FPGA Implementation of Spiking Neural Network with On-Chip Local Learning for Efficient Brain Machine Interface

Elijah Taeckens

Departmental Undergraduate Honors Thesis in Electrical and Computer Engineering

Spring 2023

Table of Contents

Introduction

Each year, there are about 18,000 new cases of spinal cord injuries in the United States [1]. These injuries frequently lead to severe degradation of motor function. Recent advancements have allowed the development of Brain Machine Interfaces (BMI), which can help restore patient mobility by decoding neural signals and using them to control prosthetic devices [2], [3]. This technology can greatly improve quality of life by improving patient mobility and freedom.

Brain machine interfaces rely on implantable neural decoding circuits that process the signals from a patient's neurons. These neural decoders are traditionally made using Kalman filters or similar linear filters to decode neural signals [4], but recent research has shown that machine learning techniques can also be highly effective [5]. However, one of the main challenges with developing neural decoding circuits is limiting power consumption. In order to avoid damaging neural tissue, implantable decoders must dissipate less than 40 mW/cm2 [6].

This project addresses the power consumption problem by using a spiking neural network (SNN) as a neural decoder. SNNs are a category of neural network that take trains of events, or spikes, as inputs, rather than real numbers. SNNs can be modeled as connections of units called neurons, which integrate incoming signals and emit outgoing spikes in ways that closely model biological neurons [7]. Due to the sparse nature of their inputs, hardware implementations of SNNs have been shown to be extremely power efficient, consuming as little as 4 μW per neuron [8].

Previous work by myself and Dr. Shah tested the accuracy of a software model of a spiking neural network when used for a neural decoding task [9]. The SNN was trained on two different data sets, one containing neural data from the premotor cortex of a monkey [10], the other containing data from the hippocampus of a rat [11], and achieved roughly equal accuracy on both data sets when compared to a standard Kalman filter and a LSTM recurrent decoder.

In order to actually be used in a clinical setting as part of a BMI, a neural decoding algorithm must be implemented on hardware so that it can be implanted in a patient's brain. Additionally, the hardware implementation should be capable of on-chip learning, so that it can be trained post-implantation and be capable of adapting to changes in the patient's brain. There are numerous examples of FPGA implementations of spiking neural networks [12], [13], [14], [15], [16], several of which successfully implement on-chip learning. These works use a variety of different neuron models and learning rules, and several make use of approximations that simplify some of the complicated dynamics involved in neural simulations, some of which will be used in this work. However, these models are used for classification tasks, such as distinguishing between different types of images, making them unsuitable for neural decoding, which requires regression-based learning to match the output of the network with the desired kinematic data.

In this work, we demonstrate the first example of an FPGA-based SNN being used for neural decoding. The implementation takes advantage of the sparse nature of spiking data to minimize the number of computations, and utilizes iterative weight update rules to greatly reduce the total memory required for learning and enable continuous online learning.

Spiking Neural Network Design

A. Neuron Models

Spiking neural networks are very intuitive to apply to the problem of neural processing because they seek to mimic the behavior of biological neurons. While there are a variety of models used to simulate the behavior of a biological neuron in a SNN, they all operate under some basic principles. SNNs take as input a sequence of discrete events known as a spike train, defined formally as $S(t) = \Sigma_{k \epsilon C}\, \delta(t{-}t^{(k)})$, where C is the set of all events and $t^{(k)}$ represents the timing of event k in C. Neurons in each layer emit their own spikes, so information is passed between layers in the form of spike trains. For this project, I used the leaky integrate-and-fire neuron model, and based the design on the one shown in [17]. The LIF neuron maintains two internal values: the synaptic current, I(t), and the membrane potential, U(t). The synaptic current and membrane potential of the i-th neuron in a layer L in the network are governed by following:

$$\tau_{syn}\frac{dI_i^l}{dt} = -I_i^l(t) + \sum_j W_{ij} S_j^{l-1}(t) \qquad (1)$$

$$\tau_{mem}\frac{dU_i^l}{dt} = (-U_i^l(t) + I_i^l(t))(1 - S_i^l(t)) \qquad (2)$$

$$S_i^l(t) = \Theta(U_i^l(t) - \nu) \qquad (3)$$

where W represents the matrix of weights applied to the input spikes and τ_{syn} and τ_{mem} are time constants. The neuron emits a spike whenever the membrane potential, U(t), crosses a threshold value, at which point the membrane potential is reset to 0.

B. SNN Architecture

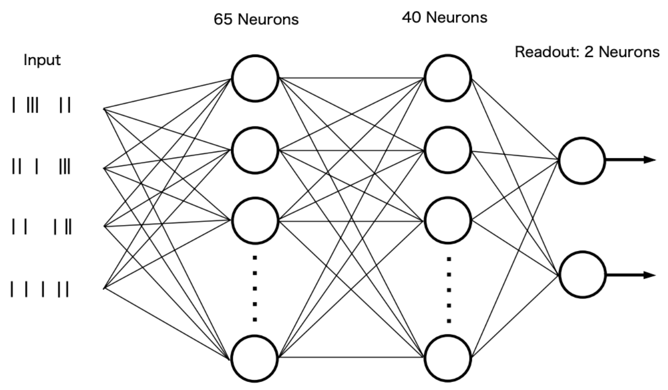

Fig 1. A graphical depiction of the spiking neural network architecture described below.

The SNN used in this study consists of an input layer, with size corresponding to the number of neural input units from each recording session, two hidden layers of 65 and 40 neurons, respectively, and a readout layer. The sizes of the hidden layers were optimized using Bayesian Optimization. The readout layer consists of 2 neurons that each output a continuous time signal, representing the predicted x and y components of the kinematic data.

C. Learning Rules

As previously mentioned, SNNs are typically used for classification tasks, in which the readout layer consists of several spiking neurons and the output is determined by selecting the neuron with the highest spiking rate [18], [15]. To perform neural decoding, the SNN must output two real valued continuous time functions corresponding to the X and Y components of the kinematic data. In [9], we explored multiple ways of modifying SNNs to perform regression. For this study, we will use the membrane potential method, in which the readout layer of the SNN consists of two LIF neurons that never emit spikes, and the output of each neuron is the membrane potential U(t). This works well as a readout function because U(t) is continuous and differentiable in time, just like real-world kinematic data. To train the model, we use surrogate gradient descent, a modified version of gradient descent in which the heaviside function, $\Theta(U_i^l(t))$, is replaced with a continuous, differentiable function $\sigma(U_i^l(t))$ to make it possible to compute a gradient over a sequence of spikes [17]. Additionally, the weights are updated using local learning, in which a readout layer is applied to each hidden layer in the network. The weights in each layer are updated according to the loss from that layer's readout, which eliminates the need for backpropagation through the network [19]. This removes the memory overhead required for backpropagation through a multi-layer neural network, and is considered more biologically plausible, since it does not require neurons to have knowledge about the spiking history of other neurons [19].

In [9], we demonstrate that a SNN trained with local learning can achieve the same results as a backpropagation-trained SNN for a neural decoding task. However, due to the recursive nature of the readout function, the software library used in that model required access to the full history of spikes generated by each hidden neuron to properly calculate the weight updates, making the total computational complexity $O(NT)$. This is extremely inefficient to implement on hardware, as the number of time-steps can often be very large in BMI applications. Additionally, it is not possible to implement such a system for online learning, in which the algorithm is trained in real time as the patient uses their BMI.

A more efficient weight update algorithm can be designed based on deep continuous local learning (Decolle) [19]. The general formula for determining updating weights via local learning is given by

$$\Delta W_{ij} = -\eta \frac{\partial L^l}{\partial W_{ij}^l} \tag{4}$$

where η is the learning rate and L^l is the loss function for the readout of layer l. When using mean squared error loss, this term becomes

$$\frac{\partial L^l}{\partial W_{ij}^l} = \sum_k (Y_k^l - \hat{Y}_k) \frac{\partial Y_k^l}{\partial W_{ij}^l} \tag{5}$$

where Y_k^l is the kth readout value and $\hat{Y}_k$ is the corresponding target value. In [19], the readout is just a weighted sum of the spiking output of the hidden layer neurons, so $\frac{\partial Y_k^l}{\partial W_{ij}^l}$ can be expressed as a constant times the derivative of the spiking output, $S_i^l(t)$, with respect to the weights. As mentioned previously, the derivative of the spiking function is not well defined, so we replace it with $\sigma'(U_i^l(t) - 1)$, the derivative of the sigmoid function, to perform surrogate gradient descent. The derivative of the spiking output can then be expressed as:

$$\frac{\partial S_i^l(t)}{\partial W_{ij}^l} = \sigma'(U_i^l(t) - 1) \frac{\partial U_i^l(t)}{\partial W_{ij}^l} \tag{6}$$

The derivative of $U_i^l(t)$ can be difficult to calculate due to its recurrent nature, so the authors of [19] use linearity to rewrite the equations for U(t) and I(t) as:

$$\begin{gathered}\tau_{syn} \frac{dQ_j^l}{dt} = -Q_j^l(t) + S_j^l(t) \\ \tau_{mem} \frac{dP_j^l}{dt} = -P_j^l(t) + Q_j^l(t) \\ U_i^l(t) = \textstyle\sum_j W_{ij}^l P_j^{l-1}\end{gathered} \tag{7}$$

Using this re-arrangement of terms, the derivative of $U_i^l(t)$ with respect to the weights is simply $P_j^{l-1}(t)$. However, unlike in [19], the equation for Y_k^l in this model is not a linear function of $S_i^l(t)$, since Y_k^l, like U_i^l, depends upon previous inputs. The linearity equivalence used to find the derivative of U_i^l cannot be used in exactly the same way here, because we are taking the derivative with respect to the spikes, not the weights, and unlike the weights the spikes are being integrated. However, it can be shown that

$$\frac{\partial Y_k^l}{\partial W_{ij}^l} = R_{ki}^l G_{ij}^l \tag{8}$$

where

$$\begin{aligned} \tau_{syn}\frac{dH_{ij}^l}{dt} &= -H_{ij}^l(t) + \frac{\partial S_i^l(t)}{\partial W_{ij}^l} \\ \tau_{mem}\frac{dG_{ij}^l}{dt} &= -G_{ij}^l(t) + H_{ij}^l(t) \end{aligned} \tag{9}$$

with the only difference between equations (7) and (9) being that H_{ij}^l integrates the derivative of the output spikes, rather than the spikes themselves. Equation (8) is derived from the fact that the dynamics governing the synaptic current and membrane potential for a spiking neuron can be equivalently expressed as the convolution of with a kernel function, K(t) [14].

$$Y_k^l(t) = R_{ki}^l \sum_{\tau} S_i^l(\tau) K(t-\tau) \tag{10}$$

$$K(t) = e^{-t/\tau_{mem}} - e^{-t/\tau_{syn}} \tag{11}$$

Using the property of derivatives of convolutions, we can state that:

$$\frac{\partial}{\partial W_{ij}}(S_i^l(t) * K(t)) = \frac{\partial S_i^l(t)}{\partial W_{ij}} * K(t) \tag{12}$$

and substituting the derivative of $S_i^l(t)$ into equation (5) yields equation (8). Combining all equations yields the full weight update equation:

$$\Delta W_{ij} = -\eta G_{ij}^l(t) \sum_k (Y_k^l - \hat{Y}_k) R_{ki} \tag{13}$$

This weight update equation captures accounts for all previous states of both the hidden layer neuron and the readout neurons without requiring any memory overhead to store the historical states of these neurons. It does require supplementary functions in the form of G_{ij}^l and H_{ij}^l, but these only increase the memory complexity to $O(N^2)$. In BMI applications, N is typically significantly less than T. Additionally, because the memory does not have to store past states, this system is capable of online continuous learning.

FPGA Implementation

A. Event-Based Logic

When working with spiking data, it is often more efficient to use event-based logic, rather than synchronous, clock-based logic. In synchronous logic, all computations are synchronized using a global clock, and each action occurs every time the clock goes high. However, in event-based logic, many computations can happen independently, and are only triggered when their input changes. This method is particularly efficient for spiking data, because spiking data is inherently sparse, meaning that there are often long stretches of time without any spiking events.

B. Asynchronous Weighted Sum

The weighted sum used in equation (2) to determine the membrane potential can be computed asynchronously significantly reducing the power consumed [13]. When computed synchronously on non-spiking data, a weighted sum with N inputs requires N multiplications and N additions. However, because of the binary nature of spiking data, this operation can be completed using sequential logic with at most N additions and no multiplications. When a new spiking input is received, the priority encoder outputs only the weight at the address corresponding to the spike; the adder then adds this value to the synaptic current, I(t). If multiple spikes are received simultaneously, the priority encoder outputs the one with the larger address. The encoder then passes a modified version of the spiking input with the selected spike removed as a new input to itself; this process repeats until all incoming spikes have been accounted for. As a result, the system performs only as many additions as needed. Additionally, the weighted sum acts independently of the clock, so it only performs computations when the spiking input changes, allowing it to save power during long stretches without spiking inputs.

C. Clocked Decay Implementation

Unfortunately, it is not possible to build a fully asynchronous LIF neuron mode. As described above, the differential equations modeling I(t) and U(t) can be expressed in discrete time as:

$$\begin{gathered} I_i^l(t+1) = \alpha I_i^l(t) + (1-\alpha)\sum_k W_{ij} S_j^{l-1}(t) \\ U_i^l(t+1) = \beta U_i^l(t) + (1-\beta) I_i^l(t) \end{gathered} \tag{14}$$

where $\alpha = e^{-1/\tau mem}$ and $\beta = e^{-1/\tau syn}$. Although the weighted sum can be implemented asynchronously, the constant rate decay requires a clock. This allows I(t) and U(t) to decay by a factor of α and β respectively, even if no spiking input occurs. Previous work on fully event-based SNNs got around this problem by removing the exponential decay [13], but for BMI applications the exponential decay function plays a critical role by efficiently encoding a representation of the time elapsed since the last spiking input. Fortunately, by using fixed point representations for I(t) and U(t), the decay functions can be executed with a single clock pulse.

D. Weight Update Logic

The calculation of the weight updates is significantly more computationally expensive than the computations described above used for the forward pass. When computing the forward pass, the sparse nature of spiking data could be exploited to remove the computationally expensive weighted sum. However, because the backward pass relies on the surrogate function $\sigma'(U_i^l(t)-1)$, the weight updates are not sparse in nature, and so a multiplication is needed for each weight update, making the number of multiplications proportional to the number of weights. Fortunately, the system can be configured so that the weight update circuits are only active during a calibration period, after which the weights become fixed and only the forward pass circuit is used. The hardware logic for the weight updates is shown in Figure 3 and implements equations (7), (9), and (13).

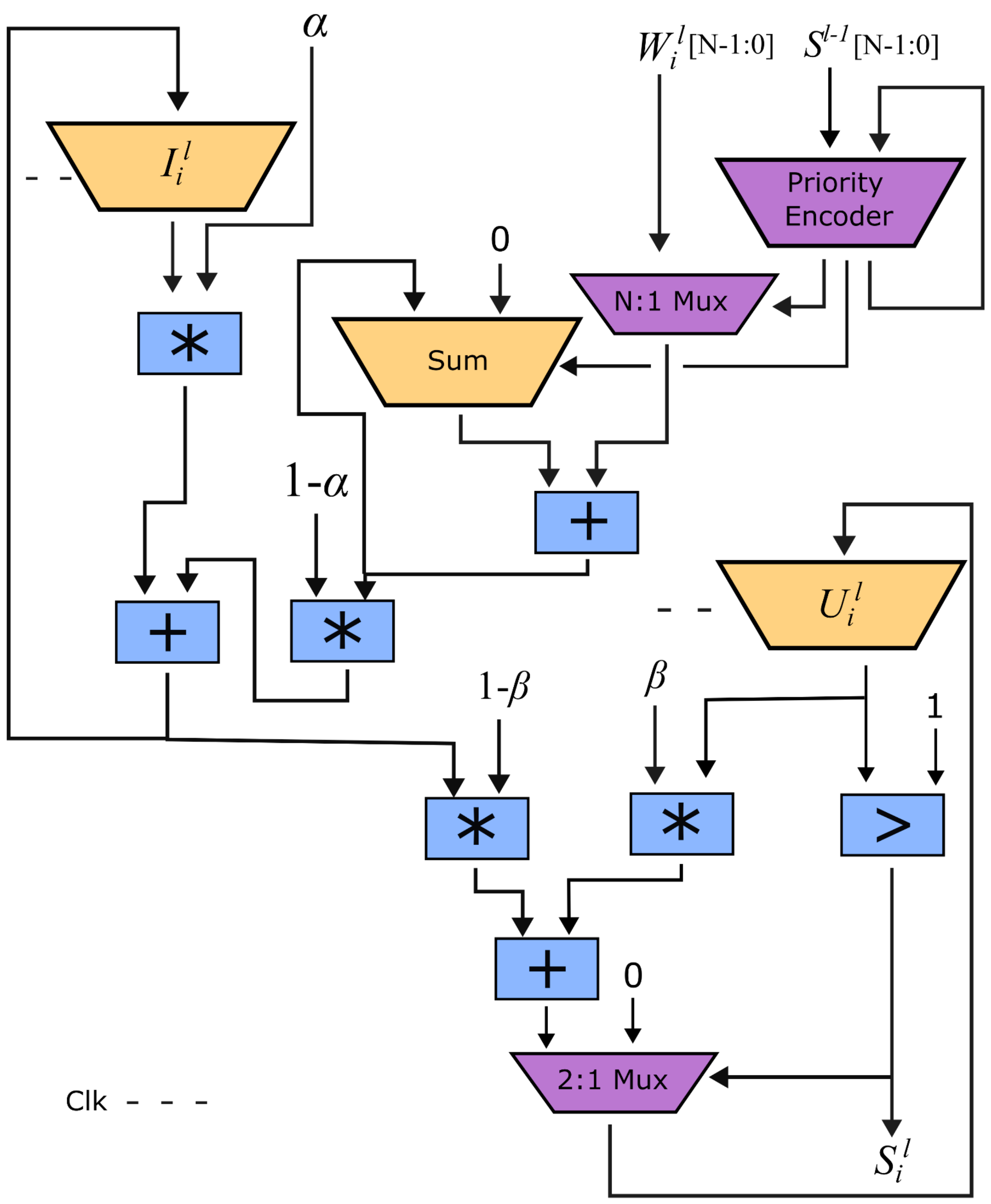

Fig 2. The data path for the forward pass of the leaky integrate-and-fire neuron. The registers with dashed inputs on the side execute update after each clock pulse, while others operate asynchronously whenever an input changes. The priority encoder module is self-triggering, allowing it to process multiple simultaneous spiking events from different channels.

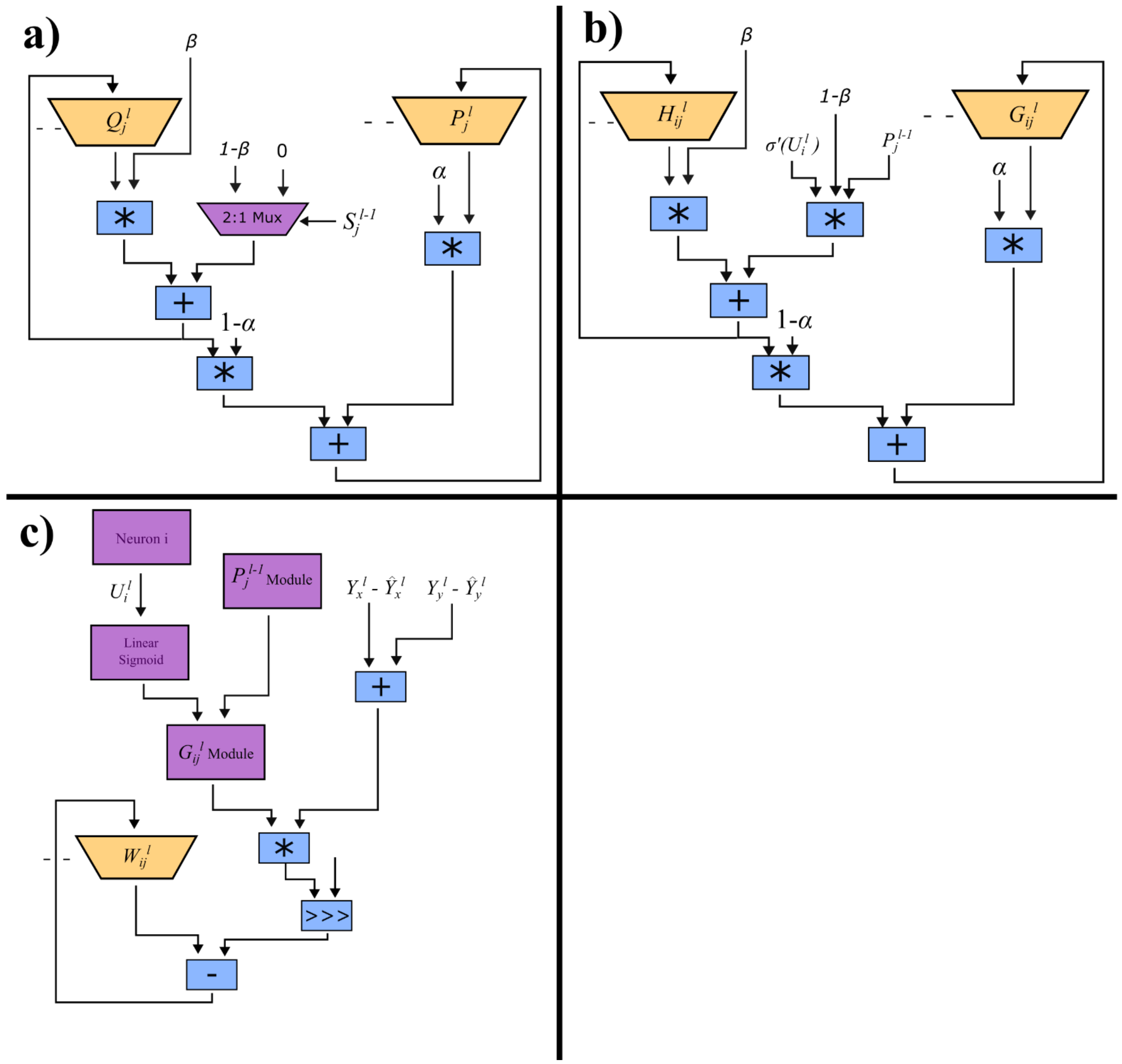

Fig 3. The data paths for the three modules involved in determining the weight updates. In each module, registers that update with each clock pulse are indicated with a dashed line. a) The module that updates the value P_j^l for each hidden and input neuron. b) The module that updates the value G_{ij}^l for each neuron pair. c) The module that updates each weight W_{ij}^l. A register shift is used to incorporate the learning rate (rather than a multiplication), with the learning rate approximated as the nearest integer power of 2.

E. Linear Sigmoid Approximation

The non-linear nature of the sigmoid function makes it very computationally expensive to execute on an FPGA compared to standard multiplications and additions. Previous studies on FPGA spiking neural networks have used piecewise linearization (PWL) to approximate the sigmoid function [14]. With PWL, the sigmoid function is divided into a finite number of segments, with each segment approximated as the least squares regression of the sigmoid function in that region. Likewise, the derivative of the sigmoid can also be approximated with PWL to calculate weight updates. Taking advantage of the symmetrical nature of the sigmoid derivative (i.e. $\sigma'(-x) = \sigma'(x)$), the computation can be expressed by:

$$\sigma'(U) = \begin{cases} m_0|U| + k_0, & |U| <= N_0 \\ m_1|U| + k_1, & N_0 < |U| <= N_1 \\ m_2|U| + k_2, & N_1 < |U| <= N_2 \\ 0, & |U| > N_2 \end{cases} \quad (15)$$

The constants N_i, m_i, and k_i were optimized via a nested binary to minimize the mean squared error between the approximation and the true sigmoid derivative. Additionally, to more closely approximate the spiking discontinuity, we use $\sigma'(10x)$ as the target function to be approximated, rather than $\sigma'(x)$, as demonstrated in [17]. The coefficient of determination for the linearization is $R^2 = 0.988$. Figure 4 compares the linearization with the true sigmoid derivative.

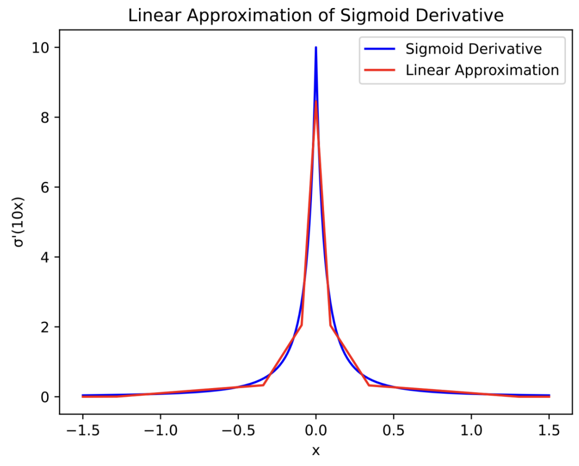

Fig. 4. Comparison of the true derivative of the sigmoid function with the piecewise linear approximation. The approximation is strong correlated with the true function, with R^2 =0.988.

Results

A. Evaluation Data

The SNN was trained and tested on a data set containing intracortical neural recordings and corresponding kinematic data from macaque primates performing reaching tasks [10], [20]. For each reaching task, the monkey moved a cursor towards an indicated target on a 20 cm by 20 cm computer screen. The velocity of the on-screen cursor was sampled at 1000 Hz. The neural recordings were preprocessed to form spike trains using threshold crossing and spike sorting algorithms. Both the neural and kinematic data were placed into 10 ms bins before being passed to the network.

B. Learning Rules Comparison

To evaluate the SNN, we first compare the accuracy and convergence time of a Python simulation of the continuous learning rules with the performance of the same SNN architecture trained using existing machine learning libraries. For both models, the data was split into testing and training data sets, and the accuracy was evaluated using the Pearson Correlation Coefficient to determine the correlation between the model and output and ground truth kinematic data. When training using basic stochastic gradient descent, the continuous learning model and the library-based model achieve approximately the same peak correlation of $\rho = 0.70$, with the continuous learning model requiring fewer training epochs to converge. When using more advanced optimization algorithms, such as Adam, the library based model is able to achieve a slightly higher correlation. Future work will explore the impact of more sophisticated optimization methods, such as momentum, to determine if the continuous learning method can match the performance of the Adam-optimized non-continuous learning.

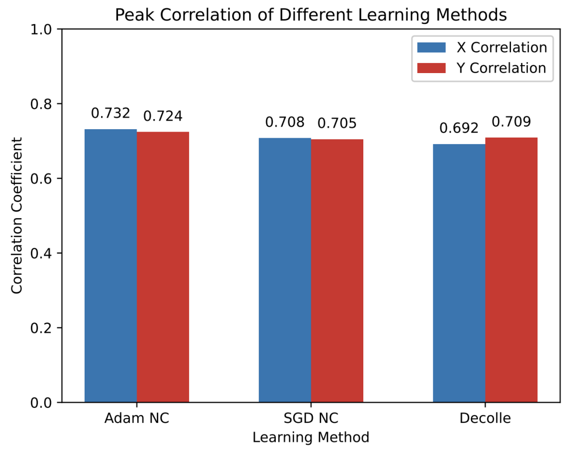

Fig. 5. Comparison of the peak correlations achieved by Python simulations of the three learning methods: non-continuous learning with the Adam optimizer, non-continuous learning with stochastic gradient descent, and deep continuous learning.

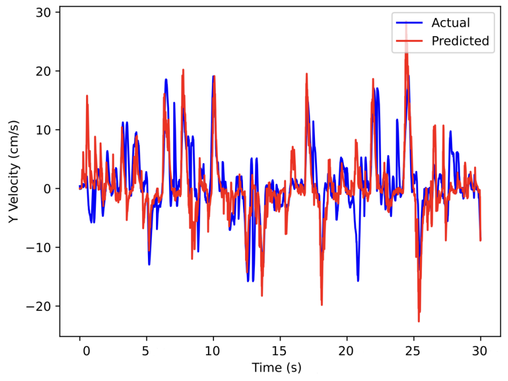

Fig. 6. A 30 second sample comparing the ground-truth kinematic data with the predicted kinematic data output by the spiking neural network. The network used to generate this example was trained for 50 epochs and simulated in Python.

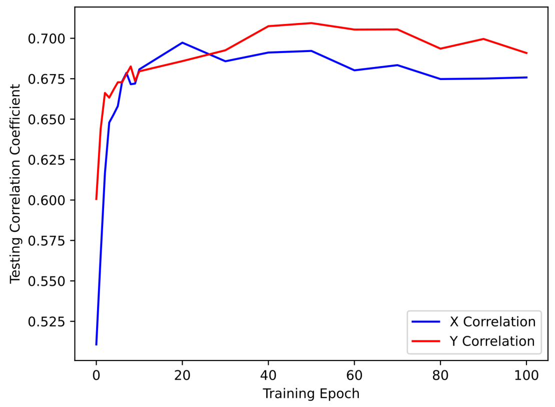

Fig. 7. A plot of the x and y correlation achieved by the Python-simulated SNN on the testing set as a function of the number of training epochs. This plot demonstrates the rapid convergence time of the continuous learning method, since the majority of learning occurs before the fifth epoch.

C. FPGA Implementation

The FPGA implementation of the neuron model was tested using Verilog simulation software. We evaluate the FPGA model on a 1.7 second sample of neural data and compare the results to the Python simulation. The results of this comparison can be seen in Figure 8. The FPGA implementation produced the same spike sequences for hidden layer neurons, as well as an identical overall output, verifying that the FPGA model works as expected on real spiking data. Additionally, the FPGA model produces the same weight updates when training, demonstrating that the weight update rules work. However, when real-time learning was enabled and the weights were actually allowed to change in response to the weight update functions, the FPGA model produced slightly different results, suggesting that there is some difference in how the FPGA system updates the registers. Future work will attempt to find the cause of this difference so that the FPGA model can properly execute complete on-chip continuous learning.

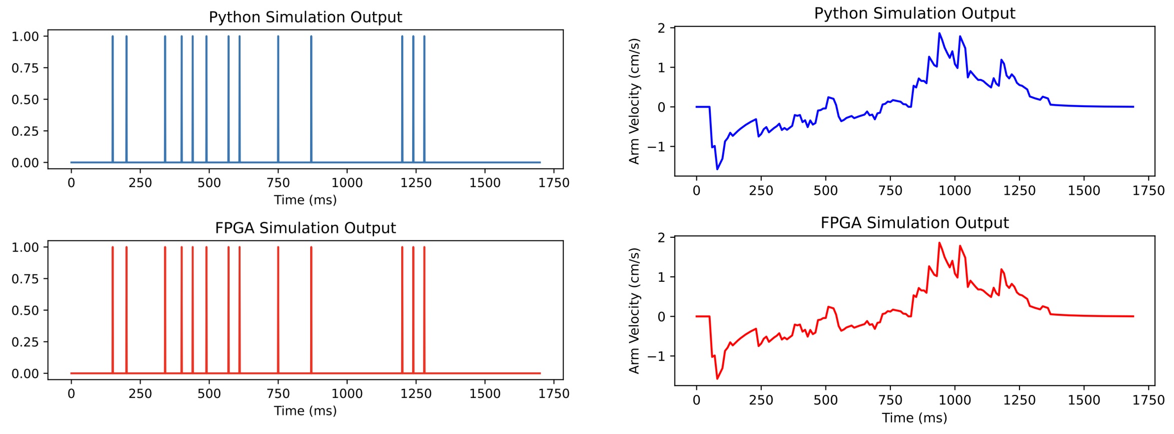

Fig 8. Left) Two plots showing the spiking output of a Python simulation and FPGA simulation of an individual LIF spiking neuron after being given real neural data as an input. The vertical lines indicate times when spikes occurred. Right) Two plots showing the predicted arm velocity output by the complete spiking neural network in response to real neural data. In both cases, the Python and FPGA simulations match very closely, demonstrating that the FPGA implementation works as expected.

Discussion

This research establishes the first example of a spiking neural network for decoding neural data on a FPGA. The FPGA implementation takes advantage of the sparsity and binary nature of spiking data to significantly reduce the number of computations. Additionally, the FPGA system implements on chip learning using local learning rules that use the recursive properties of spiking neural networks to incorporate previous temporal information without any additional memory overhead to store past states, making it possible to perform learning in real-time. In the short term, future work should go beyond FPGA simulations by testing the model on a real FPGA to evaluate the power efficiency. Additionally, future research should seek to collaborate with medical professionals to test the model in a clinical setting and determine its viability for online learning.

References

[1] N. S. C. I. S. Center, “Traumatic spinal cord injury facts and figures at a glance,” 2022.

[2] S. Musallam, B. D. Corneil, B. Greger, H. Scherberger, and R. A. Andersen, “Cognitive Control Signals for Neural Prosthetics,” *Science*, vol. 305, no. 5681, pp. 258–262, Jul. 2004, number: 5681 Publisher: American Association for the Advancement of Science Section: Report. [Online]. Available:
https://science.sciencemag.org/content/305/5681/258

[3] A. Jackson, C. Moritz, J. Mavoori, T. Lucas, and E. Fetz, “The neurochip bci: towards a neural prosthesis for upper limb function,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 14, pp. 187–190, 2006.

[4] S.-P. Kim, J. D. Simeral, L. R. Hochberg, J. P. Donoghue, and M. J. Black, “Neural control of computer cursor velocity by decoding motor cortical spiking activity in humans with tetraplegia,” *Journal of Neural Engineering*, vol. 5, pp. 455–476, 2008.

[5] S. Shah, B. Haghi, S. Kellis, L. Bashford, D. Kramer, B. Lee, C. Liu,
R. Andersen, and A. Emami, “Decoding Kinematics from Human Parietal Cortex using Neural Networks,” in *2019 9th International IEEE/EMBS Conference on Neural Engineering (NER)*, Mar. 2019, pp. 1138–1141, iSSN: 1948-3554.

[6] J. Dethier, P. Nuyujukian, S. I. Ryu, K. V. Shenoy, and K. Boahen, “Design and validation of a real-time spiking-neural-network decoder for brain–machine interfaces,” *Journal of Neural Engineering*, vol. 10, p. 036008, 2013.

[7] G. Indiveri, “A low-power adaptive integrate-and-fire neuron circuit,” in *Proceedings of the 2003 International Symposium on Circuits and Systems*. Bangkok, Thailand: IEEE, 2003, pp. IV–820–IV–823 [Online]. Available: http://ieeexplore.ieee.org/document/1206342/

[8] J. V. Arthur and K. A. Boahen, “Silicon-neuron design: A dynamical systems approach,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, pp. 1034–1043, 2011.

[9] E. Taeckens, R. Dong, and S. Shah, “A biologically plausible spiking neural network for decoding kinematics in the hippocampus and premotor cortex,” in *11th International IEEE EMBS
Conference on Neural Engineering*. IEEE, 2023. [Online]. Available: https://www.biorxiv.org/content/10.1101/2022.11.09.515838v1

[10] M. G. Perich, P. N. Lawlor, K. P. Kording, and L. E. Miller, “Extracellular neural recordings from macaque primary and dorsal premotor motor cortex during a sequential reaching task,” 2018. [Online]. Available: http://dx.doi.org/10.6080/K0FT8J72

[11] K. Mizuseki, A. Sirota, P. E, and B. G, “Multi-unit recordings from the rat hippocampus made during open field foraging,” 2009.

[12] A. Rosado-Munoz, M. Bataller-Mompean, and J. Guerrero-Martinez, “Fpga implementation of spiking neural networks,” *IFAC Proceedings Volumes*, vol. 45, pp. 139–144, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667015404562

[13] J. Stuijt, M. Sifalakis, A. Yousefzadeh, and F. Corradi, “brain: An eventdriven and fully synthesizable architecture for spiking neural networks,” 2021.

[14] T. Gao, B. Deng, J. Wang, and Y. Guosheng, “Highly efficient neuromorphic learning system of spiking neural network with multi-compartment leaky integrate-and-fire neurons,” *Frontiers in Neuroscience*, vol. 16, September 2022.

[15] G. Zhang, B. Li, J. Wu, R. Wang, Y. Lan, L. Sun, S. Lei, H. Li, and Y. Chen, “A low-cost and high-speed hardware implementation of spiking neural network,” *Neurocomputing*, vol. 382, March 2020.

[16] S. Gupta, A. Vyas, and T. Gaurav, “Fpga implementation of simplified spiking neural network,” in *27th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, 2020.

[17] E. O. Neftci, H. Mostafa, and F. Zenke, “Surrogate Gradient Learning in Spiking Neural Networks,” *arXiv:1901.09948 [cs, q-bio]*, May 2019, arXiv: 1901.09948. [Online]. Available: http://arxiv.org/abs/1901.09948

[18] S. N. Chowdhury and S. Shah, “Hardware aware modeling of mixed-signal spiking neural network,” in *20th IEEE*

Interregional NEWCAS Conference. IEEE, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9842116

[19] J. Kaiser, H. Mostafa, and E. Neftci, “Synaptic Plasticity Dynamics for Deep Continuous Local Learning (DECOLLE),” *Frontiers in Neuroscience*, vol. 14, p. 424, May 2020, arXiv:1811.10766 [cs].

[Online]. Available: http://arxiv.org/abs/1811.10766

[20] P. Lalower, M. Perich, L. Miller, and K. Kording, “Linear-nonlineartime-warp-poisson models of neural activity,” *Journal of Computational Neuroscience*, 2018. [Online]. Available: https://doi.org/10.1007/s10827018-0696-6