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

To demonstrate the value of mlGeNN in this space, we present the results of an exploration of shallow classifier architectures for the classification of the DVS gesture dataset.

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, 5 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]**(TODO: cite neko)** 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.

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 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 because they contain a temporal dimension, for instance.

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

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/XXXXXXX.XXXXXXX

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 descisions and the current feature set of mlGeNN. For more information about mlGeNN, we invite readers to visit github and explore our online documention **(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 acylical 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 topologys to be defined in terms of homogenous 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 isPoissonInput totally agnostic to how it 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 structure 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 you wish to override some of the default model parameters, use a model that does not have default parameters or use a model not built into mlGeNN, you can also specify a neuron model using a `Neuron` class instance. For example, if we wished for the poisson population to emit positive and negative spikes for positive and negative input values and for the integrate-and-fire neuron to have a higher firing threshold we could instantiate `PoissonInput` and `IntegrateFire` objects ourselves 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 you wish to use a somewhat more realistic model where inputs are *shaped* to mimic the dynamics of real ion channels, this can be swapped

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 model 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 description of 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(
            genn_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 be easily 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 enables models to be trained using 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",
    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)
```

We are also working on an EventProp compiler based on the recent work by Nowotny et al. [18].

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 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 `a` layer and the membrane voltages of the second `b` layer 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 recorded variables to be retrieved from the `cb_data` dictionary returned when evaluating (or training) the model. For example the membrane voltages emitted by all neurons during the first example 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, here we present the results of a small exploration of SNN architectures, trained using e-prop [4] on the DVS gesture dataset [2] which has been recently 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]. Here we used the Tonic library [16] to access the dataset and spatially downsample the events to 32×32. 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 (LIF and ALIF) neuron models. Of the single layer classifiers, the variant with a single layer of 512 recurrently connected ALIF neurons performs the best, achieving a mean accuracy of 89.55 % on the test set. Although this model only has a single hidden layer and around 1.3×10^6 parameters, it out-performs a two layer EGRU model [21] with over 4× as many parameters which achieved 88.02 % accuracy.

Models using ALIF neurons also perform better than those using simpler LIF neurons in all of the two layer configurations. However, interestingly, the best performing model (mean accuracy on test set of 91.29 %) had a feedforward first layer followed by a recurrent second layer – similar to the architecture employed by Yin et al. [25] – rather than having two recurrently connected layers.

- FPTT shallow recurrent (91.89 ± 0.16) % 512 neuron encoder for each channel, feeding into recurrent population of 512 neurons
- Take best architecture and explore sparsity

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

4 CONCLUSIONS

Using mlGeNN, the architecture exploration presented here was made much simpler and easier than our previous work using PyGeNN directly [15]. While it is by no means a precise measure, this is illustrated by the fact that the code used for all simulations was less than 300 lines whereas, our previous work involved one 900 line model for training as well as a seperate 500 line model for inference.

Furthermore, the results we have presented in this paper demonstrate that the approximation nature of the e-prop learning rule does not necessarily prevent it offering competitive performane on relatively complex datasets. However, e-prop does have some significant issues. Unsuprisingly for a visual task, both the original work using the IBM True North system [2] and more recent work using the Decolle and FPTT learning rules [12, 25], demonstrated that in order to get significantly better performance on the DVS gesture dataset convolutional networks are required. 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 all the synapses in each connection. Neither the FPPT [25] learning rule – where the additional $\bar{\Phi}_t$ state variable only depends on past weights so can be shared – and EventProp [23] – which does not require any additional synaptic state – have these issues so we are currently working on adding compilers for each of these learning rules to mlGeNN.

Furthermore, the e-prop learning rule requires time-driven updates which dominate the time taken to train our current although, as we demonstrate, by using sparse connectivity, this can be reduced. Therefore, we are working to implement the fully event-driven EventProp [23] learning rule in GeNN which will allow training times to also benefit from temporal sparsity. Finally, the models presented in this paper are all densely connected so are not taking advantage of connection sparsity. We are working in parallel to address this by combining these learning rules with the Deep-R [3]

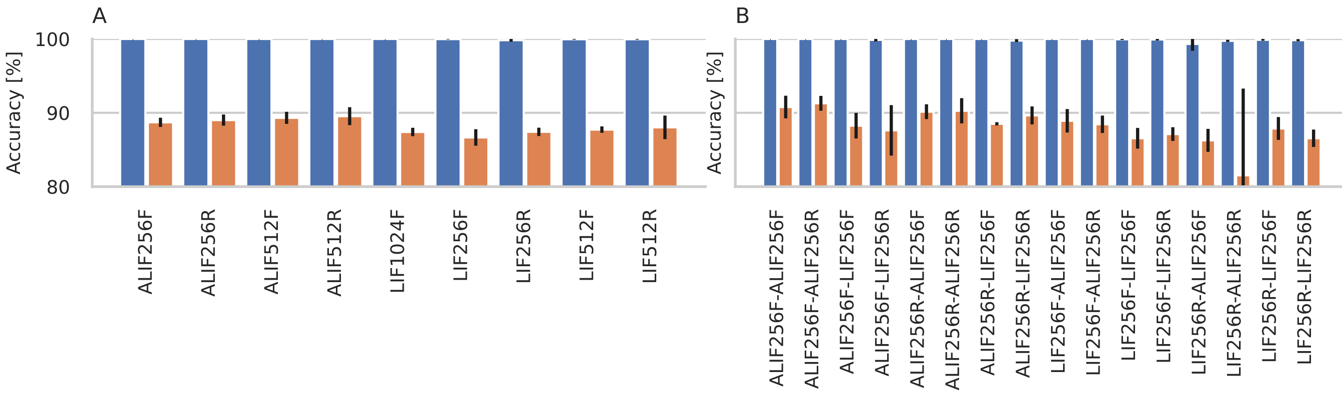

Figure 1: Accuracy on the testing and training set of the DVS gesture dataset [2] on one (A) and two (B) layer eProp classifiers. All models were trained for 100 epochs with a batch size of 512. (TODO: FINISH TRAINING 1024 NEURON MODELS) (TODO: INVESTIGATE GAPS) Bars signify the mean and error bars the standard deviation of 5 models trained with different seeds.

rewiring rule, enabling SNN classifiers to take advantage of GeNN's support for efficient sparse connectivity [14].

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).

REFERENCES

[1] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. 2019. Arbor âĂŤ 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 Bennamoun, 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 Stanislao 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 LSNNs 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 Picaud, 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$\times$128 120 dB 15 μs 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).