

Towards Realtime Stance Classification by Spiking Neural Network

Vaenthan Thiruvarduchelvan
Centre for Research in Complex Systems
Charles Sturt University
Bathurst, Australia
Email: vthiru@csu.edu.au

Terry Bossomaier
Centre for Research in Complex Systems
Charles Sturt University
Bathurst, Australia
Email: tbossomaier@csu.edu.au

Abstract—Spiking neural networks are a popular area of current research in both artificial intelligence and neuroscience. Unlike second generation networks like the multilayer perceptron (MLP), they simulate rather than emulate neuronal interactions. Spiking networks have been shown to be theoretically more powerful than earlier generation networks, and have repeatedly been suggested as ideal for realtime problems due to their time-basis. Because of their sparse nature, real neural networks are also extremely power-efficient, a pressing concern in computing today. This raises the possibility of applying sparse spiking networks for power-saving. To investigate these ideas, we wish to apply a spiking network to realtime data classification. As a first step, we use a feedforward network with the SpikeProp algorithm to classify offline skeleton data derived from a depth camera. Classifier networks were successfully trained, but we found SpikeProp considerably more complex to apply than backpropagation. There is considerable potential for optimization and power efficiency, and we hope to compare the performance of our system with more established learning techniques in a realtime setting.

I. INTRODUCTION

A. Neural Computation Efficiency

Within a few decades we will have the power of the human brain on the desktop, variously estimated at around 1 petaflop, with around 1 petabyte of memory. One respect, though, in which there is no immediate prospect of computers matching human performance is in energy efficiency. The human brain requires a minuscule 20W of power to achieve its staggering performance.

Laughlin et al. [1] showed the thermodynamic cost of biological computation to be 10^5 – 10^8 times the thermodynamic limit of $1kT$, where k is Boltzmann's constant and T the absolute temperature. In comparison, the contemporary PlayStation 3 has a power usage on the order of $10^{22}kT$ per second, and a maximum computational throughput of around 200 GFlops. Here, computation costs are around 10^{11} times the theoretical minimum, 3–6 orders of magnitude worse than the cellular case.

On the scale of an animal's energy requirements, neural computation is costly with about 20% of human energy requirements taken by the brain [1]. The brain has various adaptations to reduce its energy cost. One is its overarching network structure with some small-world or scale-free character. Another is the use of *sparse codes* where the number of

neural impulses (spikes) is kept to a minimum, with an average activity level of 1–16% balancing the costs of computation versus quiescence [2].

B. Power Aware Computing

Power management has become a significant issue in almost all forms of computing today. From the battery life of mobile devices to the energy consumption of office buildings, datacenters and their cooling systems. The power usage of servers and data centers at the scale of organizations like Google now forms a significant portion of operating costs, and innovative means are constantly sought to mitigate this [3]. Power constraints essentially imposed the current roadblock in clock rates [4], and continue to threaten the continuation of Moore's Law in other ways.

Much research has been devoted to reducing hardware power demands. Most modern general-purpose processors sport adaptive active cooling systems which monitor their temperatures to ensure that their thermal design power (TDP) is not exceeded. They may enter low-activity states or throttle back the clock rate, to reduce power output [5]. Modern multi-core processors exploit load imbalances to boost single-threaded performance. These features are now ubiquitous in consumer devices including smartphones, doing their best to conserve energy. Current research takes several directions, including bespoke and reconfigurable embedded architectures [6], [7].

At the software end of the spectrum, operating system designers have exploited these hardware features to make an impact on energy efficiency. Contemporary OS dynamically adjust clock rates and even wireless power output in response to system activity [5]. *Applications* though, have in general been power-agnostic. Since compute-intensive AI applications are becoming more commonplace by the day—especially in the burgeoning mobile-computing market—we are interested in seeing whether the energy-efficiency of the brain can be replicated in software.

C. Artificial Neural Networks

The neurons of the mammalian cortex communicate via voltage spikes. For a long time, information was assumed to travel in the *firing rate*, i.e. the average number of spikes

per second a neuron transmitted – the rate code. There were studies, notably by Opticon and Richmond in the 80s which showed much more information was available if more details of the spike distribution were analyzed.

But if this pseudo-random train of spikes is to be useful beyond the rate code, then there has to be a way of decoding it. Bialek et al [8] showed that this could be surprisingly simple. In the insect neuron it transpires that a simple linear convolution of the spike-train as a series of Dirac delta functions suffices to reproduce a continuous time-varying input, to within the maximum information available given noise.

Rieke and others built on these foundations to determine the maximum information which could be transmitted. The noise limiting the information here is the spike time variability – the error in timing of the emission of a spike.

Although artificial neural networks took off in a big way in the mid 80s with the back-propagation algorithm [9], [10], the implicit assumption in most of this work was that of a rate code. Major interest in spiking networks was delayed by several decades.

Part of the reason for this, is that spiking networks are much more computationally expensive to implement and there was no clear theory demonstrating that their computational capacity was greater in some way. Several factors have caused a resurgence in activity in spiking networks:

- 1) theoretical work by Maass [11] and others shows that in some conditions, spiking neural networks *do* have a higher computational capacity,
- 2) the computational resources available to simulate and explore different models are much greater, and
- 3) good approximations to the Hodgkin-Huxley equations [12] were developed which allow very large networks to be simulated.

Although the neural code may be read close to the limits of its information capacity by linear convolution, artificial spiking neural networks have focussed on the timing of individual spikes. Thus, static classification tasks (which could be solved by rate-coded traditional feed-forward neural networks), can be solved using such outputs. Now, instead of the firing of a particular output neuron indicating an input class, the different classes can be represented by spike times.

D. Spiking Neural Networks

Biological neurons have a wide range of behaviors. Spikes may occur in all sorts of distributions, from Poisson point processes to rhythmical activity and bursting. Such variety, comprising at least 20 types of behavior [13], shows just how unrepresentative rate codes really are.

Hodgkin and Huxley's [12] definitive model covers all these behaviors but is too computationally demanding for many applications, and thus a variety of simplifications have arisen. The simplest use the Leaky Integrate-and-Fire (LIF) model, where synaptic potentials accumulate at the soma until a firing threshold is reached:

$$\frac{1}{\tau} \frac{du}{dt} = -u + \alpha I \quad (1)$$

where τ is the membrane potential decay constant (the leakage), u the membrane potential, α a constant and I the input current (a residual basal current and the sum of all the synaptic potentials). Izhikevich [13] argues that this model reproduces very few of the behaviors of real neurons and should almost never be used.

The LIF model does not allow for two neural properties which may be of importance in artificial domains – *spike latency* and *activity dependent thresholds*. Once a neuron reaches threshold, there may be a delay, the spike latency, before the spike is actually fired. The importance of this lies in that the further inputs may bring the threshold back down, effectively cancelling the spike. The state of the neuron may also influence the voltage generated by incoming spikes, making the threshold variable. The simplest example of this is the refractory period which most neurons go through immediately after firing. These deficiencies are remedied in the Quadratic Integrate-and-Fire model, in which the dependence on the membrane potential is now quadratic:

$$\frac{1}{\tau} \frac{du}{dt} = -u^2 + \alpha I \quad (2)$$

The models so far are all expressed as differential equations in terms of currents. Gerstner's *Spike Response Model* (SRM) [14] takes a different approach, expressing the neuron's behavior in terms of response kernels. The canonical model can be written as:

$$u_j(t) = \eta^*(t) + \sum_i \left(w_{ij} \sum_f \varepsilon^*(t - t_i^f) \right) + \int_0^\infty \kappa^*(s) I(t-s) ds \quad (3)$$

Where u_j is the state (membrane) variable of neuron j with presynaptic neurons i . The kernel $\eta(t)$ describes the shape of the action potential including aftershoot, f are spike times, kernel $\varepsilon(t)$ represents the effect of a postsynaptic potential on the neuron, kernel $\kappa(t)$ models the response of the membrane potential to an input current pulse, and $I(t)$ is the input current function. The asterisks indicate that the functions can be dependent on the time of the neuron's last spike, i.e. $f^*(t) = f(t - \hat{t}_j)$.

In practice, simplified versions of the canonical model suffice for various purposes, for example by dropping dependences on last spike time. The SpikeProp algorithm we use is even simpler – external currents and action potential specifics are not used. It only calculates the dynamics of neurons in response to incoming spikes up to the point where the first spike is triggered. The spike time is noted and subsequent neuron dynamics (and hence spikes) are discarded.

E. SpikeProp

SpikeProp, presented by Bohte et al. in 2002 [15] is a gradient-descent method of learning on feedforward spiking networks, analogous to the backpropagation (BP) rule for the multilayer perceptron (MLP) [10]. Networks are arranged in the familiar layer-fashion, with input, hidden and output layers (generalizing to more layers as with BP). As presented, it

mimics somewhat the rate-coding implicit in the MLP, using the timings of spikes released by simplified SRM neurons.

A brief description of SpikeProp is provided here, but we direct the reader to the original and other papers for a more thorough treatment [15], [16]. The operation of a SpikeProp network is generally as a conventional SRM SNN simulation, where input neurons are made to spike with a delay according to the value they represent. Neurons are connected to subsequent layers by connections as with MLP, except that each link is composed of ‘subconnections’/‘synaptic terminals’/‘synapses’ with a range of delays and variable weights. The basic XOR-example [15] used 16 stepped delays from 1 to 16 ms, effectively 16 times as many connections as BP. *Neurons may fire only once*, the timings of which SpikeProp is concerned with.

When a neuron spikes, connected target neurons receive a decaying post-synaptic potential (PSP) from each interposed synapse, with the appropriate delay. The PSP is modelled as an alpha-function with $\varepsilon(t) = 0$ for $t < 0$:

$$\varepsilon(t) = \frac{1}{\tau} e^{1-t/\tau} \quad (4)$$

where τ is the (constant) synaptic time constant¹. The weighted contribution of a single synapse to the target neuron’s state variable $x(t)$ is:

$$y_i^k(t) = w_{ij}^k \varepsilon(t - t_i - d^k) \quad (5)$$

where k synapses connect individual neurons, t_i is the spike time of neuron i , and d^k is the delay of the k ’th synapse.

The decaying incoming PSPs are summed at the target neuron every timestep, and compared to a fixed threshold. When it is exceeded, the neuron fires a spike and so on. As the spike time of a neuron is related to the times of incoming spikes via the differentiable PSP function, Bohte was able to derive a backpropagation rule. First, the output deltas are calculated, using the mean-squared-error over all outputs (MSE), E :

$$\delta_j = \frac{\partial E}{\partial t_j^a} \frac{\partial t_j^a}{\partial x_j(t_j^a)} = \frac{(t_j^d - t_j^a)}{\sum_{i \in \Gamma_j} \sum_l w_{ij}^l (\partial y_i^l(t_j^a) / \partial t)} \quad (6)$$

where the superscripts a and d refer to *actual* and *desired* and Γ_j denotes the set of input neurons to neuron j . Next, earlier layer deltas are calculated:

$$\delta_i = \frac{\sum_{j \in \Gamma_i} \delta_j \sum_k \{w_{ij}^k (\partial y_i^k(t_j^a) / \partial t)\}}{\sum_{h \in \Gamma_i} \sum_l w_{hi}^l (\partial y_h^l(t_i^a) / \partial t)} \quad (7)$$

where Γ_i refers to the set of neurons which i outputs to. Finally, all weights are updated using:

$$\Delta w_{ij}^k = -\eta y_i^k(t_j) \delta_j \quad (8)$$

¹It should be noted that the derivative is discontinuous at $t = 0$. The derivative we used had $t = 0$ for $t \leq 0$ as per Moore [16], though this was not specified in the original paper. It is possible that this is the cause of some of the behavior observed in Section IV-A, and a better function could be used.

where η is the learning rate.

Although ten years have passed since it was proposed, SpikeProp is not as widely known or studied as BP. Part of this must lie with the difficulty of implementing and using it as described in the Discussion. Furthermore, the field of machine-learning with spiking networks is in relative infancy, and SpikeProp is only one of many algorithms being studied. We selected SpikeProp due to its analogy to BP, and the seemingly thoroughness of its specification.

F. Spiking Network Optimization

Conventional SNNs evaluate the state of every neuron at synchronous timesteps, commonly 0.1ms or less. These are consequently extremely compute-intensive, and ignore the sparseness of neural activity. The IBM brain simulator, for example, runs 80 and 650 times slower than realtime for models of 900 million and 1.6 billion neuron networks respectively [17].

Several groups have developed high-performance hardware simulators of varying designs [18], [19], while acceleration using GPU vector processing is a hot topic of recent [20], [21]. Most research is still conducted in software however. Some of the larger simulations mentioned require large teraflop-scale supercomputers, rapidly approaching the petaflop barrier.

To minimize the computational overhead, event-driven simulations (EDS) have been mooted by several groups beginning in 1994 [22], [23], [24], [25]. This technique reduces load by computing only those neurons which are active or may be active in subsequent timesteps. This paradigm has more recently been extended to distributed computing, which introduces the problem of synchronization [26]. Another optimization method is the lookup-table proposed by Ros et al [27], which can also improve realism.

All of these techniques have been successful in reducing computational overheads, shrinking the realtime-gap and making larger simulation sizes tractable. We will apply some of these techniques to our system in forthcoming work.

G. Depth Imaging and Gesture Detection

Depth imaging is a relatively new technology which has really burst into the mass-market and research communities with the release of the Microsoft Kinect in November 2010. It refers to imaging technology which returns frames with pixel values corresponding to the distances to objects in front of the camera. There are several underlying technologies used to deliver this data, with the Kinect using structured light. Nevertheless, depth imaging has been around before the Kinect, at a smaller scale. We use a time-of-flight camera, the SwissRanger SR-4000² which has seen some use in robotics research. Table I compares the two devices’ specifications.

An additional piece of technology which was popularized around the release of the Kinect, is a middleware SDK for extracting skeleton data from the raw depth frames. Skeleton data refers to spatial coordinates of body parts and other

²<http://www.mesa-imaging.ch/prodview4k.php>

Property	SR4000	Kinect (depth)
Technology	Time-of-Flight	Structured Light
Frame resolution	176 x 144	320 x 240
Frame rate	30 fps (50 max)	30 fps
Range	0.8m - 10m	1.2m - 3.5m
Depth accuracy	10mm	10mm
Depth precision	14-bit (16,384)	16-bit (65,536)
Field of view	44° x 35° / 69° x 56°	57° x 43°

TABLE I
COMPARISON OF SR4000 AND KINECT SPECIFICATIONS.

anchors localized on the body, for example the head, left elbow, right knee and so on. We are aware of several alternatives – OpenNI, Microsoft Kinect SDK, SoftKinetic iisu and GestureTek. Ultimately, we selected the iisu SDK³ as it supports our model of camera. The SDK is able to process frames at 30fps and return an array of skeleton and other data on a frame-by-frame basis [28].

The Kinect was initially released to provide gesture-based control of video games in the loungeroom, however researchers and enthusiasts have been using depth imaging for a myriad of different uses. It seems likely from ongoing research, industry movements and even a competition funded by Microsoft⁴ that ‘Minority Report’-style natural-interface interactions will become commonplace in future. Coupled with the need for realtime machine-learning applications and power-conscious software, efficient realtime processing of skeleton streams is a current problem.

II. METHOD

A. Model Description

We made use of an object-oriented programming paradigm, with objects representing *Simulation*, *Neuron*, *Synapse* and *PSP*. For a link between two neurons, multiple *Synapses* with differing delays were created. This simplified the summations in the equations. When *Neurons* fire, the outgoing *Synapses* enqueue *PSPs* with appropriate delays and weights to the outgoing *Neurons*. At each step, each *Neuron* calculates its state variable based on the queue of decaying *PSPs* and checks if its super-threshold. Once this has occurred, the *Neuron* no longer needs to be evaluated. The simulation is stopped when the maximum number of iterations is reached, or all neurons have fired first.

Although it could be slightly slower than a matrix-based approach as used by Moore [16], this model architecture was used because it facilitates the direction we wish to take for future work: event-driven simulation.

B. Implementation

We implemented the spiking network simulator as an multi-threaded interactive GUI application with C++ using the

Qt library. The simulation itself remains single-threaded at this stage. There were several reasons behind these design decisions. Due to the taxing computational demands of spiking networks, we need a highly optimized system which can meet realtime demands. A bespoke C++ system lends itself to various optimizations, including event-driven simulation. The GUI interface allows direct, realtime inspection of the network’s operation which is useful for detecting unintended or unpredictable behavior. This feature allowed us to quickly recognize programming errors, and several behaviors not disclosed by earlier implementers [15], [16]. This approach did however come with some challenging concurrency issues to negotiate.

Special attention was not paid to optimizations at this stage, which will be the subject of future work. Besides language-specific optimizations (beyond the scope of this paper) the only optimization was to bypass neurons that had already fired.

C. Data Source

A previous software project by V.T. [29] was developed to recognize the stances of a performer on stage using the SR-4000, and manipulate stage lighting. The system took the skeleton data from the iisu SDK and applied a rule-based pattern recognizer to detect the presence of a particular stance. Eight stances were specified for that proof-of-concept, including standing straight, arms-outstretched and arms-akimbo as illustrated in Figure 1. The application created, *tofStage*, made use of the scripting language Lua embedded in the SDK [28], and was able successfully to classify the stances in realtime at 30fps.

Of the array of skeleton data available from the SDK, the system used the following nine datapoints: *position*, *mass_center*, *head*, *left_hand*, *right_hand*, *pelvis*, *sternum*, *left_foot*, *right_foot*. As we were using the free licence of the SDK, elbow, knee and other data was unavailable. Note that the SDK performs the required transforms on camera data to return linear (x,y,z) coordinates in space. For the proof-of-concept software, we limited ourselves to front-on stances, allowing us to ignore the Z-dimension (distance from camera) data.

This method had several drawbacks though:

- 1) The rules (e.g. `leftHand.x - massCenter.x > 0.5`) are very brittle, and require hand-tweaking for the particular performer and environment.
- 2) Writing rules to detect many more poses and complicated poses quickly becomes impractical, and reliable detection can be expected to diminish.
- 3) The performance of a rule-based system scales in terms of rule count. System performance can be expected to degrade as stances are added.
- 4) Programming and testing new rules or adjusting for different performers takes considerable developer time.

Therefore, the problem was amenable to a machine learning technique, and ideal for our investigations here. Since this earlier system was able to classify stances, it provided us with a labelled dataset for supervised learning.

³<http://www.softkinetic.com/solutions/iisusdk.aspx>

⁴<http://www.kaggle.com/c/GestureChallenge>

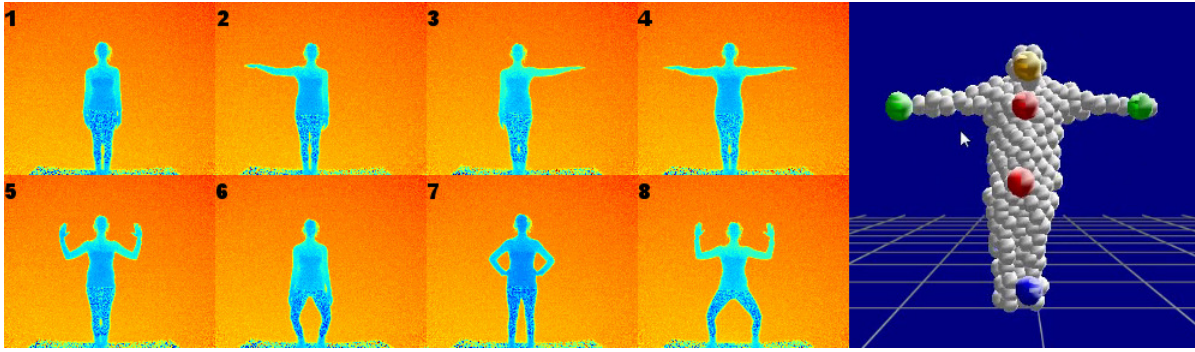


Fig. 1. Left: SR4000 images of the eight stances detected by *tofStage*, which provided the labelled training data. Right: A frame processed by the iisu SDK, showing some of the body datums used.

D. Experiments

Our first goal was to establish that our simulator performed correctly, and could reproduce the learning behavior of SpikeProp on the basic XOR problem. Source code was not provided by Bohte, and that provided by Moore [16] was considerably different in architecture to ours, and would require considerable refactoring to generalize.

The main experiment was to see if the system could learn a real-valued function performing stance detection. From an input array of coordinates from the skeleton SDK, we wanted the system to output a single real value representing the stance. First we used the simplest input/output decoding with a minimal network size of 18 inputs, 9 hidden neurons and a single output neuron; no reference neuron was used⁵. In half of our experiments, one of the nine hidden neurons was switched from excitatory to inhibitory (PSP contribution inverted).

We kept most of the parameters used for this experiment comparable to those used by Moore for the XOR problem, as shown in Table II. The step size was kept at 0.1ms to make the simulations run with adequate speed. Earlier results showed that output granularity is on the order of this value, which was adequate.

E. Input Encoding/Output Decoding

The inputs to the network are a series of spike times, which are quantized in the simulation step size. As with Bohte, we aligned these on millisecond-boundaries. As we used similar parameters, the input spike times were mapped from 0ms to 8ms, i.e. for each of the 18 input neurons, there were 9 possible input spike times.

The *tofStage* application (Section II-C) was modified to print out for each frame, a length-28 vector of each of the (x,y,z) values of each the nine points of interest, followed by the corresponding stance value. A recorded video of the author performing each stance in sequence for two runs was processed to give a total of 1143 vectors (the *full* dataset), of which a stance was detected in 511 (the *sans0* dataset). The remaining 632 vectors in effect contain “negative” examples,

⁵This is an extra input neuron added by Bohte to his XOR network, which always fired at $t=0$.

Parameter	Value	Description
network topology	18,9,1	†No. input,hidden,output neurons
random seed	3	†Weight initialization seed
η	1	†Learning rate
τ	7.0	Spike response time-constant
Vthresh	50	Spike threshold
ΔT	8	Input spike time range
reference neuron	no	Additional input with spike time 0
inhibitory neurons	0, 1	No. of inhibitory hidden neurons
subconnections	16	Synaptic terminals per connection
weight_init_method	2	Weight init. method [16]
negative weights	yes	Permit negative weights
sim_max_steps	5000	Max. steps per simulation
step duration	0.1	Simulation step size (ms)
step interval	0	Delay time between steps

TABLE II
LISTING OF PARAMETERS USED FOR SPIKEPROP TRAINING.
†PARAMETERS IN COMMON WITH MLP BACKPROP.

i.e. poses not matching the eight sought. From each, a random sample of 50 records was chosen for training (containing examples of all stances), as well as a ‘minimal’ sample containing one example of each stance. The idea with the latter was to see how well the network could learn from one training example acting like a ‘support vector’.

For each vector, the Z-values were discarded, as we are limiting ourselves to front-on 2D classification here. For each of the 18 (x,y) real values in the resulting length-19 vector, the values were normalized across all records to a value between [0..1]. These real values were then converted to spike-time integers in the range [0..8] by multiplying by 8 and selecting the closest integer.

The 19th value of the vector, the output stance (an integer stance value between [0..8]) was added to 10 to bring it in the range of [10..18], similar to the original XOR problem. Output decoding involved selecting the closest integer in that range to the network’s real-valued output.

We are also currently simulating a network utilizing ‘receptive fields’ as described by Bohte, with eight separate neurons

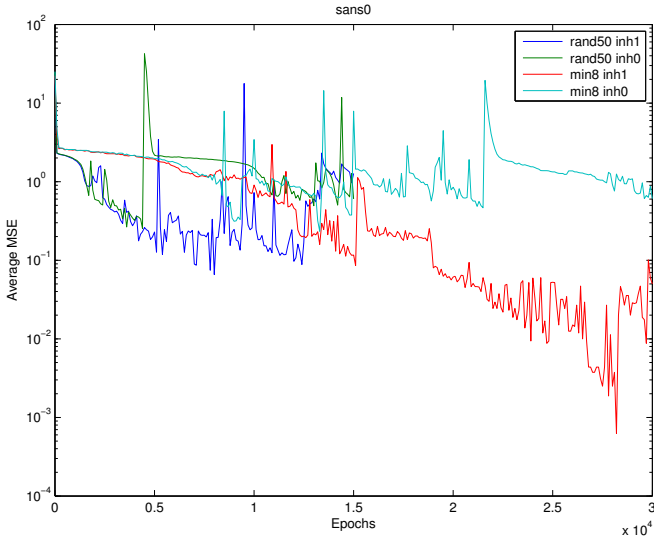


Fig. 2. Learning progress of networks using the `sans0` dataset.

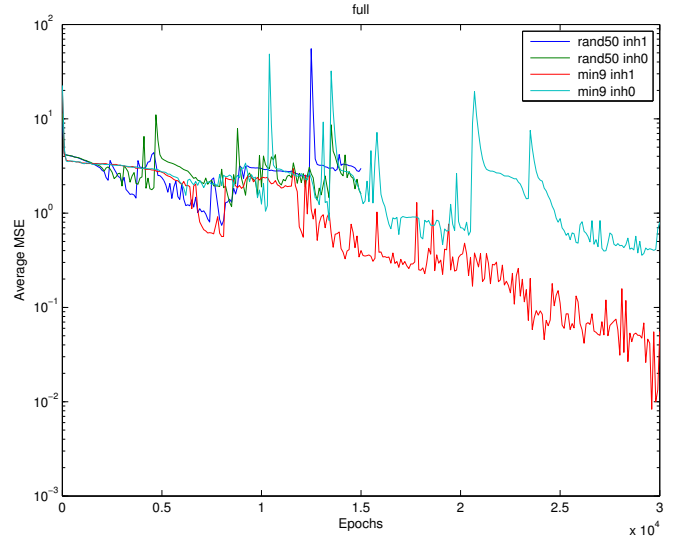


Fig. 3. Learning progress of networks using the `full` dataset.

for each input, totalling 144 input neurons. This network has 217 neurons in total and takes considerably longer to train, as such completed results were not available at the time of writing. Following their completion, we intend on simulating a further network with ‘classification’ neurons, i.e. one for each stance detectable.

III. RESULTS

SpikeProp was run on the training data for 1500 epochs on the random-50 sets, and 3000 epochs for the minimal sets as they ran considerably faster. The average mean-squared-error (MSE) over all records in the training set was calculated every 10 epochs to monitor the training process. At the end of the maximum number of epochs, the network was validated against the entire dataset (`full` or `sans0` respectively) to obtain the average MSE. From these runs, the best MSE figures were also extracted (Table III).

Figures 2 and 3 show the gradual learning process, but also surges in error which usually meant that the final MSE over the entire set is not usually the best. The networks were clearly able to learn, with those having an inhibitory neuron performing better, and minimal datasets being learned to greater accuracy. The FullSet MSE figures and inspection of the prediction results indicated that the network did not generalize sufficiently over the full dataset to be used for reliable stance detection at this stage.

An additional performance metric needs to be devised to better capture performance after the real-valued outputs are discretized to the closest integer. The average MSE cost function does not adequately reflect how fit for purpose the networks currently are. Additional tuning of this, and network parameters, should allow us to achieve adequate performance.

IV. DISCUSSION

Our first attempt at adapting SpikeProp to this realtime problem gave mainly positive results. We were able to replicate

SpikeProp using an object-oriented approach amenable to event-driven simulation. The present network was able to learn reliably, but validation results show that it is not yet to the standard required for accurate stance detection. Because of the way it is used—by discretizing the output to the nearest integer—a better cost function than MSE could be used to improve performance.

By looking at the best MSE figures, which ignore the ‘surge’ behavior, we see that having an inhibitory neuron generally improves performance. Why this should be so when negative weights are allowed is most probably due to the weight initialization method; at present only positive weights are used. Therefore, increasing the proportion of inhibitory connections—as found in the cortex—may yield improvements.

We believe that both better classification performance and better power conservation (spike minimization) can be achieved by using receptive fields as per Bohte [15] and having multiple output neurons. In this case, the timing resolution of spikes is less critical and less likely to be influenced by input noise. Additionally, shunting inhibition and Bayesian-prior techniques can be used to further minimize spiking. This work lays the foundation for these upcoming experiments.

It became evident during this work that SpikeProp was considerably harder to implement and use compared with the MLP. It has many more tunable parameters (Table II) which need to be adjusted for application to any particular problem. If using an MLP requires careful network dimensioning and close parameter tuning, SpikeProp compounds the problem. However, the fewer training epochs required and potential optimizations mitigate these issues. We are still in the process of figuring out a framework for most effectively applying SpikeProp, and some questions remain.

#	TrainSet	Network	Epochs	Best MSE*	Final MSE†	FullSet MSE‡
1	sans0_18_rand50	18,9,1 inh=1	1500	0.0658	1.271	1.313
2	sans0_18_rand50	18,9,1 inh=0	1500	0.254	0.602	0.730
3	sans0_18_minimal8	18,9,1 inh=1	3000	0.001	0.050	3.496
4	sans0_18_minimal8	18,9,1 inh=0	3000	0.233	0.607	2.298
5	full_18_rand50	18,9,1 inh=1	1500	0.742	2.986	3.134
6	full_18_rand50	18,9,1 inh=0	1500	1.170	1.594	3.018
7	full_18_minimal9	18,9,1 inh=1	3000	0.008	0.056	13.020
8	full_18_minimal9	18,9,1 inh=0	3000	0.357	0.800	10.609

TABLE III

TRAINING PERFORMANCE. †AVERAGE MSE OVER TRAINING DATA. ‡AVERAGE MSE AT FINAL EPOCH, OVER ENTIRE FULL OR SANS0 DATASET.

A. Observed Behaviors

We observed the following behaviors with SpikeProp in our experiments:

- The output results are on the order of the stepsize, as originally pointed out by Bohte. We used a timestep of 0.1ms to keep training times manageable.
- ‘Surge’ or instability behavior as examined by [30]. SpikeProp generally converges, while occasionally diverging greatly.
- At the outset, less than half the weights are altered during each iteration, but this number gradually increases with the number of epochs. It is possible that this is the cause of the preceding observation – as a neuron’s spike gets delayed, the contribution of previously-ignored “late” incoming synapses begin to contribute to that neuron.
- The simulator gradually slows down (epochs/s) seemingly exponentially with epochs. This is probably due to after-spike optimization and the previous point. As more synapses need to be computed per epoch, computation takes longer.
- The experiments with no inhibitory hidden neurons slowed down (epochs/s) much more quickly than ones with them.
- With some initial weight configurations, two failure modes were observed:
 - Neurons may fail to spike at all, a requirement for the SpikeProp algorithm.
 - The denominator of Equations 6 and 7 occasionally evaluated to zero. This can arise in the case of a local minimum, or as a result of the spike response function.

Neither occurrences were explicitly mentioned by others, but could fall under the umbrella of “did not converge”. Nor is either encountered in backpropagation, but appear regularly in long-running SpikeProp training sessions. It is possible that using a fully continuous and/or non-zero spike response function (Eqn. 4) could eliminate these.

- It is evident that the network performs acceptably without a reference neuron, so it is unclear why Bohte included one in his XOR experiments. All spike times can be considered relative to simulation start time.

The complexity of SpikeProp coupled with undocumented implementational details mean that different implementations may diverge slightly. Differences in random-number generator and the sensitivity to initial weights may mean that the exact results from Bohte and Moore [15], [16] may not be exactly reproducible. Further investigations will be required to explain and characterize these issues.

As highlighted by Moore [16], several things were not specified by Bohte in his original paper. Namely, the method for initializing weights and setting the (constant) spike response threshold. Apart from Moore’s experiments, others have proposed various methods for improving the learning performance e.g. RProp/QuickProp [31] and multidimensional learning [32], and for reducing stability problems [30], [33]. We need to investigate if these more recent findings will improve our system.

V. CONCLUSIONS

Towards our goal of employing a spiking neural network for realtime stance detection, we were able to implement SpikeProp successfully and train it to learn from offline recorded data. From the experiments conducted, we observed various interesting behaviors with SpikeProp, some of which may have been recently investigated, and others which will require further investigation. Using the input/output encoding method employed, we were able to train at least one network which was able to recognize the eight target stances regularly enough, which is a good baseline for further work. The SpikeProp algorithm was found to be an order of magnitude more complex in terms of implementing, training and tuning than the multilayer perceptron, highlighting the difficulties in applying spiking networks.

Spiking networks have been shown to be computationally more powerful than second-generation networks [11], and are touted as ideal for realtime problems. At the same time, their discrete, time-based nature offers various avenues for optimization. One of these is event-based computation wherein energy is mostly expended on active neurons only, as the highly-efficient brain does. Given that power requirements now dominate decisions at all levels of computing, and the pervasion of machine learning in today’s world, the application of efficient spiking networks to machine learning is appealing.

Future Work: The next steps of this project are to tune the parameters of SpikeProp, and try other means of input/output decoding, both to improve performance and reduce computing overheads.

It is evident that considerable optimization is possible with the simulator, which in addition to saving power will allow faster processing and longer testing. Among the possibilities are:

- pruning PSPs once below a certain threshold
- bypassing neuron evaluation where the number of PSPs enqueued can not possibly trigger a spike
- event-driven evaluation of neurons
- implementing the PSP kernel using a lookup-table

In light of the possibility of event-driven simulation, new network topologies and coding schemes which minimize spiking (for instance, suppressing spikes for missing inputs or detection failure) would be desirable.

Ultimately, we wish to interface a trained network to the realtime feed from the depth camera, allowing classification in realtime. We intend to see how its power performance compares to second-generation networks and other established techniques. In the process, we will expand the capabilities of our system to varying persons and activities.

VI. ACKNOWLEDGEMENTS

V.T. was supported by a Ph.D. scholarship from the Center for Research in Complex Systems (CRiCS) at Charles Sturt University.

REFERENCES

- [1] S. B. Laughlin, R. R. de Ruyter van Steveninck, and J. C. Anderson, "The metabolic cost of neural computation," *Nature Neuroscience*, vol. 1, no. 1, pp. 36–41, 1998.
- [2] S. B. Laughlin and T. Sejnowski, "Communication in neuronal networks," *Science*, vol. 301, pp. 1870–1874, Sep. 2003.
- [3] J. Carter and K. Rajamani, "Designing energy-efficient servers and data centers," *Computer*, vol. 43:7, pp. 76–78, July 2010.
- [4] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.
- [5] S. Albers, "Energy-efficient algorithms," *Communications of the ACM*, vol. 53, no. 05, pp. 86–96, May 2010.
- [6] D. Donofrio, L. Oliker, J. Shalf, M. F. Wehner, C. Rowen, J. Krueger, S. Kamil, and M. Mohiyuddin, "Energy-efficient computing for extreme-scale science," *IEEE Computer*, vol. 42, no. 11, pp. 62–71, Nov. 2009.
- [7] S. Swanson and M. Taylor, "Greendroid: Exploring the next evolution in smartphone application processors," *Communications Magazine, IEEE*, vol. 49, no. 4, pp. 112–119, april 2011.
- [8] W. Bialek, F. Rieke, R. de Ruyter van Steveninck, and D. Warland, "Reading a neural code," *Science*, vol. 252, no. 5014, pp. 1854–1857, 1991.
- [9] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. dissertation, Harvard University, Cambridge, MA, 1974.
- [10] D. Rumelhart and J. McClelland, *Parallel Distributed Processing (Two Volumes)*. MIT Press, 1986.
- [11] W. Maass and M. Schmitt, "On the complexity of learning for spiking neurons with temporal coding," *Information and Computation*, vol. 153, pp. 26–46, 1999.
- [12] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500–544, 1952.
- [13] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.
- [14] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
- [15] S. M. Bohte, J. N. Kok, and H. L. Poutre, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1–4, pp. 17–37, 2002.
- [16] S. C. Moore, "Backpropagation in spiking neural networks," Master's thesis, University of Bath, 2002.
- [17] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: Cortical simulations with 10^9 neurons, 10^{13} synapses," in *SC '09 Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.
- [18] K. Boahen, "Neuromorphic microchips," *Scientific American*, vol. 292, no. 5, pp. 56–63, May 2005.
- [19] X. Jin, M. Lujan, L. Plana, S. Davies, S. Temple, and S. Furber, "Modeling spiking neural networks on spinnaker," *Computing in Science Engineering*, vol. 12, no. 5, pp. 91–97, sept.-oct. 2010.
- [20] A. Fidjeland and M. Shanahan, "Accelerated simulation of spiking neural networks using GPUs," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, July 2010, pp. 1–8.
- [21] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, "GPU-based simulation of spiking neural networks with real-time performance and high accuracy," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, July 2010, pp. 1–8.
- [22] L. Watts, "Event-driven simulation of networks of spiking neurons," in *Advances in neural information processing systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan Kaufmann Publishers, 1994, ch. 6, pp. 927–934.
- [23] M. Mattia and P. D. Giudice, "Efficient event-driven simulation of large networks of spiking neurons and dynamical synapses," *Neural Computation*, vol. 12, no. 10, pp. 2305–2329, 2000.
- [24] A. Delorme and S. J. Thorpe, "Spikenet: An event-driven simulation package for modeling large networks of spiking neurons," *Network: Computation in Neural Systems*, vol. 14, no. 4, pp. 613–627, 2003.
- [25] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter, "Parallel event-driven neural network simulations using the Hodgkin-Huxley neuron model," *Parallel and Distributed Simulation, Workshop on*, vol. 0, pp. 16–25, 2005.
- [26] A. Mouraud, H. Paugam-Moisy, and D. Puzenat, "A distributed and multithreaded neural event driven simulation framework," in *Proc. of PDCN06, Parallel and Distributed Computing and Networks*, Feb. 2006.
- [27] E. Ros, R. Carrillo, E. M. Ortigosa, B. Barbour, and R. Agis, "Event-driven simulation scheme for spiking neural networks using lookup tables to characterize neuronal dynamics," *Neural Computation*, vol. 18, no. 12, pp. 2959–2993, Dec. 2006.
- [28] "iisu Developer Guide ver. 2.8," SoftKinetic SA/NV, Tech. Rep., Mar 2011.
- [29] J. Carroll and V. Thiruvarduchelvan, *Performance: Developing New Visual languages (Forthcoming)*. Rodopi, Amsterdam/New York, 2011, ch. Time of Flight: biodigital feedback and performance design.
- [30] H. Takase, M. Fujita, H. Kawanaka, S. Tsuruoka, H. Kita, and T. Hayashi, "Obstacle to training spikeprop networks — cause of surges in training process," in *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, June 2009, pp. 3062–3066.
- [31] S. McKennoch, D. Liu, and L. Bushnell, "Fast modifications of the spikeprop algorithm," in *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, 0-0 2006, pp. 3970–3977.
- [32] B. Schrauwen and J. Van Campenhout, "Extending spikeprop," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 1, July 2004, pp. 4 vol. (xlvii+3302).
- [33] W. Tshiki, T. Haruhiko, K. Hiroharu, and T. Shinji, "A training algorithm for spikeprop improving stability of learning process," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, 31 2011-Aug. 5 2011, pp. 951–955.