Probabilistic spike propagation for FPGA implementation of spiking neural networks

Abinand Nallathambi [1] and Nitin Chandrachoodan [2]

[1]Affiliation not available
[2]Indian Institute of Technology Madras

October 30, 2023

Abstract

The results presented in the paper are based on simulations on benchmark spiking neural networks. The methodology is described in the paper.

PROBABILISTIC SPIKE PROPAGATION FOR FPGA IMPLEMENTATION OF SPIKING NEURAL NETWORKS

A PREPRINT

Abinand Nallathambi
IIT Madras
ee16s032@ee.iitm.ac.in

Nitin Chandrachoodan
IIT Madras
nitin@ee.iitm.ac.in

January 10, 2020

ABSTRACT

Evaluation of spiking neural networks requires fetching a large number of synaptic weights to update postsynaptic neurons. This limits parallelism and becomes a bottleneck for hardware.

We present an approach for spike propagation based on a probabilistic interpretation of weights, thus reducing memory accesses and updates. We study the effects of introducing randomness into the spike processing, and show on benchmark networks that this can be done with minimal impact on the recognition accuracy.

We present an architecture and the trade-offs in accuracy on fully connected and convolutional networks for the MNIST and CIFAR10 datasets on the Xilinx Zynq platform.

1 Introduction

Spiking neural networks are often referred to as the third generation of neural network models [10], and have several characteristics that make them attractive from the viewpoint of hardware design. They follow an event-driven model of computation, where the work done (hence energy consumed) can be made proportional to the number of spike events, and do not require the arrays of multiply-accumulate (MAC) operations that characterize conventional artificial neural networks (including convolutional networks, referred to here as ANNs) This makes ANNs well-suited to parallel implementation on architectures such as GPUs.

In contrast, spiking neural networks may require other types of computations to determine whether a neuron is to fire or not. Hardware architectures for spiking networks (eg. [2, 4, 7, 11]) therefore differ considerably from those for regular ANNs, and focus more on features that enable efficient event-driven computation. This usually requires the network to be trained specifically for the target architecture (due to restrictions on permitted connections or weights), and it is not efficient to take a network trained for one architecture and directly run it on another.

Despite being event driven, spiking networks still require a large number of memory accesses, primarily for two purposes [11]: determining the recipient neurons of a spike, and fetching weights of the corresponding synapses. Recent data (eg. [8]) indicates that fetching data from memory (especially off-chip) is much more expensive in energy than arithmetic computations. For reasonably sized networks, the neuron weight and index information becomes too much to store on-chip – the resulting off-chip memory accesses thus end up dominating the energy consumed for the computation, and can also lead to increased latency.

The regular view of a spiking network is that if a presynaptic neuron spikes, it increases the membrane potential of the postsynaptic neuron by an amount equal to the weight of the synapse. Alternatively, similar to the ideas in [9, 16], we could view weight as a measure of how likely it is that a spike will propagate across a synapse.

In this paper, we present a probabilistic method of spike propagation that can significantly reduce the number of memory accesses required for evaluation of a spiking neural network, thus saving both time and energy. The specific contributions of this paper are as follows:

- We show that by interpreting synaptic weights as probabilities, it is possible to implement spiking networks, and show that as the number of timesteps in the computation increases, the behaviour converges to that of the original (deterministically evaluated) network.
- By altering the way the weights are stored and indexed, we achieve significant decrease in the number of memory accesses required to evaluate a network.
- We present a hardware architecture that can be used as an accelerator on a System-on-Chip (SoC) platform, and can implement this model of computation efficiently. Several optimizations on the architecture are presented that allow significant speedups over the software implementation.
- The impact of this approach is quantified on well known benchmark circuits (MNIST and CIFAR-10).

The paper is organized as follows: we next present the motivation for the probabilistic interpretation of weights, and show through experiments that this can be implemented with minimal impact on accuracy of the network. We quantify the reduction in memory accesses that would result even in a pure software implementation of the scheme. Next, we present an architecture suitable for implementation as a hardware accelerator for an SoC, and study several variants of the probabilistic spike propagation that provide different accuracy *vs* performance tradeoffs. We then discuss the results in the context of previous approaches from the literature, and finally present our conclusions.

2 Motivation

A spiking neural network involves three kinds of operations, namely (a) injection, (b) generation and (c) propagation of spikes. Spike injection involves injecting input spikes into the network, which would excite and initiate activity in the network. This is done either by generating spike trains from static inputs by following some probability distribution, or by input mechanisms that naturally generate spike trains.

Spike generation is the process of evaluating the neurons and their spiking in response to stimuli based on some mathematical model of the neuron. In this work, we have used the Integrate-and-Fire neuron model, but the approach is largely independent of the underlying model. The computational load of this part grows with the number of neurons.

Spike propagation decides which output synapses should be affected by a spike on a neuron. This requires identifying the recipients of the spikes and fetching the corresponding synaptic weights. The computational load for this grows with the number of synapses, making it the bottleneck of SNN evaluation.

Consider a neuron i with N outgoing connections: every time it spikes, all of its outgoing connections must be updated. This requires accessing the N synaptic weights and updating the state variables of the postsynaptic neurons. In the case of fully connected networks, or the fully connected layers of a deep network, the number of connections and weights can be very large, and the memory accesses here can dominate the overall performance of the network. In the present work, we focus on these layers as they are the ones with the highest ratio of memory to computation.

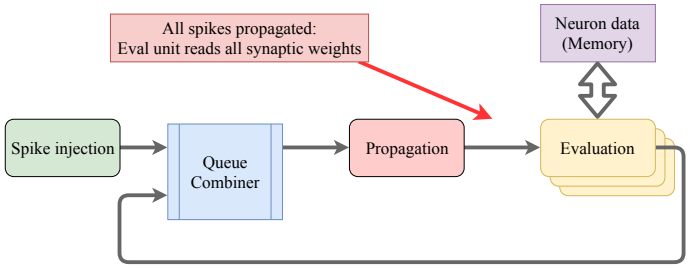
Fig. 1a shows a schematic architecture for this approach: the spike injection unit as well as the neuron evaluation unit feed into a *queue* of spikes, that are in turn processed by the evaluation unit. This unit needs to read in the weights of all outgoing synapses for each spiking neuron, and update the target's membrane potential by an amount equal to the weight of the synapse. In this architecture, almost all the work (and memory access) happens in the evaluation unit, and the propagation unit just passes generated spikes through to the queue for the next timestep.

Fig. 1b shows an alternate view where the decision on which neurons are to be updated in the next timestep is made by the spike propagation unit, which decides whether a spike propagates across a synapse, while the evaluation unit makes the final decision on spiking and inserts entries into the queue for the next timestep. In the next section, we will see how this change can be used to reduce memory accesses.

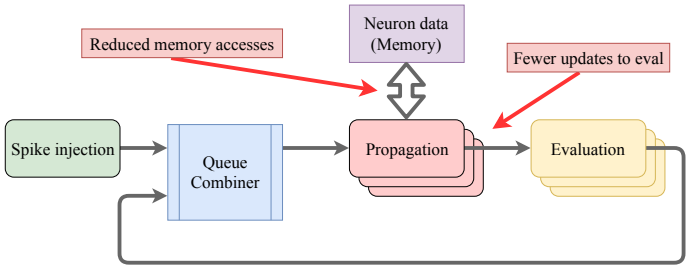
2.1 Probabilistic spike propagation

The typical weight distribution for a neuron shows a few synapses with large weights, tapering down to a relatively large number of synapses of low weight. For example, fig. 2 shows the outgoing synaptic weights of a few sample neurons in a network for the MNIST dataset. Note that if the weights were uniformly distributed, we would expect this chart to be a straight (diagonal) line. For a weight distribution skewed towards the lower bound the curve would be below this diagonal.

One way to take advantage of this is to quantize the weights, and suppress those below a threshold. However this impacts accuracy, and the resulting loss cannot be recovered.



(a) All synapses evaluated



(b) Probabilistic synapse evaluation

Figure 1: Schematic architectures for spike processing.

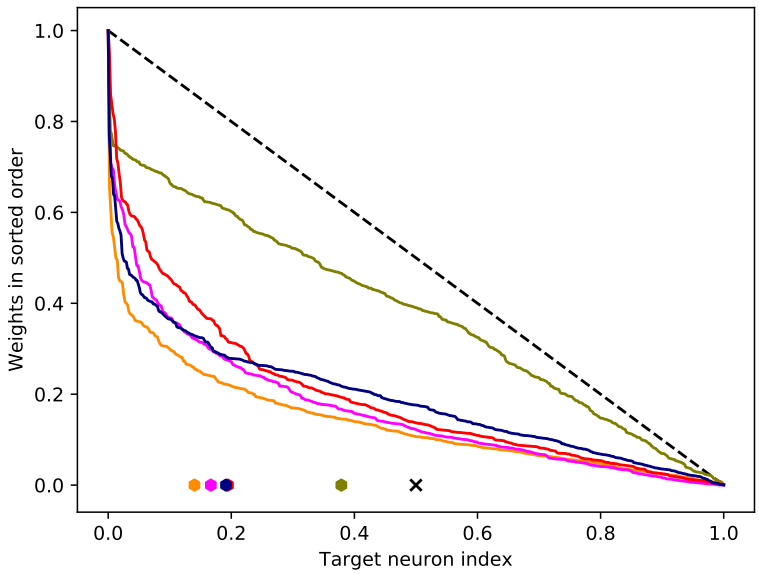


Figure 2: Weights of sample neurons in descending order, normalized against maximum outgoing weight for that neuron; x-axis is index of target neuron normalized by number of outgoing synapses. Expected *termination points* are marked as dots on the x-axis.

We propose an alternate interpretation of a synaptic weight in terms of the probability of propagating a spike on that synapse. By limiting the randomness to the propagation of spikes alone and leaving other aspects of the neuron model unchanged, we show that we can use weights from existing spiking networks without major changes.

In the deterministic approach, when neuron i spikes, the applied weight for every postsynaptic neuron update is equal to the actual weight w_{ij}. For the integrate-and-fire neuron model, over a set of N_i spikes, this will cause an increase in membrane potential of $N_i \times w_{ij}$ on neuron j. If this exceeds the threshold voltage, a spike is produced, and the membrane potential is reset. We hypothesize that it is enough that the temporal sum of the applied weights, across multiple spikes of i, for each neuron j should approach $N_i \times w_{ij}$. In other words, if we apply a weight $\hat{w}_{ij}$ with probability p_{ij}, then it is sufficient that

$$N_i \times w_{ij} = N_i \times p_{ij} \times \hat{w}_{ij} \tag{1}$$

This can be achieved by letting $p_{ij} = \frac{w_{ij}}{w_i^{max}}$ and $\hat{w}_{ij} = w_i^{max}$, where w_i^{max} is the maximum weight for outgoing synapses of neuron i. When neuron i spikes, we generate a random number r uniformly distributed between $[0, w_i^{max}]$, compare r with w_{ij} and update only those postsynaptic neurons j for which the comparison succeeds, with an applied weight of w_i^{max}.

Note that we process *excitatory* and *inhibitory* synapses separately as they correspond to positive and negative weights respectively. Both types are treated in the same way, but for the inhibitory synapses we let $p_{ij} = \frac{w_{ij}}{w_i^{min}}$ and $\hat{w}_{ij} = w_i^{min}$.

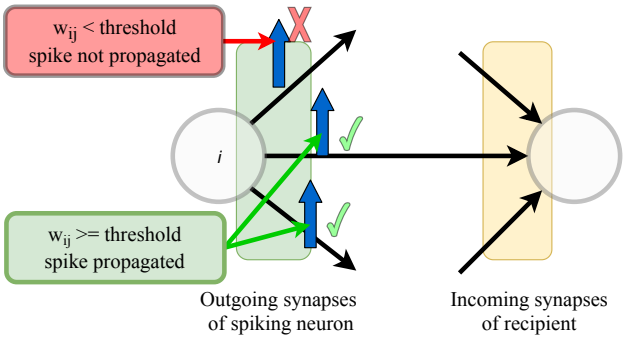


Figure 3: Probabilistic spike propagation.

2.2 Reducing memory accesses

Probabilistic spike propagation can be leveraged to reduce memory fetches for outgoing synaptic weights by sorting the weight array. We generate a random value r per spiking neuron, and propagate the spike to those targets with $w_{ij} > r$. This requires reading the weight and index of the postsynaptic neuron, but as soon as the comparison fails for one synapse, all remaining synapses can be skipped.

Note that we need to store both the actual weight value and the index of the target neuron, potentially requiring twice the amount of memory, and twice the number of memory accesses. Therefore, for this approach to be useful, we will need to reduce both these requirements.

Assume the target neuron indices for the outgoing synapses of neuron i are numbered from 1 to N_i^{max}. Note that N_i^{max}could vary from one neuron to another – this can account for differences between layers or even in the number of excitatory and inhibitory connections a given neuron has. We define the *termination point* of the weight update for neuron i as the smallest index $t \in [1, N_i^{max}]$ for which $sorted\_weight_i[t] < r$. It is clear that we would like the average value of t over all neurons and all timesteps to be small in order to save on memory accesses. The expectation of this t approaches $\sum_j w_{ij}/w_i^{max}$. In particular, if $E[t] < N_i^{max}/2$, then even if we access weights and indices separately to propagate spikes, we still need fewer memory accesses than the original deterministic approach.

This condition is satisfied even for the simple case that the weight distribution for a given neuron is uniform. Since r is drawn from uniform distribution between 0 and w_i^{max}, across multiple spikes on i the expected value ($E[r]$) would approach $w_i^{max}/2$. On the other hand, for the type of distribution discussed in sec. 2.1, the termination point would shift to smaller values, as can be seen in fig. 2. Here, most neurons have termination points much lower than $N_i^{max}/2$, which implies that the probabilistic method would need fewer total memory accesses than the deterministic approach.

2.3 Figure of merit

We introduce the term *Memory accesses per spike* (MAPS) to quantify and compare different implementations of a spiking network. When a neuron spikes, we measure the number of memory accesses (synaptic weights or indices) that are required to *process* the spike, or update neurons affected by this spike.

For example, if the output of a given neuron in a SNN is connected to N other neurons, then in the baseline deterministic approach, we would need to read in the weights for all N synapses, and perform updates on all N of the output neurons. However, with the probabilistic approach, it may be possible to process only a subset of these for a given spike. This will result in a lower MAPS.

Note: it is quite possible that the neuron indices and the synaptic weights require different numbers of bits for storage, which would also impact the energy consumed for reading one of these values. This would vary significantly from one network to another, and there are also known techniques to try and further compress the storage and bandwidth requirements. However, our purpose in the study here is to quantify the savings in memory accesses themselves, and we ignore differences in numbers of bits.

3 Implementation Issues

We now consider how the probabilistic approach can be implemented, both on pure software platforms, as well as on custom hardware.

Algorithm 1: Scan-based termination

Input: i, $sorted\_weight$, $sorted\_index$
$r \leftarrow UniformRandom(0, w_i^{max})$
$j \leftarrow 1$
while $sorted\_weight[j] \geq r$ **do**
  $UpdateNeuron(sorted\_index[j], w_i^{max})$
  j++

Alg. 1 implements the probabilistic approach by scanning weights until the value drops below the randomly generated threshold. This approach has the disadvantage that both the $sorted\_weight[i]$ and $sorted\_index[i]$ need to be read to process each outgoing synapse.

Algorithm 2: Termination with binary search

Input: i, $sorted\_weight$, $sorted\_index$
$r \leftarrow UniformRandom(0, w_i^{max})$
$termpt \leftarrow BinarySearch(sorted\_weights, r)$
for $j \in [1, termpt]$ **do**
  $UpdateNeuron(sorted\_index[j], w_i^{max})$

Alg. 2 modifies this to determine the termination point by running a binary search on the sorted weight array. This can be done much faster ($O(\log N_i^{max})$ comparisons) than the scanning based method (which will end up performing $termpt$ comparisons), and once the termination point has been determined, only the neuron indices ($sorted\_index$) need to be read.

3.1 Hardware Architecture

Though the proposed approach already shows advantages in a software implementation, it is possible to extract even more benefit from a hardware architecture that is able to exploit the memory access patterns appropriately. Rather than designing a full architecture for spiking network processing, we designed a hardware accelerator in an SoC system built around the ARM processor core of a Xilinx Zynq FPGA. A schematic of the architecture is shown in fig. 4.

The main tradeoff we are investigating is to store some weights on-chip in exchange for much lower off-chip access. FPGAs are well suited to this kind of architecture, as they have built-in RAM blocks that can be used for such storage, whereas this could be quite expensive in an ASIC. So even though the core ideas here would apply to an ASIC as well,

the benefits are easiest to realize on an FPGA platform where the approach is implemented as a hardware co-processor for the system CPU.

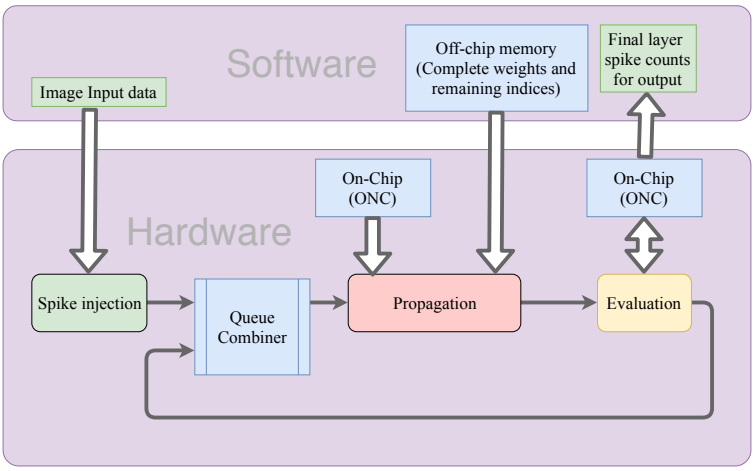


Figure 4: Hardware accelerator architecture

We now examine some variants of the probabilistic method, and how they would impact the resulting architecture and performance.

Binary search

The basic binary search based method allows us to quickly compute a termination point and then use a simple iteration through indices to update the outgoing synapses. While this is a good fit for software, in hardware the random access to weights at different indices that is required for binary search can be a problem, since off-chip memory accesses in hardware are usually best performed in a sequential burst.

Random index

Since the sole purpose of the binary search is to compute a termination point for alg. 2, we examine alternative approaches to solve this problem. The most obvious solution is to just generate $termpt$ randomly, rather than generating r and mapping it to $termpt$. This completely eliminates the need to access weights to decide how many synapses are to be updated, but the drawback is that it also loses any information contained in the weight distribution. As a result, the random index based termination method performs poorly on networks where the majority of weights follow distributions other than uniform.

Weight transformed indexing

Another possibility is to generate a random index $x \in [1, N_i^{max}]$, and use a function $fw_i(x)$ to translate that into a termination point:

$$termpt = fw_i(x) \tag{2}$$

With this formulation, it can be shown that if the normalized weight distribution (weights normalized against w_i^{max}as a function of neuron index normalized against N_i^{max}) satisfies the equation

$$f(f(x)) = x \tag{3}$$

then we can use a transform function of the form

$$termpt = w_i[x] \times \frac{N_i^{max}}{w_i^{max}} \tag{4}$$

It turns out that eq. 3 is reasonably closely satisfied for several practical sorted weight distributions, similar to those seen in fig. 2. For example, a linearly decaying sorted weight distribution or a hyperbolic distribution will both satisfy this condition.

This implies that we can compute the $termpt$ using a single weight lookup: generate a random index x, look up $w_i[x]$ and transform it to $termpt = w_i[x] \times N_i^{max}/w_i^{max}$. Although this requires one random access to the weight memory,

Network	Architecture	Layers
MNIST1	Fully connected	784-1200-1200-10
MNIST2	Convolutional	5x5x32c-2x2p-5x5x64c- -2x2p-2048-10
CIFAR10	Convolutional	3x3x64c-2x2p-3x3x128c- -2x2p-3x3x256c-2x2p-1024-10

Table 1: Benchmark networks: the *Layers* indicate the number of neurons in each layer, with *c* for convolutional, *p* for average-pooling, and others fully connected.

Technique	Termination point
DET	Deterministic (baseline)
BS	Binary search
RI	Random index chosen
TR	Weight transform function
PWL	Piecewise linear with 5 segments

Table 2: Techniques to determine the termination point for the probabilistic approach. DET is the deterministic baseline where all neurons are updated.

this is considerably less expensive than the binary search, and in most cases is found to perform better than the pure random $termpt$ method described previously.

For a hardware implementation, we need to store both sorted weights and sorted indices in memory (which could be off-chip high density DRAM for example), but only the sorted indices will need to be read in large quantities: weights will only be accessed to map the $termpt$.

Piecewise linear approximation

The final approach we consider further trades off additional storage for accuracy. Here, we use the observation from fig. 2 that several of the sorted weight distributions seem to show a pattern of piecewise linear segments. This means that if we can use some additional storage to keep track of indices where the weight distribution changes slope, we could get better accuracy, and closer fidelity to the original sorted weight distribution, while not incurring the full cost of binary search or other methods.

3.2 Experimental results

The proposed approaches were validated on well-known benchmark problems (MNIST and CIFAR-10), using multiple networks that were trained as ANNs and converted to SNNs using the methods in [6]. The networks and the different probabilistic approaches are listed in tables 1 and 2.

We conducted experiments on the different approaches to quantify the loss in accuracy compared to the deterministic approach, and to estimate how many more timesteps are required by the probabilistic method to reach the same accuracy as the deterministic approach. The piecewise linear approximation seems to be the best compromise between hardware complexity (storage required for the piecewise linear breakpoints) and the accuracy.

Table 3 summarizes the reduction in off-chip memory access that can be obtained through use of the probabilistic methods. As off-chip memory typically consumes considerably more energy (and has higher latency) [8] than on-chip memory, we would like to move as many of the weight accesses to on-chip storage as possible. Assuming we have a fixed amount of memory is available on-chip, we would like to know which weights should be stored on-chip to get the best benefit.

The deterministic approach has to read all outgoing synaptic weights in any case, so cannot benefit from this, but the probabilistic methods can gain considerably here. As we can see from the table, for the benchmark networks considered, if we can store around 40% of the weights on-chip, the number of off-chip accesses drops to very low levels.

Most importantly, we can trade off the on-chip storage for reduction in off-chip accesses much more effectively than when doing deterministic evaluation.

Table 4 shows how the accuracy of the probabilistic methods converges to that of the deterministic method as we run for more simulation steps. Even though this looks like it may be a negative point for the probabilistic approach, the fact that the number of memory accesses has been reduced disproportionately (table 3) means that the total memory accesses are in fact lower than the deterministic approach, even after accounting for the additional simulation time.

Network	Prop	Avg MAPS	Fraction of weights on-chip				
			0	0.2	0.4	0.6	0.8
MNIST1	DET	1200	1	0.8	0.6	0.4	0.2
	PWL	240	0.20	0.11	0.06	0.03	0.01
MNIST2	DET	2048	1	0.9	0.8	0.7	0.6
	PWL	684	0.33	0.21	0.12	0.05	0.01
CIFAR10	DET	1024	1	0.9	0.8	0.7	0.6
	PWL	250	0.24	0.25	0.08	0.00	0.00

Table 3: Average memory accesses per spike (MAPS) when a fraction of the weights/indices are stored on-chip (MAPS normalized against the baseline deterministic value)

Network	Steps	DET	BS	RI	TR	PWL
MNIST1	100	98.55	98.48	97.49	98.29	98.43
	200	98.53	98.5	97.51	98.29	98.47
	300	98.54	98.5	97.5	98.29	98.49
	1000	98.56	98.52	97.5	98.29	98.51
MNIST2	100	99.13	98.96	99.05	99.07	99
	200	99.17	99.05	99.13	99.11	99.1
	300	99.16	99.09	99.12	99.12	99.13
	1000	99.17	99.12	99.14	99.11	99.14
CIFAR10	200	74.35	71.73	72.51	73.13	72.4
	400	76.05	74.79	75.63	75.14	74.92
	600	76.44	75.4	76.87	75.62	75.74

Table 4: Accuracy of the probabilistic methods converges towards that of deterministic as the number of simulation steps increases.

Fig. 5 highlights the accuracy convergence: for the two variants of MNIST, we see how the difference between the deterministic and two probabilistic methods decreases with more timesteps.

4 Discussion

The focus of this work is on reducing memory accesses in implementation of spiking neural networks. We have achieved this by introducing a probabilistic approach to spike propagation, that allows weights of known pre-trained spiking networks to be used, without requiring retraining or restructuring the network. We now relate this to previously published works in this area, and bring out the differences in our approach.

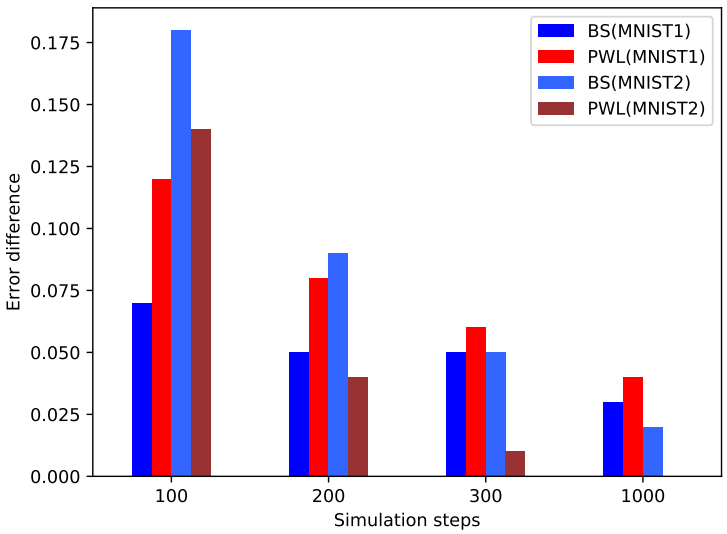


Figure 5: Difference in error percentage over increasing run times between deterministic and probabilistic spike propagation for the MNIST1 and MNIST2 networks.

Use of pre-trained networks

Training spiking neural networks is not as well developed as training conventional artificial neural networks (ANNs), and the level of accuracy obtained using methods such as spike timing dependent plasticity (STDP) [6] are generally lower than the state of the art ANNs. Previous research (eg. [6, 14]) has shown how weights obtained by training an ANN can be converted for use with a spiking network model, with minimal impact on accuracy. A problem with this approach is that the resulting spiking networks are often quite complex. In particular, the expected benefits of spiking networks: less complex compute units, fewer and sparser weights and activations etc. may not be obtained by this method of conversion. However, since they provide high accuracy spiking networks, they are often used as a reference to compare a spiking network against an ANN.

In this paper, we are not concerned with how the spiking network is obtained, only whether we can interpret the synaptic weight as probability of transmission of a spike. We try to show how the benchmark circuit's accuracy can be retained at a high level with the conversion to our probabilistic approach, and not to try and improve on it in any way. In fact, we do not attempt to compare against the original ANN, and are only interested in retaining fidelity to the converted SNN, and comparing the number of memory accesses that would be required for implementing it.

Custom hardware architectures

There have been several custom hardware accelerators designed expressly to implement spiking networks, such as [2–4, 11, 18] *etc.*. These accelerators usually aim to minimize power or energy, for which they introduce certain restrictions on the type of spiking network that can be realized effectively. For example, [4] specifically takes up a custom benchmark example of the LASSO algorithm to demonstrate the effectiveness of the spiking architecture when compared to a software implementation, and similar custom benchmarks are considered by [2]. This is not in itself a shortcoming of those architectures – it could in fact point to a problem with how spiking networks are currently being applied to problems they are not well suited to.

The memory access reduction we consider in this work applies uniformly to any spiking network architecture, including the custom approaches mentioned above. In fact, most of the work on the custom architectures uses techniques such as bitwidth reduction or weight truncation to reduce memory traffic. Our approach is complementary to such techniques, but would require the weights and indices to be stored in sorted order. If this is possible for a given hardware architecture, then it is possible to apply our ideas to these custom hardware systems as well and further improve their performance.

Stochastic techniques

Stochastic computation techniques apply randomness to the process of computation itself [17]. Variants of this approach have been applied to spiking neural networks (eg. [1, 13, 19]). These are mostly orthogonal to the ideas we discuss, since a different (stochastic) hardware architecture for individual compute units can also be incorporated into our approach.

[1] considers the probabilistic model of the neuron itself, but here the neuron is modified so that the spiking behavior itself is stochastic. We want to use pre-existing neuron models without changing the intrinsic spiking behavior, so only the spike propagation is made random in such a way as to exploit the time averaging.

Bayesian spiking neurons [5, 12] apply probabilistic techniques for the neuron models themselves. These and similar probabilistic neuron models such as [9] focus on improving the functionality and scope of neuron models, rather than hardware implementation.

Approximate and Emerging technologies

Approximate computing is well known in the area of signal processing and neural network hardware, but has seen limited application to spiking networks. One example is [15], where neurons are progressively trimmed from evaluation as time progresses. Again, our approach is orthogonal to this, and could be used to further reduce computations even for those neurons that are being evaluated.

Finally, there are approaches that rely on the use of new and emerging technologies, such as spin-based computing ([20]). These are out of the scope of the present work as they completely change the way in which networks are implemented, and everything from the neuron model to the training is different.

5 Conclusions

Repeated accesses to synaptic weights forms the main bottleneck in the evaluation of spiking neural networks, especially in fully connected layers involving large numbers of weights. The probabilistic approach to spike propagation presented in this paper can result in significant savings in the number of memory accesses required to evaluate a spiking neural network. The approach can be applied to a pre-trained spiking network without imposing restrictions on the type of network or the weights, and without requiring retraining. This is made possible by the observation that the long-term average weight applied by this probabilistic approach over a number of timesteps converges to the actual synaptic weight that should have been applied in a deterministic approach.

Experiments on benchmark circuits show that the proposed approach is able to achieve equivalent results to a deterministic spiking network, given enough timesteps. Even though the number of timesteps may be more, the probabilistic approach is able to achieve the same level of accuracy as a deterministic approach using fewer total memory accesses, which would translate directly into a lower total energy of computation. For the benchmark circuits considered, by storing just 40% of the weights from the fully connected layer on chip, we can reduce the number of off-chip memory accesses by close to 90%.

References

[1] K. Ahmed et al. Probabilistic inference using stochastic spiking neural networks on a neurosynaptic processor. In *IJCNN 16*, pages 4286–4293. IEEE, 7 2016.

[2] F. Akopyan et al. TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Trans. on CAD*, 2015.

[3] K. Cheung et al. NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors. *Frontiers in Neuroscience*, 9:516, 1 2016.

[4] M. Davies et al. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1), 2018.

[5] S. Deneve. Bayesian Spiking Neurons I: Inference. *Neural Computation*, 20(1):91–117, 1 2008.

[6] P. Diehl et al. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *Proc. Intl. Joint Conf. on Neural Networks*, 2015.

[7] S. Furber et al. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, 5 2014.

[8] S. Han et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proc. ISCA '16*, volume 44, pages 243–254. IEEE, 6 2016.

[9] N. Kasabov. To spike or not to spike: A probabilistic spiking neuron model. *Neural Networks*, 23(1), 1 2010.

[10] W. Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 12 1997.

[11] D. Neil and S.-C. Liu. Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator. *IEEE Trans. on VLSI*, 22(12):2621–2628, 12 2014.

[12] M. G. Paulin and A. Van Schaik. Bayesian Inference with Spiking Neurons. In *arXiv: 1406.5115*, 2014.

[13] J. L. Rossello et al. Probabilistic-based neural network implementation. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 6 2012.

[14] B. Rueckauer et al. Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. *Frontiers in Neuroscience*, 11:682, 12 2017.

[15] S. Sen et al. Approximate computing for spiking neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 193–198. IEEE, 3 2017.

[16] H. Seung. Learning in Spiking Neural Networks by Reinforcement of Stochastic Synaptic Transmission. *Neuron*, 40(6):1063–1073, 12 2003.

[17] N. R. Shanbhag et al. Stochastic computation. In *Proc. DAC '10*, page 859. ACM Press, 2010.

[18] G. Smaragdos et al. BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations. *J. of Neural Engg.*, 14(6):066008, 12 2017.

[19] S. Smithson et al. Stochastic Computing Can Improve Upon Digital Spiking Neural Networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 309–314. IEEE, 10 2016.

[20] G. Srinivasan et al. Magnetic tunnel junction enabled all-spin stochastic spiking neural network. In *Proc. of DATE, 2017*, pages 530–535. IEEE, 3 2017.