Energy-Based Analog Neural Network Framework

Mohamed Watfa, Alberto Garcia-Ortiz, Gilles Sassatelli

Energy-Based Analog Neural Network Framework

Mohamed Watfa*†, Alberto Garcia-Ortiz†, Gilles Sassatelli*
*LIRMM, University of Montpellier, CNRS, Montpellier, France
†ITEM, University of Bremen, Bremen, Germany
{mwatfa, sassatelli}@lirmm.fr {agarcia}@item.uni-bremen.de

***Abstract*—With the impressive success of deep learning, a recent trend is moving towards neuromorphic mixed-signal approaches to improve energy efficiency. However, the process of building, training, and evaluating mixed-signal neural models is slow and laborious. In this paper, we introduce an open-source framework, called EBANA, that provides a unified, modularized, and extensible infrastructure for building and validating analog neural networks (ANNs). It already includes the most common building blocks and maintains sufficient modularity and extensibility to easily incorporate new concepts. It uses Python as interface language with a syntax similar to Keras, while hiding the complexity of the underlying analog simulations. These features make EBANA suitable for researchers and practitioners to experiment with different design topologies and explore the various tradeoffs that exist in the design space.**

***Index Terms*—deep learning, neural networks, training, analog, SPICE, framework, processing-in-memory**

I. Introduction

The past decade has seen a remarkable series of advances in deep learning (DL) approaches based on artificial neural networks. In the drive towards better accuracy, the complexity and resource utilization of state-of-the-art models have been increasing at such an astounding rate that most of the training and processing is being done in large data centers. However, energy cost, scalability, latency, data privacy, etc., pose serious challenges to existing cloud computing. Alternatively, edge computing has emerged as an attractive possibility [1].

With energy-efficiency being a primary concern, the success of bringing intelligence to the edge is pivoted on innovative circuits and hardware that simultaneously take into account the computation and communication that are required. Consequently, recent hardware architectures for DL show an evolution towards "in/near-memory" computing with the goal of reducing data movement as much as possible. One category of such architectures, the so-called Processing-In-Memory (PIM), consists in removing the necessity of moving data to the processing units by performing the computations inside the memory. This approach is commonly implemented by exploiting the analog characteristics of emerging non-volatile memories (NVM) such as ReRAM crossbars. Furthermore, as ANN inference is inherently resilient to noise, this opens the opportunity to embrace analog computing, which can be much more efficient than digital especially in the low SNR (signal-to-noise ratio) regime [2].

Due to the highly demanding device and circuit requirements for accurate neural network training [3], most mixed-signal implementations are inference-only. While the optimal implementation of the memory devices is an on-going challenge, there is an opportunity to simplify the circuit requirements by considering learning algorithms that are well-matched with the underlying hardware. One such algorithm is the Equilibrium Propagation (EP) algorithm that leverages the fact that the equilibrium point of a circuit corresponds to the minimization of an energy function [4]. By allowing the bidirectional flow of signals, the EP method forgoes the need for a dedicated circuit during the backward phase of training, while also keeping the overhead of the periphery circuit that supports it to a minimum as there is no need for analog-to-digital converters between layers.

Given the growing rate of machine learning workloads, it is of paramount importance to have a framework that is capable of performing a comprehensive comparison across different accelerator designs and identify those that are most suitable for performing a particular ML task. Thanks to ML frameworks such as Google's Tensorflow and Keras, the ease of creating and training models is far less daunting than it was in the past. While training an analog neural network with EP could in theory be possible in Tensorflow, there are three major difficulties:

- First, the I-V characteristic of each circuit element has to be completely defined.
- Second, the network layers have to be designed in such a way that they can influence each other in both directions.
- Finally, implementing procedures that involve iterative updates, like differential equations, within automatic differentiation libraries like Tensorflow, would mean that we need to store all the temporary iterates created during this solution. This requires storing a great deal of information in memory.

To address the above issues, this work introduces an exploratory framework called EBANA (Energy-Based ANAlog neural networks), built in the spirit of Keras[1] with two goals in mind: ease-of-use and flexibility. By hiding the complexity inherent to machine learning and analog electronics behind a simple and intuitive API, the framework facilitates experimentation with different network topologies and the exploration of the various trade-offs that exist in the design space.

This paper is organized as follows. In sec. II, we give a very brief introduction into energy based learning, and explain why it is a natural fit for analog systems. In sec. III, we provide an overview of the internals of our API, and illustrate with

[1]https://keras.io/guides/functional_api/

an example how quickly and easily models can be created. In sec. IV, we validate our framework by training an analog circuit on a nontrivial ML task, evaluate the performance, and show how the framework can be extended. Finally, we discuss the conclusions and further work.

II. Energy Based Learning

The main goal of deep learning or statistical modelling is to find the dependencies between variables. Energy Based Models (EBMs) encode these dependencies in the form of an energy function E that assigns low energies to correct configurations and high energies to incorrect configurations. However, unlike statistical models which must be properly normalized, EBMs have no such requirements [5], and, as such, can be applied to a wider set of problems.

There are two aspects that must be considered for training EBMs. The first is finding the energy function that produces the best output $\boldsymbol{y}$ for an input $\boldsymbol{x}$. This is usually tied to the architecture of the network. The second is shaping the energy function so that the desired value of $\boldsymbol{y}$ has lower energy than all other (undesired) values. This idea is depicted in Fig. 1. In the following sections, we consider one example of such a method, explain how it works, and discuss how it can be used to train analog neural networks.

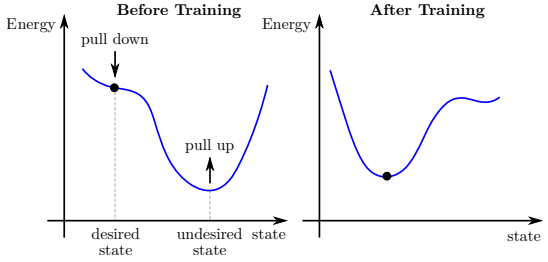

Fig. 1. Training EBMs consists in associating low energies with desired configurations and high energies with undesired configurations

A. Introducing Equilibrium Propagation

The success of deep neural networks can be attributed to the backpropagation (BP) algorithm. In spite of that, BP poses a few difficulties for implementation in hardware. The requirement for different circuits in both phases of training is one of the core issues that the EP learning framework sets out to address [4]. It involves only local computations while leveraging the dynamics of energy-based physical systems. It has been used to train Spiking Neural Networks [6] and in bidirectional learning of Recurrent Neural Networks [7].

The EP algorithm is a contrastive learning method in which the gradient of the loss function is defined as the difference between the equilibrium state energies of two different phases of the network. The two phases are as follows. In the *free* phase, the input is presented to the network and the network is allowed to settle into a *free* equilibrium state (thereby minimizing its energy). Once equilibrium is reached, inference result is available at the output neurons. In the second, *nudging* phase, an error is introduced to the output neurons, and the network settles into a *weakly-clamped* equilibrium state, which has a slightly lower energy than the *free* equilibrium state. The parameters of the network are then updated based on these two equilibrium states.

B. Equilibrium Propagation Algorithm

In the EP algorithm, the activity or state s_i of each neuron is governed by the network energy function

$$F(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, \beta, \boldsymbol{s}) = E(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{s}) + \beta C(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{s}). \quad (1)$$

where $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b})$ are the network parameters, $\boldsymbol{x}$ is the input to the network, $\boldsymbol{y}$ is the target output, and $\boldsymbol{s} = \{s\} = \{x, h, \hat{y}\}$ is the collection of neuron states, comprised of input, hidden and output neurons, respectively.

The total energy function F is composed of two sub-parts: the internal energy E, which is a measure of the interaction of the neurons in the absence of any external force, and the external energy or cost function C, modulated by the influence parameter β. The states are gradually updated over time to minimize the overall energy.

The EP training algorithm is presented in algorithm 1. For proof, see [4]. While it looks similar to backpropagation, propagating the errors towards the input does not require a special computational circuit, making this approach especially attractive for implementation in hardware. However, unlike backpropagation, the state of the network is an implicit function of $\boldsymbol{\theta}$, meaning that the state and gradient of the objective function cannot be calculated efficiently and exactly. The requirement for long phases of numerical optimization of the energy function makes applications based on the EP framework less practical on digital hardware.

Algorithm 1 Equilibrium Propagation

1) Fix the inputs and allow the system to settle at $\boldsymbol{s}^0$ that corresponds to the local minimum of $E(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{s})$ or $F(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, 0, \boldsymbol{s})$. Collect $\frac{\partial F}{\partial \theta}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, 0, \boldsymbol{s}^0)$.
2) With the input still fixed, nudge the output units toward their target values. Allow the system to settle at a new but nearby fixed point $\boldsymbol{s}^\beta$ that corresponds to slightly smaller prediction error. Collect $\frac{\partial F}{\partial \theta}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, \beta, \boldsymbol{s}^\beta)$.
3) Update the parameter θ according to:

$$\Delta\theta \propto -\frac{1}{\beta}\left(\frac{\partial F}{\partial \theta}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, \beta, \boldsymbol{s}^\beta) - \frac{\partial F}{\partial \theta}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}, 0, \boldsymbol{s}^0)\right) \quad (2)$$

III. Exploratory Framework

Our framework, shown in Fig. 2, is largely made up of two parts: one for defining the network model, and the other for training in the analog domain.

The interface to EBANA is Python due to its popularity and its rich ecosystem of libraries for data processing and data analysis. Furthermore, almost all the operations but circuit simulation are performed in Python: this includes netlist generation, gradient currents calculation, and weight update.

We employ the open-source SPICE simulator Ngspice[2] for realistic simulation of the circuit dynamics. The interoperability between Python and Ngspice is provided through the Python library PySpice[3], which provides the means to generate SPICE netlists, send them to Ngspice for simulation, and communicate the result back to Python.

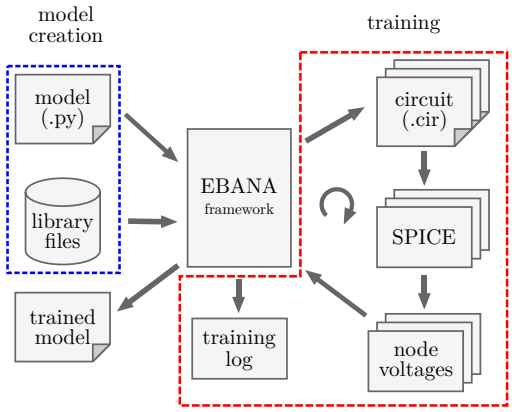

Fig. 2. Block diagram of EBANA framework

A. Network Structure

The process of designing and training a model in our framework starts with defining the model. The general structure of an analog neural network that can be trained with the EP framework is shown in Fig. 3. It consists of an input layer, several hidden layers, and an output layer. It looks similar to a regular neural network that can be trained by the backpropagation algorithm except for two major differences. First, the layers can influence each other bidirectionally; i.e., the information is not processed step-wise from inputs to outputs but in a global way. Second, the output nodes are linked to current sources which serve to inject loss gradient signals during training.

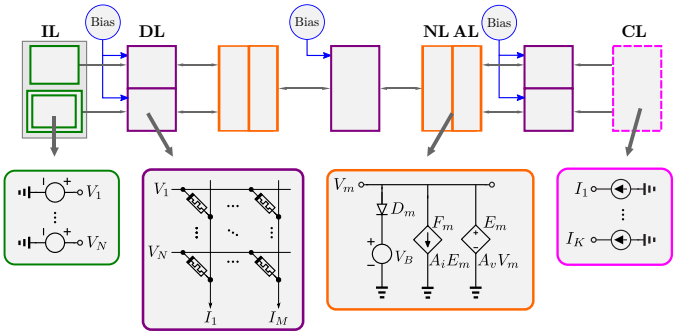

Fig. 3. Analog neural network in the EP framework. IL: Current Layer, DL: Dense Layer, NL: Nonlinearity Layer, AL: Amplification Layer, CL: Current Layer

[2]http://ngspice.sourceforge.net/
[3]https://pypi.org/project/PySpice/

B. Creating a Model

Layers form the core data structure of our framework. They are expressed as Python classes whose constructors create and initialize the connections between the circuit components, and whose `call` methods build the netlist. The process of creating a model is heavily inspired by Keras's functional API due to its flexibility at composing layers in a non-linear fashion. In this manner, the user is able to construct models with multiple inputs/outputs, share layers, combine layers, disable layers, and much more. An example of this is given in Fig. 4, which follows the structure shown in Fig. 3. In the following subsections, we provide details on only those layers that have a unique interpretation in the analog domain.

```
# input layer
xp = InputVoltageLayer(units=4, name='xp')
xn = InputVoltageLayer(units=4, name='xn')
b1 = BiasVoltageLayer(units=1, name='b1', bias_voltage=bias1)
j1 = ConcatenateLayer(name='j1')([xp, xn, b1])

# hidden dense layer
d1 = DenseLayer(units=10, lr=0.0001, name='d1', init_type='glorot')(j1)
a1_1 = DiodeLayer(name='act1_1', direction='down', bias_voltage=a1_bias)(d1)
a1_2 = DiodeLayer(name='act1_2', direction='up', bias_voltage=a2_bias)(a1_1)
g1 = AmplificationLayer(gain=4, name='amp1')(a1_2)

# output dense layer
b2 = BiasVoltageLayer(units=1, name='b2', bias_voltage=bias2)
j2 = ConcatenateLayer(name='j2')([g1, b2])
d2 = DenseLayer(units=6, lr=0.0001, name='d2', init_type='glorot')(j2)
c1 = CurrentLayer(name='c1')(d2)

# define model inputs and outputs
model = Model(inputs=[xp, xn, b1, b2], outputs=[c1])

# train model
optimizer = optimizers.Adam(model)
loss_fn = losses.MSE()
model.fit(train_dataloader, beta=0.01, epochs=100,
    loss_fn=loss_fn, optimizer=optimizer, test_dataloader=test_dataloader
)
```

Fig. 4. Example of a model in the EBANA framework (Iris model)

1) Input layer: This defines the number of inputs to the circuit, which are typically represented by voltage sources. Generally, the input layer is defined according to the dataset. However, the input layer can be defined slightly differently in the analog domain.

- First, since the weights are implemented by resistances, and resistances cannot be negative, a second set of inputs with the opposite polarity of the voltages defined in the dataset is added to the input layer. This accounts for negative weights and effectively doubles the number of inputs. This idea is depicted with two green rectangles in Fig. 4.
- Second, in typical software-based models, the bias, when used, is implicitly set to 1. However, since circuits can work with a wide range of voltages, setting the bias voltage to values other than 1 is necessary. Hence, we provide the option to independently set the bias voltage in each layer. Note that it is also possible to learn the bias voltages.

2) Weight layers: Two kinds of weight layers are defined in the framework. The `Dense` class is the implementation of

the fully-connected layer, which means that the neurons of the layer are connected to every neuron of its preceding layer. Because of this fixed connection between inputs and weights, resistive crossbar arrays are well-suited for implementing the dot product operation in the analog domain.

While fully-connected layers are straight-forward to implement, the implementation of a convolutional layer is difficult in the analog domain because spatial invariance is imposed through weight sharing. A common scheme to map convolutional layers to resistive crossbars is to decompose the convolution operation into vector-matrix-multiplications. This is done by unrolling the filters along the columns and applying the input sliding windows sequentially across the rows. This operation can only be parallelized by replicating the weights across multiple crossbars. Because of this, we implemented a slightly different variant of the convolutional layer, called the `LocallyConnected2D` layer, that avoids the weight sharing problem.

Another issue that is specific to ANNs is the weight initialization problem. Neural networks are very sensitive to the initial weights, and thus selecting an appropriate weight initialization strategy is critical to stabilize the training process. As a result, a lot of research has gone into finding optimal weight initialization strategies [8]. However, since conductances cannot be negative, these methods cannot be applied directly. The default option is to initialize the conductances by drawing samples uniformly at random from the range $[10^{-4}\text{S}, 10^{-1}\text{S}]$. It should be noted that some circuit topologies have difficulty converging using the default initialization.

3) Nonlinearity layer: The nonlinearity layer is implemented as two separate layers: an activation layer, and an amplification layer, which has the effect of normalizing the swing of the signal as it goes through the network.

With the help of diodes, it is possible to create nonlinearities similar to the tanh and sigmoid activation functions. Several options are available for the nonlinearity.

- **Diode orientation** (`direction`): This specifies the orientation of the anode and cathode of the diode with respect to ground.
- **Bias voltage** (`bias`): By choosing a `bias` value other than zero, we can change the voltage at which the diode saturates, and therefore alter the shape of the nonlinearity.
- **SPICE model** (`model`): This is a text description that is passed to the SPICE simulator that defines the behavior of the diode. The default option is to use an ideal diode.

The amplification layer is implemented with ideal behavioral sources. The only option we have available is to change the gain A of the stage. This means that the voltages get boosted in the forward direction by a factor of A, and the currents get amplified in the backward direction by a factor of $\frac{1}{A}$.

4) Current Source layer: This layer simply adds current sources at each output node to inject current into the network during the nudging phase. It is implemented with ideal current sources. During the forward phase, the current sources are set to 0.

C. Training

The training process that is implemented by the fit method is illustrated in Fig. 5.

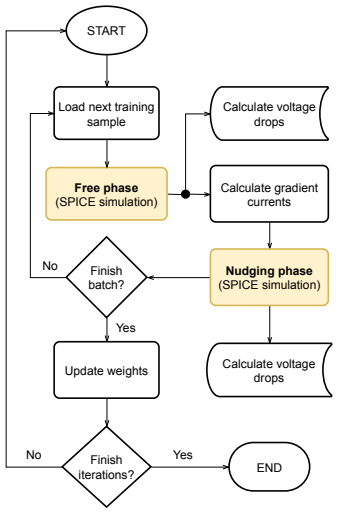

Fig. 5. Implementation of the `fit` method

1) Weight gradient calculation: The current gradients are calculated according to the chosen loss function. For instance, in the case of the mean squared-error (MSE), the loss is given by $C(\hat{Y}_k, Y) = \frac{1}{2}(\hat{Y}_k - Y_k)^2$, where k is the index of output node, $\hat{Y}_k$ is the output of the node, and Y_k is the target value. Other loss functions such as the cross-entropy loss are also available.

The current that is injected into output node k is some multiple β of the derivative of the loss with respect to that node: i.e., $-\beta\frac{\partial C}{\partial \hat{Y}_k}$. The negative sign implies gradient descent.

To address the constraint of non-negative weights, the number of output nodes are doubled. That is, the output node $\hat{Y}_k$ is represented as the difference between two nodes: $\hat{Y}_k = \hat{Y}_k^+ - \hat{Y}_k^-$. The currents, I_k^+ and I_k^-, that are to be injected into Y_k^+ and Y_k^-, respectively, are:

$$\begin{aligned} I_k^+ &= -\beta\frac{\partial C}{\partial Y_k^+} = \beta(Y_k + Y_k^- - Y_k^+) \\ I_k^- &= -\beta\frac{\partial C}{\partial Y_k^-} = \beta(Y_k^+ - Y_k^- - Y_k) \end{aligned} \tag{3}$$

2) Weight update: During the free phase, the current sources at the output nodes are set to 0. The inputs are applied and circuit is allowed to settle. We then collect the node voltage $V^0 = (V_1^0, \ldots, V_N^0)$ and calculate the voltage drop ΔV_{ij}^0 across each conductance.

In the nudging phase, the current given by equation (3) is injected into each output node. After the circuit settles, we collect the node voltages $V^\beta = (V_1^\beta, \ldots, V_N^\beta)$ and calculate the voltage drop ΔV_{ij}^β across each conductance once again. We then update each conductance according to the equation below [9].

$$G_{ij} \leftarrow G_{ij} - \frac{\alpha}{\beta}[(\Delta V_{ij}^\beta)^2 - (\Delta V_{ij}^0)^2] \tag{4}$$

where α is the learning rate.

The weight update rule as defined by (4) is one of the options available in the `optimizer` class, and is defined under the name `SGD` (stochastic gradient descent). Other weight update mechanisms such as `SGDMomentum` (stochastic gradient descent with momentum) and `ADAM` are also available.

D. Parallelism

Training with EP requires performing the free phase and nudging phase, after which the conductances are updated. Both of these phases are done sequentially in SPICE, and are the critical path in the pipeline. While SPICE simulations are always going to be time consuming, the overall simulation time can be reduced by running many simulations in parallel. This is achieved by noting that all the samples in a mini-batch are independent and, therefore, could be simulated independently. As a result, the simulation time could in theory be limited only by the time it takes to simulate a single sample.

IV. Evaluation

In this section, we evaluate our framework focusing on three aspects: correctness, extensibility, and performance.

A. Illustrative Example: Learning the Iris Dataset

As a first step in the evaluation, we built a model that could learn the Iris dataset. This example is a well-known problem of moderate complexity, containing 150 samples, with 4 input variables and 3 output variables.

Two preprocessing steps are needed before the data is ready for training. First, the input variables have to be normalized. Second, since the output variable is categorical and ordering is unimportant, we associate each unique output value with a 3 bit one-hot encoded value. Hence, after the preprocessing step, the dataset has 4 inputs and 3 outputs.

We constructed a model with 1 input layer, 1 hidden layer, and 1 output layer, as shown in Fig. 4. The input layer has 9 nodes; 4 for the regular inputs, 4 for the inverted set, and 1 for the bias. In the preprocessing step, the data was scaled to have a mean of 0 and standard deviation of 4. Therefore, the voltages at the input take real values in the range $[-4\mathrm{V}, 4\mathrm{V}]$.

The hidden layer was implemented with 10 nodes and the output layer with 6 nodes. The weights were initialized from samples drawn randomly from the range $\left[10^{-7}\mathrm{S}, \frac{0.08}{\sqrt{n_{\mathrm{in}}+n_{\mathrm{out}}+1}}\mathrm{S}\right]$, where n_{in} is the size of the inputs and n_{out} is the number of nodes. The learning rate of both layers was set to 0.0001.

The dataset was split into two parts: 105 samples for training, and 45 samples to evaluate the model on new data while training. The optimizer was set to `ADAM` and the model was trained for 400 iterations. It achieved an accuracy of 100% on the test dataset. A plot of the loss and accuracy as a function of the number of the training epochs is shown in Fig. 6. This validates the correctness of our framework.

To gain more insight into the training process, a sample of 8 conductances were chosen at random and their evolution in the course of the training is plotted in Fig. 7. We observe a complex but smooth evolution of the conductances. Also, despite achieving an accuracy of 100%, the circuit has yet to arrive at the most optimal equilibrium state for the given dataset.

For a machine learning task with higher difficulty, we trained a larger model on the Fashion-MNIST dataset. To reduce the simulation time, we encoded the sample images into vectors of length 128 (originally 784) before feeding them into the analog model. We achieved an accuracy of 86% on the test dataset after just 1 epoch with the cross-entropy loss. A similar sized model in Keras achieved an accuracy of 86.7% on the test dataset after 1 epoch and 88.9% after 100 epochs.

B. Extensibility

Even thought it is possible to design fully functional ANNs with the EBANA framework, we provide sufficient system encapsulation and model extensibility to meet the individual requirements of incorporating new models and extending the functionality of the framework, beyond Energy-Based Models. This includes adding new layers, defining new loss functions, changing the training loop, and much more.

To demonstrate the extensibility capabilities of our framework, we consider the example shown in Fig. 9. Here, we show that by subclassing the `SubCircuit` class, and with a just a few lines of code, a new kind of nonlinearity can be defined using MOSFET transistors and voltages sources.

Our library is modularized to easily plug in or swap out components. For instance, to investigate the circuit behavior with this new kind of nonlinearity, all we have to do is replace the `DiodeLayer` in Fig. 2 with `MOSDiode` layer in Fig. 9 and rerun the simulation. Moreover, while the circuit in Fig. 2 is setup for training, it can be easily converted to one that measures the compatibility of an input-output pair by simply swapping the `current` layer with a voltage layer that represents the output.

C. Performance

To evaluate the performance of the simulator, two experiments were conducted. The first experiment was conducted on the Iris model with the goal of measuring the speed-up gained through parallelism. We fixed the number of samples in the mini-batch and ran the simulation for the same number of epochs on a single thread, followed by 2, and then 4. The result is shown in Fig. 8. While the speed-up factor was indeed almost doubled when the thread count was increased from 1 to 2, doubling the thread count further resulted in just 1.5x increase in speed. Due to the resulting circuit being relatively simple, and the small batch size, the overhead of starting new processes for every batch is a nontrivial percentage of the overall simulation time. To investigate this further, we artificially increased the dataset size 8 folds to keeo a thread busy for a longer period. The orange line plot of Fig. 8 confirms this intuition.

For the second experiment, we wanted to measure the simulation performance as a function of problem complexity.

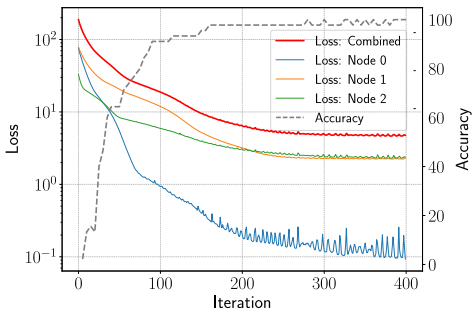

Fig. 6. Plot of loss and accuracy vs. Iteration

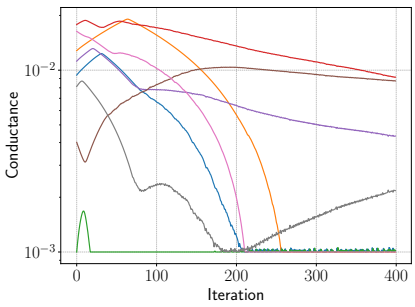

Fig. 7. Evolution of the conductances

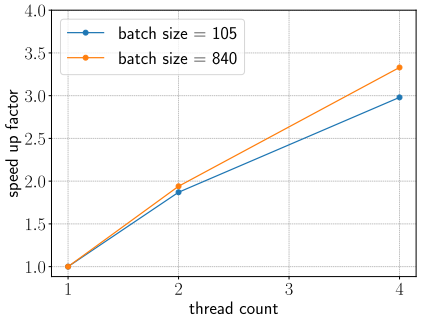

Fig. 8. Speed up factor vs. number of threads

TABLE I
SIMULATION TIME AS A FUNCTION OF CIRCUIT SIZE AND TRAINING DATASET SIZE

datasets	input units	output units	circuit nodes (N)	training dataset size (D)	epochs (E)	time (T)	thread count (P)	K(10^{-4})
xor	5	2	16	4	85	14s	1	25.74
iris	9	6	56	105	155	182s	4	7.99
wine	25	4	111	5000	2	217s	4	7.92

```
class MOSDiode(SubCircuit):
def __init__(self, name, in_subnet, out_subnet, direction, bias, spice_model):
    SubCircuit.__init__(self, name, *in_subnet)
    mod_name = 'NMOS-model'

    if direction == 'down':
        for i in range(len(out_subnet)):
            net = in_subnet[i]
            self.M(i+1, f"N{i+1}", net, net, net, model=mod_name, l=length, w=width)
            self.V(i+1, f"N{i+1}", self.gnd, bias)
    else:
        for i in range(len(out_subnet)):
            net = f"N{i+1}"
            self.M(i+1, in_subnet[i], net, net, net, model=mod_name, l=length, w=width)
            self.V(i+1, f"N{i+1}", self.gnd, bias)
```

Fig. 9. Example of defining a new kind of layer

To this end, we considered 3 datasets; xor, iris, and wine. To obtain an estimate for the complexity of the circuit, we counted the number of nodes only in those models that achieved greater than 95% accuracy on the test dataset. This is due to the fact that the bias-variance trade-off is a property of the model size.

The circuits were simulated and the average simulation time in seconds is recorded in table I. For a measure of the intrinsic speed of the simulator, a column with a calculated property K is added. The property is calculated according to equation (5) and takes into account the simulation time T, the number of allocated threads P, the number of nodes in the generated circuit N, the number of epochs E , and the size of the training dataset D. We can see from table I that K is about the same for the two examples whose simulation time is not dominated by the overhead of starting the SPICE simulator. We expect this to hold true for larger datasets.

$$K = \frac{T \cdot P}{N \cdot E \cdot D} \tag{5}$$

Training for all the experiments was carried out in a laptop with an Intel i7-6700HQ CPU and 32 GB of RAM.

V. CONCLUSION

In this paper, we presented an open-source unified, modularized, and extensible framework called EBANA [10], that can be used to easily build, train, and validate analog neural networks. While it is already possible to build fully functional models, more features and functionalities will be added in future iterations, including more realistic hardware blocks for proper evaluation of the energy consumption of the system. Furthermore, we plan on improving the training speed by optimizing the training loop and adding methods for distributed training over multiple machines.

REFERENCES

[1] Xiaofei Wang et al. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey. *IEEE Communications Surveys & Tutorials*, 22(2):869–904, 2020.

[2] Boris Murmann. Mixed-Signal Computing for Deep Neural Network Inference. 29(1):3–13.

[3] Tayfun Gokmen and Yurii Vlasov. Acceleration of Deep Neural Network Training with Resistive Cross-Point Devices: Design Considerations. *Frontiers in Neuroscience*, 10, July 2016.

[4] Benjamin Scellier and Yoshua Bengio. Equilibrium Propagation: Bridging the Gap between Energy-Based Models and Backpropagation. 11:24.

[5] Yann LeCun, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu Jie Huang. A Tutorial on Energy-Based Learning. page 59.

[6] Erwann Martin et al. EqSpike: Spike-driven equilibrium propagation for neuromorphic implementations. 24(3):102222.

[7] Axel Laborieux et al. Scaling Equilibrium Propagation to Deep ConvNets by Drastically Reducing its Gradient Estimator Bias. *arXiv:2006.03824 [cs]*, June 2020.

[8] Huimin Li et al. A Comparison of Weight Initializers in Deep Learning-Based Side-Channel Analysis. In Jianying Zhou et al., editors, *Applied Cryptography and Network Security Workshops*, volume 12418 of *LNS*, pages 126–143. Springer International Publishing.

[9] J. Kendall, R. Pantone, K. Manickavasagam, Y. Bengio, and B. Scellier. Training End-to-End Analog Neural Networks with Equilibrium Propagation.

[10] EBANA Framework. https://github.com/mawatfa/ebana.