GPU - compare to real-time
- Show effect of sparsity on performance - slower sparse inference due to lack of parallelism in small model with relatively small batch size due to small testing set.

## 4 CONCLUSIONS

We hope that the mlGeNN library described in this paper will be as valuable to the community as it was in generating the results presented here. While it is by no means a precise measure, the simplicity and ease of using mlGeNN is illustrated by the fact that the code used for all simulations presented here was less than 300 lines whereas, when we used PyGeNN directly in our previous work [15] one 900 line model was required for training and a separate 500 line model for inference.

The e-prop learning rule is an approximation to gradient descent, both in terms of using a surrogate gradient and in neglecting some terms relating to recurrent connections. The results we have presented in this paper demonstrate that this approximate nature of the e-prop learning rule does not necessarily prevent it from offering competitive performance on relatively complex datasets. However, e-prop has properties that could be seen as disadvantageous for fast and efficient learning and inference. Firstly, e-prop requires time-driven updates of synaptic variables which dominates the time taken to train models. We demonstrate that by using sparse connectivity, this problem can be reduced and, in initial unpublished experiments, we have found that by combining sparse connectivity with the Deep-R [3] rewiring rule, much of the performance lost due to the sparser connectivity can be recovered. Secondly and unsurprisingly for a visual task, both the original experiments on the DVS gesture dataset [2] and more recent work using the Decolle and FPTT learning rules [12, 25], showed that a convolutional SNN can achieve significantly higher performance than shallow recurrent networks. However, because the eligibility traces which drive e-prop weight updates are calculated from the product of pre and postsynaptic activity, they cannot be shared across the corresponding synapses in a convolutional layer. Each synapse needs individual state variables. Neither the FPPT [25] learning rule – where the additional  $\Phi_i$  state variable only depends on past weights so can be shared – nor EventProp [23] – which does not require any additional synaptic state – have these issues so are much more suitable for training convolutional architectures.

Therefore, based on the promising initial results presented by Nowotny et al. [18], we are working to develop an mlGeNN compiler for the EventProp [23] learning rule which will allow convolutional models to be trained and to reduce training times by taking advantage of temporal sparsity through purely event-driven training.

## ACKNOWLEDGMENTS

This work was funded by the EPSRC (grant numbers EP/V052241/1 and EP/S030964/1) and the EU's Horizon 2020 program (grant agreement 945539). Compute time was provided through Gauss Centre for Supercomputing application number 21018 and EPSRC (grant

number EP/T022205/1) and local GPU hardware was provided by an NVIDIA hardware grant award.

## REFERENCES

- [1] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. 2019. Arbor – A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 274–282. <https://doi.org/10.1109/EMPDP.2019.8671560> arXiv:1901.07454
- [2] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey McKinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, Jeff Kusnitz, Michael Debole, Steve Esser, Tobi Delbruck, Myron Flickner, and Dharmendra Modha. 2017. A Low Power, Fully Event-Based Gesture Recognition System. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 7388–7397. <https://doi.org/10.1109/CVPR.2017.781> ISSN: 1063-6919.
- [3] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. 2018. Deep rewiring: Training very sparse deep networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (2018), 1–24. arXiv:1711.05136
- [4] Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. 2020. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications* 11, 1 (dec 2020), 3625. <https://doi.org/10.1038/s41467-020-17236-y>
- [5] Nicholas T Carnevale and Michael L Hines. 2006. *The NEURON book*. Cambridge University Press.
- [6] TensorFlow Developers. 2022. TensorFlow. <https://doi.org/10.5281/zenodo.4724125>
- [7] Jason K Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bannamoun, Doo Seok Jeong, and Wei D Lu. 2021. Training spiking neural networks using lessons from deep learning. *arXiv preprint arXiv:1906.09395* (2021).
- [8] Wei Fang, Yanqi Chen, Jianhao Ding, Ding Chen, Zhaofei Yu, Huihui Zhou, Yonghong Tian, and other contributors. 2020. SpikingJelly. <https://github.com/fangwei123456/spikingjelly>.
- [9] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural Simulation Tool). *Scholarpedia* 2, 4 (2007), 1430.
- [10] Bruno Golosio, Gianmarco Tiddia, Chiara De Luca, Elena Pastorelli, Francesco Simula, and Pier Stanislaw Paolucci. 2021. Fast Simulations of Highly-Connected Spiking Cortical Models Using GPUs. *Frontiers in Computational Neuroscience* 15, February (feb 2021), 1–17. <https://doi.org/10.3389/fncom.2021.627620>
- [11] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. 2018. BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python. *Frontiers in Neuroinformatics* 12, December (dec 2018), 1–18. <https://doi.org/10.3389/fninf.2018.00089> arXiv:1806.01423
- [12] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. 2020. Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE). *Frontiers in Neuroscience* 14, May (may 2020), 1–11. <https://doi.org/10.3389/fnins.2020.00424> arXiv:1811.10766
- [13] James C Knight, Anton Komissarov, and Thomas Nowotny. 2021. PyGeNN: A Python Library for GPU-Enhanced Neural Networks. *Frontiers in Neuroinformatics* 15, April (apr 2021). <https://doi.org/10.3389/fninf.2021.659005>
- [14] James C. Knight and Thomas Nowotny. 2018. GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a Highly-Connected Cortical Model. *Frontiers in Neuroscience* 12, December (2018), 1–19. <https://doi.org/10.3389/fnins.2018.00941>
- [15] James C Knight and Thomas Nowotny. 2022. Efficient GPU training of LSNs using eProp. In *Neuro-Inspired Computational Elements Conference*. ACM, New York, NY, USA, 8–10. <https://doi.org/10.1145/3517343.3517346>
- [16] Gregor Lenz, Kenneth Chaney, Sumit Bam Shrestha, Omar Oubari, Serge Piccaud, and Guido Zarrella. 2021. Tonic: event-based datasets and transformations. <https://doi.org/10.5281/zenodo.5079802> Documentation available under <https://tonic.readthedocs.io>.
- [17] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. 2008. A 128×128 120 dB 15  $\mu$ m<sup>2</sup> Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE Journal of Solid-State Circuits* 43, 2 (2008), 566–576. <https://doi.org/10.1109/JSSC.2007.914337>
- [18] Thomas Nowotny, James P. Turner, and James C. Knight. 2022. Loss shaping enhances exact gradient learning with EventProp in Spiking Neural Networks. (Dec. 2022). <http://arxiv.org/abs/2212.01232> arXiv:2212.01232 [cs].
- [19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

- [20] Christian Pehle and Jens Egholm Pedersen. 2021. *Norse - A deep learning library for spiking neural networks*. <https://doi.org/10.5281/zenodo.4422025> Documentation: <https://norse.ai/docs/>.
- [21] Anand Subramoney, Khaleelulla Khan Nazeer, Mark Schöne, Christian Mayr, and David Kappel. 2022. EGRU: Event-based GRU for activity-sparse inference and learning. *arXiv preprint arXiv:2206.06178* (2022).
- [22] James Paul Turner, James C Knight, Ajay Subramanian, and Thomas Nowotny. 2022. mlGeNN: accelerating SNN inference using GPU-enabled neural networks. *Neuromorphic Computing and Engineering* (2022), 024002. <https://doi.org/10.1088/2634-4386/ac5ac5>
- [23] Timo C Wunderlich and Christian Pehle. 2021. Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports* 11, 1 (dec 2021), 12829. <https://doi.org/10.1038/s41598-021-91786-z>
- [24] Esin Yavuz, James Turner, and Thomas Nowotny. 2016. GeNN: a code generation framework for accelerated brain simulations. *Scientific Reports* 6, 1 (may 2016), 18854. <https://doi.org/10.1038/srep18854>
- [25] Bojian Yin, Federico Corradi, and Sander M Bohte. 2021. Accurate online training of dynamical spiking neural networks through Forward Propagation Through Time. *arXiv preprint arXiv:2112.11231* (2021).
- [26] Zixuan Zhao, Nathan Wycoff, Neil Getty, Rick Stevens, and Fangfang Xia. 2021. Neko: a Library for Exploring Neuromorphic Learning Rules. <https://doi.org/10.48550/arXiv.2105.00324> arXiv:2105.00324 [cs].