CSE 237D/145

Embedded Systems Design Project

Final Project Report

Power of SNNs

Authors:

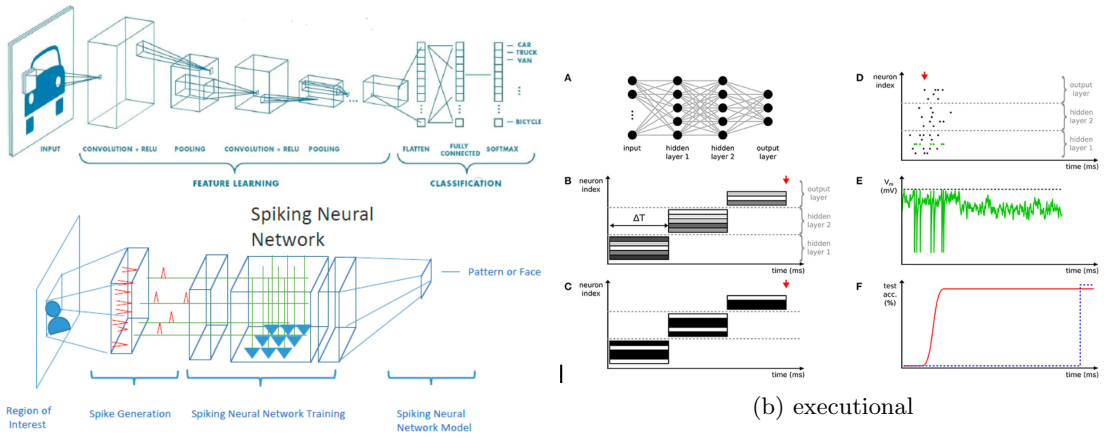
Fatemehsadat Mireshghallah

Akhil Birlangi

Instructor:
Dr.Ryan Kastner

June 15, 2019

UC San Diego



(a) Architectural

(b) executional

Figure 1: This image compares the architecture and the execution of DNNs and SNNs. (a) Shows how in DNNs data is fed forward through layers, but in an SNN spikes are moving, in time. (b) Compares the execution of the two models. In DNNs, the output is generated all at once, whereas in SNNs, it is gradually achieved through time.

1 Abstract

Embedded systems have constrained power and memory resources, therefore full-precision deep neural networks are not easily deployable on these devices. Binarized Neural Networks (BNN) and Spiking Neural Networks (SNN) have low computational costs which makes them amenable for embedded devices that are computationally constrained. In BNNs both activations and weights are only one bit wide which bring about bitwise XOR and population count operations, instead of multiply-accumulate. On the other hand, SNNs are event based. Specifically, their computation is based on the arrival of on-off events through time. In this project, we aim to compare these two different light-weighted computational models of neural networks in terms of consumed power and inference accuracy.

2 Introduction

Recently, neural network models have become prevalent in edge devices for diverse applications such as object detection, computer vision, image processing , etc. In order to make neural networks faster and more power efficient for embedded devices, new neural network computation models such as Binary Neural Networks (BNNs) and Spiking Neural Networks (SNNs) have been introduced. BNN [1] is a model in which the precision of all the parameters are lowered to just one bit. Therefore, the operations can be reduced to logical gate operations such as exclusive or (xor). On the other hand, SNNs are event based. More specifically, their computation is based on arrival of on and off events throughout the computation time. We aim to compare these two different light-weighted network models in terms of latency, consumed power and inference accuracy.

2.1 DNNs Vs. SNNs

In this section, we want to compare SNNs and DNNs. In spiking networks, the input has to become a train of spikes in time. There are many ways to carry this out, we have decided to change the pixel numbers into step signals. The timing nature of SNNs makes them very similar to Recurrent Neural Networks (RNNs). Figure 1a [10] shows an architectural and structural comparison of SNNs and DNNs.

Some benefits of SNNs over DNNs are [2]:

- Low energy usage

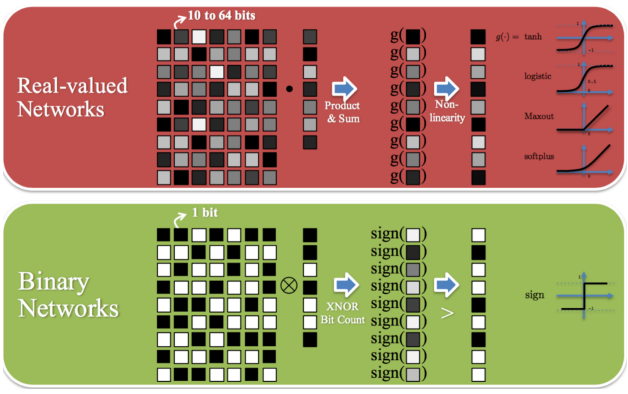


Figure 2: A comparison of Real-valued networks and Binary networks

- Greater parallelizability due to local-only interactions
- Better able to learn non-differentiable functions

Figure 1b [14] shows a comparison of deep spiking neural networks (SNNs) to conventional deep neural networks (DNNs), specifically a fully connected network. A) shows a fully connected 2 hidden layer network. B) shows the data as it flows through the layers, in time. Different shades of grey show different values. C) shows the same network flow, with single bits (binary network). D) shows a fast and asynchronous propagation of spikes in neurons of a similar Deep SNN, the green spikes show potential that has crossed a certain threshold. E) shows the potential of the membrane of the neuron shown in green in D. F) compares the accuracy of two models (SNN in red and DNN in dashed blue). As it is shown, the asynchronous updates in spikes of the SNN cause a faster, but less accurate convergence.

2.2 BNNs Vs. SNNs

Binary Neural Networks (BNNs): These networks operate on single bits, not full precision (10-64 bit) numbers. This helps alleviate a lot of the computational complexity of neural networks.

As Figure 2 [12] shows, real-valued networks take in full-precision numbers, calculate the product and sum, apply activation functions and then feed the data to next layer. BNNs, however, use +1 and -1 bit values, do bitwise XNOR and then population count. The population count output is in full-precision. Then, to feed the data forward to the next layer, sign function is applied, to take the numbers back to -1,+1 domain. SNNs are event-based and they operate considering time. The network input and output are usually a series of spikes (delta function or more complex shapes).

In this project we aim to analyze spiking neural network operation thoroughly, and compare its performance to BNN and CNNs.

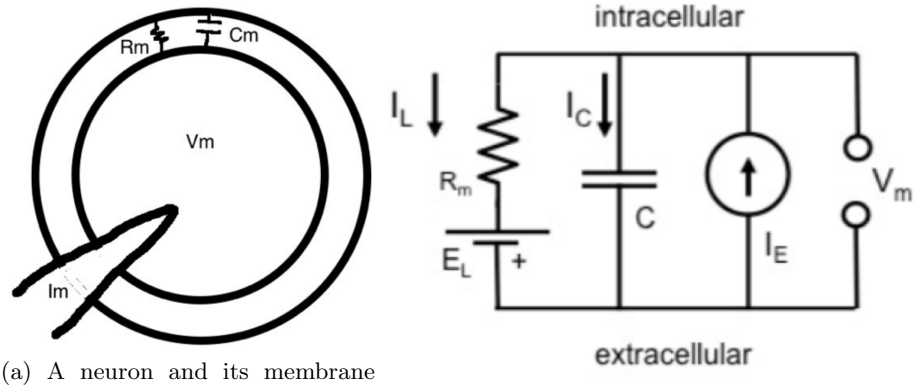
3 Technical Material

In this section we delve deeper into the details of the project. We first describe the neuron model we chose and its formulation. Then we explore two architectures that we tried, and our hardware implementation. Finally, we see our results.

3.1 Neuron Model

Figure 3 shows a classification of different neuron models. We have put our focus on the Leaky Integrate and Fire (LIF), and tried to implement that.

Figure 3: Different neuron models [11]



(a) A neuron and its membrane modeled as an RC circuit [3]

(b) RC circuit that a neuron is modeled with [4]

Figure 4: LIF neuron modeling

As seen in Figure 4a, in the LIF model, the neuron membrane can be depicted as a capacitor with a resistor, which causes leakage. The input spikes cause the accumulation of charge in the capacitor, while at the same time leaking through the resistor. Figure 4b shows the RC circuit.

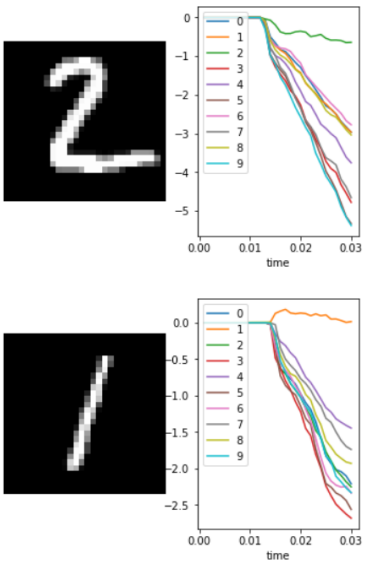
The cell membrane acts as an electrical insulator between the intracellular fluid and extracellular fluid (Figure 4b) that are two conductors. This creates a capacitor, accumulating charge through the membrane and therefore has the ability to integrate inputs over time. The accumulated charge creates a voltage difference across the membrane (Vm).
To summarize [5]:

- The cell has a single voltage, V_m
- The membrane has capacitance C_m
- The membrane leaks charge through R_m
- The charge carriers are driven by V_e

Spikes occur whenV_m exceeds some threshold value V_th. When a spike occurs, the voltage is artificially dropped to a value V_reset.

```python
# add the first convolutional layer
x = nengo_dl.tensor_layer(
    inp, tf.layers.conv2d, shape_in=(28, 28, 1), filters=32,
    kernel_size=3)
# apply the neural nonlinearity
x = nengo_dl.tensor_layer(x, neuron_type)
# add another convolutional layer
x = nengo_dl.tensor_layer(
    x, tf.layers.conv2d, shape_in=(26, 26, 32),
    filters=64, kernel_size=3)
x = nengo_dl.tensor_layer(x, neuron_type)
# add a pooling layer
x = nengo_dl.tensor_layer(
    x, tf.layers.average_pooling2d, shape_in=(24, 24, 64),
    pool_size=2, strides=2)
# another convolutional layer
x = nengo_dl.tensor_layer(
    x, tf.layers.conv2d, shape_in=(12, 12, 64),
    filters=128, kernel_size=3)
x = nengo_dl.tensor_layer(x, neuron_type)
# another pooling layer
x = nengo_dl.tensor_layer(
    x, tf.layers.average_pooling2d, shape_in=(10, 10, 128),
    pool_size=2, strides=2)
# linear readout
x = nengo_dl.tensor_layer(x, tf.layers.dense, units=10)
```

(a) Nengo model code



(b) Nengo output over time

Figure 5: Nengo model.

We model this with the following formulation [5, 11]:

$$Vm(t+1) = Vm(t) + dt * (-(Vm(t) - V_e) + Im * Rm)/tau_m \tag{1}$$

Table 1 shows the constants we used for our neuron.

3.2 Network Architectures

We worked on two network architectures for the MNIST[18] dataset, the first one was from NengoDl [6]. This network architecture is shown in Figure 5a. This network's architecture is different from the usual DNN architecture(LeNet [13])used for MNIST dataset. LeNet, like many other DNNs, has two sections, a feature extraction (3 convolutions) and a classifier (2 fully connected layers). But this network has 4 convolutions and one fully connected layer, and each convolution is followed by a set of LIF neurons, used as non-linearities.

We trained this model using SoftLIF and evaluated it using our LIF implementation which we tuned to act similar to Nengo neurons. As input, for each pixel we make a step, with the magnitude of the pixel intensity and we feed it to the network. It is important to note that unlike DNNs, SNNs do not produce the correct results all at once (even when trained). As shown in Figure 1b, the SNNs generate results over time, and the accuracy keeps getting better and better at each time step. The output of our simulation can be seen in 5b.

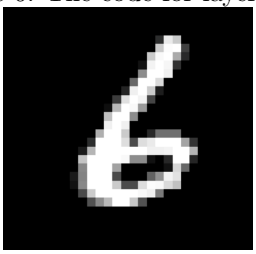
The aforementioned model is huge, and not suitable for the hardware we had in mind. Therefore, we decided to switch to a light-weight model, a two layer feature extractor. This new model, consisted only of two LIF

Table 1: Constants used

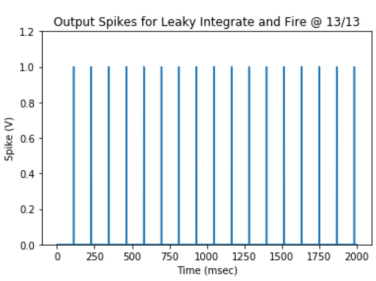
	Value
dt	125 ms
V_e	0 V
R_m	1 kOhm
tau_m	10 ms

```
l2x, l2y = 0,0
for ry in range (0, len_y, stride[0]):
    l2x = 0
    for rx in range(0, len_x, stride[0]):
        stimulus_ret_unit = np.zeros(time)
        print('Generating stimulus for L2 neurons {}/{}'.format(l2y, l2x))
        for ny in range(stride[0]):
            for nx in range(stride[0]):
                x = rx + nx
                y = ry + ny
                stimulus_ret_unit += neurons[y][x].spikes[:time] * mult_factor
        print ('Adding stimulus for l2 neuron {},{}, duration={}\n'.format(l2y, l2x, len(stimulus_ret_unit)))
        l2_neuron_stimulus[l2y,l2x,:] = stimulus_ret_unit
        l2x += 1
    l2y += 1
```

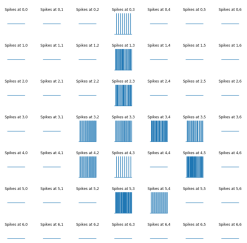
Figure 6: The code for layer 2 of the feature extraction network that acts as convolution.



(a) Input image to feature extractor network



(b) output of the first layer



(c) output of the second layer

Figure 7: Feature extraction model.

neuron layers that acted as convolutions. The first layer is a 2D LIF neuron layer of size 28×28, so each neuron takes in the intensity of a single pixel. The input to the neuron is a vector, and all the elements of this vector are the intensity of that pixel. The length of the vector shows the number of time steps we want to let the network run. The input goes through the first layer and the outputs come out from each neuron. The output for neuron $(13, 13)$ can be seen in 7b. This output is for all time steps, and the neuron has transformed the step signal to spikes.

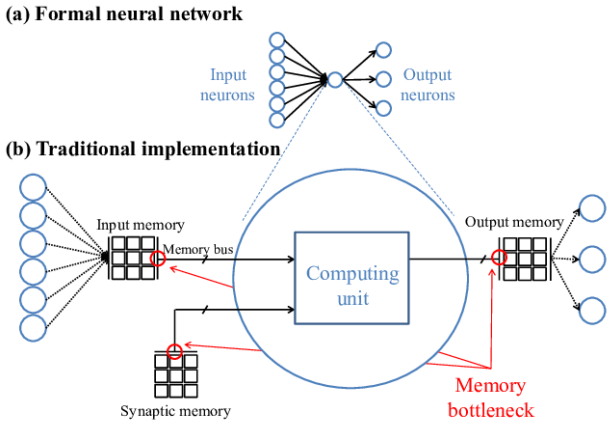
The outputs of the first layer are then added together, in a way that resembles convolution. The code for this can be seen in Figure 6. There is a sliding window, and all the neuron outputs are accumulated for that window. The results of each accumulation becomes input to the second layer, which is a 7×7 LIF neuron layer. The output of this layer looks like Figure 7c. Each of the small plots are the outputs of each neuron, over time. As shown, more intense parts that have a higher density of spikes,are the lighter parts of the number.

3.3 Hardware Implementation

Implementation on hardware was accomplished by utilizing a scheduler that fed synaptic inputs into neural compute engines. Each of these engines would perform a simulation of an LIF neuron for a given time interval. The output spikes of these engines would then be fed into synapse engines by the scheduler. The synapse engines accumulate the spike trains from a layer of neurons and apply the given weights to those trains. Then, the spike trains are summed element-wise in order to create the input spike train for the next layer of neurons. This input train is then fed back into the neuron engine to compute the output of the next neuron.

This engine scheduler architecture was initially implemented as a single hardware neuron and utilized software to handle the scheduling and synapse mechanics. This hardware neuron was linked up to an AXI DMA controller and connected to the Zynq processing shell. From here data could be sent and received in order to process a single neuron at a time. It serves as a good proof of concept but suffers from severe slowdowns due to AXI data movement overhead.

This setup can, albeit slowly, perform the state update calculations for a full neural network. We performed



(a) Neuron engine [16]

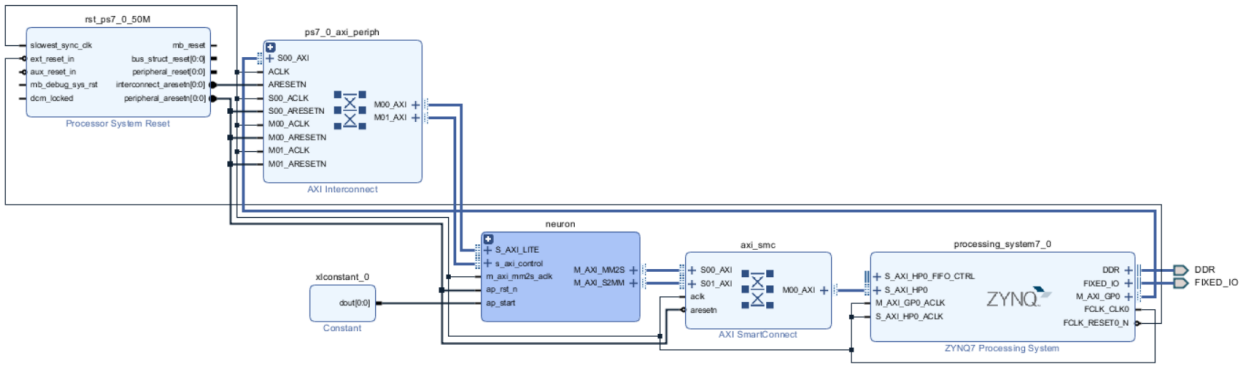


Figure 8: Neural Engine Model.

testing on this neuron to verify its functionality and ensure that it behaved identical to its software counterpart. We continued testing to verify if the hardware neuron still continued to retain its similarity in response to the known-good software LIF neuron utilized in NengoDL. As of now, the neuron is somewhat similar to its software counterpart but significantly different than the known good. Further tweaking of the neural engine in HLS is required to ensure that the neuron's spike response is accurate.

3.4 Results

Table 2 shows a summary of our results. The SNN execution time on software (the one on Intel Corei7) is highest, since it was produced through 30 timesteps, which is 30 runs of the network. Although the execution time is high, the error is the lowest.

On software the CNN is faster than BNN, since Intel's CPU architecture is not compatible with bit operations, so it cannot exploit the speedup opportunity that BNNs bring about. The reason that it is not even the same, but worse, is that time is pent to quantize and the quantize numbers for this network.

The two final results which are for the PYNQ board, the first one is when the light-weightedness of BNNs is exploited through hardware, and the second is when the BNN is run using the software, in a general purpose way.

4 Milestones

Our inital milestones:

1. To have a network implemented for MNIST using both BNN computation model and the SNN Integrate and Fire (LIF) neuron model[17]. A network means the layers and the trained weights.
2. Deliver a table, containing the inference accuracy, time and consumed power for a network on the MNIST dataset using both BNN and SNN on an FPGA, we have the pynq board in mind
3. for a longer term goal, we would like to also apply steps 1 and 2 for other datasets, such as SVHN and CIFAR. We would also like to try changing the BNN model to act more similar to SNNs.

During the quarter, after doing a literature review and getting a better grip on the material, we updated our milestones to the more fine-grained ones, as listed below:

1. Get a single LIF neuron to take in a spike and output a spike in hardware.
2. Create a layer of neurons in hardware to input spikes and output spikes.
3. Feed the spikes forward from one neuron to another neuron in hardware.
4. Create a two layer network of neurons to classify the MNIST dataset to get an output on python and train it to get desirable accuracy.
5. No.4 on hardware: port the network and the weights and biases on hardware .
6. Collecting latency and accuracy data on 100 inference runs of MNIST on the BNN.
7. No.6 for our SNN.

We achieved milestone 1 on software first. The main challenge we had was to tune our neuron such that our results are similar to those of the NengoDL[7] neurons. The difficulty stems in different mathematical formulations in which the neurons behavior could be modeled. We chose this formula 1 [5, 11], which seemed easier to implement on hardware. But another way to model it is [7, 11]:

$$Vm(t+1) = Vm(t) - (Im - Vm(t)) * exp(-d_t/\ tau_m) \tag{2}$$

which was employed in NengoDL. Here, the index m, means membraine, tau is the time constant of the rc model and I is the input to the neuron. Because of the slight difference in the formulations, we had some difficulty tuning the parameters of our neuron to match the outputs. When we were done with the tuning, we ported the software neuron to hardware, by making a neuron engine.

We also achieved milestone number 4 to some extent. The network we got was more than two layers (5 layers)[6]. We trained it for two epochs and got an accuracy of 97.25% on MNIST. Therefore milestone number 5, we decided to use a smaller network that only executed feature extraction[8]. We implemented this network with our neuron and it only has two layers and all the weights for the convolutions (it uses the LIF neurons to mimic convolutions) are set to 1. We also have step 6 on Software [1] and hardware (Pynq

Table 2: Results summary for MNIST [18]

	Inference Time	Error	Platform
SNN	995	0.75%	Intel Corei7
CNN	0.4	1.00%	Intel Corei7
BNN	10.9	2.00%	Intel Corei7
BNN	0.028	1.60%	Xilinx Pynq-Z2- Hardware
BNN	1428.57	1.60%	Xilinx Pynq-Z2- Software

board) and 7 on software, executed on an Intel Corei7 CPU.

To broaden our comparisons, we have also included the results of software latency and accuracy results for a full-precision CNN (LeNet) on an Intel Corei7 CPU.

We achieved our hardware implementation goals partially, and we believe that was due to underestimating the amount of studying we needed to carry out to fully understand how a spiking network operates. We also lost time switching back and fort with different network structures. We first tried to change the Nengo model to strip it of its convolution, we could not get good results from that, so we decided to work with the model with convolutions, but the model size was too big. Then we decided to switch to the 2 layer feature extraction network.

We also had difficulty trying to choose the way we wanted to generate spikes for input to our network. We wanted to transform the pixel intensity to frequency, and generate spikes from that. Going through literature[9] we found the following a logarithmic function. That would scale the frequency in a way that a neuron can handle it. To model a synapse and generate a current based on this input, we employed the equation aand alpha function in table 1 of [15]. implementing that took some time. But when this did not work in the network, we looked deeper in the two other SNN implementations we had found, we realized that they are not transforming the input image to spikes themselves, they are just setting the input as a step, and this goes straight to the neuron, without need for any extra transformations.

5 Conclusion

In this project we carried out a comparison of binarized neural networks and spikining neural networks. We compared their execution time and accuracy for the MNIST dataset using software on an Intel Corei7 CPU. We also compared the execution time of a Binarized Neural Network on a Xilinx PYNQ-Z2 board using a software implementation and a hardware one. We then implemented our neuron engine on the PYNQ board.

As future work, we want to extend our hardware implementation from a single neuron engine to synapse and offer energy comparisons as well.

References

[1] online accessed May 2019 https://github.com/itayhubara/BinaryNet.pytorch.

[2] online accessed May 2019 https://skymind.ai/wiki/spiking-neural-network-snn.

[3] online accessed May 2019 http://tips.vhlab.org/techniques-and-tricks/matlab/integrate-and-fire.

[4] online accessed May 2019 https://ocw.mit.edu/resources/res-9-003-brains-minds-and-machines-summer-cours tutorials/tutorial-2.-matlab-programming/MITRES_9_003SUM15_fire.pdf.

[5] online accessed May 2019 http://tips.vhlab.org/techniques-and-tricks/matlab/integrate-and-fire.

[6] online accessed May 2019 https://www.nengo.ai/nengo-dl/examples/spiking-mnist.html.

[7] online accessed May 2019 https://www.nengo.ai/nengo/_modules/nengo/neurons.html.

[8] online accessed May 2019 https://github.com/markstrefford/Spiking-Neural-Network.

[9] online accessed May 2019 https://www.telescope-optics.net/eye_intensity_response.htm.

[10] Nowak P Tessadori J Massobrio P Chiappalone M. Bruzzone A, Pasquale V. Interfacing in silico and in vitro neuronal networks. online accessed May 2019 https://www.ncbi.nlm.nih.gov/pubmed/26737020.

[11] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, New York, NY, USA, 2014.

[12] Minje Kim. Bitwise neural networks. online accessed May 2019 `https://minjekim.com/demo_bnn.html`.

[13] Yann LeCun. Gradient-based learning applied to document recognition. 1998.

[14] Thomas Pfeil Michael Pfeiffer. Deep learning with spiking neurons: Opportunities and challenges. online accessed May 2019 `https://www.frontiersin.org/articles/10.3389/fnins.2018.00774/full`.

[15] Ammar Mohemmed, Stefan Schliebs, Satoshi Matsuda, and Nikola Kasabov. Method for Training a Spiking Neuron to Associate Input-Output Spike Trains. In Lazaros Iliadis and Chrisina Jayne, editors, *12th Engineering Applications of Neural Networks (EANN 2011) and 7th Artificial Intelligence Applications and Innovations (AIAI)*, volume AICT-363 of *Engineering Applications of Neural Networks*, pages 219–228, Corfu, Greece, September 2011. Springer. Part 12: Spiking ANN.

[16] Damien Querlioz, Olivier Bichler, Adrien Francis Vincent, and Christian Gamrat. Bioinspired programming of memory devices for implementing an inference engine. *Proceedings of the IEEE*, 103:1398–1416, 08 2015.

[17] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A survey of neuromorphic computing and neural networks in hardware. *CoRR*, abs/1705.06963, 2017.

[18] NYU) Yann LeCun (Courant Institute and New York) Corinna Cortes (Google Labs. The mnist dataset of handwritten digits. online accessed May 2019 `http://www.pymvpa.org/datadb/mnist.html`.