mlGeNN: user-friendly and efficient spike-based Machine Learning

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. mlGeNN: user-friendly and efficient spike-based Machine Learning. In *Neuro-Inspired Computational Elements Conference (NICE 2023), April 11-April 14, 2023, San Antonio, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The development of efficient SNN simulators has been a key area of computational neuroscience research for several decades [1, 3, 6, 7, 12]. However, these simulators are not well-suited to the types of model and the workflows required for spike-based machine learning research. As such, many ML researchers have chosen to build libraries [4, 5, 8, 10]**(TODO: cite neko)** on top of frameworks like PyTorch which allow SNNs to be defined in a familiar environment for ML researchers,. When using 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 spiking and zeros for non-spiking neurons. This representation allows one to apply the existing infrastructure of the underlying ML library to SNNs but, as real neurons often spike at comparatively low rates, propagating the activity of inactive neurons through the network leads to unnecessary computation. Additionally, the connections between populations of real neurons are sparse which, using a standard ML library, would typically be implemented as a weight matrix containing many zeros.

Both choices are inefficient so, 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 **(TODO: cite ourselves!)**. We previously presented an initial version of mlGeNN **(TODO: cite)** which provided workflows for converting ANNs trained using TensorFlow **(TODO: cite)** into SNNs for simulation using GeNN. However, while we found that performing inference using our converted SNNs was faster than competing libraries, SNNs are inherantly at a disadvantage for static image classification as by their nature they turn what an ANN library could perform in a single inference step into something that needs simulating for several timesteps.

2 MLGENN

2.1 Describing models

- functional vs sequential abstractions -> connection based and sequential
- Abstraction - synapses/feedback connections vs compilers

Synaptic connectivity in the brain is structured but highly recurrent and this makes the underlying Directed Acyclical Graphs (DAG) model used by typical deep learning libraries a poor fit as

One of the key aims of mlGeNN is to make it simple to define Spiking Neural Networks (SNNs) with arbitrary topologys that are awk. Networks consist of homogenous groups of neurons described by `Population` objects connected together with `Connection` objects. All populations and connections are owned by a `Network` which acts as a context manager so a network with two populations of neurons could be simply created like:

```
from ml_genn import (Connection, Population,
                     Network)

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:

```
from ml_genn import (Connection, Population,
                     Network)
from ml_genn.neurons import (PoissonInput,
                             IntegrateFire)

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 This same general principle is also used for configuring many other aspects of mlGeNN, including loss functions, metrics and readouts.

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:

```
from ml_genn import (InputLayer, Layer,
                     SequentialNetwork)

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 **(TODO: cite pygenn)**. Nonetheless, the following example illustrates how a minimal Integrate-and-Fire neuron model could be defined for use in mlGeNN:

```
from ml_genn.neuron import Neuron
from ml_genn.utils.model import NeuronModel
from ml_genn.utils.value import ValueDescriptor

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 Compiling and using models

Once a network structure has been defined using the functionality described in the previous section, mlGeNN provides a range of *compilers* to produce GeNN models which can 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: spikes},
                                   {output: labels})
```

By default, the evaluation method calculates sparse categorical accuracy against the labels but mlGeNN provides alternative metrics for regression tasks and custom metrics can be easily implemented.

(TODO: talk a little about eProp)

```
compiler = EPropCompiler(
    example_timesteps=1500,
    losses="sparse_categorical_crossentropy",
    optimiser="adam", batch_size=512)

compiled_net = compiler.compile(network)
```

(TODO: talk a little about callbacks etc for recording etc)

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 **(TODO: cite maas)** on the DVS gesture dataset which has been recently used to evaluate the EGRU **(TODO: cite)** and FPTT **(TODO: cite)** learning rules. This dataset was provided by the Tonic library **(TODO: cite)** and, while mlGeNN does not depend on this library, it includes a `preprocess_tonic_spikes` helper function which converts spike trains from Tonic datasets into mlGeNN’s internal format.

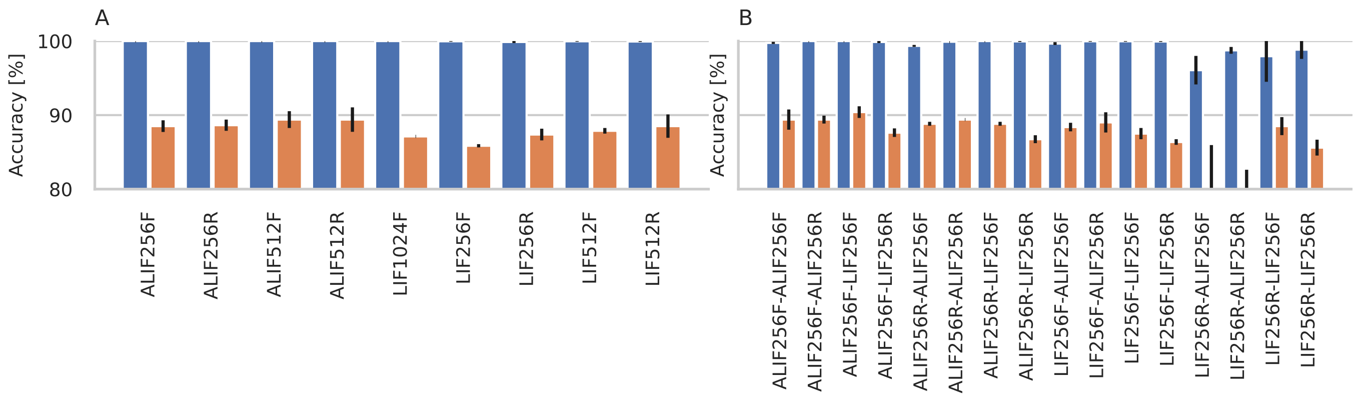

Figure 1: Accuracy on the testing and training set of the DVS gesture dataset (TODO: cite) on one (A) and two (B) layer eProp classifiers. All models were trained for 100 epochs with a batch size of 512. (TODO: finish training 1024 neuron models) (TODO: investigate gaps) Bars signify the mean and error bars the standard deviation of 5 models trained with different seeds.

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 89.39 % 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 (TODO: cite) with over 4× as many parameters which achieved 88.02 % accuracy.

- FPTT shallow recurrent (91.89 ± 0.16) % 512 neuron encoder for each channel, feeding into recurrent population of 512 neurons
- EGRU two EGRU layers of 512 88.02 % (5.5M parameters), two EGRU layers of 1024 90.22 % (15.75M parameters)
- 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

The results we have presented in this paper demonstrate that the approximation used by the e-prop learning rule do not necessarily prevent it enablig competitive performane in relatively complex datasets, e-prop does have some significant issues.

- Eligibility traces are unique to each pair of connected pre and postsynaptic neurons so cannot be efficiently added to convolutional models
- The e-prop learning rule requires time-driven updates which dominate the time taken to *train* these models although, by using sparse connectivity, this

However, Therefore, we are working to implement the fully event-driven EventProp [11] 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 [2] rewiring rule, enabling SNN classifiers to take advantage of GeNN's support for efficient sparse connectivity [9].

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] 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
[3] Nicholas T Carnevale and Michael L Hines. 2006. *The NEURON book.* Cambridge University Press.
[4] 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).
[5] Wei Fang, Yanqi Chen, Jianhao Ding, Ding Chen, Zhaofei Yu, Huihui Zhou, Yonghong Tian, and other contributors. 2020. SpikingJelly. https://github.com/fangwei123456/spikingjelly.
[6] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural Simulation Tool). *Scholarpedia* 2, 4 (2007), 1430.
[7] 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
[8] 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

[9] 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

[10] 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/.

[11] 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

[12] 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