UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Deep Spiking Networks

O'Connor, P.; Welling, M.

Publication date
2016

Document Version
Submitted manuscript

Link to publication

Citation for published version (APA):
O'Connor, P., & Welling, M. (2016). *Deep Spiking Networks*. arXiv.org. https://arxiv.org/abs/1602.08323v2

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:26 May 2022

arXiv:1602.08323v2 [cs.NE] 7 Nov 2016

Deep Spiking Networks

Peter O'Connor
QUVA Lab
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
p.e.oconnor@uva.nl

Max Welling
QUVA Lab
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
m.welling@uva.nl

Abstract

We introduce an algorithm to do backpropagation on a spiking network. Our network is "spiking" in the sense that our neurons accumulate their activation into a potential over time, and only send out a signal (a "spike") when this potential crosses a threshold and the neuron is reset. Neurons only update their states when receiving signals from other neurons. Total computation of the network thus scales with the number of spikes caused by an input rather than network size. We show that the spiking Multi-Layer Perceptron behaves identically, during both prediction and training, to a conventional deep network of rectified-linear units, in the limiting case where we run the spiking network for a long time. We apply this architecture to a conventional classification problem (MNIST) and achieve performance very close to that of a conventional Multi-Layer Perceptron with the same architecture. Our network is a natural architecture for learning based on streaming event-based data, and is a stepping stone towards using spiking neural networks to learn efficiently on streaming data.

1 Introduction

In recent years the success of Deep Learning has proven that a lot of problems in machine-learning can be successfully attacked by applying backpropagation to learn multiple layers of representation. Most of the recent breakthroughs have been achieved through purely supervised learning.

In the standard application of a deep network to a supervised-learning task, we feed some input vector through multiple hidden layers to produce a prediction, which is in turn compared to some target value to find a scalar cost. Parameters of the network are then updated in proportion to their derivatives with respect to that cost. This approach requires that all modules within the network be differentiable. If they are not, no gradient can flow through them, and backpropagation will not work.

An alternative class of artificial neural networks are Spiking Neural Networks. These networks, inspired by biology, consist of neurons that have some persistent "potential" which we refer to as ϕ, and alter each-others' potentials by sending "spikes" to one another. When unit i sends a spike, it increments the potential of each downstream unit j in proportion to the synaptic weight $W_{i,j}$ connecting the units. If this increment brings unit j's potential past some threshold, unit j sends a spike to its downstream units, triggering the same computation in the next layer. Such systems therefore have the interesting property that the amount of computation done depends on the contents of the data, since a neuron may be tuned to produce more spikes in response to some pattern of inputs than another.

In our flavour of spiking networks, a single forward-pass is decomposed into a series of small computations provide successively closer approximations to the true output. This is a useful feature for real time, low-latency applications, as in robotics, where we may want to act on data quickly, before it is fully processed. If an input spike, on average, causes one spike in each downstream layer of the network, the average number of additions required per input-spike will be $\mathcal{O}(\sum_{l=1}^{L} N_l)$, where N_l is the number of units in the layer l. Compare this to a standard network, where the

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

basic messaging entity is a vector. When a vector arrives at the input, full forward pass will require $\mathcal{O}(\sum_{l=1}^{L}(N_{l-1} \cdot N_l))$ multiply-adds, and will yield no "preview" of the network output.

Spiking networks are well-adapted to handle data from event-based sensors, such as the Dynamic Vision Sensor (a.k.a. Silicon Retina, a vision sensor) Lichtsteiner et al. [2008] and the Silicon Cochlea (an audio sensor) Chan et al. [2007]. Instead of sending out samples at a regular rate, as most sensors do, these sensors asynchronously output events when there is a change in the input. They can thus react with very low latency to sensory events, and produce very sparse data. These events could be directly fed into our spiking network (whereas they would have to be binned over time and turned into a vector to be used with a conventional deep network).

In this paper, we formulate a deep spiking network whose function is equivalent to a deep network of Rectified Linear (ReLU) units. We then introduce a spiking version of backpropagation to train this network. Compared to a traditional deep network, our Deep Spiking Network has the following advantageous properties:

1. Early Guessing. Our network can make an "early guess" about the class associated with a stream of input events, before all the data has been presented to the network.

2. No multiplications. Our training procedure consists only of addition, comparison, and indexing operations, which potentially makes it very amenable to efficient hardware implementation.

3. Data-dependent computation. The amount of computation that our network does is a function of the data, rather than the network size. This is especially useful given that our network tends to learn sparse representations.

The remainder of this paper is structured as follows: In Section 2 we discuss past work in combining spiking neural networks and deep learning. In 3 we describe a Spiking Multi-Layer Perceptron. In 4 we show experimental results demonstrating that our network behaves similarly to a conventional deep network in a classification setting. In 5 we discuss the implications of this research and our next steps.

2 Related Work

There has been little work on combining the fields of Deep Learning and Spiking neural networks. The main reason for this is that there is not an obvious way to backpropagate an error signal through a spiking network, since output is a stream of discrete events, rather than smoothly differentiable functions of the input. Bohte et al. [2000] proposes a spiking deep learning algorithm - but it involves simulating a dynamical system, is specific to learning temporal spike patterns, and has not yet been applied at any scale. Buesing et al. [2011] shows how a somewhat biologically plausible spiking network can be interpreted as an MCMC sampler of a high-dimensional probability distribution. Diehl et al. does classification on MNIST with a deep event-based network, but training is done with a regular deep network which is then converted to the spiking domain. A similar approach was used by Hunsberger and Eliasmith [2015] - they came up with a continuous unit which smoothly approximated the the firing rate of a spiking neuron, and did backpropagation on that, then transferred the learned parameters to a spiking network. Neftci et al. [2013] came up with an event-based version of the contrastive-divergence algorithm, which can be used to train a Restricted Boltzmann Machine, but it was never applied in a Deep-Belief Net to learn multiple layers of representation. O'Connor et al. [2013] did create an event-based spiking Deep Belief Net and fed it inputs from event-based sensors, but the network was trained offline in a vector-based system before being converted to run as a spiking network.

Spiking isn't the only form of discretization. Courbariaux et al. [2015] achieved impressive results by devising a scheme for sending back an approximate error gradient in a deep neural network using only low-precision (discrete) values, and additionally found that the discretization served as a good regularizer. Our approach (and spiking approaches in general) differ from this in that they sequentially compute the inputs over time, so that it is not necessary to have finished processing all the information in a given input to make a prediction.

Algorithm 1 Spiking Vector Quantization

1: **Input:** $\vec{v} \in \mathbb{R}^d, T \in \mathbb{N}$
2: **Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
3: **for** t $\in$ 1...T **do**
4: $\quad \vec{\phi} \leftarrow \vec{\phi} + \vec{v}$
5: $\quad$ **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
6: $\quad\quad i \leftarrow argmax(|\vec{\phi}|)$
7: $\quad\quad s \leftarrow sign(\phi_i)$
8: $\quad\quad \vec{\phi}_i \leftarrow \vec{\phi}_i - s$
9: $\quad\quad$ FireSpike(source = i, sign = s)

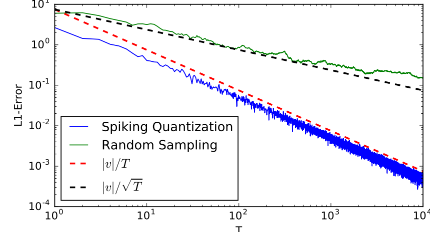

Figure 1: Variable-Spike Quantization shows $1/T$ convergence, while ordinary sampling converges at a rate of $1/\sqrt{T}$. Note both x and y axes are log-scaled.

3 Methods

In Sections 3.1 to3.3 we describe the components used in our model. In Section 3.5 we will use these components to put together a Spiking Multi-Layer Perceptron.

3.1 Spiking Vector Quantization

The neurons in the input layer of our network use an algorithm that we refer to as Spiking Vector Quantization (Algorithm 1) to generate "signed spikes" - that is, spikes with an associated positive or negative value. Given a real vector: $\vec{v}$, representing the input to an array of neurons, and some number of time-steps T, the algorithm generates a series of N signed-spikes: $\langle(i_n, s_n) : i_n \in [1..len(\vec{v})], s_n \in \{\pm1\}, n \in [1..N]\rangle$, where N is the total number of spikes generated from running for T steps, i_n is the index of the neuron from which the n'th spike fires (note that zero or more spikes can fire from a neuron within one time step), $s_n \in \{\pm1\}$ is the sign of the n'th spike.

In Algorithm 1, we maintain an internal vector of "neuron potentials" $\vec{\phi}$. Every time we emit a spike from neuron i we subtract s_i from the potential ϕ_i until $\vec{\phi}$ is in the interval bounded by $(-\frac{1}{2}, \frac{1}{2})^{len(\vec{v})}$. We can show that as we run the algorithm for a longer time (as $T \to \infty$), we observe the following limit:

$$\lim_{T\to\infty} : \vec{v} = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{1}$$

Where $\vec{e_{i_n}}$ is an one-hot encoded vector with index i_n set to 1. The proof is in the supplementary material.

Our algorithm is simply doing a discrete-time, bidirectional version of Delta-Sigma modulation - in which we encode floating point elements of our vector $\vec{v}$ as a stream of signed events. We can see this as doing a sort of "deterministic sampling" or "herding" Welling [2009] of the vector v. Figure 1 shows how the cumulative vector from our stream of events approaches the true value of v at a rate of $1/T$. We can compare this to another approach in which we stochastically sample spikes from the vector $\vec{v}$ with probabilities proportional to the magnitude of elements of $\vec{v}$, (see the "Stochastic Sampling" section of the supplementary material), which has a convergence of $1/\sqrt{T}$.

3.2 Spiking Stream Quantization

A small modification to the above method allows us to turn a stream of vectors into a stream of signed-spikes.

If instead of a fixed vector $\vec{v}$ we take a stream of vectors $v_{stream} = \{\vec{v_1}, ...\vec{v_T}\}$, we can modify the quantization algorithm to increment $\vec{\phi}$ by $\vec{v_t}$ on timestep t. This modifies Equation 8 to:

$$\lim_{T\to\infty} : \frac{1}{T}\sum_{t=1}^{T} \vec{v_t} = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{2}$$

So we end up approximating the running mean of v_{stream}. See "Spiking Stream Quantization" in the supplementary material for full algorithm and explanation. When we apply this to implement a neural network in Section 3.5, this stream of vectors will be the rows of the weight matrix indexed by the incoming spikes.

3.3 Rectifying Spiking Stream Quantization

We can add a slight tweak to our Spiking Stream Quantization algorithm to create a spiking version of a rectified-linear (ReLU) unit. To do this, we only fire events on positive threshold-crossings, resulting in Algorithm 7.

Algorithm 2 Rectified Spiking Stream Quantization

1: **Input:** $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
2: **Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
3: **for** t $\in 1...T$ **do**
4: $\quad \vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
5: $\quad$ **while** $max(\vec{\phi}) > \frac{1}{2}$ **do**
6: $\quad\quad i \leftarrow argmax(\vec{\phi})$
7: $\quad\quad \vec{\phi}_i \leftarrow \vec{\phi}_i - 1$
8: $\quad\quad$ FireSpike(source = i, sign = +1)

We can show that if we draw spikes from a stream of vectors in the manner described in Algorithm 7, and sum up our spikes, we approach the behaviour of a ReLU layer:

$$\lim_{T \to \infty} : \max\left(0, \frac{1}{T}\sum_{t=1}^{T} \vec{v_t}\right) = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} \tag{3}$$

See "Rectified Stream Quantization" in the supplementary material for a more detailed explanation.

3.4 Incremental Dot-Product

Thus far, we've shown that our quantization method transforms a vector into a stream of events. Here we will show that this can be used to incrementally approximate the dot product of a vector and a matrix. Suppose we define a vector $\vec{u} \leftarrow \vec{v} \cdot W$, Where W is a matrix of parameters. Given a vector $\vec{v}$, and using Equation 8, we see that we can approximate the dot product with a sequence of additions:

$$\vec{u} = \vec{v} \cdot W \approx \frac{1}{T}\big(\sum_{n=1}^{N} \vec{e}_{i_n} s_n\big) \cdot W = \frac{1}{T}\big(\sum_{n=1}^{N} s_n \vec{e}_{i_n} \cdot W\big) = \frac{1}{T}\big(\sum_{n=1}^{N} s_n \vec{W}_{i_n,\cdot}\big) \tag{4}$$

Where $W_{i,\cdot}$ is the i'th row of matrix W.

3.5 Forward Pass of a Neural Network

Using the parts we've described so far, Algorithm 8 describes the forward pass of a neural network. The InputLayer procedure demonstrates how Spike Vector Quantization, shown in Algorithm 1 transforms the vector into a stream of events. The HiddenLayer procedure shows how we can combine the Incremental Dot-Product (Equation 4) and Rectifying Spiking Stream Quantization (Equation 3) to approximate the a fully-connected ReLU layer of a neural network. The Figure in the "MLP Convergence" section of the supplimentary material shows that our spiking network, if run for a long time, exactly approaches the function of the ReLU network.

3.6 Backward Pass

In the backwards pass we propagate error spikes backwards, in the same manner as we propagated the signal forwards, so that the error spikes approach the true gradients of the ReLU network as

Algorithm 3 Pseudocode for a forward pass in a network with one hidden layer

```
1:  function FORWARDPASS(x⃗ ∈ ℝ^{d_in}, T ∈ ℕ)
2:  |  Variable: u⃗ ∈ ℝ^{d_out} ← 0⃗
3:  |  for t ∈ 1...T do
4:  |  |  InputLayer(x⃗)
5:  |  return u⃗/T
6:  |  procedure INPUTLAYER(v⃗ ∈ ℝ^{d_in})
7:  |  |  Internal: φ⃗ ∈ ℝ^{d_in}
8:  |  |  φ⃗ ⇐ φ⃗ + v⃗
9:  |  |  while max(|φ⃗|) > 1/2 do
10: |  |  |  i ← argmax(|φ⃗|)
11: |  |  |  s = sign(φ_i)
12: |  |  |  φ_i ← φ_i − s
13: |  |  |  HiddenLayer(i, s)
14: |  procedure HIDDENLAYER(i ∈ [1..d_in], s ∈ [−1, +1])
15: |  |  Internal: φ⃗ ∈ ℝ^{d_hid}, W ∈ ℝ^{d_in × d_hid}
16: |  |  φ⃗ ⇐ φ⃗ + s · W_{i,·}
17: |  |  while max(φ⃗) > 1/2 do
18: |  |  |  i ← argmax(φ⃗)
19: |  |  |  φ_i ← φ_i − 1
20: |  |  |  OutputLayer(i)
21: |  procedure OUTPUTLAYER(i ∈ [1..d_hid])
22: |  |  Internal: W ∈ ℝ^{d_hid × d_out}
23: |  |  Global: u⃗ ← u⃗ + W_{i,·}                          ▷ changes u⃗
```

$T \to \infty$. Pseudocode explaining the procedure is provided in the “Training Iteration” Section of the supplementary material, and a diagram explaining the flow of signals is in the “Network Diagram” section.

A ReLU unit has the function and derivative:

$$\begin{aligned} f(x) &= [x > 0] \cdot x \\ f'(x) &= [x > 0] \end{aligned} \tag{5}$$

Where:
$[x > 0]$ denotes a step function (1 if $x > 0$ otherwise 0).

In the spiking domain, we express this simply by blocking error spikes on units for which the cumulative sum of inputs into that unit is below 0 (see the "filter" modules in the “Network Diagram” section of the supplementary material).

The signed-spikes that represent the backpropagating error gradient at a given layer are used to index columns of that layer’s weight matrix, and negate them if the sign of the spike is negative. The resulting vector is then quantized, and the resulting spikes are sent back to previous layers.

One problem with the scheme described so far is that, when errors are small, it is possible that the error-quantizing neurons never accumulate enough potential to send a spike before the training iteration is over. If this is the case, we will never be able to learn when error gradients are sufficiently small. Indeed, when initial weights are too low, and therefore the initial magnitude of the backpropagated error signal is too small, the network does not learn at all. This is not a problem in traditional deep networks, because no matter how small the magnitude, some error signal will always get through (unless all hidden units are in their inactive regime) and the network will learn to increase the size of its weights. We found that a surprisingly effective solution to this problem is to simply not reset the ϕ of our error quantizers between training iterations. This way, after some burn-in period, the quantizer’s ϕ starts each new training iteration at some random point in the interval $[-\frac{1}{2}, \frac{1}{2}]$, and the unit always has a chance to spike.

A further issue that comes up when designing the backward pass is the order in which we process events. Since an event can move a ReLU unit out of its active range, which blocks the transmission of itself or future events on the backward pass, we need to think about the order in which we

processing these events. The topic of event-routing is explained in the "Event Routing" section of the supplementary material.

3.7 Weight Updates

We can collect spike statistics and generates weight updates. There are two methods by which we can update the weights. These are as follows:

Stochastic Gradient Descent The most obvious method of training is to approximate stochastic gradient descent. In this case, we accumulate two spike-count vectors, $\vec{c}_{in}$ and $\vec{c}_{error}$ and take their outer product at the end of a training iteration to compute the weight update:

$$\Delta W \leftarrow -\frac{\eta}{T} \cdot \vec{c_{in}} \otimes \vec{c}_{error} \approx -\frac{\eta}{T} \frac{\partial \mathcal{L}}{\partial W} \tag{6}$$

Fractional Stochastic Gradient Descent (FSGD) We can also try some thing new. Our spiking network introduces a new feature: if a data point is decomposed as a stream of events, we can do parameter updates even before a single data point has been observed. If we do updates whenever an error event comes back, we update each weight based on only the input data that has been seen *so far*. This is described by the rule:

$$\Delta W_{:,i} \leftarrow -\frac{\eta}{T} \cdot s \cdot \vec{c_{in}} \tag{7}$$

Where $\vec{c_{in}}$ is an integer vector of counted input spikes, $\Delta W_{:,i}$ is the change to the $i'th$ column of the weight matrix, $s \in \{-1, 1\}$ is the sign of the error event, and i is the index of the unit that produced that error event, and T is the number of time-steps per training iteration. Early input events will contribute to more weight updates than those seen near the end of a training iteration. Experimentally (see Section 4.2, we see that this works quite well. It may be that the additional influence given to early inputs causes the network to learn to make better predictions earlier on, compensating for the approximation caused by finite-runtime of the network.

3.8 Training

We chose to train the network with one sample at a time, although in principle it is possible to do minibatch training. We select a number of time steps T, to run the network for each iteration of training. At the beginning of a round of training, we reset the state of the forward-neurons (all $\vec{\phi}$'s and the state of the running sum modules), and leave the state of the error-quantizing neurons (as described in 3.6). On each time step t, we feed the input vector to the input quantizer, and propagate the resulting spikes through the network. We then propagate an error spike back from the unit corresponding to the correct class label, and update the parameters by one of the two methods described in 3.7. See the "Network Diagram" section of the supplementary material to get an idea of the layout of all the modules.

4 Experiments

4.1 Simple Regression

We first test our network as a simple regressor, (with no hidden layers) on a binarized version of the newsgroups-20 dataset, where we do a 2-way classification between the electronics and medical newsgroups based word-count vectors. We split the dataset with a 7-1 training-test ratio (as in Crammer et al. [2009]) but do not do cross-validation. Table 1 shows that it works.

4.2 Comparison to ReLU Network on MNIST

We ran both the spiking network and the equivalent ReLU network on MNIST, using an architecture with 2 fully-connected hidden layers, each consisting of 300 units. Refer to the "Hyperparameters" section of the Supplimentary Material for a full description of hyperparameters.

Table 2 shows the results of our experiment, after 50 epochs of training. We find that the conventional ReLU network outperforms our spiking network, but only marginally. In order to determine how

Network	% Test / Training Error
1 Layer NN	2.278 / 0.127
Spiking Regressor	2.278 / 0.82
SVM	4.82 / 0

Table 1: Scores on 20 newsgroups, 2-way classification between 'med' and 'electronic' newsgroups. We see that, somewhat surprisingly, our approach outperforms the SVM. This is probably because, being trained through SGD and tested at the end of each epoch, our classifier had more recently learned on samples at the end of the training set, which are closer in distribution to the test set than those at the beginning.

Network	% Test / Training Error
Spiking SGD:	3.6 / 2.484
Spiking FSGD:	2.07 / 0.37
Vector ReLU MLP	1.63 / 0.426
Spiking with ReLU Weights	1.66 / 0.426
ReLU with Spiking FSGD weights	2.03 / 0.34

Table 2: Scores of various implementations on MNIST after 50 epochs of training on a network with hidden layers of sizes [300, 300]. "Spiking SDG" and "Spiking FSGD" are the spiking network trained with Stochastic Gradient Descent and Fractional Stochastic Gradient descent, respectively, as described in Section 3.7. "Vector ReLU MLP" is the score of a conventional MLP with ReLU units and the same architecture and training scheme. "Spiking with ReLU Weights" is the score if we set the parameters of the Spiking network to the already-trained parameters of the ReLU network, then use the Spiking Network to classify MNIST. "ReLU with Spiking weights" is the inverse - we take the parameters trained in the spiking FSGD network and map them to the ReLU net.

much of that difference was due to the fact that the Spiking network has a discrete forward pass, we mapped the learned parameters from the ReLU network onto the Spiking network (spiking with ReLU Weights"), and used the Spiking network to classify . The performance of the spiking network improved nearly to that of the ReLU network , indicating that the difference was not just due to the discretization of the forward pass but also due to the parameters learned in training. We also did the inverse (ReLU with Spiking-FSGD-trained weights) - map the parameters of the trained Spiking Net onto the ReLU net, and found that the performance became very similar to that of the original Spiking (FSGD-Trained) Network. This tells us that most of the difference in score is due to the approximations in training, rather than the forward pass. Interestingly, our Spiking-FSGD approach outperforms the Spiking-SGD - it seems that by putting more emphasis on early events, we compensate for the finite runtime of the Spiking Network. Figure 2 shows the learning curves over the first 20-epochs of training. We see that the gap between training and test performance is much smaller in our Spiking network than in the ReLU network, and speculate that this may be to the regularization effect of the spiking. To confirm this, we would have to show that on a larger network, our regularization actually helps to prevent overfitting.

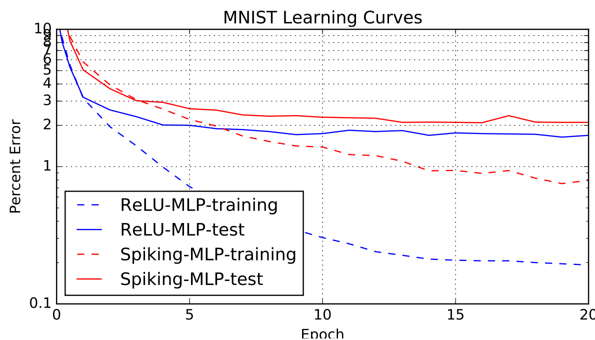

Figure 2: The Learning curves of the ReLU network (blue) and the Spiking network (red). Solid lines indicate test error and dashed lines indicate training error.

4.3 Early Guessing

We evaluated the "early guess" hypothesis from Section 1 using MNIST. The hypothesis was that our spiking network should be able to make computational cheap “early guesses" about the class of the input, before actually seeing all the data. A related hypothesis was under the “Fractional” update scheme discussed in Section 3.7, our networks should learn to make early guesses more effectively than networks trained under regular Stochastic Gradient Descent, because early input events contribute to more weight updates than later ones. Figure 3 shows the results of this experiment. We find, unfortunately, that our first hypothesis does not hold. The early guesses we get with the spiking network cost more than a single (sparse) forward pass of the input vector would. The second hypothesis, however, is supported by the right-side of Figure 5. Our networks trained with Fractional Stochastic Gradient Descent make better early guesses than those trained on regular SGD.

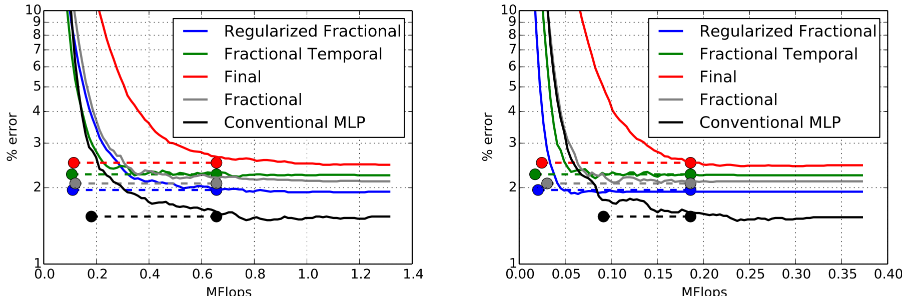

Figure 3: Left: We compare the total computation (x-axis, in MegaFlops) required to achieve a given score (y-axis, in percent error), between differently trained predictors. Each curve represents a differently trained spiking network (Regularized Fraction was trained with a regularization term, Fractional Temporal was trained with higher learning rate for early events). The black line is the convergence curve of a spiking network with parameters learned in a conventional ReLU net of the same architecture (784-300-300-10). The dots show the computation time and performance of a conventional ReLU network, with the right dot indicating the cost of a full feedforward pass, and the left indicating the cost when one removes units with 0-activation when computing the cost of a matrix multiplication. We see that, when considering the full network, our spiking approach does give a good early guess compared to a naively implemented deep net, but not after considering sparsity. Right: However, when only considering layers after the input layer (whose sparsity we do not control), we can see that there is an advantage to our spiking training scheme: In the low-flop range our spiking nets have lower error. The networks that were trained as spiking networks are better at making early guesses than the conventionally trained network.

5 Discussion

We implemented a Spiking Multi-Layer Perceptron and showed that our network behaves very similarly to a conventional MLP with rectified-linear units. However, our model has some advantages over a regular MLP, most of which have yet to be explored in full. Our network needs neither multiplication nor floating-point numbers to work. If we use Fractional Stochastic Gradient Descent, and scale all parameters in the network (initial weights, thresholds, and the learning rate) by the inverse of the learning rate, the only operations used are integer addition, indexing, and comparison. This makes our system very amenable to efficient hardware implementation.

The Spiking MLP brings us one step closer to making a connection between the types of neural networks we observe in biology and the type we use in deep learning. Like biological neurons, our units maintain an internal potential, and only communicate when this potential crosses some firing threshold. We believe that the main value of this approach is that it is a stepping stone towards a new type of deep learning. The way that deep learning is done now takes no advantage of the huge temporal redundancy in natural data. In the future we would like to adapt the methods developed here to work with nonstationary data. Such a network could pass spikes to keep the output distribution in “sync” with an ever-changing input distribution. This property - efficiently keeping an online track of latent variables in the environment, could bring deep learning into the world of robotics.

References

Sander M Bohte, Joost N Kok, and Johannes A La Poutré. Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, pages 419–424, 2000.

Lars Buesing, Johannes Bill, Bernhard Nessler, and Wolfgang Maass. Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons. *PLoS Comput Biol*, 7 (11):e1002211, 2011.

Vincent Chan, Shih-Chii Liu, and André Van Schaik. Aer ear: A matched silicon cochlea pair with address event representation interface. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54(1):48–59, 2007.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015. URL `http://arxiv.org/abs/1511.00363`.

Koby Crammer, Alex Kulesza, and Mark Dredze. Adaptive regularization of weight vectors. In *Advances in neural information processing systems*, pages 414–422, 2009.

Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing.

Xiequan Fan, Ion Grama, and Quansheng Liu. Hoeffding's inequality for supermartingales. *Stochastic Processes and their Applications*, 122(10):3545–3559, 2012.

Eric Hunsberger and Chris Eliasmith. Spiking deep networks with lif neurons. *arXiv preprint arXiv:1510.08829*, 2015.

Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128$\times$ 128 120 db 15 μs latency asynchronous temporal contrast vision sensor. *Solid-State Circuits, IEEE Journal of*, 43(2): 566–576, 2008.

Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in neuroscience*, 7, 2013.

Peter O'Connor, Daniel Neil, Shih-Chii Liu, Tobi Delbruck, and Michael Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in neuroscience*, 7, 2013.

Max Welling. Herding dynamical weights to learn. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1121–1128. ACM, 2009.

A Algorithms

A.1 Proof of convergence of Spike-Vector Quantization

Here we show that if we obtain events $\langle(i_n, s_n) : n \in [1..N]\rangle$ given a vector $\vec{v}$ and a time T from the Spiking-Vector Quantization Algorithm then:

$$\lim_{T\to\infty} \vec{v} = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{8}$$

Algorithm 4 Spiking Vector Quantization

1: **Input:** $\vec{v} \in \mathbb{R}^d, T \in \mathbb{N}$
2: **Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
3: **for** t $\in 1...T$ **do**
4: │ $\vec{\phi} \leftarrow \vec{\phi} + \vec{v}$
5: │ **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
6: │ │ $i \leftarrow argmax(|\vec{\phi}|)$
7: │ │ $s \leftarrow sign(\phi_i)$
8: │ │ $\vec{\phi}_i \leftarrow \vec{\phi}_i - s$
9: │ │ FireSpike(source = i, sign = s)

Since $\forall i : -\frac{1}{2} < \phi_i < \frac{1}{2}$ the L1 norm is bounded by:

$$\|\phi_T\|_{L1} = \|\sum_{t=1}^{T} v - \sum_{n=1}^{N} e_{i_n} s_n\|_{L1} < \frac{l(\vec{v})}{2} \tag{9}$$

where $l(\vec{v})$ is the number of elements in vector $\vec{v}$. We can take the limit of infinite time, and show that our spikes converge to form an approximation of $\vec{v}$:

$$\begin{aligned} &\lim_{T\to\infty} : \frac{1}{T}\|\phi_T\|_{L1} = \lim_{T\to\infty} : \|\frac{1}{T}\sum_{t=1}^{T} v - \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n\|_{L1} = 0 \\ &\lim_{T\to\infty} : \vec{v} = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n \end{aligned} \tag{10}$$

A.2 Stochastic Sampling

Algorithm 5 Stochastic Sampling of events from a vector

1: **Input:** vector v, int T
2: $mag \leftarrow sum(abs(\vec{v}))$
3: $\vec{p} = abs(\vec{v})/mag$
4: **for** t ∈ $1...T$ **do**
5: │ $N = poisson(mag)$
6: │ **for** n ∈ $1...N$ **do**
7: │ │ $i \leftarrow DrawSample(\vec{p})$
8: │ │ $s \leftarrow sign(v_i)$
9: │ │ FireSignedSpike(index = i, sign = s)

A.3 Spiking Stream Quantization

In our modification to Spiking Vector Quantization, we instead feed in a stream of vectors, as in Algorithm 6.

Algorithm 6 Spiking Stream Quantization

1: **Input:** $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
2: **Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
3: **for** t $\in 1...T$ **do**
4: $\quad \vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
5: $\quad$ **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
6: $\quad\quad i \leftarrow argmax(|\vec{\phi}|)$
7: $\quad\quad s \leftarrow sign(\phi_i)$
8: $\quad\quad \vec{\phi_i} \leftarrow \vec{\phi_i} - s$
9: $\quad\quad$ FireSpike(source = i, sign = s)

If we simply replace the term $\sum_{t=1}^{T} \vec{v}$ in Equation 10 with $\sum_{t=1}^{T} \vec{v_t}$, and follow the same reasoning, we find that we converge to the running mean of the vector-stream.

$$\begin{aligned} \lim_{T\to\infty} : \frac{1}{T}\|\phi_T\|_{L1} = \lim_{T\to\infty} : \|\frac{1}{T}\sum_{t=1}^{T} v_t - \frac{1}{T}\sum_{n=1}^{N} e_{\vec{i_n}} s_n\|_{L1} = 0 \\ \lim_{T\to\infty} : \frac{1}{T}\sum_{t=1}^{T} \vec{v} = \frac{1}{T}\sum_{n=1}^{N} e_{\vec{i_n}} s_n \end{aligned} \tag{11}$$

A.4 Rectified Stream Quantization

We can further make a small modification where we only send positive spikes (so our $\vec{\phi}$ can get unboundedly negative.

Algorithm 7 Rectified Spiking Stream Quantization

1: **Input:** $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
2: **Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
3: **for** t $\in 1...T$ **do**
4: $\quad \vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
5: $\quad$ **while** $max(\vec{\phi}) > \frac{1}{2}$ **do**
6: $\quad\quad i \leftarrow argmax(\vec{\phi})$
7: $\quad\quad \vec{\phi_i} \leftarrow \vec{\phi_i} - 1$
8: $\quad\quad$ FireSpike(source = i, sign = +1)

To see why this construction approximates a ReLU unit, first observe that the total number of spikes emitted can be computed by considering the total cumulative sum $\sum_{t=1}^{T} v_{j,t}$. More precisely:

$$N_{j,T} = \max\big(0, \lfloor \max_{T'\in[1...T]} \big(\sum_{t=1}^{T'} v_{j,t} + \frac{1}{2}\big)\rfloor\big) \tag{12}$$

where $N_{j,T}$ indicates the number of spikes emitted from unit j by time T and $\lfloor\cdot\rfloor$ indicates the integer floor of a real number.

Assume the $v_{j,t}$ are IID sampled from some process with mean $E[v_{j,t}] = \mu_j$ and finite standard deviation σ_j. Define $\zeta_{j,t} = v_{j,t} - \mu_j$ which has zero mean and the cumulative sum $\xi_{j,T} = \sum_{t=1}^{T} \zeta_{j,t}$ which is martingale. There are a number of concentration inequalities, such as the Bernstein concentration inequalities Fan et al. [2012] that bound the sum or the maximum of the sequence $\xi_{j,T}$ under various conditions. What is only important for us is the fact that in the limit $T \to \infty$ the sums $\xi_{j,T}$ concentrate to a delta peak at zero in probability and that we can therefore conclude $\sum_{t=1}^{T} v_{j,t} \to T\mu_j$ from which we can also conclude that the maximum, and thus the number of spikes will grow in the same way. From this we finally conclude that $\frac{N_{j,T}}{T} \to \max(0, \mu)$, which is the ReLU non-linearity. Thus the mean spiking rate approaches the ReLU function of the mean input.

B MLP Convergence

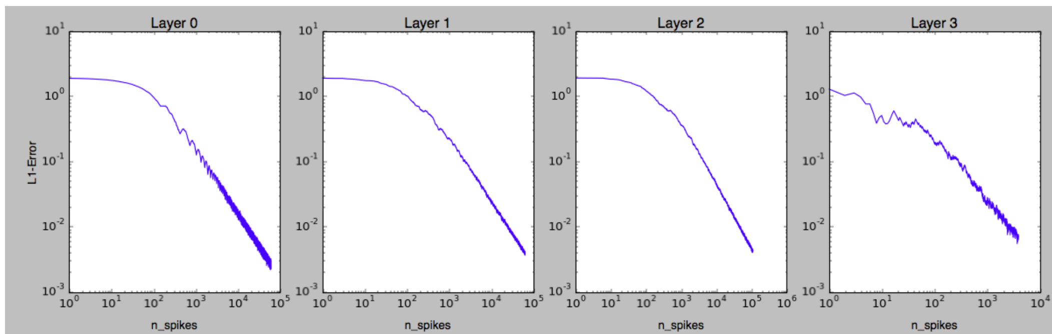

Figure 4: A 3-layer MLP (784-500-500-10) MLP with random weights ($\sim \mathcal{N}(0, 0.1)$) is fed with a random input vector, and a forward pass is computed. We compare the response of the ReLU network to the counts of spikes from our spiking network, and see that over all layers, the responses converge as $T \to \infty$. Note both x and y axes are log-scaled.

C A Training Iteration

Algorithm 8 Pseudocode for a single Training Iteration on a network with one hidden layer with Fractional Stochastic Gradient Descent

1: **procedure** TRAININGITERATION($\vec{x} \in \mathbb{R}^{d_{in}}$, $\vec{y} \in \mathbb{R}^{d_{out}}$, $T \in \mathbb{N}$, $\eta \in \mathbb{R}$)
2: **Variables:** $\vec{u} \in \mathbb{R}^{d_{out}} \leftarrow \vec{0}$, $W_{hid} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W_{out} \in \mathbb{R}^{d_{hid} \times d_{out}}$
3: **for** t $\in 1...T$ **do**
4: InputLayer.forward($\vec{x}$)
5: OutputLayer.backward($\vec{u} - \vec{y}$)
6: **procedure** INPUTLAYER($\vec{v} \in \mathbb{R}^{d_{in}}$)
7: **Internal:** $\vec{\phi} \in \mathbb{R}^{d_{in}}$
8: $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v}$
9: **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
10: $i \leftarrow argmax(|\vec{\phi}|)$
11: s = $sign(\phi_i)$
12: $\phi_i \leftarrow \phi_i - s$
13: HiddenLayer(i, s)
Object: HiddenLayer
14: **Internal:** $\vec{c_{in}} \in \mathbb{R}^{d_{in}} \leftarrow \vec{0}$, $\vec{c_{out}} \in \mathbb{R}^{d_{hid}} \leftarrow \vec{0}$
15: **procedure** FORWARD($i \in [1..d_{in}]$, $s \in [-1, +1]$)
16: **Internal:** $\vec{\phi} \in \mathbb{R}^{d_{hid}}$
17: $c_{in_i} \leftarrow c_{in_i} + s$
18: $\vec{\phi} \Leftarrow \vec{\phi} + s \cdot [W_{hid}]_{i,\cdot}$
19: $\vec{c_{out}} \leftarrow \vec{c_{out}} + [W_{hid}]_{i,\cdot}$
20: **while** $max(\vec{\phi}) > \frac{1}{2}$ **do**
21: $i \leftarrow argmax(\vec{\phi})$
22: $\phi_i \leftarrow \phi_i - 1$
23: OutputLayer(i, +1)
24: **procedure** BACKWARD($\vec{v} \in \mathbb{R}^{d_{hid}}$)
25: **Internal:** $\vec{\phi} \in \mathbb{R}^{d_{hid}}$
26: $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v} \odot [\vec{c_{out}} > 0]$
27: **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
28: $j \leftarrow argmax(|\vec{\phi}|)$
29: s = $sign(\phi_i)$
30: $\phi_j \leftarrow \phi_j - s$
31: $[W_{hid}]_{\cdot,j} \leftarrow [W_{hid}]_{\cdot,j} - \eta \cdot s \cdot \vec{c_{in}}$ ▷ Update to W_{hid}
Object: OutputLayer
32: **Internal:** $\vec{c_{in}} \in \mathbb{R}^{d_{hid}} \leftarrow \vec{0}$
33: **procedure** FORWARD($i \in [1..d_{hid}]$, $s \in [-1, +1]$)
34: $c_{in_i} \leftarrow c_{in_i} + s$
35: **Global:** $\vec{u} \leftarrow \vec{u} + s \cdot [W_{out}]_{i,\cdot}$ ▷ Update to $\vec{u}$
36: **procedure** BACKWARD($v \in \mathbb{R}^{d_{out}}$)
37: **Internal:** $\vec{\phi} \Leftarrow \vec{\phi} + s \cdot W_{i,\cdot}$
38: **if** $c_j > 0$ **then**
39: $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v}$
40: **while** $max(|\vec{\phi}|) > \frac{1}{2}$ **do**
41: $j \leftarrow argmax(|\vec{\phi}|)$
42: s = $sign(\phi_i)$
43: $\phi_i \leftarrow \phi_i - s$
44: $[W_{out}]_{\cdot,j} \leftarrow [W_{out}]_{\cdot,j} - \eta \cdot s \cdot \vec{c_{in}}$ ▷ Update to W_{out}
45: HiddenLayer.Backward($s \cdot [W_{out}]_{\cdot,j}$)

D Network Diagram

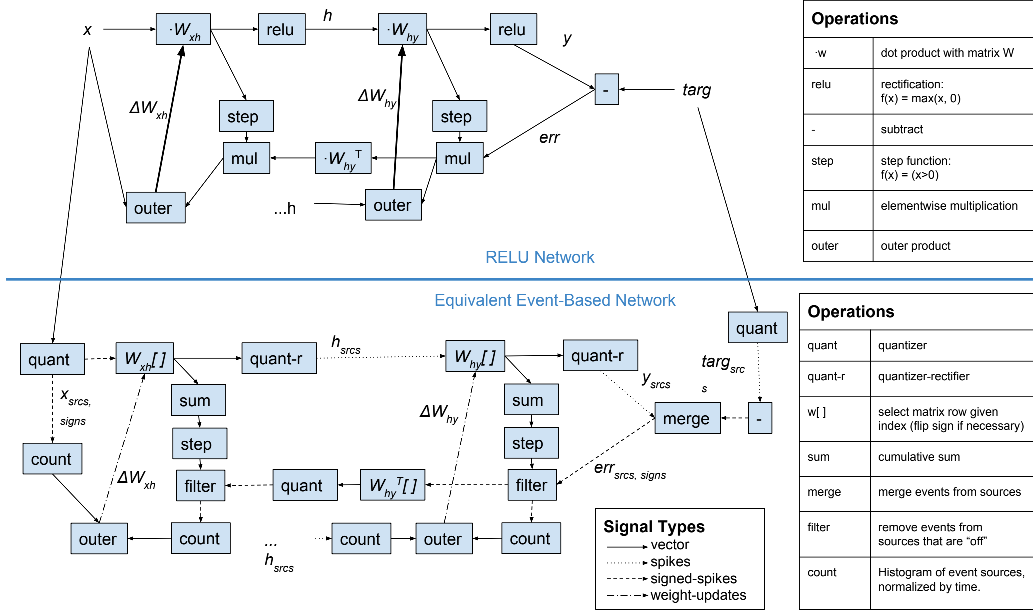

Figure 5: The architecture of the Spiking MLP. On the top, we have a conventional neural network of rectified linear units with one hidden layer. On the bottom, we have the equivalent spiking network.

E Hyperparameters

Our spiking architecture introduced a number of new hyperparameters and settings that are unfamiliar with those used to regular neural networks. We chose to evaluate these empirically by modifying them one-by-one as compared to a baseline.

- Fractional Updates.
  - False (Baseline): We use the standard stochastic-gradient descent method
  - True: We use our new Fractional Stochastic Gradient Descent method - described in section 3.7
- Depth-First
  - False (Baseline): Events are propagated "Breadth-first", meaning that, at a given time-step, all events are collected from the output of one module before any of their child-events are processed.
  - True: If an event from module A creates child-events from module B, those are processed immediately, before any more events from module A are processed.
- Smooth Weight Updates
  - False (Baseline): The weight-update modules take in a count of spikes from the previous layer as their input.
  - True: The weight-update modules take the rectified cumulative sum of the pre-quantized vectors from the previous layer - resulting in a smoother estimate of the input.
- Backwards-Quantization:
  - No-Reset-Quantization (Baseline): The backwards quantization modules do not reset their $\vec{\phi}$s with each training iteration.
  - Random: Each element of $\vec{\phi}$ is randomly selection from the interval $[-\frac{1}{2}, \frac{1}{2}]$ at the start of each training iteration.
  - Zero-Reset: The backwards quantizers reset their $\vec{\phi}$s to zero at the start of each training iteration.

- Number of time-steps: How many time steps to run the training procedure for each sample (Baseline is 10).

Since none of these hyperparameters have obvious values, we tested them empirically with a network with layer sizes [784-200-200-10], trained on MNIST. Table 3 shows the affects of these hyperparameters.

Table 3: Percent error on MNIST for various settings. We explore the effects of different network settings by changing one at a time, training on MNIST, and comparing the result to a baseline network. The baseline network has layer-sizes [784, 200, 200, 10], uses regular (non-fractional) stochastic gradient descent, uses Breadth-First (as opposed to Depth-First) event ordering, does not use smooth weight updates, uses the "No-reset" scheme for its backward pass quantizers, and runs for 10 time-steps on each iteration.

Variant	**% Error**
Baseline	3.38
Fractional Updates	3.10
Depth-First Propagation	81.47
Smooth Gradients	2.85
Smooth & Fractional	3.07
Back-Quantization = Zero-Reset	87.87
Back-Quantization = Random	3.15
5 Time Steps	4.41
20 Time Steps	2.65

Most of the Hyperparameter settings appear to make a small difference. A noteable exception is the Zero-Reset rule for our backwards-quantizing units - the network learns almost nothing throughout training. The reason for this is that the initial weights, which were drawn from $\mathcal{N}(0, 0.01)$ are too small to allow any error-spikes to be sent back (the backward-pass quantizers never reach their firing thresholds). As a result, the network fails to learn. We found two ways to deal with this: “Back-Quantization = Random” initializes the $\vec{\phi}$ for the backwards quantizers randomly at the beginning of each round of training. “Back-Quantization = No-Reset” simply does not reset $\vec{\phi}$ in between training iterations. In both cases, the backwards pass quantizers always have some chance at sending a spike, and so the network is able to train. It is also interesting that using Fractional Updates (FSGD) gives us a slight advantage over regular SGD (Baseline). This is quite promising, because it means we have no need for multiplication in our network - As Section 3.7 explains, we simply add a column to the weight matrix every time an error spike arrives. We also observe that using the rectified running sum of the pre-quantization vector from the previous layer as our input to the weight-update module (Smooth Gradients) gives us a slight advantage. This is expected, because it is simply a less noisy version of the count of the input spikes that we would use otherwise.

F Event Routing

Since each event can result in a variable number of downstream events, we have to think about the order in which we want to process these events. There are two issues:

1. In situations where one event is sent to multiple modules, we need to ensure that it is being sent to its downstream modules in the right order. In the case of the SMLP, we need to ensure that, for a given input, its child-events reach the filters in the backward pass before its other child-events make their way around and do the backward pass. Otherwise we are not implementing backpropagation correctly.
2. In situations where one event results in multiple child-events, we need to decide in which order to process these child events and their child events. For this, there are two routing schemes that we can use: Breadth-first and depth-first. We will outline those with the example shown in Figure 6. Here we have a module A that responds to some input event by generating two events: a_1 and a_2. Event a_1 is sent to module B and triggers events b_1 and

b_2. Event a_2 is sent and triggers event b_3. Table 4 shows how a breadth-first vs depth-first router will handle these events.

Breadth-First	Depth-First
$B(a_1)$	$B(a_1)$
$B(a_2)$	$D(b_1)$
$C(a_1)$	$D(b_2)$
$C(a_2)$	$C(a_1)$
$D(b_1)$	$B(a_2)$
$D(b_2)$	$D(b_3)$
$D(b_3)$	$C(a_2)$

Table 4: Depth-First and Breadth-First routing differ in their order of event processing. This table shows the order of event processed in each scheme. Refer to 6.

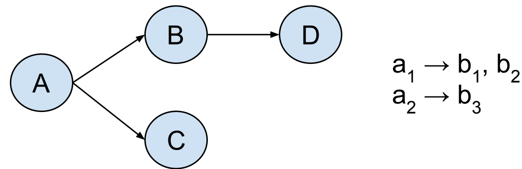

Figure 6: A simple graph showing 4 modules. Module A generates an event a_1 that causes two events: b_1 and b_2. These are then distributed to downstream modules. The order in which events are processed depends on the routing scheme.

Experimentally, we found that Breadth-First routing performed better on our MNIST task, but we should keep an open mind on both methods until we understand why.