Winter 3-11-2016

The Design of a Simple, Spiking Sparse Coding Algorithm for Memristive Hardware

Walt Woods
Portland State University

The Design of a Simple, Spiking Sparse Coding Algorithm for Memristive Hardware

by

Walt Woods

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Christof Teuscher, Chair
Melanie Mitchell
Marek Perkowski

Portland State University
2016

Abstract

Calculating a sparse code for signals with high dimensionality, such as high-resolution images, takes substantial time to compute on a traditional computer architecture. Memristors present the opportunity to combine storage and computing elements into a single, compact device, drastically reducing the area required to perform these calculations. This work focused on the analysis of two existing sparse coding architectures, one of which utilizes memristors, as well as the design of a new, third architecture that employs a memristive crossbar. These architectures implement either a non-spiking or spiking variety of sparse coding based on the *Locally Competitive Algorithm* (LCA) introduced by Rozell *et al.* in 2008. Each architecture receives an arbitrary number of input lines and drives an arbitrary number of output lines. Training of the dictionary used for the sparse code was implemented through external control signals that approximate Oja's rule. The resulting designs were capable of representing input in real-time: no resets would be needed between frames of a video, for instance, though some settle time would be needed. The spiking architecture proposed is novel, emphasizing simplicity to achieve lower power than existing designs.

The architectures presented were tested for their ability to encode and reconstruct 8×8 patches of natural images. The proposed network reconstructed patches with a normalized, root-mean-square error of 0.13, while a more complicated CMOS-only approach yielded 0.095, and a non-spiking approach yielded 0.074. Several outputs competing for representation of the input was shown to improve reconstruction quality and preserve more subtle components in the final encoding; the proposed algorithm lacks this feature. Steps to address this were proposed for future work by scaling input spikes according to the current expected

residual, without adding much complexity. The architectures were also tested with the MNIST digit database, passing a sparse code onto a basic classifier. The proposed architecture scored 81% on this test, a CMOS-only spiking variant scored 76%, and the non-spiking algorithm scored 85%. Power calculations were made for each design and compared against other publications. The overall findings showed great promise for spiking memristor-based ASICs, consuming only 28% of the power used by non-spiking architectures and 6.6% as much power as a CMOS-only spiking architecture on this task. The spike-based nature of the novel design was also parameterized into several intuitive parameters that could be adjusted to prefer either performance or power efficiency.

The design and analysis of architectures for sparse coding should greatly reduce the amount of future work needed to implement an end-to-end classification pipeline for images or other signal data. When lower power is a primary concern, the proposed architecture should be considered as it surpassed other published algorithms. These pipelines could be used to provide low-power visual assistance, highlighting objects within high-definition video frames in real-time. The technology could also be used to help self-driving cars identify hazards more quickly and efficiently.

Acknowledgements

I am very grateful for the support and advice of my advisor, Christof Teuscher. Thanks also to Mohammad Taha, Patrick Sheridan, and Jens Bürger for helpful discussions and collaborations which contributed to this work. This work was supported in part by the National Science Foundation under award # 1028378 and by the Defence Advanced Research Projects Agency (DARPA) under award # HR0011-13-2-0015. The views expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defence or the U.S. Government.

Contents

Abstract i

Acknowledgements iii

Contents iv

List of Figures viii

1 Introduction 1

1.1 Motivation 1

1.2 Sparse Coding 2

1.2.1 Definition 2

1.2.2 Encoding a Signal 3

1.2.3 Learning the Dictionary 5

1.3 Related Work 7

1.3.1 CMOS-Only Architectures 7

1.3.2 Architectures With Novel Components 8

1.4 Contributions 10

2 Models 12

2.1 Analog Locally Competitive Algorithm (LCA) 12

2.2 Spiking Locally Competitive Algorithm (SLCA) 14

2.3 Simplified, Spiking Locally Competitive Algorithm (SSLCA) 17

2.4 Memristor 26

3 Reconstruction **31**

3.1 Methodology 31

3.2 Results 33

3.2.1 LCA 33

3.2.2 SLCA 38

3.2.3 SSLCA 38

3.3 Discussion 40

4 Classification of Handwritten Digits **42**

4.1 Methodology 42

4.2 Results 43

4.2.1 SLP 43

4.2.2 LCA 47

4.2.3 SLCA 49

4.2.4 SSLCA 49

4.3 Discussion 51

5 Power Consumption **52**

5.1 Methodology 52

5.2 Results 54

5.3 Discussion 58

6 Conclusion **60**

7 Future Work **62**

References **63**

A Job Stream: Easy Pipeline Processing **69**

A.1 Introduction . . . 71

A.2 Requirements . . . 74

A.3 Building job_stream . . . 74

A.3.1 Building and Installing the Python Module . . . 74

A.3.2 Building the C++ Shared Library . . . 74

A.3.3 Build Paths . . . 75

A.4 Python . . . 75

A.4.1 The Inline Module . . . 75

A.4.2 Running External Programs (job_stream.invoke) . . . 86

A.4.3 Recipes . . . 87

A.5 C++ Basics . . . 92

A.6 Reducers and Frames . . . 95

A.7 Words of Warning . . . 101

A.8 Appendix . . . 102

A.9 Running the Tests . . . 102

A.10 Running a job_stream C++ Application . . . 102

A.11 Running in Python . . . 103

B git-results: Accountable Organization for Experiments **104**

B.1 Installation . . . 104

B.2 Usage . . . 105

B.3 Special Files . . . 106

B.3.1 `git-results-build` . . . 106

B.3.2 `git-results-run` . . . 106

B.3.3 `git-results-progress` . . . 106

B.4 What does `git-results` put in the output folder and the folders above it? . . . 107
B.4.1 Meta information . . . 107
B.4.2 Experiment results . . . 108
B.5 Comparing code from two experiments . . . 109
B.6 Resuming / Re-Entrant git-results-run files . . . 109
B.7 Special Directories . . . 109
B.8 Moving / Linking results . . . 110

List of Figures

1.1 Sparse coding example. A 3×3 input image (input vector of length 9) is shown with 5 non-zero coefficients. Using a dictionary of 6 vectors representing horizontal and vertical lines, this same image can be represented with only 2 non-zero coefficients. 3

1.2 Sparse coding example with reconstruction error. The dictionary elements selected cannot perfectly express the input: the green pixel (horizontal stripes) is present in the reconstruction but not the original, and the red block (vertical stripes) is present in the original but not the reconstruction. These errors between the original input and the reconstruction constitute the residual $\vec{r}$, which can be quantized via the *root-mean-square error* (RMSE). Here, there are 2 incorrect values in the reconstruction out of 9. If each incorrect value is represented by a residual of 1 on the corresponding axis, then the RMSE would be $\sqrt{\frac{2(1)^2}{9}} = 0.47$. 4

1.3 Learning via Oja's rule [23]. By repeatedly adding residuals back into the dictionary elements used in a reconstruction, those dictionary elements adjust to better suit the input. 7

2.1 Basic LCA network displaying relation of input currents $\vec{s}$, receptive field weights $W = (\vec{w_0}, \vec{w_1}, ...)$, internal states $\vec{u}$, thresholded outputs $\vec{a}$, and inhibitory connections of strength $G_{m,n}a_n, m \neq n$. The field weights $w_{i,m}$ are realized as memristive devices at the crossbar junctions. Illustrated in solid gray is the bias column, populated with memristive devices set to weight w_{bias}. The bias column corrects for leak current that would otherwise skew the calculated dot products. For systems needing to calculate dot products between inputs and weights both spanning positive and negative values, such as our reconstruction task, three modifications are required: w_{bias} is set somewhere between 0 and 1, such as 0.4; the illustrated dotted gray rows must be added with complementary weights ($w = 0.7 \rightarrow \overline{w} = 2w_{bias} - w = 0.1$); and only one of s_i or $\overline{s_i}$ must be set to a positive voltage dependent on the sign of the original input. These three modifications modify the dot product calculation to account for negative and positive weights and inputs. 13

2.2 ADADELTA uses the history of an individual parameter's changes to infer its present learning rate η. Shown here are results from training the LCA discussed in this work on the MNIST dataset and evaluating reconstruction performance. Digits trained is on a logarithmic scale to make it clear that ADADELTA outperforms other learning rate schedules early on and maintains its lead. Compared with a static learning rate or an exponentially decaying learning rate, ADADELTA converges much more quickly to an optimal solution. 15

2.3 Shapero *et al.*'s modified SLCA architecture, Fig. 2(b) in the original paper [30]. In contrast with the LCA where local competition is constantly enforced based on the activity of each output neuron, the SLCA uses a *Low-Pass Filter* (LPF) of the spiking activity to determine which neurons need to be inhibited; no inhibition will occur before a spike. This architecture was realized in Shapero *et al.* through a capacitor attached to an inverter fed by the spikes; the voltage on the capacitor controls a PMOS transistor which drains a neuron's state in a rate proportional to the $\Phi^T\Phi - I$ term. For more information see Fig. 5 in the original paper [30]. 16

2.4 The proposed Simple, Spiking Locally Competitive Algorithm architecture. Note that neurons do not need any communication or configuration from other neurons, unlike the LCA and SLCA. The only state shared within the architecture is a single bit indicating whether or not any neuron is currently firing. Black dots at the crossbar junctions represent memristive devices. Row headers were, in this work, a simple passthrough for the input spikes. In future work, the row header would be responsible for providing inhibition amongst neurons without requiring increased network connectivity; this is discussed in Section 2.3. Column headers in this work were a simple capacitor, a switch to drain the capacitor instantly in the event that any neuron fires, and a Schmitt Trigger which detected when the capacitor's voltage exceeding a firing threshold, triggering an output spike. In a true hardware implementation, both row headers and column headers would be responsible for providing write voltages to update the memristive devices as part of the training cycle. 18

2.5 Plot showing the relation of $\bar{\chi}$ to V_{neuron} to fire after 1 ns. Error bars are from estimation of χ for Q_1 and Q_2. 23

2.6 Plot demonstrating effect of specifying capacitance on the trigger voltage used to determine if a neuron is firing, while keeping an average firing rate of 1 GHz. Error bars are from estimation of χ for Q_1 and Q_2. 24

2.7 The firing thresholds required to achieve different untrained to trained firing time ratios. This is dependent on the statistics of the dataset. Instability in firing thresholds with a high untrained to trained fire time ratio is caused by very small differences in voltage creating a large difference in ratio as the trigger voltage approaches the maximum neuron voltage of Q_2/Q_1 (from Fig. 2.7). Approaching this limit is also what causes the firing threshold to flatten out as on the right side of the graph. 25

2.8 The SSLCA is parametrized according to Spike Resolution (average number of spikes in response to an input stimulus) and Spike Density (duty cycle of an input signal with maximum intensity). Spike Resolution is determined by the product of simulation time and spike rate; running the same spike rate for longer will generate more spikes, enhancing the resolution of the output. Spike Resolution is divided out when computing the analog-equivalent output of the SSLCA, such that a neuron with an output of 1 means that it was the only neuron that spiked. Spike Density dictates what percentage of the time a spiking line (input or output) will be active high when it is maximally saturated. For example, a Spike Density of 100% would look like a DC voltage at maximum input, and a square wave with equal high and low times for an input of 50%. With a Spike Density of 50%, a maximum input would look like the aforementioned square wave, while an input of 50% would create a square wave with a low time three times as large as its high time. . 27

2.9 Device resistance varies for some models as voltage increases, while for other models it remains the same. This makes it very important for crossbar architectures to standardize on a consistent read voltage V_{read} when discussing resistances, or account for the non-linearity. Yang *et al.*'s model breaks down around 2.6 V, causing that line to abruptly end. Eshraghian's model breaks down around 2 V in our experiments. Ratios for all models between 0.1 V and their breakdown voltages (or 4 V) are in Table 2.2. 28

3.1 Example images from the NAT10 dataset used to evaluate image reconstruction in this work. Each image in this dataset was scaled to 128×128 pixels with full RGB color. 32

3.2 Reconstruction performance throughout training on 8×8 patches of the NAT10 dataset. The analog, non-spiking algorithm performed the best, followed closely by Shapero *et al.*'s spiking algorithm. The architecture designed for this work performed marginally worse. The reasons for this are addressed in Section 3.2.3. 33

3.3 Activity (portion of neurons actively contributing to output coefficients) for each algorithm when reconstructing 8×8 patches of the NAT10 dataset. Algorithms exhibiting higher activity produced less sparse of an encoding; that is, they are likely to have a lower NRMSE because more neurons were contributing to each reconstruction. Higher activity allows neurons to learn parts of an image rather than an entire image. Too high of activity leads to each neuron learning a single element of the input - essentially re-encoding the input identically to how it was presented. Algorithm parameters (λ for LCA and SLCA) were adjusted to target 20% activity. SSLCA had no parameter to adjust this metric, which is discussed in Section 3.2.3. 34

3.4 Progression of LCA training for two different testing patches; leftmost is the target patch. The five following images are after 34, 136, 644, 1088, and 4096 patches trained. An excess of activity initially (Fig. 3.3) lead to excessive brightness early in training. This was quickly learned out, and the final reproduction had a similar color quality to the original patch. The LCA also did a reasonable job of reconstructing a bright streak in the second example patch. 35

3.5 Example elements from the final dictionary for the LCA. Note how the 20% activity indicated in Fig. 3.3 leads to each neuron representing a large swatch of color at a specific location. Several of these neurons added together can successfully reproduce broad characteristics from any input. 35

3.6 Progression of SLCA training on two different test patches: leftmost is the target patch, and the remaining five images are reconstructions after 34, 136, 644, 1088, and 4096 training patches. Note that the first reconstruction after 34 patches is clamped to all-white. Similarly to the LCA (Fig. 3.4), an initial pattern of over-activity lead to excessive reconstruction brightness. As training progressed, the reconstructions approached a smoothed version of the original input, very similarly to the LCA. The main difference between LCA and SLCA is that SLCA takes longer to converge. The SLCA also demonstrated slightly poorer resolution, lightening the upper-right corner on the second test patch but not reproducing the streak. . . 37

3.7 Example elements from the final dictionary for the SLCA. Similarly to the LCA, the 20% target activity lead to each neuron representing large patches of color at different locations. The SLCA's patches are less smoothed than the LCA's in large part due to the SLCA exhausting more training examples before converging to 20% activity. 37

3.8 Progression of SSLCA training on two different test patches; leftmost is the target patch, and the remaining five images are reconstructions after 34, 136, 644, 1088, and 4096 training patches. Unlike LCA and SLCA (Figs. 3.4 and 3.6), the SSLCA matches brightness and color very quickly. This is partly due to the algorithm being configured using the average statistics of the input dataset. Lower activity also contributes; the SSLCA exhibits only 8% activity versus the LCA and SLCA's 20% (Fig. 3.3). Since there are fewer nonzero coefficients, Oja's rule dictates that fewer neurons get trained each step, resulting in faster convergence. The second test patch is notably worse with SSLCA; the changing of color between the last two reproductions indicates that one of the 50 output neurons was contested, a side-effect of an inadequate number of neurons participating in the reconstructions. 39

3.9 Example elements from the final dictionary for the SSLCA. Unlike the LCA or SLCA where dictionary elements represent locational patches of color, the SSLCA elements come to represent whole images. This results from higher output sparsity (Fig. 3.3), which occurs because the SSLCA as implemented in this paper has no means of inhibiting parts of the input that are already represented by previous neuronal firings. Another aspect worthy of note: these dictionary elements are far brighter than those of either the LCA or SLCA (Figs. 3.5 and 3.7). This is due to the firing threshold V_{neuron} being determined from a perfect match to the input. This is further explained in Section 3.2.3. 39

4.1 Classification performance throughout training on the MNIST dataset with 50 neurons. Performance of the SLP being superior to the others was a result of 50 neurons being too small for the task at hand; however, for a comparative analysis between algorithms doing the same thing, the number was sufficient. Ultimately, the LCA outperformed both the SLCA and SSLCA. More interestingly, the SSLCA significantly outperformed the SLCA, even though the SLCA had higher activity and produced a similar NRMSE on this task (Figs. 4.2 and 4.3). This phenomenon is discussed between Figs. 4.6 and 4.8. 44

4.2 Reconstruction performance throughout training on the MNIST dataset. All algorithms settle to a similar NRMSE despite very different levels of activity (Fig. 4.3). 45

4.3 Activity of sparse coding layer throughout training on the MNIST dataset. Similar to Fig. 3.3 from Section 3.2, the LCA and SLCA both demonstrated higher activity than the SSLCA. The LCA and SLCA were both λ-adjusted in the same way, however the LCA would not go lower than 0.14 activity on this task. As shown, the λ chosen was quite high and initially suppressed most of the LCA activity. Once the algorithm adapted, its dictionary trended towards lines rather than whole digits as with the SLCA and SSLCA (Figs. 4.4, 4.6 and 4.8). 46

4.4 LCA reconstruction performance throughout training on the MNIST dataset. A very high λ inhibited visible reproductions early, but yielded to reasonable reconstructions. High initial λ was necessary for a fair comparison amongst algorithms; otherwise the LCA would learn large pixels and demonstrate high activity). 47

4.5 Sample dictionary elements from LCA after training. Even with a high λ enforcing low activity, LCA learned digit edges as opposed to the whole digits learned by the SLCA and SSLCA (Figs. 4.7 and 4.9). 47

4.6 SLCA reconstruction performance throughout training on the MNIST dataset. The SLCA performed notably worse than the LCA with only marginally worse NRMSE. From these reconstructions, it is clear that the SLCA often reproduced several digits with the same neuron combination. While this helped the SLCA minimize reconstruction error in its initial high-activity state (Fig. 4.3), the ambiguity confused the SLP. 48

4.7 Sample dictionary elements from SLCA after training. The duality of some of these neurons is apparent; the 3^{rd} receptive field is primarily a 7, but also has the loop element from a 9. The 5^{th} element could be either a 2 or an 8. This ambiguity prevented the SLP from effectively differentiating certain digits. 48

4.8 SSLCA reconstruction performance throughout training on the MNIST dataset. Unlike the LCA and SLCA which both had higher activity than the SSLCA, these reconstructions are very targeted to be a single digit. This the effect of combining Oja's rule with very low activity: each receptive field was affected by training infrequently, and learned an average representation of a specific digit. This can also be seen in Fig. 4.9. 50

4.9 Sample dictionary elements from SSLCA after training. Unlike the LCA or SLCA, each dictionary element very clearly learns to represent a single digit. This is due to the combination of Oja's rule and lower activity, leading to each receptive field being updated fewer times, in more specific conditions. 50

5.1 Power consumption on the Reconstruction and Classification tasks. The Reconstruction task from Chapter 3 consists of mapping an input signal of 192 values to a sparse code with 50 values; the Classification task from Chapter 4 consists of mapping an input signal with 784 values to a sparse code with 50 values. 54

5.2 Comparative score on the Reconstruction task in this work, calculated by balancing the NRMSE from Chapter 3 against estimated power consumption for each architecture (SLCA's power comes from Shapero *et al.*). Power and NRMSE, both quantities which are worse when larger, were normalized by dividing out the worst architecture's value for each. Each quantity was then scaled by α and $1-\alpha$, respectively, to achieve the given score. LCA is the clear winner on this task, until power becomes about 70% of the importance criteria, at which point the SSLCA's substantially lower power gives it the lead. Alterations to the SSLCA that might help decrease the NRMSE on this task, making it the clear choice across the board, are discussed in Sections 2.3 and 3.2.3 as well as Chapter 7. 56

5.3 Comparative score on the Classification task in this work, calculated by balancing the reciprocal of classification accuracy from Chapter 4 against estimated power consumption for each architecture (SLCA's power comes from Shapero *et al.*). The reciprocal of classification accuracy was chosen so that large values would indicate worse performance. Both the power and reciprocal of classification accuracy were normalized by dividing out the worst architecture's value for each. The remaining quantities were then scaled by α and $1 - \alpha$, respectively, to achieve the given score. While the LCA's superior classification rate gives it the edge when power is unimportant, power needs to be only 22% of the performance criteria for the SSLCA to outperform the other algorithms. The SLCA is consistently the worst choice for this task, owing both to the large power required by Shapero *et al.*'s implementation as well as the ambiguity in its receptive fields as discussed in Section 4.2.3. 57

A.1 A job_stream job takes some input, transforms it, and emits zero or more outputs 92

A.2 A job that adds one to the input and emits it 92

A.3 Estimating pi 95

A.4 Estimating pi 96

1

Introduction

1.1 Motivation

Sparse coding architectures have been around for a long time and have been used for a number of applications, including image classification [18,26], compression [6], and reinforcement learning [1]. The ability to reduce a large number of inputs into a sparse code of higher order features has been primarily performed on traditional computer architectures or in CMOS technology. Incorporating novel devices, such as the memristor, into these architectures promises greater efficiency. Since these architectures perform generic tasks (a sparse coding algorithm does not dictate a specific type of input, e.g. video, pictures, or audio could all be processed on the same hardware), these algorithms are a great target for optimized, next-generation ASICs.

Memristive devices, first fabricated in 2008, offer the ability to represent a broad range of values in a single, two-terminal device [32]. The simplicity and flexibility of these devices promise a significant reduction in both power and area for integrated circuits [37,38]. Memristors have also given new life to the study of neuromorphic architectures: in the past few years, there have been many publications detailing their unique ability to function similarly to a biological synapse, whose sensitivity to input can be adjusted throughout the synapse's life [10,26,40]. In neuromorphic architectures, the strengths of these synapses are represented by weight matrices. With memristors, the weight matrices can be stored as a crossbar with memristors connecting each row and column, and evaluated directly by applying voltages to

rows and measuring the current through the columns. This approach provides a compact, analog means of representing the various weights within a neuromorphic architecture.

This work explored neuromorphic architectures specifically related to sparse coding. A common problem in image recognition is the high dimensionality of the input signal. Sparse coding describes the process of translating a multi-dimensional input vector into a new basis where a smaller number of basis vectors have non-zero coefficients, effectively increasing the significance of each non-zero coefficient in the resulting vector. In this work, the basis vectors were learned from training data on the MNIST digit database [15] as well as patches on selected natural images (Fig. 3.1). The *Locally Competitive Algorithm* (LCA) introduced by Rozell *et al.* provides the baseline for this analysis [27]. LCA works well as it implements inhibition between different basis vectors, preventing them from becoming too similar to one another, while at the same time calculating this inhibition factor rather than needing to learn it (in contrast with, e.g., SAILnet [13, 42]).

1.2 Sparse Coding

1.2.1 Definition

Sparse coding is a process by which input vectors are encoded more sparsely (fewer non-zero coefficients) in a different basis than the input's natural basis. By reducing the number of non-zero coefficients needed to describe an input, sparse coding effectively increases the significance of each non-zero coefficient in the output. Intuitively, this means that the new basis maps significant features in the original input space onto single axes in the new space, which can be linearly combined to reconstruct members of the original input space. Often, the new basis is called a

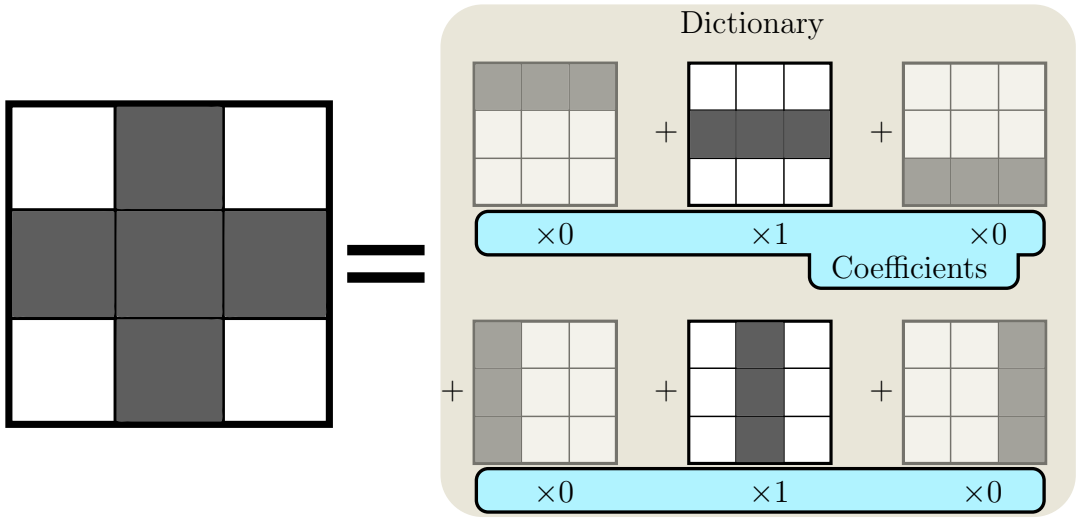

Figure 1.1: Sparse coding example. A 3×3 input image (input vector of length 9) is shown with 5 non-zero coefficients. Using a dictionary of 6 vectors representing horizontal and vertical lines, this same image can be represented with only 2 non-zero coefficients.

"dictionary" due to the fact that its axes encode key features of the input space, with each basis vector being called a "dictionary element." Figure 1.1 demonstrates an example of sparse coding.

Sparse coding can be broken down into two steps: encoding an input given a predefined dictionary, shown in Section 1.2.2, and learning a dictionary to better encode a set of inputs, explored in Section 1.2.3.

1.2.2 Encoding a Signal

Encoding is the process of generating coefficients for the sparse code's dictionary elements that best represent a given input vector. One possible set of coefficients generated by encoding an image using a sparse code is shown in Fig. 1.1.

Since sparsity is the desired attribute in a sparse code, often the active (non-zero coefficient) dictionary elements do not perfectly describe the input. This leads

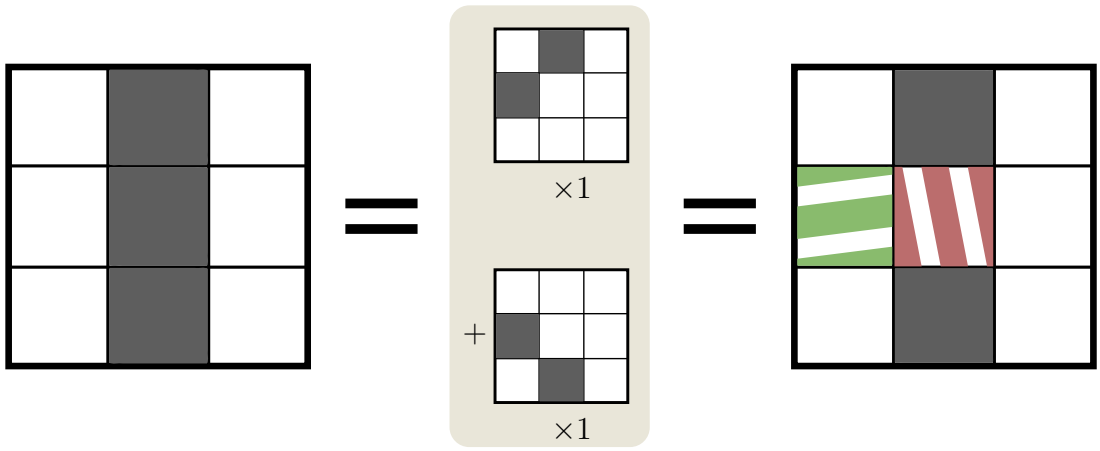

Figure 1.2: Sparse coding example with reconstruction error. The dictionary elements selected cannot perfectly express the input: the green pixel (horizontal stripes) is present in the reconstruction but not the original, and the red block (vertical stripes) is present in the original but not the reconstruction. These errors between the original input and the reconstruction constitute the residual $\vec{r}$, which can be quantized via the *root-mean-square error* (RMSE). Here, there are 2 incorrect values in the reconstruction out of 9. If each incorrect value is represented by a residual of 1 on the corresponding axis, then the RMSE would be $\sqrt{\frac{2(1)^2}{9}} = 0.47$.

to a loss of details that can not be represented in the new basis. These lost details can be quantized as reconstruction error: $R(\vec{r} = \vec{x} - \hat{\vec{x}})$, where R is some function of the residual $\vec{r}$ between the original input $\vec{x}$ and its analog that has gone through the sparse coding process $\hat{\vec{x}}$. For this work, R was defined as the *root-mean-square error* (RMSE), $R(\vec{r}) = \sqrt{\frac{\sum_{i=1}^{n} r_i^2}{n}}$. This function has the benefit of being the distance between the input vectors in the space of its standard basis. This concept is demonstrated in Fig. 1.2. Algorithms dealing with mapping an input vector often offer a trade-off between the sparsity of the final solution and its accuracy. In the LCA, this is accomplished through the λ threshold parameter [27].

There are many algorithms that address the encoding part of sparse coding. One of the oldest known algorithms is Matching Pursuit, which selects the next

non-zero coefficient based on which coefficient would be largest [17]. More complicated algorithms, such as the LCA presented by Rozell *et al.*, attempt to simultaneously solve for all coefficients by integrating a system of ordinary differential equations [27]. Algorithms such as *Spike-Timing-Dependent Plasticity* (STDP) implement the encoding step alongside dictionary learning without any separation of the two parts [18].

The focus of this work was on algorithms based on the LCA, as it implements inhibition between dictionary elements in such a way that the vectors making up the elements are at a larger angle to one another than without inhibition.

1.2.3 Learning the Dictionary

Sparse codes allow any input to be represented with any dictionary. However, a poor dictionary leads to a large residual, implying that the encoded vector does not capture much of the input information. Iteratively improving a sparse code requires tweaking the dictionary in such a way that it encodes future inputs more sparsely and accurately.

Different algorithms exist for updating the dictionary. One family of such algorithms is Hebbian learning, where updates are applied according to the analog strengths of the inputs in tandem with the strengths of the outputs. The traditional Hebbian rule for updates, proposed by Donald Hebb in 1949, implements behavior where neurons become more and more responsive to the input combination that caused them to fire in the first place [9]. However, this rule is unstable on its own, requiring the weights to be constantly renormalized. This work was extended by Erkki Oja in 1982, who proposed a variant that adjusts the learning rate according to the residual of the input rather than the input itself [23]. This avoids the

instability problems seen with Hebb's rule. Oja's rule was further extended by Terence Sanger in 1989 to produce a layer of neurons that are arranged to avoid linear dependence between the receptive fields in the layer [28]. However, Sanger's rule is more complicated to implement in hardware, as well as not being suitable for sparse coding: the usage of principle components infers orthogonality in each receptive field, leading to greater loss of accuracy when only a subset of coefficients are allowed to be non-zero.

An extension to the Hebbian family, algorithms such as STDP also provide means of updating the receptive fields in tandem with encoding the input. In STDP, any inputs that are active when an output is triggered will have their weights increased, while any inactive inputs will have their weights decreased. This is triggered by having the column voltage varied between a suitably low value to trigger an increase in weight for active inputs, and a suitably high value to trigger a decrease in weight for inactive inputs. Such an approach was previously demonstrated with a memristive crossbar by Querlioz *et al.* [26].

This work used Oja's rule to update the dictionary elements, which is illustrated in Fig. 1.3 and defined as [23]:

$$\Delta w_{i,j} = \eta y_j r_i. \tag{1.1}$$

Oja's rule provides stable weights, and also lends itself to an intuitive understanding that the dictionary's ultimate goal is to minimize reconstruction error (realized through the residual term r_i in Eq. (1.1); a smaller residual produces a smaller $\Delta w_{i,j}$).

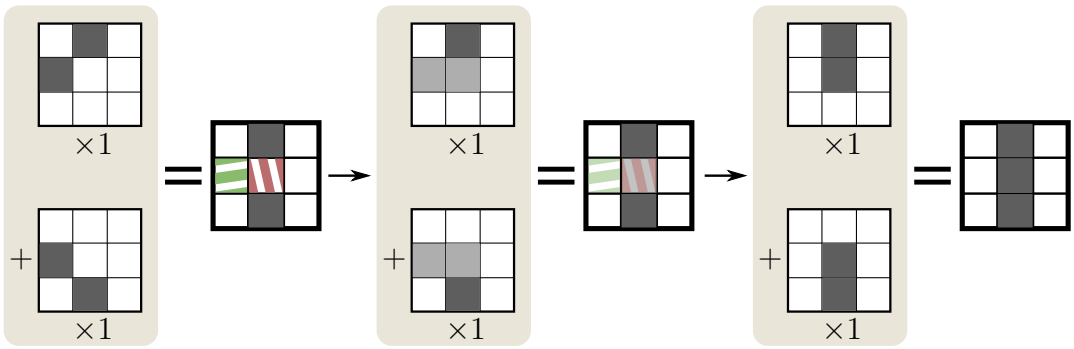

Figure 1.3: Learning via Oja's rule [23]. By repeatedly adding residuals back into the dictionary elements used in a reconstruction, those dictionary elements adjust to better suit the input.

1.3 Related Work

Sparse coding is a technique with a number of applications, including image classification [18,26], compression [6], and reinforcement learning [1]. The wide applicability of sparse coding has lead researchers to investigate architectures implementing algorithms that address sparse coding. Some of these prior works use conventional CMOS techniques, while others use more novel nanodevices to realize the same algorithms.

1.3.1 CMOS-Only Architectures

Traditional CMOS architectures can be used to implement sparse coding. Kim *et al.* implemented a spiking ASIC in 65nm CMOS in 2014 [13], achieving a throughput of 952 Mpixels/s at 0.486 nJ/pixel using a sparse code of 512 receptive fields. Their architecture was based on SAILnet, a sparse coding algorithm similar to LCA where the inhibition between output columns is learned rather than computed [42].

Shapero *et al.*, the proposers of the *Spiking Locally Competitive Algorithm* demonstrated in this work, implemented both their spiking LCA and non-spiking

LCA using a *Field-Programmable Analog Array* (FPAA) leveraging 350 nm technology [29, 30]. They cited 3 mW of power consumed by the spiking algorithm for a 12×18 network. However, the chip they used idles at 1.7 mW of power, making the spiking LCA consume 1.3 mW of power on its own. The non-spiking LCA consumed 28.3 µW of power for a 2×3 network. They claim power scaling of $\mathcal{O}(n)$ and $\mathcal{O}(n\sqrt{n})$ for their spiking and non-spiking architectures. Scaling up the network sizes to match the number of neurons used in this work, 50, the non-spiking algorithm would require around 1.93 mW and the spiking algorithm would require 3.72 mW of power. However, these figures omit the additional increase in input lines; the tasks in this work use 192 and 768 input lines, whereas those estimates are for 33 input lines. Shapero *et al.* do not address input line scaling independently. It is reasonable to assume that input line scaling is $\mathcal{O}(n)$, given neither the non-spiking nor spiking architectures presented by Shapero *et al.* use additional logic between input lines. Therefore, adding the remaining 159 input lines by scaling the original, measured figures, these architectures are estimated at 4.18 mW for non-spiking and 21.0 mW for spiking. For the larger problem in this work, with 768 input lines, these architectures are estimated at 11.8 mW and 79.0 mW, respectively.

1.3.2 Architectures With Novel Components

Other researchers have produced sparse coding architectures using novel components such as memristors, using CMOS for the traditional logic parts of their architectures [3, 8, 24, 31, 33, 40].

One of the first memristive sparse coding architectures was designed and simulated by Zamarreño-Ramos *et al.* in 2011 [40]. Using data recorded from a fabricated spiking retina chip to drive the rows, different action potential shapes were explored in the columns. Unfortunately, the encoding accuracy of this setup was not evaluated in this work. Power is discussed, but the memristor model that they are using leads to the derivation of power in excess of 2 kW.

Soudry *et al.* looks at using memristors for implementing multilayer neural networks [31]. The overall architecture is not defined, instead specifying two transistors and a single memristor per synapse, regardless of other configuration. Sparse coding accuracy is said to be comparable to that in software, limited by the accuracy of the weight update. Power benefits of chips using memristors are briefly discussed, revealing a figure of 13-50× better than "standard CMOS technology," though actual numbers are not presented.

Payvand *et al.* recently described and simulated a memristor-based neuromorphic chip based on STDP [24]. This paper describes leveraging CMOL techniques to combine CMOS with memristors. Power is not discussed, nor is accuracy. This work also presents an algorithm which only takes the first spike into account; future spikes are discarded. As shown in Section 3.2.3, algorithms relying on a single spike perform significantly worse during reconstruction as they are unable to combine multiple dictionary elements to better represent the input.

Bennett *et al.* also recently used memristors in tandem with 45nm CMOS to identify patterns [3]. Their work focused on binary functions, with both input and output dimensionalities well below that needed for image recognition. Power and accuracy figures are not discussed.

Garbin *et al.* investigated the variability of memristors in a convolutional neural

network, accompanied by 28nm CMOS [8]. They confirmed that device variability does not significantly degrade classification on either of the MNIST or GTSRB databases. On MNIST, they achieved 99% accuracy, although this is with a very large, convolutional network.

1.4 Contributions

This work contains evidence of my work over the last two years, including:

- Implemented Rozell *et al.*'s original, analog LCA for sparse coding (Section 2.1).
- Implemented Shapero *et al.*'s modified, spiking LCA for sparse coding (Section 2.2).
- Designed and implemented a Simple Spiking algorithm using memristors in a crossbar for sparse coding (Section 2.3).
- Introduced a meaningful vocabulary for controlling the trade-offs between accuracy and efficiency for spiking models, and parametrized the Simple Spiking variant proposed in this work to that vocabulary (Section 2.3).
- Compared different memristive devices in an environment with a 500 MHz clock (Section 2.4).
- Evaluated sparse coding algorithms' accuracy with natural image reconstruction (Chapter 3).
- Evaluated sparse coding algorithms' utility for digit classification on the MNIST handwritten digit database (Chapter 4).

- Evaluated the power consumption properties of these algorithms (Chapter 5).
- Published two conference papers and one journal article with collaborators from teuscher.:Lab [34–36]; the conference paper from NANOARCH 2014 in Paris, France received the "Best Student Paper" award.
- Designed, developed, and packaged job_stream software package for simple, extensible parallelization (Appendix A).
- Designed, developed, and distributed git-results software package for experiment cataloguing and diffing (Appendix B).

2

Models

This work focused on three different algorithms for sparse coding: Rozell *et al.*'s *Locally Competitive Algorithm* (LCA), Shapero *et al.*'s spiking extension of that algorithm, the *Spiking Locally Competitive Algorithm* (SLCA), and a novel architecture designed and presented as part of this work, the *Simplified, Spiking Locally Competitive Algorithm* (SSLCA). This chapter explores the theory behind each architecture, as well as how the dictionaries used by these sparse coding architectures were trained.

2.1 Analog Locally Competitive Algorithm (LCA)

Rozell *et al.* introduced their *Locally Competitive Algorithm* (LCA) in 2008 as a means of solving the sparse coding problem [27]. Motivated by the sparse neuronal activity found in biological brains, the LCA minimizes a given cost function in order to achieve sparse coding. Sparseness within the algorithm is achieved by modeling local inhibitory connections across neurons. This locality well resembles biological brains and makes it a suitable algorithm for hardware implementations. The LCA is an unsupervised learning approach that calculates an output neuron's excitation by integrating an ODE as shown in [27]:

$$\dot{u_m}(t) = \frac{1}{\tau}\left[b_m(t) - u_m(t) - \sum_{n \neq m} G_{m,n} a_n(t)\right], \tag{2.1}$$

where u_m is the m^{th} output neuron's membrane potential, b_m represents a neuron's excitatory inputs and is equal to the dot-product of the neuron's input vector $\vec{s}$

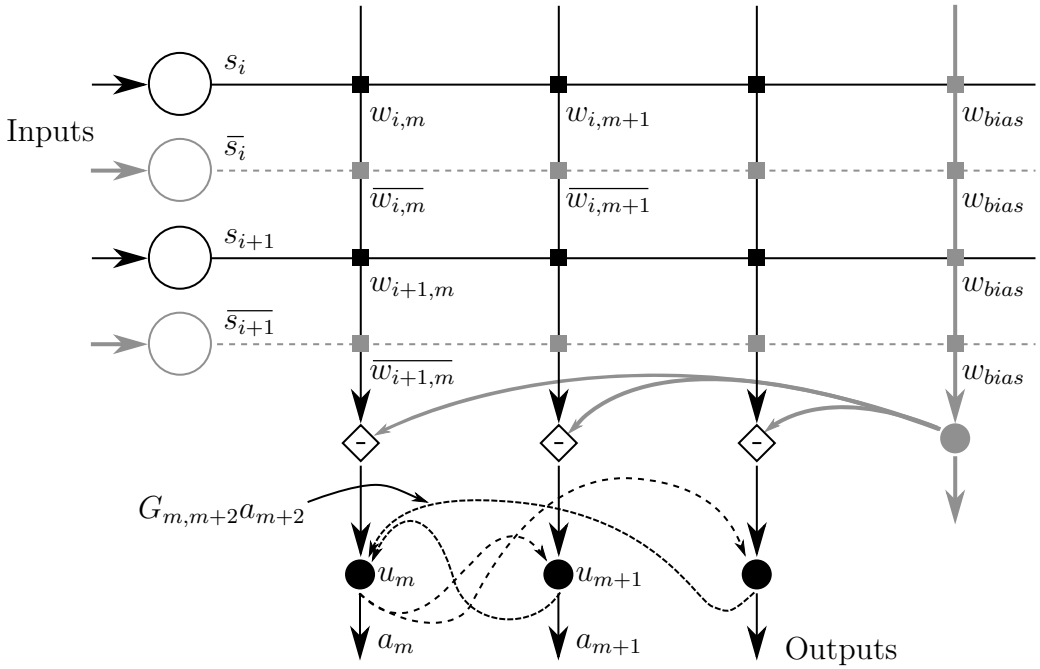

Figure 2.1: Basic LCA network displaying relation of input currents $\vec{s}$, receptive field weights $W = (\vec{w_0}, \vec{w_1}, ...)$, internal states $\vec{u}$, thresholded outputs $\vec{a}$, and inhibitory connections of strength $G_{m,n}a_n, m \neq n$. The field weights $w_{i,m}$ are realized as memristive devices at the crossbar junctions. Illustrated in solid gray is the bias column, populated with memristive devices set to weight w_{bias}. The bias column corrects for leak current that would otherwise skew the calculated dot products. For systems needing to calculate dot products between inputs and weights both spanning positive and negative values, such as our reconstruction task, three modifications are required: w_{bias} is set somewhere between 0 and 1, such as 0.4; the illustrated dotted gray rows must be added with complementary weights ($w = 0.7 \rightarrow \overline{w} = 2w_{bias} - w = 0.1$); and only one of s_i or $\overline{s_i}$ must be set to a positive voltage dependent on the sign of the original input. These three modifications modify the dot product calculation to account for negative and positive weights and inputs.

and the corresponding weight vector $\vec{w_m}$, $G_{m,n}$ describes the mutual representation of the m and n^{th} receptive fields and is defined as $\phi^T\phi - I$, and a_n is a thresholding function applied to u_n to achieve sparsity. This setup is further explored in Fig. 2.1.

As stated in the original paper, this ODE minimizes an energy function that is the combination of the difference between a reconstruction and the original input signal, plus a sparseness term [27]. The result is that the product of the dictionary ϕ with the activation vector $\vec{a}$ is an optimal approximation $\hat{\vec{s}}$ of the input signal $\vec{s}$ with the dictionary ϕ and the given sparsity penalty (achieved by the thresholding function used for the activation vector $\vec{a}$).

In this work, the dictionary was updated after each reconstruction according to Oja's rule as in Section 1.2.3. However, to achieve a near-optimal learning schedule, the work of Zeiler *et al.* on the ADADELTA algorithm was used to generate a dynamic learning rate [41]. The difference between these schedules can be seen in Fig. 2.2.

2.2 Spiking Locally Competitive Algorithm (SLCA)

Shapero *et al.* extended Rozell's analog algorithm into a spiking variant in 2013 [30]. Utilizing CMOS technology, their approach implements an inhibitory response to residual activity measured from spikes. This is demonstrated in Fig. 2.3.

Equations 5-7 from [30] were implemented to test this architecture:

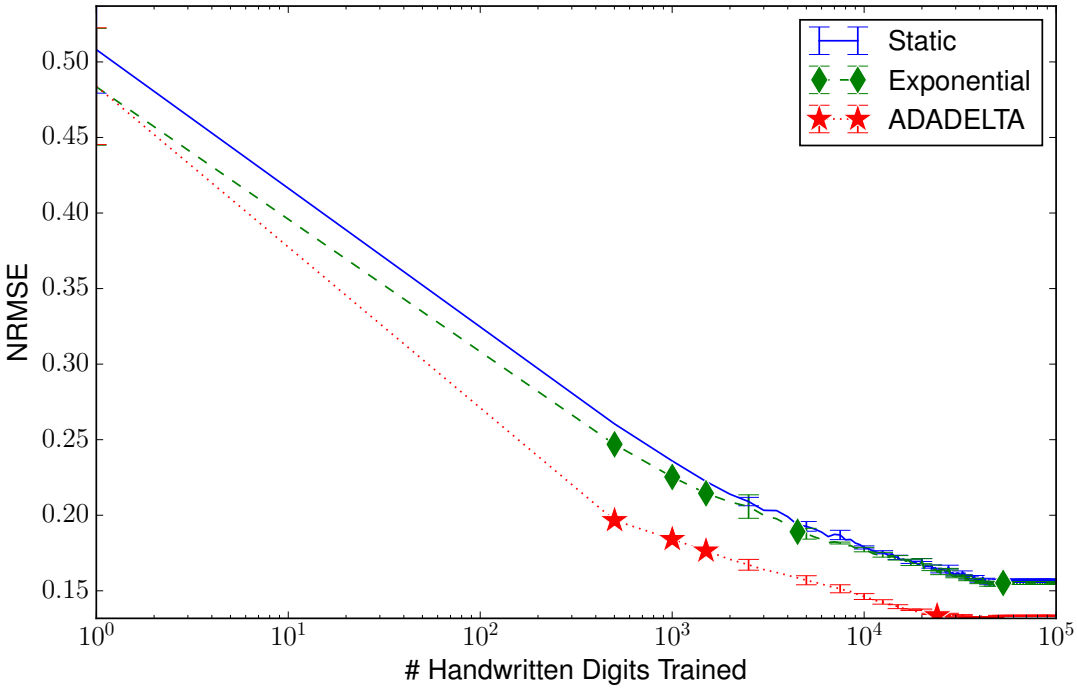

Figure 2.2: ADADELTA uses the history of an individual parameter's changes to infer its present learning rate η. Shown here are results from training the LCA discussed in this work on the MNIST dataset and evaluating reconstruction performance. Digits trained is on a logarithmic scale to make it clear that ADADELTA outperforms other learning rate schedules early on and maintains its lead. Compared with a static learning rate or an exponentially decaying learning rate, ADADELTA converges much more quickly to an optimal solution.

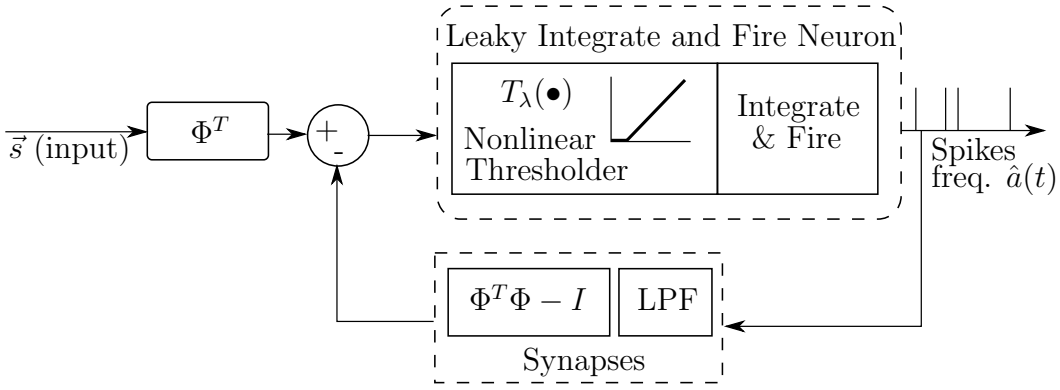

Figure 2.3: Shapero *et al.*'s modified SLCA architecture, Fig. 2(b) in the original paper [30]. In contrast with the LCA where local competition is constantly enforced based on the activity of each output neuron, the SLCA uses a *Low-Pass Filter* (LPF) of the spiking activity to determine which neurons need to be inhibited; no inhibition will occur before a spike. This architecture was realized in Shapero *et al.* through a capacitor attached to an inverter fed by the spikes; the voltage on the capacitor controls a PMOS transistor which drains a neuron's state in a rate proportional to the $\Phi^T\Phi - I$ term. For more information see Fig. 5 in the original paper [30].

$$\dot{v}(t) = u(t) - \lambda, \qquad v(t^-) > 1 \Rightarrow v(t^+) = 0,$$

$$\hat{a}(t) = max(u(t) - \lambda, 0) = T_\lambda(u(t)),$$

$$u_i(t) = b_i - \sum_{j \neq i} \left(H_{i,j} \sum_k \alpha(t - t_{j,k}^{FB}) \right),$$

where v is analogous to the voltage of a capacitor indicating that neuron's state, u is the current into that capacitor, $\hat{a}$ is the firing rate of each neuron (the output of the network; the sparse code), b is the dot product of the neuron's receptive field and the input vector, $\alpha(t) = U(t)e^{-t/\tau}$ where $U(t)$ is the heaviside step function, τ is a time constant, $H_{i,j}$ is $\phi^T\phi - I$ just like G in the analog algorithm, and $t_{j,k}^{FB}$ is the k^{th} firing of the j^{th} neuron.

The $\alpha(t - t_{j,k}^{FB})$ term in $u_i(t)$ provides an impulse response for inhibition after each spike, preventing the over-stimulation of any individual element in the reconstruction. The original LCA's self-inhibitory term $-u_m(t)$ is replaced by the spiking behavior, and explicit thresholding of the output is no longer necessary since a spike might not be generated even on a column exhibiting a non-zero charge.

Dictionary updates with the SLCA were accomplished identically to the LCA: Oja's rule produces changes in the weight matrix based on the residual, using ADADELTA to adjust the learning rate.

2.3 Simplified, Spiking Locally Competitive Algorithm (SSLCA)

The *Simplified, Spiking Locally Competitive Algorithm* (SSLCA) is a modified version of the LCA with emphasis on low power and implementation simplicity rather than accuracy. Approaching the problem from this angle helps to cement understanding of the trade-offs involved in these algorithms, as well as better defining the trade-offs available.

To implement the original LCA in hardware, a memristive crossbar is required, as well as isolation circuitry to deal with sneak paths and convert current through the memristive devices into a voltage [36]. Additional circuitry would need to be added to implement the subtraction of competing representations as in Eq. (2.1). The circuitry to implement inhibition based on local competition would be substantial: for M neurons, M^2 inhibitory forces are required, as each neuron exerts a force on each other neuron including itself to work with non-normalized dictionary elements. Each inhibitory force has its own weight and is multiplied by the current activity of one of the neurons. Shapero *et al.*'s SLCA also requires M^2 transistive

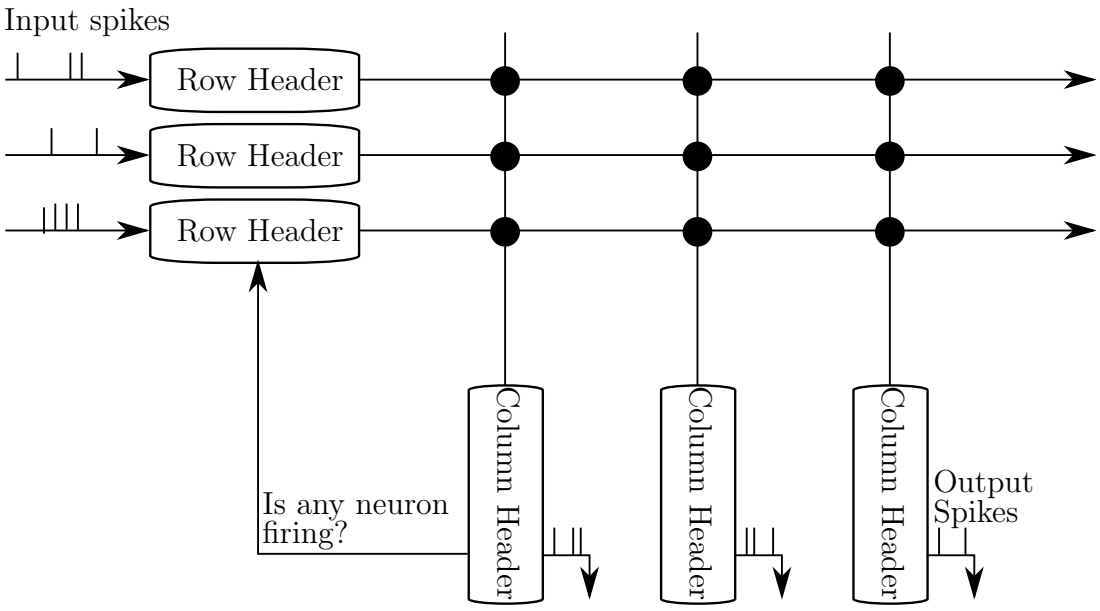

Figure 2.4: The proposed Simple, Spiking Locally Competitive Algorithm architecture. Note that neurons do not need any communication or configuration from other neurons, unlike the LCA and SLCA. The only state shared within the architecture is a single bit indicating whether or not any neuron is currently firing. Black dots at the crossbar junctions represent memristive devices. Row headers were, in this work, a simple passthrough for the input spikes. In future work, the row header would be responsible for providing inhibition amongst neurons without requiring increased network connectivity; this is discussed in Section 2.3. Column headers in this work were a simple capacitor, a switch to drain the capacitor instantly in the event that any neuron fires, and a Schmitt Trigger which detected when the capacitor's voltage exceeding a firing threshold, triggering an output spike. In a true hardware implementation, both row headers and column headers would be responsible for providing write voltages to update the memristive devices as part of the training cycle.

devices to implement inhibition [30]. The primary fault of these architectures is a lack of scalability: even though power consumption does not scale as M^2, the area required to implement the circuit does.

SSLCA avoids this issue by opting to eventually implement inhibition through the memristive crossbar itself rather than with additional hardware. Neurons and inputs are handled by row and column headers attached directly to the memristive crossbar that do not communicate with one another. This avoids the aforementioned scalability issue, while also being a very simple architecture to reason about. The resulting architecture is demonstrated in Fig. 2.4. The SSLCA consists of row headers between the input spikes and a memristive crossbar, feeding into column headers (neurons) which fire when sufficiently stimulated. The row headers in this work were implemented as simple passthroughs; in a real architecture, the row header would be responsible for allowing input spikes through during encoding as well as setting appropriate voltages to change memristor states during training. The column headers in this work were implemented as a capacitor attached directly to the crossbar, a Schmitt Trigger to detect when the capacitor is sufficiently charged to trigger an output spike, and a switch to drain the capacitor instantly when any neuron fires. In a real architecture, the column header would also be responsible for setting appropriate voltages to write the memristive devices as needed by the learning algorithm.

One downside of the SSLCA implementation used in this work is that, unlike the LCA or SLCA, there is no mechanism for inhibition amongst neurons. Every neuron firing is independent of all previous firings, meaning that there was limited collaboration during the tasks investigated in this work (e.g., Chapter 3). The SSLCA's architecture does not preclude this functionality: the row header may be

modified by using a PMOS transistor to gate input spikes, where the gate of the transistor is charged through the memristive crossbar when a neuron fires. Since connections between neurons and input rows have higher conductance when the associated input is an important part of the associated receptive field, the result would be that inputs that are well-represented by the current spiking pattern would have subsequent input spikes dampened, allowing other neurons to fire in response to the patterns in the input not covered by previous neuron firings. The implementation and tuning of this part of the network was outside of the scope of this work, and as such was not included.

For a crossbar with neuron capacitors directly attached to the nanowires, and all input lines treated as voltage sources, the capacitor charges may be solved directly. Given a C for the capacitance of a neuron, V_{neuron} as the voltage of that capacitor, and for each input row a voltage V_i and a conductance G_i provided by a memristive device connecting the two nanowires, the general formula is:

$$C\frac{\partial V_{neuron}}{\partial t} = \sum_i (V_i - V_{neuron})G_i.$$

By assuming an input row i spikes to voltage V_{set} with a mean of K_i activity (on for K_i, off for $1 - K_i$), and is grounded the rest of the time, this becomes:

$$
\begin{aligned}
C\frac{\partial V_{neuron}}{\partial t} &= \sum_i \Big(K_i(V_{set} - V_{neuron})G_i \\
&\qquad + (1 - K_i)(0 - V_{neuron})G_i \Big), \\
&= \sum_i (K_i V_{set} - V_{neuron})G_i, \\
&= V_{set} \sum_i K_i G_i - V_{neuron} \sum_i G_i.
\end{aligned}
$$

The Laplace transform may be used to solve this:

$$Q_1 = \sum_i G_i, \tag{2.2}$$

$$Q_2 = V_{set} \sum_i K_i G_i, \tag{2.3}$$

$$
\begin{aligned}
\mathcal{L}\{V_{neuron}\}s(Cs + Q_1) &= CsV_{neuron,t=0} + Q_2, \\
V_{neuron}(t) &= \frac{Q_2}{Q_1}(1 - e^{\frac{-tQ_1}{C}}) + V_{neuron,t=0}e^{\frac{-tQ_1}{C}}.
\end{aligned} \tag{2.4}
$$

Using this equation to parameterize the network required fixing all but one of the variables. To do this, the first step was to establish sensible values for Q_1 and Q_2. From Eqs. (2.2) and (2.3), Q_1 is the full conductance of the row, and Q_2 is a combination of V_{set}, which is fixed based on the crossbar's specification, and each element's conductance multiplied by the anticipated activity of that element's input row. As illustrated in Section 2.1, the training algorithm used to derive the dictionaries in this work attempts to minimize the reconstruction error. If an image is processed by the network repeatedly such that the resulting sparse code

consists of a single, non-zero coefficient, then the training algorithm will adapt the dictionary element corresponding to the non-zero coefficient to be a perfect representation of the input image. Assuming therefore that the average response should have a single spike at time t, the asymptotic behavior of each column will be itself a representation of the input vector. Therefore, the conductance G_i and the input intensity K_i will be linearly related. By letting $G_i = G_{max} g_i$ and $K_i = K_{max} k_i$, we can define some distribution χ that describes the distribution of values in the input space. For a network with M inputs, Q_1 and Q_2 can then be asymptotically inferred to be:

$$\begin{aligned} Q_1 &= M G_{max} \bar{\chi}, \\ Q_2 &= M V_{set} K_{max} G_{max} \overline{\chi^2}. \end{aligned} \tag{2.5}$$

In practice, χ was approximated as a beta distribution with the same mean as the set of expected input values. The effects of changing the mean of χ are shown in Fig. 2.5.

To fully parametrize the network, C and t must also be defined. Setting $t = t_{avgFire}$ (the desired average time between spikes) enables the calculation of $V_{neuron}(t_{avgFire})$, a threshold voltage that will spike in the average, trained case at the desired rate. Furthermore, leaving the capacitance C free is beneficial as it allows us to trade between stability and accuracy. Low values of C produce higher firing voltages (Eq. (2.4)) which may be beneficial to overcome op-amp input offset voltages, but the capacitor charges quicker, relying more on sporadic activity and less on the average patterns of the input spikes. The relationship between firing

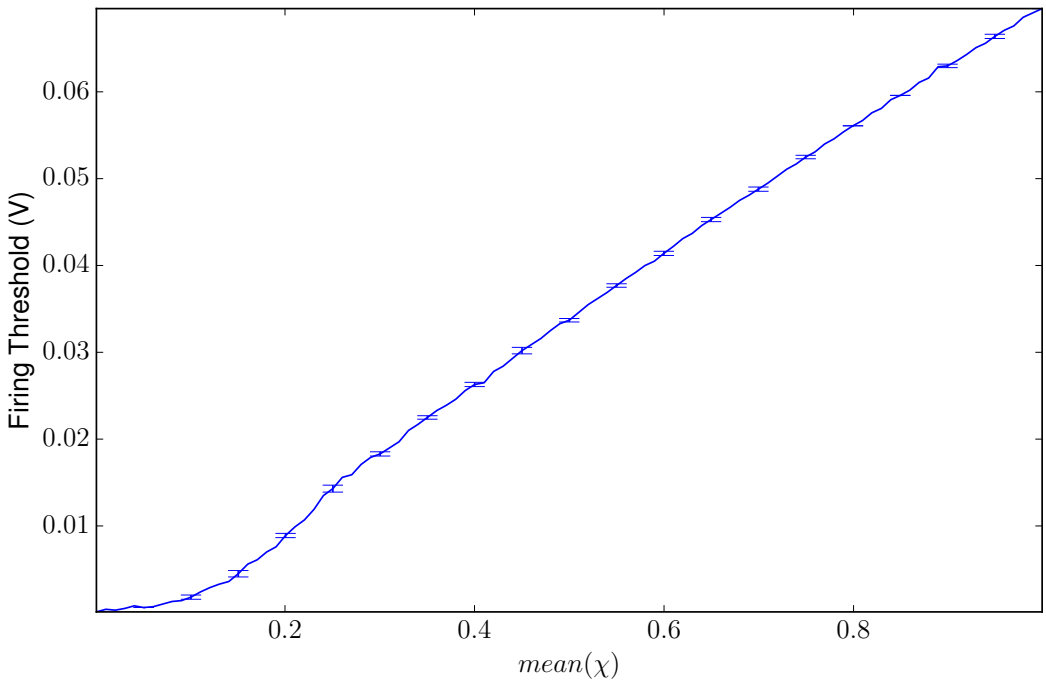

Figure 2.5: Plot showing the relation of $\bar{\chi}$ to V_{neuron} to fire after 1 ns. Error bars are from estimation of χ for Q_1 and Q_2.

voltage and capacitance is demonstrated in Fig. 2.6.

A statistic worth knowing about the chosen firing threshold is the ratio of time it takes an untrained (randomly initialized) neuron to fire when compared to a trained neuron. Equation (2.4) can be rearranged to solve for t when $V_{neuron,t=0} = 0$, yielding:

$$t = \frac{-C}{Q_1} ln \left(1 - V_{neuron}(t)\frac{Q_1}{Q_2}\right). \tag{2.6}$$

Taking the ratio of $t_{untrained}$ to $t_{trained}$ (which will have different values for Q_1 and Q_2) reveals that capacitance does not affect the ratio of firing between an untrained neuron and a trained neuron. The only relevant variables for the ratio

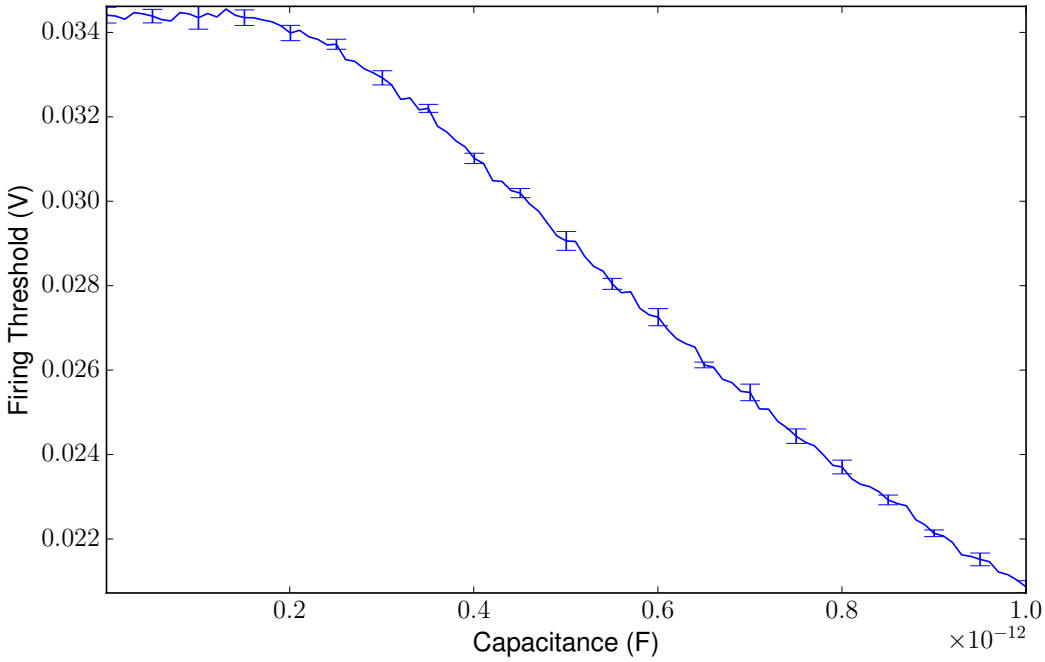

Figure 2.6: Plot demonstrating effect of specifying capacitance on the trigger voltage used to determine if a neuron is firing, while keeping an average firing rate of 1 GHz. Error bars are from estimation of χ for Q_1 and Q_2.

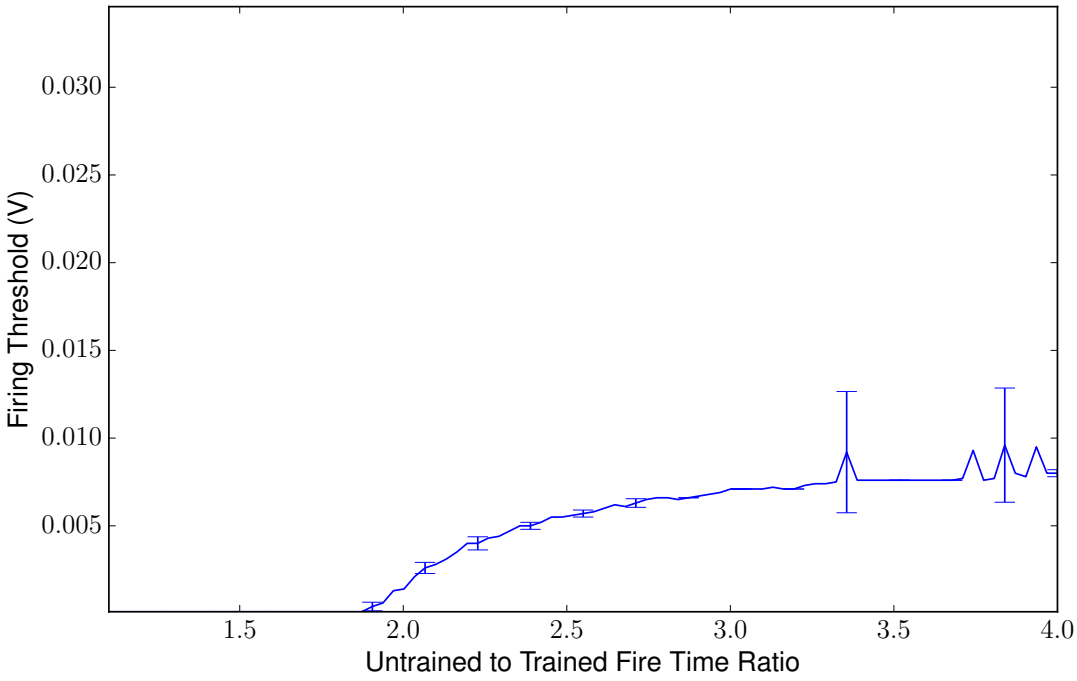

Figure 2.7: The firing thresholds required to achieve different untrained to trained firing time ratios. This is dependent on the statistics of the dataset. Instability in firing thresholds with a high untrained to trained fire time ratio is caused by very small differences in voltage creating a large difference in ratio as the trigger voltage approaches the maximum neuron voltage of Q_2/Q_1 (from Fig. 2.7). Approaching this limit is also what causes the firing threshold to flatten out as on the right side of the graph.

are the firing threshold, V_{neuron}, and the receptive field estimators Q_1 and Q_2. The trained estimators are evaluated identically to Eq. (2.5); untrained estimators may be obtained by modifying Q_2 to use $\overline{\chi_1}\,\overline{\chi_2}$ in place of $\overline{\chi^2}$, indicating that the receptive field and the input do not match. The effect of different firing thresholds on the firing ratio is explored in Fig. 2.7.

The neuron trigger voltage, $V_{fire} = V_{neuron}(t_{avgFire})$, may thus be parametrized by area (choosing C) or by op-amp bias voltage (choosing V_{fire} directly and solving for C; Eq. (2.4)). The chosen trigger voltage can be evaluated for viability by

calculating the untrained to trained firing time ratio, ensuring that this does not exceed the expected number of spikes (Eq. (2.6)). What remains is a network that is fully parametrized based on the desired performance characteristics and the physical properties of the network and input datasets (resistive range of the memristive device and the expected statistics of the input data). For greater resolution in the output, the network must only be simulated for a longer time.

The experiments in this work were parametrized by the average number of output spikes (spike resolution) and the relative density of the input spikes (spike density). These concepts are demonstrated in Fig. 2.8. In practice, spike resolution would dictate the quotient of the simulation time and $t_{avgFire}$. Spike density would dictate the duty cycle of input spikes coming from a signal of maximum intensity, and is identical to K_{max}. Experimental parameters are explored more in Section 3.2.3.

Dictionary updates with the SSLCA were accomplished identically to the LCA and SLCA: Oja's rule produced changes in the weight matrix based on the residual, using ADADELTA to adjust the learning rate.

2.4 Memristor

There are many different memristor models available; 14 of these were surveyed in my recent collaboration for NANOARCH 2015 [36]. These models differ in terms of resistive range, switching characteristics, physical viability, and a number of other factors. That work investigated each model in the context of a 500 MHz clock, with the goal of accomplishing state transitions within a single clock cycle (2 ns). Reproduced in Table 2.1 are the memristors models surveyed as well as their read characteristics. Figure 2.9 demonstrates how some memristor models

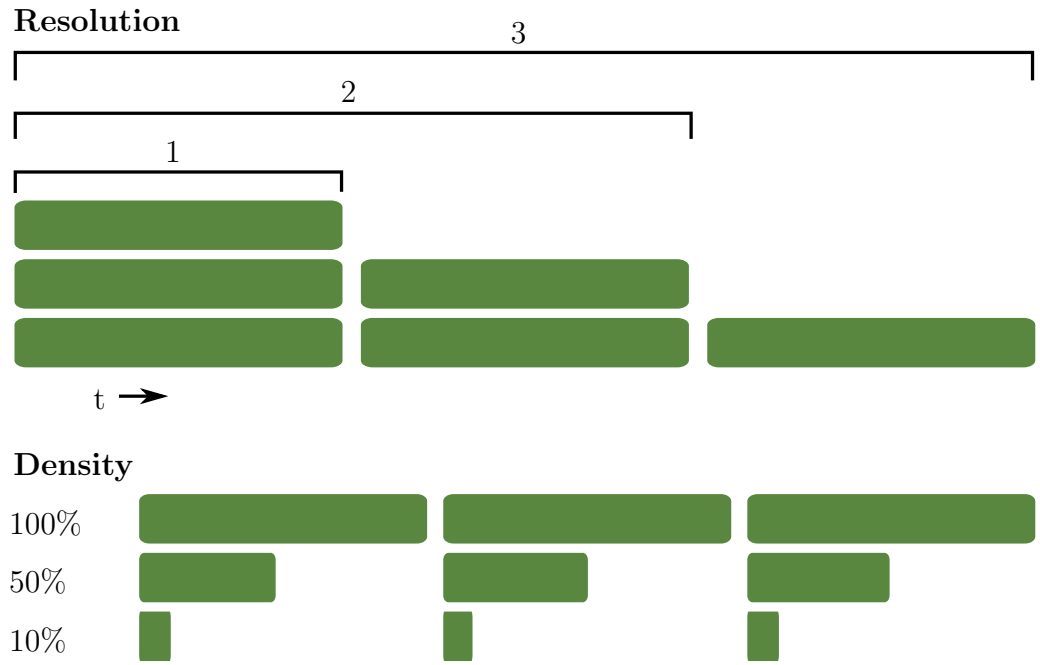

Figure 2.8: The SSLCA is parametrized according to Spike Resolution (average number of spikes in response to an input stimulus) and Spike Density (duty cycle of an input signal with maximum intensity). Spike Resolution is determined by the product of simulation time and spike rate; running the same spike rate for longer will generate more spikes, enhancing the resolution of the output. Spike Resolution is divided out when computing the analog-equivalent output of the SSLCA, such that a neuron with an output of 1 means that it was the only neuron that spiked. Spike Density dictates what percentage of the time a spiking line (input or output) will be active high when it is maximally saturated. For example, a Spike Density of 100% would look like a DC voltage at maximum input, and a square wave with equal high and low times for an input of 50%. With a Spike Density of 50%, a maximum input would look like the aforementioned square wave, while an input of 50% would create a square wave with a low time three times as large as its high time.

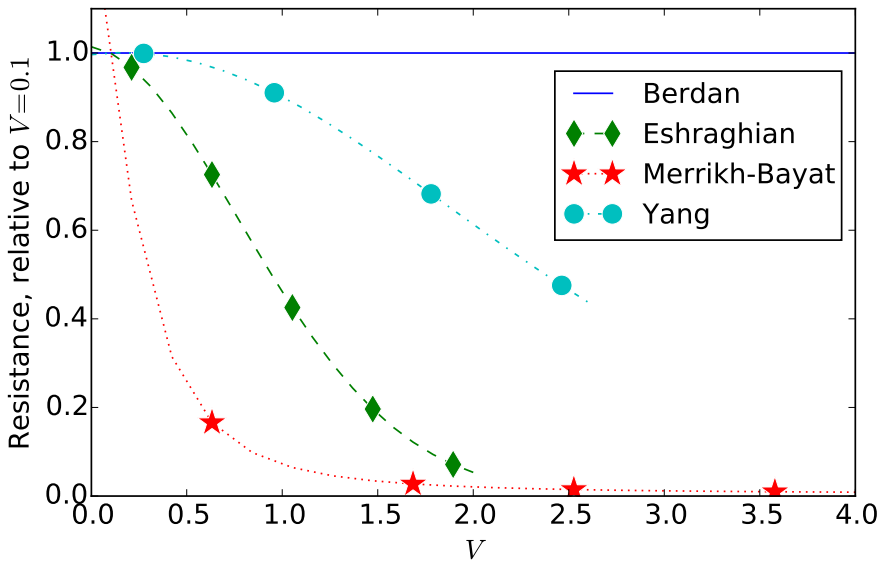

Figure 2.9: Device resistance varies for some models as voltage increases, while for other models it remains the same. This makes it very important for crossbar architectures to standardize on a consistent read voltage V_{read} when discussing resistances, or account for the non-linearity. Yang *et al.*'s model breaks down around 2.6 V, causing that line to abruptly end. Eshraghian's model breaks down around 2 V in our experiments. Ratios for all models between 0.1 V and their breakdown voltages (or 4 V) are in Table 2.2.

change restistance when exposed to different voltages. Table 2.2 shows the power consumption from a crossbar configuration learning the MNIST dataset, the same task as in Chapter 4.

For the experiments in this thesis, it was desirable to use a memristor that was based on a physical device (such that a fabricated version would have similar performance to simulations), demonstrated a quick switching time (suitable for 500 MHz), and was low power. Out of all of the models surveyed, the Yang *et al.* model best satisfies these three criteria, and so was chosen for this work.

One problem that a hardware implementation with memristors might also face is an inability to precisely represent different weight values. In my other published work, this is investigated in detail [34,35]. That research looks at how fine of control

Table 2.1: Models studied and their characteristics

Memristor	Type	$max(V_{read})$ (V)	R_{max} (kΩ)	R_{min} (kΩ)
Batas [2]	*	4.0	87	5.5
Berdan ***[4]***	TiO_2	4.0	94	5.0
Biolek [5]	*	3.9	9.4	0.59
Merrikh-Bayat ***[19]***	TiO_2	1.2	280	17
Eshraghian [7]	†	0.031	1 400 000	79 000
Lehtonen [16]	*	0.82	410 000	28 000
Pershin [25]	*	0.000 007 0	9.6	1.5
TEAM [14]	*	0.48	0.13	0.061
Yang ***[39]***	$Ag, Cu; TiO_2$	1.4	180	54
Jo ***[11]***	Ag/Si	4.0	370 000	24 000
Miao ***[20]***	TaO_x	4.0	17	1.1
Miller ***[21]***	TiO_2	4.0	1.4	0.085
Oblea ***[22]***	$Ge2Se3; Ag$	0.32	11	0.70
Jo & Lu ***[12]***	Ag/Si	2.9	12	0.72

* These models are not based on a physical device. Models that are based on a physical device will have their group name **in bold** in all tables from this paper.

† *Eshraghian* is based on experimental data from a physical device, but they did not have access to the device to further test their model.

Each device in the survey is listed here along with its chemistry, if applicable. $max(V_{read})$ is defined as the positive voltage at which 1000 cycles on a 500 MHz clock will produce a change in the logical weight W of the device (bounded on $[0, 1]$) of 0.01, or 4 V, whichever is smaller. R_{max} and R_{min} are the states of the device when mapped to $W = 0$ and $W = 1$, respectively. The range represented by these values is 90% of the device's physical limits. The overall range was constrained to prevent excessive switching times at the extremes. Resistances were evaluated at $V_{read} = min(0.1\,\text{V}, max(V_{read}))$, in order to better allow comparisons between devices that change resistance with voltage. Table 2.2 and Fig. 2.9 explore the effects of voltage on resistance for different models.

is needed over the memristor's state in order to effectively accomplish the sparse coding task. The results showed that 16 states, or 4-bit resolution, was sufficient to reasonably reconstruct analog datasets and perform well on the MNIST task.

While precision affects hardware realizations of memristive algorithms, it was not considered as part of this work's simulations. Having already investigated the

Table 2.2: Power during crossbar evaluation

Memristor	$\frac{R(V_{max})}{R(0.1)}$	V_{read} (V)	Power (µW)
Batas	1.0	0.10	68
Berdan	1.0	0.10	74
Biolek	1.0	0.10	630
Merrikh-Bayat	0.009	0.10	22
Eshraghian	0.05	0.031	0.000 46
Lehtonen	0.000 06	0.10	0.013
Pershin	1.0	7×10^{-6}	0.000 001 4
TEAM	1.0	0.10	11 000
Yang	0.4	0.10	9.9
Jo	0.3	0.10	0.015
Miao	0.6	0.10	340
Miller	1.0	0.10	4400
Oblea	1.0	0.10	530
Jo & Lu	1.0	0.10	520

Evaluation properties of different memristor models. The second column, $\frac{R(V_{max})}{R(0.1)}$, denotes a sample ratio of each device's resistance when evaluated at the device's breakdown voltage or 4 V (whichever is smaller) and 0.1 V. A value of 1.0 indicates that $R = f(W)$, and is independent of voltage. V_{read} is the normalized read voltage used to compute power draw during crossbar evaluation, and is the minimum of $max(V_{read})$ and 0.1 V to allow for easier comparison between devices. Power is the average power consumed per output column (784 devices) from a network trained via LCA on the MNIST dataset. *Pershin*'s power is substantially lower than the others due to its incredibly low V_{read}, deriving from the fact that it is a binary model and was not constrained by actual device measurements.

effects of precision, and recognizing that each algorithm would suffer similarly, it was deemed that comparing the algorithms with analog (floating point) weights would be sufficient for this thesis.

3

Reconstruction

The quality of a sparse coding algorithm can be determined partly from how much loss occurs between the original signal and a reconstruction based on the sparse code. Significant loss between the original and the reconstruction indicates that the algorithm does a poor job retaining specific details from the input, and might be inefficient at conveying information needed for machine learning tasks further down, e.g., an image recognition pipeline. This chapter explores the three architectures presented in Chapter 2 in the context of generating sparse codes and reconstructing the original input.

3.1 Methodology

Sparse coding involves translating an input signal to a different basis which can represent those input signals with fewer non-zero coefficients. This sparse basis may be translated back into the original signal's space, resulting in a reconstruction of the original signal. The difference between the original signal and the reconstruction is called the residual.

In this work, the *Normalized Root-Mean-Square Error* (NRMSE) of the residual is used to evaluate how accurately a sparse coding system can encode image patches from a *Natural Image Dataset Containing 10 Images* (NAT10; Fig. 3.1). The NRMSE's value is a representation of the inaccuracy of each individual pixel in the reconstructed patch, with extra weight given to outliers (because it is calculated as an L^2 norm).

Figure 3.1: Example images from the NAT10 dataset used to evaluate image reconstruction in this work. Each image in this dataset was scaled to 128×128 pixels with full RGB color.

Each of the 10 images in the NAT10 dataset used for this task were divided into patches of 8×8 non-overlapping regions. Of these patches, 2048 were used to train LCA, SLCA, and SSLCA models. Training patches were shuffled to avoid over-training a specific feature (e.g. the color blue) too early. The remaining 512 patches were used to evaluate the reconstruction fidelity of each algorithm at various points throughout training. Training occurred across two iterations of the dataset to give algorithms ample time to demonstrate asymptotic behavior. All experiments were repeated 5 times; error bars shown indicate the standard deviation.

Experiments were run using the job_stream parallelization library (Appendix A) and organized using the git-results plugin (Appendix B); both of these packages were products of my work leading up to this thesis.

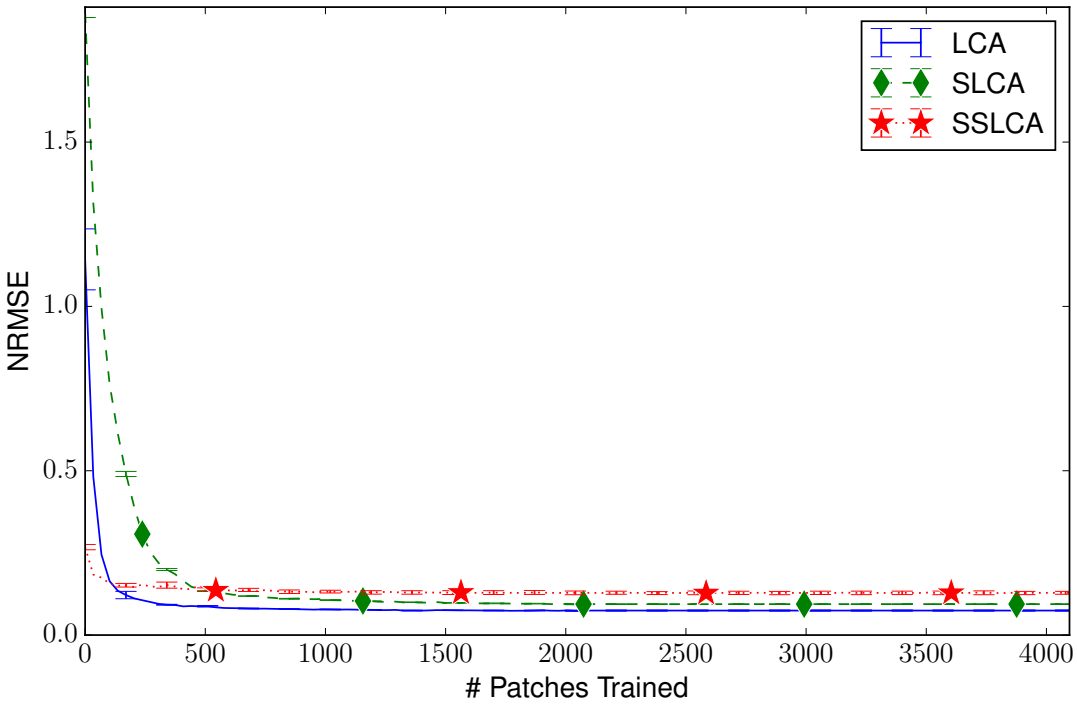

Figure 3.2: Reconstruction performance throughout training on 8×8 patches of the NAT10 dataset. The analog, non-spiking algorithm performed the best, followed closely by Shapero *et al.*'s spiking algorithm. The architecture designed for this work performed marginally worse. The reasons for this are addressed in Section 3.2.3.

3.2 Results

NRMSE results can be seen in Fig. 3.2. The corresponding activity (the portion of active neurons contributing non-zero coefficients to the sparse code) is demonstrated in Fig. 3.3. An analysis of each algorithm's performance follows.

3.2.1 LCA

The LCA implemented for this work had no accuracy restraints other than the chosen sparsity and the number of neurons. That sparsity is shown through output

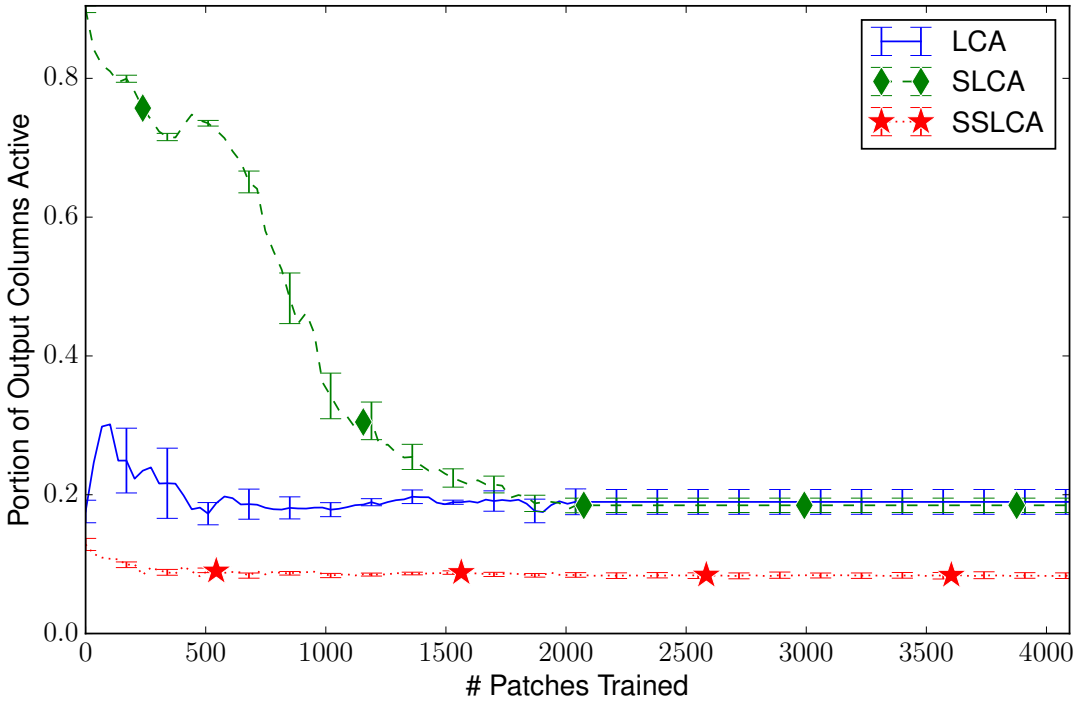

Figure 3.3: Activity (portion of neurons actively contributing to output coefficients) for each algorithm when reconstructing 8×8 patches of the NAT10 dataset. Algorithms exhibiting higher activity produced less sparse of an encoding; that is, they are likely to have a lower NRMSE because more neurons were contributing to each reconstruction. Higher activity allows neurons to learn parts of an image rather than an entire image. Too high of activity leads to each neuron learning a single element of the input - essentially re-encoding the input identically to how it was presented. Algorithm parameters (λ for LCA and SLCA) were adjusted to target 20% activity. SSLCA had no parameter to adjust this metric, which is discussed in Section 3.2.3.

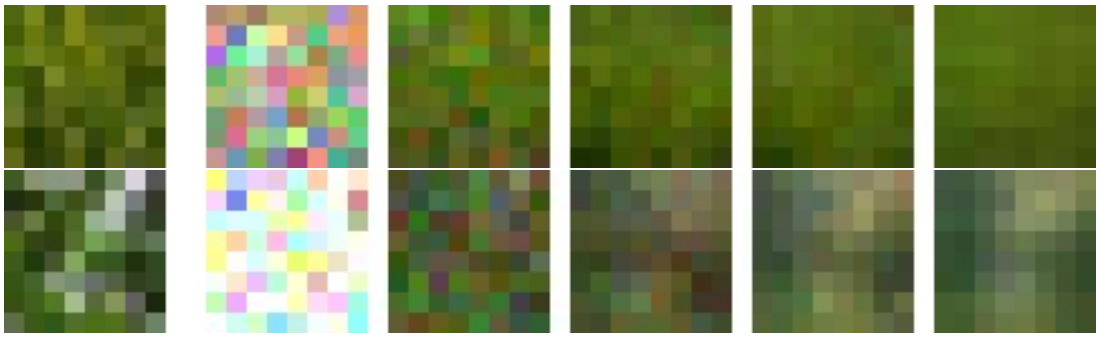

Figure 3.4: Progression of LCA training for two different testing patches; leftmost is the target patch. The five following images are after 34, 136, 644, 1088, and 4096 patches trained. An excess of activity initially (Fig. 3.3) lead to excessive brightness early in training. This was quickly learned out, and the final reproduction had a similar color quality to the original patch. The LCA also did a reasonable job of reconstructing a bright streak in the second example patch.

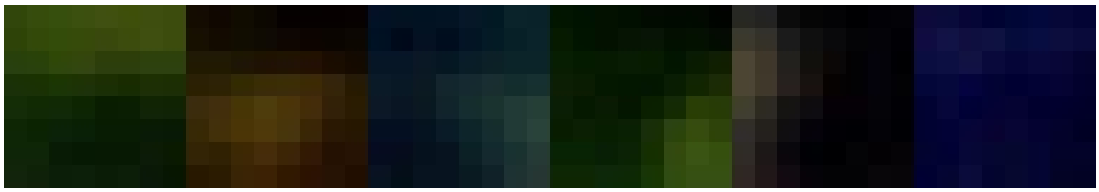

Figure 3.5: Example elements from the final dictionary for the LCA. Note how the 20% activity indicated in Fig. 3.3 leads to each neuron representing a large swatch of color at a specific location. Several of these neurons added together can successfully reproduce broad characteristics from any input.

activity in Fig. 3.3. Performance of the LCA is the optimal baseline for reconstruction in this work. The spiking variants presented attempt to approximate the algorithm (which includes local competition), but as they are spiking and not analog, the resolution of each neuron's output response is diminished. The progression of the LCA reproducing a particular test patch throughout the training process can be seen in Fig. 3.4.

The LCA utilizes competition between neurons and prevents overrepresentation of the input by including an inhibition term based on the other outputs. This resulted in each receptive field learning to represent a distinct position and color quality of each patch (Fig. 3.5). Since the algorithms in this work only have 50 receptive fields to train, while there are $8 \times 8 \times 3 = 192$ inputs, the ability to effectively combine several receptive fields during reconstruction helps the LCA achieve a final NRMSE of 0.074, substantially lower than the SSLCA's 0.13 or the SLCA's 0.095.

It should be noted that in some instances of sparse coding, it is expected that the learned receptive fields will be gabor filters [26]. Other works force gabor filters onto the learned fields [18]. However, to achieve these patterns, the learned weights must be allowed to go negative as well as positive. This work enforced non-negative weights as well as inputs, as this is easier to reason about and produces simpler hardware implementations. Versions of these architectures featuring both negative inputs and weights have been investigated in other works [35]. For convenience, Fig. 2.1 demonstrates the necessary modifications to compensate for negative weights and inputs in an LCA architecture.

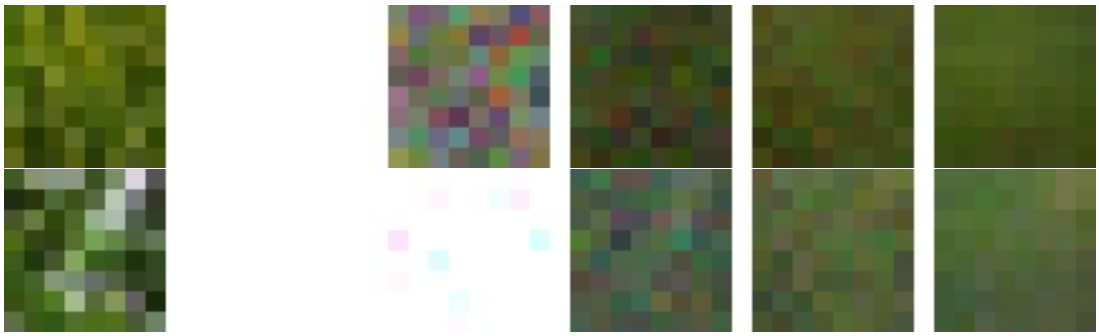

Figure 3.6: Progression of SLCA training on two different test patches: leftmost is the target patch, and the remaining five images are reconstructions after 34, 136, 644, 1088, and 4096 training patches. Note that the first reconstruction after 34 patches is clamped to all-white. Similarly to the LCA (Fig. 3.4), an initial pattern of over-activity lead to excessive reconstruction brightness. As training progressed, the reconstructions approached a smoothed version of the original input, very similarly to the LCA. The main difference between LCA and SLCA is that SLCA takes longer to converge. The SLCA also demonstrated slightly poorer resolution, lightening the upper-right corner on the second test patch but not reproducing the streak.

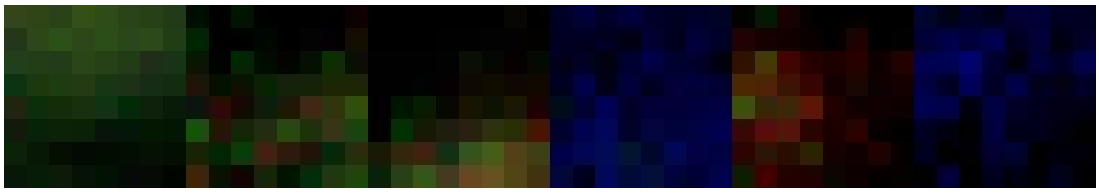

Figure 3.7: Example elements from the final dictionary for the SLCA. Similarly to the LCA, the 20% target activity lead to each neuron representing large patches of color at different locations. The SLCA's patches are less smoothed than the LCA's in large part due to the SLCA exhausting more training examples before converging to 20% activity.

3.2.2 SLCA

Shapero *et al.*'s Spiking LCA performed fairly well, achieving a final NRMSE of 0.095. It converged much more slowly than either the LCA or SSLCA (Fig. 3.2). Like the LCA, the SLCA implements inhibition in terms of activity amongst other neurons. This lead to the receptive fields representing patches of color at different locations, combining several receptive fields to yield an accurate reconstruction (Figs. 3.6 and 3.7).

Overall, performance for the SLCA was quite similar to the LCA. There are two reasons that the SLCA does not reach the LCA's performance. The first reason is that counting spikes has a much more coarse resolution than the analog values used by the LCA. The second reason is that inhibition can only occur after a spike is already triggered. Initially, this leads to much higher activity than that seen in the LCA. Once neurons have trained to represent more specific receptive fields, this does not pose a problem for the algorithm. Still, this added substantially to the initial training time needed by the SLCA to reach peak performance.

3.2.3 SSLCA

The SSLCA reconstructed patches less accurately than both LCA and SLCA. While less accurate, it converged the fastest and also produced significantly more sparse encodings (Figs. 3.2 and 3.3). The NRMSE of 0.13 indicates that the average RGB pixel value was off by 13% of the spectrum, which is significant. However, the improved sparsity might be advantageous.

The main shortcoming of the SSLCA as implemented for this work was that it did not address inhibition as a result of representation by other active neurons. This lead to overrepresentation of certain image areas, and meant that each neuron

Figure 3.8: Progression of SSLCA training on two different test patches; leftmost is the target patch, and the remaining five images are reconstructions after 34, 136, 644, 1088, and 4096 training patches. Unlike LCA and SLCA (Figs. 3.4 and 3.6), the SSLCA matches brightness and color very quickly. This is partly due to the algorithm being configured using the average statistics of the input dataset. Lower activity also contributes; the SSLCA exhibits only 8% activity versus the LCA and SLCA's 20% (Fig. 3.3). Since there are fewer non-zero coefficients, Oja's rule dictates that fewer neurons get trained each step, resulting in faster convergence. The second test patch is notably worse with SSLCA; the changing of color between the last two reproductions indicates that one of the 50 output neurons was contested, a side-effect of an inadequate number of neurons participating in the reconstructions.

Figure 3.9: Example elements from the final dictionary for the SSLCA. Unlike the LCA or SLCA where dictionary elements represent locational patches of color, the SSLCA elements come to represent whole images. This results from higher output sparsity (Fig. 3.3), which occurs because the SSLCA as implemented in this paper has no means of inhibiting parts of the input that are already represented by previous neuronal firings. Another aspect worthy of note: these dictionary elements are far brighter than those of either the LCA or SLCA (Figs. 3.5 and 3.7). This is due to the firing threshold V_{neuron} being determined from a perfect match to the input. This is further explained in Section 3.2.3.

firing needed to represent the image as a whole rather than a part of the input image. In other words, multiple receptive fields did not collaborate significantly to reproduce the input; this is evident from the final receptive fields, shown in Fig. 3.9. Each receptive field represented a whole image rather than a color and location pairing as was the case with LCA and SLCA. Steps to address this difficiency are discussed in both Section 2.3 and Chapter 7; briefly, dampening the effects of input spikes when that particular input is well-represented by the current spiking pattern should suffice.

Another aspect of the SSLCA reconstructions worth noticing is how much brighter the final dictionary elements were than the input patches (compare Fig. 3.9 to Fig. 3.8). This was because the firing threshold V_{neuron} was determined based on an optimal match between the input and the receptive field (Section 2.3). The training set and the test set both violated this assumption, resulting in a lower average spiking rate than was anticipated. Since reconstructions are based on the product of the dictionary elements and the spike rate, Oja's rule translated lower spike rates into brighter receptive fields. This could be corrected in future work by making more accurate assumptions for Q_1 and Q_2, recognizing the collaboration between several neurons for the ideal (trained) case.

3.3 Discussion

The final NRMSEs for the LCA, SLCA, and SSLCA were 0.074, 0.095, and 0.13, respectively. The primary difference between the LCA and SLCA was revealed to be due to a difference in spiking resolution as well as the inability for a spiking algorithm to implement inhibition before any spikes fire. The SSLCA performed worse than either of the other algorithms as it did not represent inhibition on

account of patterns in the input that were already represented by previous spikes. Even so, all three algorithms successfully reproduced the main characteristics of each patch, and the final NRMSEs were reasonably close together. Chapter 4 looks at the effect of these reconstruction quality differences on a machine learning task.

4

Classification of Handwritten Digits

While the LCA family of algorithms were designed to minimize the sum of the error in the residual and a sparsity term [27], this does not guarantee the transmission of information necessary for machine learning tasks. Details that are significant for reconstruction may not aid classification, and broad patterns that might not significantly affect a reconstruction's NRMSE might be important for classification. Keeping the reconstruction qualities of each algorithm in mind, this chapter investigates each architecture's ability to provide information needed to classify handwritten digits in the MNIST database.

4.1 Methodology

A sparse coding algorithm's ability to accurately encode the input signal is not necessarily related to its ability to retain meaningful information for classification. To demonstrate this, the sparse code from each algorithm explored in this work was passed to a *Single-Layer Perceptron Network* (SLP) which was trained to classify digits from the MNIST handwritten digit database [15]. The MNIST database has 60 000 digits, 50 000 of which are used for training, and 10 000 of which are used for testing.

Similarly to Chapter 3, each algorithm had 50 dictionary elements and was trained across two iterations of the MNIST database and evaluated for classification accuracy. The scaling of performance across number of nodes was demonstrated in prior work [35]; this work focused on the differences between algorithms instead.

For this reason, 50 output neurons were used for each algorithm. The classification score presented is the percentage of correct classifications from the SLP based on the training data. All experiments were repeated 5 times; error bars shown indicate the standard deviation.

Experiments were run using the job_stream parallelization library (Appendix A) and organized using the git-results plugin (Appendix B); both of these packages were products of my work leading up to this thesis.

4.2 Results

Classification results are shown in Fig. 4.1. These are supported by the NRMSE on this task in Fig. 4.2 and the activity for the encoding passed to the supervised layer in Fig. 4.3. Each algorithm's performance is discussed in the subsequent sections.

4.2.1 SLP

For a baseline, the raw MNIST data was passed directly to the SLP layer used for classification. Figure 4.1 clearly demonstrates that an SLP network using the pixel data as inputs outperformed the other algorithms presented in this work. This is due to the restriction that those algorithms were limited to 50 neurons, and thus 50 receptive fields. In other words, the resolution of the sparse codes was not sufficient for the task at hand. Performance was not significantly lower for any sparse coding algorithm than the SLP on its own though, and the number is sufficient to compare the unsupervised algorithms to one another.

In practice, the sparse coding algorithms with only 50 neurons might still provide benefits over the SLP on its own, as they would only need to transmit a sparse

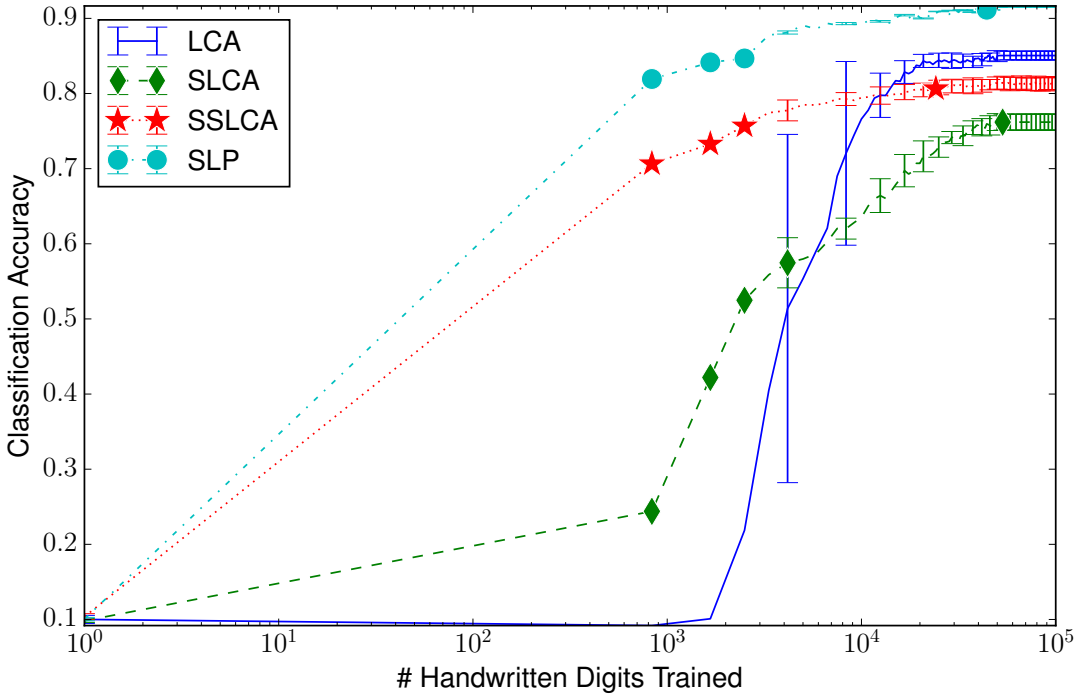

Figure 4.1: Classification performance throughout training on the MNIST dataset with 50 neurons. Performance of the SLP being superior to the others was a result of 50 neurons being too small for the task at hand; however, for a comparative analysis between algorithms doing the same thing, the number was sufficient. Ultimately, the LCA outperformed both the SLCA and SSLCA. More interestingly, the SSLCA significantly outperformed the SLCA, even though the SLCA had higher activity and produced a similar NRMSE on this task (Figs. 4.2 and 4.3). This phenomenon is discussed between Figs. 4.6 and 4.8.

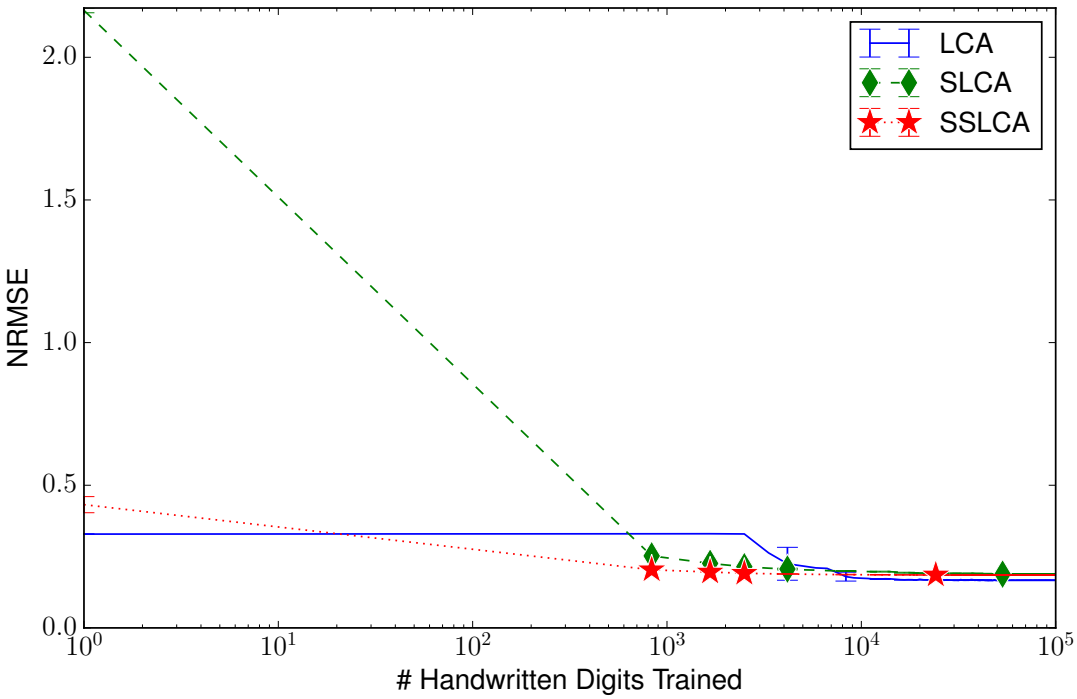

Figure 4.2: Reconstruction performance throughout training on the MNIST dataset. All algorithms settle to a similar NRMSE despite very different levels of activity (Fig. 4.3).

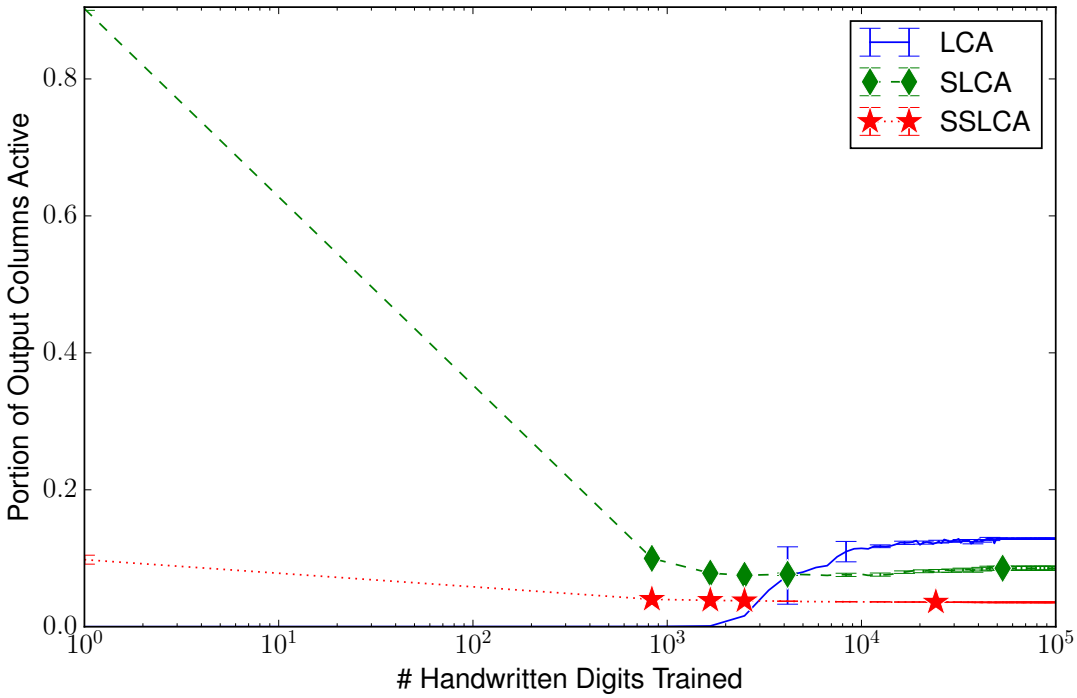

Figure 4.3: Activity of sparse coding layer throughout training on the MNIST dataset. Similar to Fig. 3.3 from Section 3.2, the LCA and SLCA both demonstrated higher activity than the SSLCA. The LCA and SLCA were both λ-adjusted in the same way, however the LCA would not go lower than 0.14 activity on this task. As shown, the λ chosen was quite high and initially suppressed most of the LCA activity. Once the algorithm adapted, its dictionary trended towards lines rather than whole digits as with the SLCA and SSLCA (Figs. 4.4, 4.6 and 4.8).

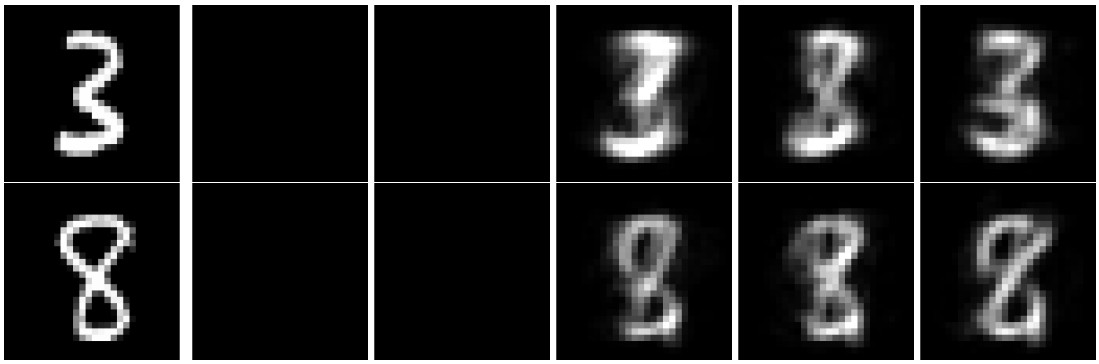

Figure 4.4: LCA reconstruction performance throughout training on the MNIST dataset. A very high λ inhibited visible reproductions early, but yielded to reasonable reconstructions. High initial λ was necessary for a fair comparison amongst algorithms; otherwise the LCA would learn large pixels and demonstrate high activity).

Figure 4.5: Sample dictionary elements from LCA after training. Even with a high λ enforcing low activity, LCA learned digit edges as opposed to the whole digits learned by the SLCA and SSLCA (Figs. 4.7 and 4.9).

subset of 50 values to a classification network rather than 784 values.

4.2.2 LCA

Classification accuracy by the LCA was decent, surpassing 85% with only 50 neurons. This matches prior results [35]. Figure 4.4 is provided to demonstrate the reconstruction quality of the LCA networks that achieved this result.

In this work, the LCA networks began with extremely low activity due to a high λ. The high initial λ was necessary to get the overall activity at the end of the experiments to better match across algorithms. Without a high λ, the LCA

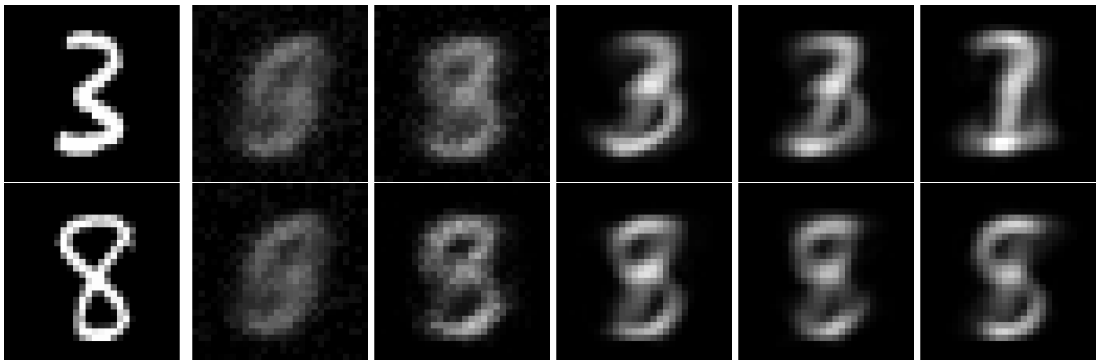

Figure 4.6: SLCA reconstruction performance throughout training on the MNIST dataset. The SLCA performed notably worse than the LCA with only marginally worse NRMSE. From these reconstructions, it is clear that the SLCA often reproduced several digits with the same neuron combination. While this helped the SLCA minimize reconstruction error in its initial high-activity state (Fig. 4.3), the ambiguity confused the SLP.

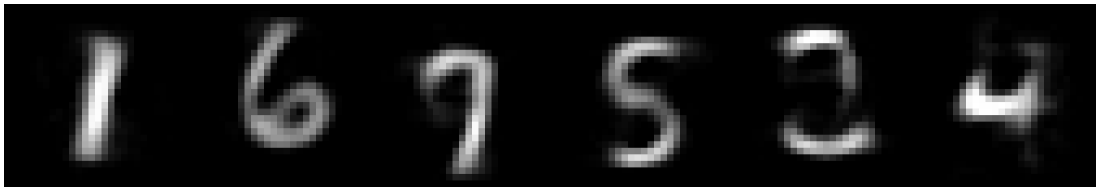

Figure 4.7: Sample dictionary elements from SLCA after training. The duality of some of these neurons is apparent; the 3^{rd} receptive field is primarily a 7, but also has the loop element from a 9. The 5^{th} element could be either a 2 or an 8. This ambiguity prevented the SLP from effectively differentiating certain digits.

would learn large dots analogous to pixels in this task, very similar to the NAT10 reconstructions from Section 3.2.1. Interestingly, while both the SLCA and SSLCA exhibit low activity and learned whole digits as their receptive fields, the LCA's final dictionary featured digit edges instead (Fig. 4.5). Even higher values of λ than the one used resulted in no activity at all.

4.2.3 SLCA

The spiking model of the LCA as proposed by Shapero *et al.* performed the worst, achieving only 76% on the classification task. This was a surprising result since its NRMSE was virtually identical to the SSLCA (Fig. 4.2) and it produced more activity (Fig. 4.3). Looking at its final dictionary in Fig. 4.7 and its reconstructions throughout training in Fig. 4.6, the reason for this poor performance is apparent: most of the neurons learned to represent a combination of two digits, confusing the SLP. While this resulted in a lower reconstruction error with the initially high activity of the SLCA, in the long term it greatly hurt the SLCA's viability for classifying digits. Realistically, this effect could have happened to any of the algorithms in this paper. What left the SLCA particularly vulnerable to it was its extremely high activity early in training, which led to many neurons learning together. The LCA and SSLCA, whose initial activities were both much lower, avoided this.

4.2.4 SSLCA

The SSLCA outperformed the SLCA but not the LCA, finishing with an average accuracy of 81% on MNIST. The main reason it outperformed the SLCA was its lower initial activity, avoiding the problem where each receptive field learned to represent more than one digit, which confuses the supervised SLP doing the actual classification. The LCA most likely outperformed the SSLCA because of its higher activity and better collaboration amongst neurons, similar to its performance characteristics on the reconstruction task in Chapter 2. Higher activity combined with meaningful parts of digits meant that the SLP received meaningful combinations from the LCA, creating more than 50 distinct combinations. The SSLCA, on

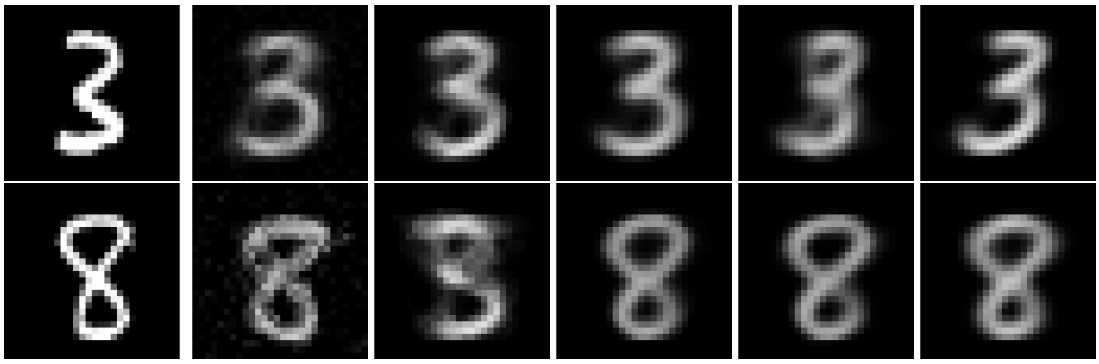

Figure 4.8: SSLCA reconstruction performance throughout training on the MNIST dataset. Unlike the LCA and SLCA which both had higher activity than the SSLCA, these reconstructions are very targeted to be a single digit. This the effect of combining Oja's rule with very low activity: each receptive field was affected by training infrequently, and learned an average representation of a specific digit. This can also be seen in Fig. 4.9.

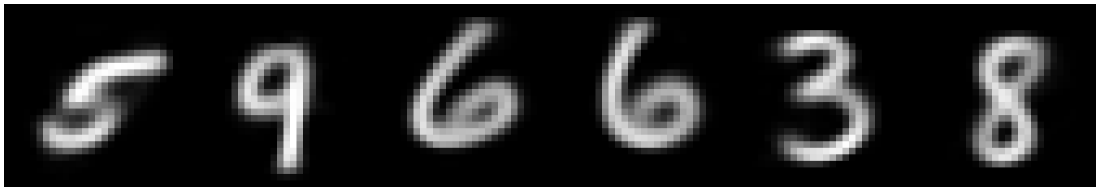

Figure 4.9: Sample dictionary elements from SSLCA after training. Unlike the LCA or SLCA, each dictionary element very clearly learns to represent a single digit. This is due to the combination of Oja's rule and lower activity, leading to each receptive field being updated fewer times, in more specific conditions.

the other hand, produced very clean receptive fields, but no combination of those receptive fields would imply a different digit classification.

4.3 Discussion

This chapter demonstrated that reconstruction performance does not necessarily correlate to an algorithm's ability to convey information necessary for classification. The LCA still performed best, with 85% accuracy. To accomplish this high figure, it leveraged multiple receptive fields working together to provide meaningful information. The SSLCA followed with 81% accuracy, demonstrating very clean receptive fields that correlated to specific digits. The SLCA performed the worst with 76% accuracy. This was shown to be a symptom of the SLCA's tendency to have very high activity early in the training process, resulting in neurons that were pulled several different directions and learned to represent hybrids of two different digits. That situation would be avoidable by using a different λ threshold, which would allow the algorithm to have a larger set of active neurons.

5

Power Consumption

The motivation for a simple, spiking algorithm stemmed not only from ease of implementation, but also from the thought that simplicity could yield substantial power savings. The premise of the design was to connect the neuron state capacitors to the memristive crossbar without an *operation amplifier* (op-amp) specifically to avoid the power cost of operating an op-amp. This chapter investigates the power consumption of the architectures presented.

5.1 Methodology

From the previously published memristor survey, the maximum read voltage for the Yang *et al.* memristive device when using 2 ns read cycles is 1.4 V [36]. However, using a lower voltage both consumes less power and produces a higher resistance in the memristive device. Therefore, the experiments in this work used 0.7 V as the read voltage. At 0.7 V, the resistive range of this device is 52 kΩ to 207 kΩ.

Power for the analog LCA was calculated assuming a virtual ground and omitting the *operational amplifiers* (op-amps). Input voltage was scaled from 0 V to 0.7 V according to the intensity of the input signal. This setup lead to the crossbar power consumption being equivalent to the sum of power at each junction in the crossbar, including the bias column. Also missing is the power draw required to subtract the bias column from each column's output. While this could be done with an op-amp for each column, another approach would be to digitize the voltage from each column, and subtract the bias' digitization from each column's in

software.

For each memristive device in the LCA, the logical weight was converted to an actual resistance based on the formula from my 2015 NANOARCH publication [36]:

$$R_{i,j} = R(W_{i,j}) = \frac{R_{max}R_{min}}{W_{i,j}R_{max} + (1 - W_{i,j})R_{min}}.$$

This equation was shown to exactly reproduce the desired dot product, contingent on the ability to precisely set individual resistances.

The SSLCA's power was calculated somewhat more precisely. Input spikes of 0.7 V were applied to input lines; while inactive, these lines are grounded. Charged capacitors are discharged after each spike. To make comparison with the LCA reasonable, the fire trigger op amp was also not included in the power consumption of this design. Spikes were applied with a maximum duty cycle of 10% (spike density 0.1).

In the SSLCA, memristive devices' resistances were set based on conductance: a maximum weight (1.0) mapped to the maximum conductance (19.2 µS). Smaller weights are the appropriately scaled-down quantities of this conductance, down to a minimum logical weight of $W = \frac{1}{207}/\frac{1}{52} = 0.251$, or 4.83 µS.

The SLCA model was not experimentally evaluated for power as part of this work, as it uses a large number of transistors which could not be simulated within the scope of this work. However, Shapero *et al.*, the proposers of the architecture, implemented both the LCA and the SLCA on a *Field-Programmable Analog Array* (FPAA) using 350 nm CMOS technology [29,30]. This was discussed in Section 1.3.1. The figures derived for a task the same size as the Reconstruction task presented in this work were 4.18 mW for the LCA and 21.0 mW for the SLCA. For the Classification task, the projections were 11.8 mW for the LCA and 79.0 mW

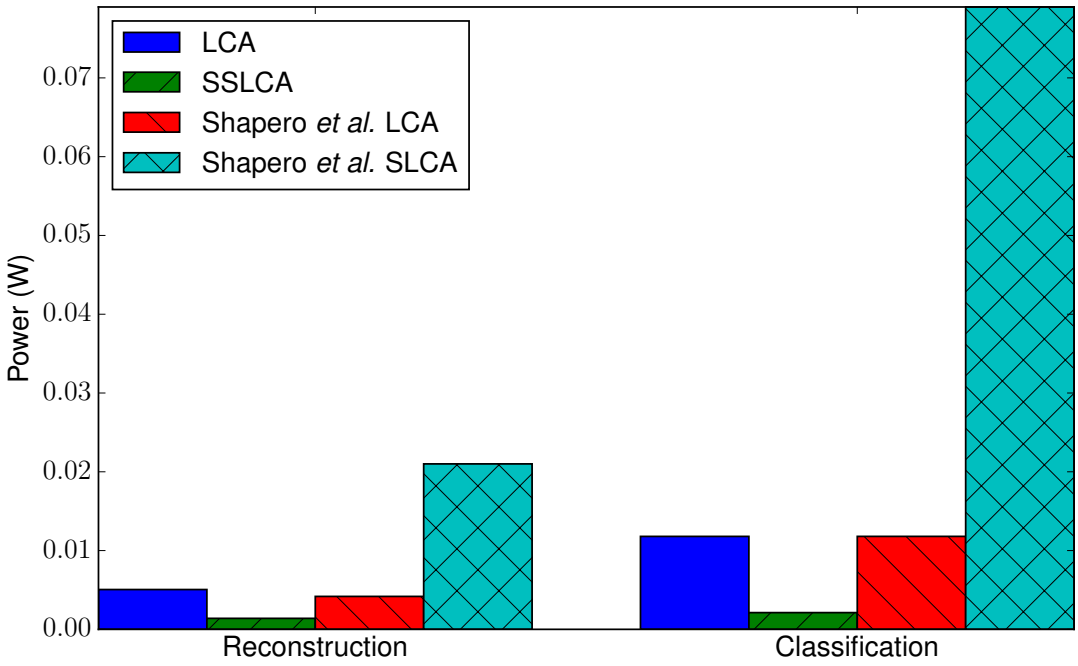

Figure 5.1: Power consumption on the Reconstruction and Classification tasks. The Reconstruction task from Chapter 3 consists of mapping an input signal of 192 values to a sparse code with 50 values; the Classification task from Chapter 4 consists of mapping an input signal with 784 values to a sparse code with 50 values.

for the SLCA.

Experiments were run using the job_stream parallelization library (Appendix A) and organized using the git-results plugin (Appendix B); both of these packages were products of my work leading up to this thesis.

5.2 Results

Figure 5.1 demonstrates the measured power consumption for the LCA and SSLCA as presented in this paper as well as the extrapolated power from Shapero *et al.*'s implementation of the LCA and SLCA. Shapero *et al.* notes that their SLCA would eventually consume less power than their LCA implementation, as SLCA's

requirements grow as $\mathcal{O}(n)$ while LCA's requirements grow as $\mathcal{O}(n\sqrt{n})$ as the number of neurons is increased [30].

Comparing the results between this work's LCA and SSLCA on the Reconstruction and Classification tasks (Fig. 5.1), it is apparent that the spiking nature of SSLCA produced substantial power savings. A maximum-valued input had a duty cycle of 10%; actual power savings were smaller because spikes always produce a full 0.7 V potential on the input line to the memristive crossbar, whereas the LCA used voltage scaling to achieve different input values. Since Ohm's law dictates $P = \frac{V^2}{R}$, voltage scaling is more effective than duty cycling. However, voltage scaling requires more complex circuitry to achieve, potentially resulting in further power penalties outside of the scope of this work.

The Shapero *et al.* estimates of power consumption on these tasks revealed that efforts to improve the accuracy of the SSLCA would be well-founded: the SSLCA consumed only 6.6% of the power as Shapero *et al.*'s SLCA and 28% of their LCA implementation. Both the SLCA presented in this work and Shapero *et al.*'s SLCA grow in complexity as $\mathcal{O}(n)$, so these savings should be consistent as the network size grows.

Another key consideration is how well the measured power for each architecture combines with each architecture's performance on the Reconstruction and Classification tasks. Each architecture's score from these sections was combined with the estimated power in Figs. 5.2 and 5.3. For the Reconstruction task, the LCA is the clear winner until power is around three times as important as NRMSE. At that point, the SSLCA's significantly lower power consumption makes it the algorithm of choice. The Classification task demonstrated a smaller spread between

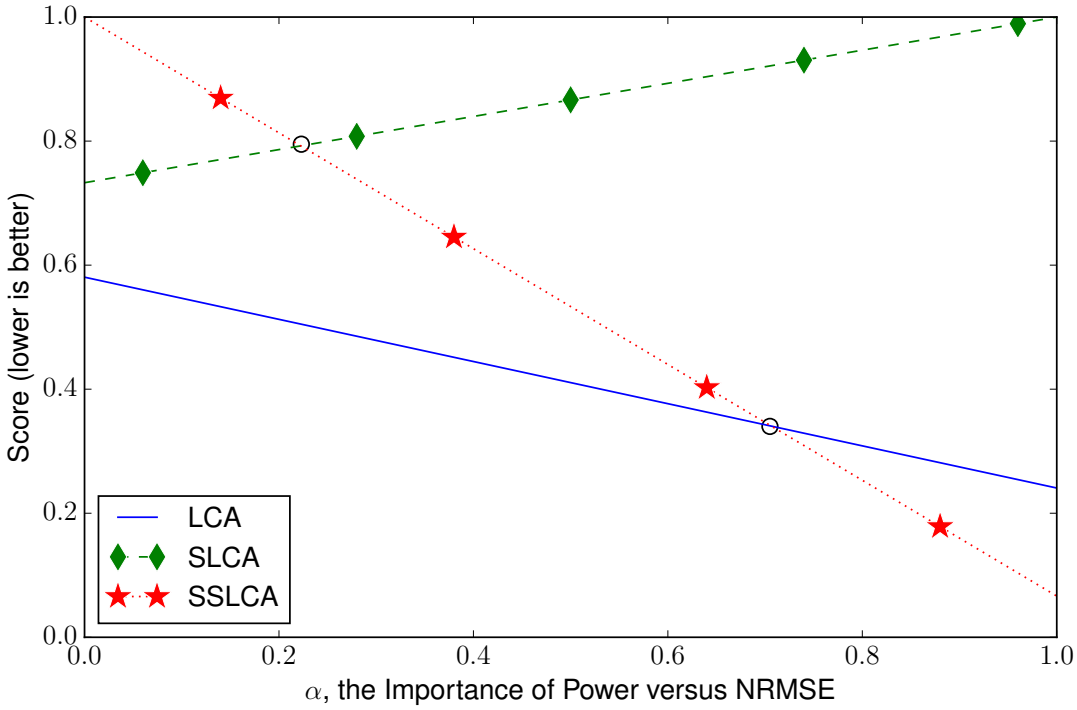

Figure 5.2: Comparative score on the Reconstruction task in this work, calculated by balancing the NRMSE from Chapter 3 against estimated power consumption for each architecture (SLCA's power comes from Shapero *et al.*). Power and NRMSE, both quantities which are worse when larger, were normalized by dividing out the worst architecture's value for each. Each quantity was then scaled by α and $1-\alpha$, respectively, to achieve the given score. LCA is the clear winner on this task, until power becomes about 70% of the importance criteria, at which point the SSLCA's substantially lower power gives it the lead. Alterations to the SSLCA that might help decrease the NRMSE on this task, making it the clear choice across the board, are discussed in Sections 2.3 and 3.2.3 as well as Chapter 7.

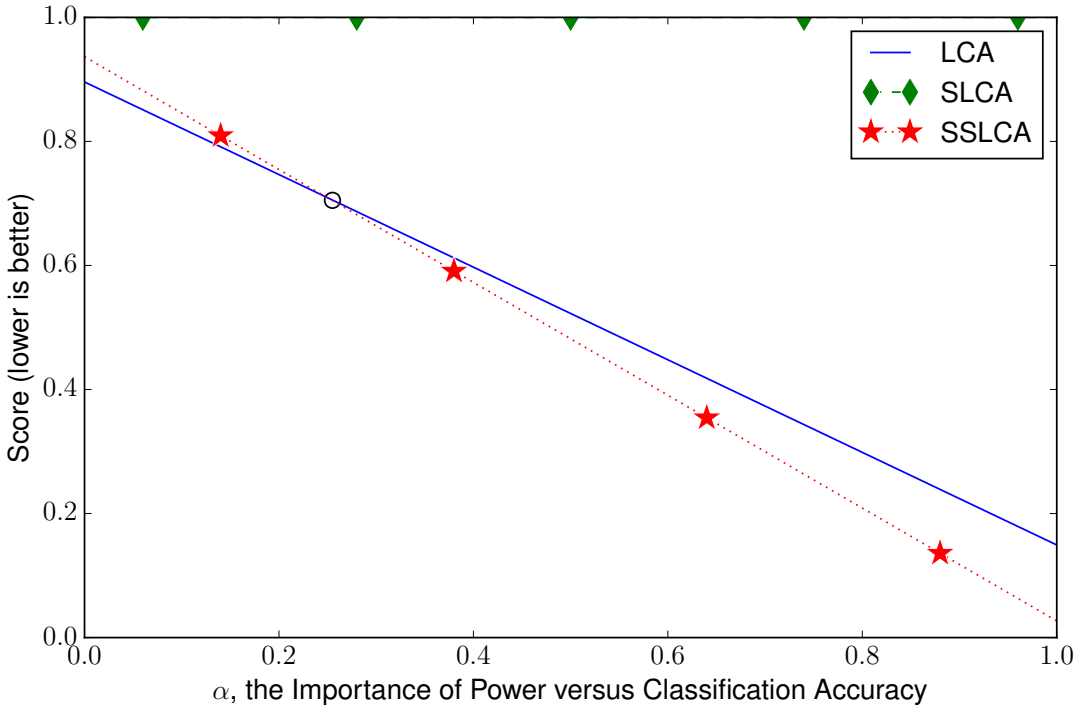

Figure 5.3: Comparative score on the Classification task in this work, calculated by balancing the reciprocal of classification accuracy from Chapter 4 against estimated power consumption for each architecture (SLCA's power comes from Shapero *et al.*). The reciprocal of classification accuracy was chosen so that large values would indicate worse performance. Both the power and reciprocal of classification accuracy were normalized by dividing out the worst architecture's value for each. The remaining quantities were then scaled by α and $1-\alpha$, respectively, to achieve the given score. While the LCA's superior classification rate gives it the edge when power is unimportant, power needs to be only 22% of the performance criteria for the SSLCA to outperform the other algorithms. The SLCA is consistently the worst choice for this task, owing both to the large power required by Shapero *et al.*'s implementation as well as the ambiguity in its receptive fields as discussed in Section 4.2.3.

the algorithms' ability to classify digits. While the LCA is the most performant algorithm when only classification accuracy is needed, the SSLCA becomes a better choice when power is a third as important as classification accuracy. These metrics demonstrate that the SSLCA is already a viable choice in applications where power is a concern. Improvements to bring the SSLCA into the same NRMSE and classification accuracy categories as the LCA are discussed in Sections 2.3 and 3.2.3 as well as Chapter 7.

5.3 Discussion

The SSLCA significantly outperformed the other architectures evaluated for power on both tasks, consuming a maximum of 28% of the power as other architectures. Spiking algorithms also demonstrated improved scalability over the non-spiking LCA, growing with $\mathcal{O}(n)$ rather than $\mathcal{O}(n\sqrt{n})$.

When combined with the results from Chapters 3 and 4, it was shown that the SSLCA is very viable for the Reconstruction task when power is three times as important as NRMSE. For the Classification task, the SSLCA should be chosen when power is only a third as important as classification accuracy. These combination metrics showed the significance of the power savings for the SSLCA, and motivate future work on improving the SSLCA while maintaining the low power provided by its simplicity.

An important hardware quantity not evaluated in this work is the rate at which each architecture can encode inputs. If an architecture consumes twice as much power as another but processes data twice as fast, then the two architectures are equivalent in terms of the energy required to finish a task. This was omitted from this work due to time constraints. However, there is no immediate reason why any

of these architectures would be significantly slower than any of the others, were they all updated to modern clock speeds and CMOS technology.

6

Conclusion

This work investigated three sparse coding architectures: the *Locally Competitive Algorithm* (LCA) proposed by Rozell *et al.* in 2008 [27]; the *Spiking Locally Competitive Algorithm* (SLCA), an extension of the LCA designed by Shapero *et al.* in 2013 [30]; and a novel architecture that was the product of this work dubbed the *Simplified, Spiking Locally Competitive Algorithm* (SSLCA). Designs for the LCA and SSLCA using memristors alongside traditional CMOS technology were presented. The SLCA's design was presented. Each architecture was simulated with 50 neurons in two tasks: reconstructing 8×8 patches from 10 natural images, and classifying handwritten digits from the MNIST database.

Results showed that the SSLCA was a worthwhile contender when power consumption is a consideration. On the Reconstruction task, the SSLCA performed slightly worse than the other two algorithms. While targeting approximately the same output layer activity of 20%, it was found that the LCA reconstructed patches with an NRMSE of 0.074, the SLCA with an NRMSE of 0.095, and the SSLCA with an NRMSE of 0.13. On the Classification task, the SSLCA outperformed the SLCA, potentially due to a technicality: the SLCA demonstrated high initial activity, causing receptive fields to settle on a combination of two digits rather than a single digit or parts of a digit as were the cases with SSLCA and LCA, respectively. The final classification accuracies were 85% for the LCA, 81% for the SSLCA, and 76% for the SLCA. Where the SSLCA significantly outperformed both of the other algorithms was in power: the SSLCA consumed a maximum of 28% of

the power as the other architectures, and demonstrated the best scaling properties of $\mathcal{O}(n)$ (SLCA also grows with $\mathcal{O}(n)$, while LCA grows as $\mathcal{O}(n\sqrt{n})$). Combining the results from the Reconstruction and Classification tasks with the power figures, it was shown that the non-spiking LCA is ideal where optimal NRMSE or classification accuracy is the only concern. However, when power is a third as important as classification accuracy, the SSLCA became the algorithm of choice. When considering reconstruction quality only, the SSLCA became the algorithm of choice when power was three times as important as NRMSE.

Overall, the SSLCA was successful as a novel architecture. The version used in this work lacked the ability to deliberately combine several receptive fields to represent the input. The LCA and SLCA both demonstrated this quality, and it was discussed as future work for the SSLCA. While the SSLCA lacking this quality led to decreased task performance, the power savings of the SSLCA made the algorithm viable, particularly for classification of handwritten digits.

The findings in this work showed that memristors should be strongly considered when looking for ways to optimize existing algorithms. They are power and area-efficient, and led to the design of a novel architecture which demonstrated substantial power savings over conventional architectures. While more work on the SSLCA would be needed to match the performance of the other algorithms, the version simulated in this paper was shown to be more suitable than the LCA and SLCA for applications employing sparse coding with a strong requirement for low power.

7

Future Work

As discussed in Section 3.2.3, the SSLCA's main deficiency stems from the lack of a residual layer. The LCA realizes this through the penalty term based on similarity between receptive fields; the SLCA keeps track of this information by keeping a record of spikes and penalizing appropriately. However, the SSLCA has no such mechanism.

Future designs for the SSLCA include row headers with a transistor whose gate is charged proportionally to the representation of that input in the current output. This should almost entirely mitigate the performance difference between LCA and SSLCA, while keeping the benefits of using memristors and a simple, power-efficient architecture. This was discussed in Section 2.3.

Another flaw in the implementation of SSLCA for this work was discussed in Section 3.2.3: the assumptions for Q_1 and Q_2 of a perfect match between a trained receptive field and the input are not realistic. The algorithm would perform better with a larger number of spikes, encouraged by choosing a less ideal Q_1 and Q_2. For example, the assumptions for Q_1 and Q_2 could be based on each receptive field accounting for 25% of the input. This would help the algorithm adjust to a dictionary that would look more like the colored patches from the LCA and SLCA.

References

[1] H. B. Ammar, K. Tuyls, M. E. Taylor, K. Driessens, and G. Weiss. Reinforcement learning transfer via sparse coding. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 383–390. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[2] D. Batas and H. Fiedler. A Memristor SPICE Implementation and a New Approach for Magnetic Flux-Controlled Memristor Modeling. *IEEE Transactions on Nanotechnology*, 10(2):250–255, March 2011.

[3] C. H. Bennett, D. Chabi, T. Cabaret, B. Jousselme, V. Derycke, D. Querlioz, and J.-O. Klein. Supervised learning with organic memristor devices and prospects for neural crossbar arrays. In *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 181–186, July 2015.

[4] R. Berdan and C. Lim. A Memristor SPICE Model Accounting for Volatile Characteristics of Practical ReRAM. *Electron Device Letters, IEEE*, 35(1):2013–2015, 2014.

[5] Z. Biolek, D. Biolek, and V. Biolkova. SPICE model of memristor with nonlinear dopant drift. *Radioengineering*, 18(1):210–214, 2009.

[6] O. Bryt and M. Elad. Compression of facial images using the K-SVD algorithm. *Journal of Visual Communication and Image Representation*, 19(4):270–282, 2008.

[7] K. Eshraghian, O. Kavehei, K. Cho, J. M. Chappell, A. Iqbal, S. F. Al-Sarawi, and D. Abbott. Memristive Device Fundamentals and Modeling: Applications to Circuits and Systems Simulation. *Proceedings of the IEEE*, 100(6):1991–2007, June 2012.

[8] D. Garbin, E. Vianello, O. Bichler, M. Azzaz, Q. Rafhay, P. Candelier, C. Gamrat, G. Ghibaudo, B. DeSalvo, and L. Perniola. On the impact of oxram-based synapses variability on convolutional neural networks performance. In *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 193–198. IEEE, 2015.

[9] DO Hebb. The organization of behavior; a neuropsychological theory. 1949.

[10] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, 10(4):1297–1301, 2010.

[11] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Letters*, 10(4):1297–1301, 2010.

[12] S. H. Jo and W. Lu. CMOS compatible nanoscale nonvolatile resistance switching memory. *Nano Letters*, 8(2):392–397, 2008.

[13] J. K. Kim, P. Knag, T. Chen, and Z. Zhang. Efficient Hardware Architecture for Sparse Coding. *IEEE Transactions on Signal Processing*, 62:4173–4186, August 2014.

[14] S. Kvatinsky and E. G. Friedman. TEAM: threshold adaptive memristor model. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 60(1):211–221, 2013.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[16] E. Lehtonen and M. Laiho. CNN using memristors for neighborhood connections. *2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010)*, pages 1–4, February 2010.

[17] S. G. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *Signal Processing, IEEE Transactions on*, 41(12):3397–3415, Dec 1993.

[18] T. Masquelier and S. J. Thorpe. Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS Comput Biol*, 3(2):e31, 02 2007.

[19] F. Merrikh-Bayat, B. Hoskins, and D. B. Strukov. Phenomenological modeling of memristive devices. *Applied Physics A*, 118(3):779–786, 2015.

[20] F. Miao, J. P. Strachan, J. J. Yang, M. Zhang, I. Goldfarb, A. C. Torrezan, P. Eschbach, R. D. Kelley, G. Medeiros-Ribeiro, and R. S. Williams. Anatomy of a nanoscale conduction channel reveals the mechanism of a high-performance memristor. *Advanced materials*, 23(47):5633–5640, December 2011.

[21] K. Miller, K. S. Nalwa, A. Bergerud, N. M. Neihart, and S. Chaudhary. Memristive Behavior in Thin Anodic Titania. *IEEE Electron Device Letters*, 31(7):737–739, July 2010.

[22] A. S. Oblea, A. Timilsina, D. Moore, and K. A. Campbell. Silver chalcogenide based memristor devices. *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–3, July 2010.

[23] E. Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[24] M. Payvand and L. Theogarajan. Exploiting local connectivity of CMOL architecture for highly parallel orientation selective neuromorphic chips. In *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 187–192. IEEE/ACM, July 2015.

[25] Y. V. Pershin and M. D. Ventra. SPICE model of memristive devices with threshold. *arXiv preprint arXiv:1204.2600*, pages 11–15, April 2012.

[26] D. Querlioz, W. S. Zhao, P. Dollfus, J. Klein, O. Bichler, and C. Gamrat. Bioinspired networks with nanoscale memristive devices that combine the unsupervised and supervised learning approaches. In *Nanoscale Architectures (NANOARCH), 2012 IEEE/ACM International Symposium on*, pages 203–210, July 2012.

[27] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen. Sparse coding via thresholding and local competition in neural circuits. *Neural computation*, 20(10):2526–2563, 2008.

[28] T. D. Sanger. Optimal unsupervised learning in feedforward neural networks. *Neural Networks*, 2(6):459–473, 1989.

[29] S. Shapero, A. S. Charles, C. J. Rozell, and P. Hasler. Low power sparse approximation on reconfigurable analog hardware. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 2(3):530–541, 2012.

[30] S. Shapero, C. Rozell, and P. Hasler. Configurable hardware integrate and fire neurons for sparse approximation. *Neural Networks*, 45:134–143, 2013.

[31] D. Soudry, D. Di Castro, A. Gal, A. Kolodny, and S. Kvatinsky. Memristor-based multilayer neural networks with online gradient descent training. 2015.

[32] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 05 2008.

[33] J. C. Tapson, G. K. Cohen, S. Afshar, K. M. Stiefel, Y. Buskila, T. J. Hamilton, and A. van Schaik. Synthesis of neural networks for spatio-temporal spike pattern recognition and processing. *Frontiers in Neuroscience*, 7(153), 2013.

[34] W. Woods, J. Bürger, and C. Teuscher. On the influence of synaptic weight states in a locally competitive algorithm for memristive hardware. In *Proceedings of the 2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH 2014)*, pages 19–24. IEEE Press, July 2014.

[35] W. Woods, J. Bürger, and C. Teuscher. Synaptic weight states in a locally competitive algorithm for neuromorphic memristive hardware. *Nanotechnology, IEEE Transactions on*, 14(6):945–953, Nov 2015.

[36] W. Woods, M. M. A Taha, D. Tran, J. Bürger, and C. Teuscher. Memristor Panic - A Survey of Different Device Models in Crossbar Architectures. *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*, pages 106–111, 2015.

[37] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie. Design implications of memristor-based RRAM cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[38] J. J. Yang, D. B. Strukov, and D. R. Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13–24, 2013.

[39] Y. Yang and W. Lu. Nanoscale resistive switching devices: mechanisms and modeling. *Nanoscale*, 5(21):10076–92, November 2013.

[40] C. Zamarreño-Ramos, L. A. Camuñas-Mesa, J. A. Pérez-Carrasco, T. Masquelier, T. Serrano-Gotarredona, and B. Linares-Barranco. On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex. *Frontiers in Neuroscience*, 5(26):1–22, 2011.

[41] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, abs/1212.5701, 2012.

[42] J. Zylberberg, J. T. Murphy, and M. R. DeWeese. A sparse coding model with synaptically local plasticity and spiking neurons can account for the diverse shapes of V1 simple cell receptive fields. *PLoS Comput Biol*, 7(10):e1002250, October 2011.

Appendix A

Job Stream: Easy Pipeline Processing

job_stream is a C++ and Python package available from PyPI (the Python Package Index) and is hosted on github.com: `https://github.com/wwoods/job_stream`. The library was developed to be novice-friendly, leveraging MPI to work seamlessly with the processing clusters available at PSU. It features no-delay checkpointing, load balancing, and performance reporting. This thesis leveraged the Python job_stream.inline module, and achieved acceleration in excess of 300× across 16 different machines. The README for job_stream follows:

README Contents:

- Introduction
- Requirements
- Building job_stream
  - Building and Installing the Python Module
  - Building the C++ Shared Library
  - Build Paths
    - Linux
- Python
  - The Inline Module

        * inline.Work
            · inline.Work.init
            · inline.Work.job
            · inline.Work.finish
            · inline.Work.frame
            · inline.Work.reduce
            · inline.Work.result
            · inline.Work.run
        * inline.Object
        * inline.Multiple
    - Running External Programs (job_stream.invoke)
    - Recipes
        * for x in ...
        * Nested for i in x
        * Aggregating outside of a for loop
        * Aggregating multiple items outside of a for loop
- C++ Basics
    - Reducers and Frames
- Words of Warning
- Appendix
    - Running the Tests
    - Running a job_stream C++ Application
    - Running in Python

A.1 Introduction

job_stream is a straightforward and effective way to implement distributed computations. How straightforward? Well, if we wanted to find all primes between 0 and 999:

```
# Import the main Work object that makes using job_stream dead
 simple
from job_stream.inline import Work
import math

# Start by declaring work based on the list of numbers between 0
 and 999 as a
# piece of 'Work'. When the w object goes out of context, the
 job_stream will
# get exectued
with Work(range(1000)) as w:
   # For each of those numbers, execute this method to see if that
    number is prime
   @w.job
   def isPrime(x):
      for i in range(2, int(math.sqrt(x)) + 1):
         if x % i == 0:
            return
      print(x)
```

Neat, huh? Or for more of a real-world example, if we wanted line counts for all of the files in a directory:

```
# Import the inline library of job_stream (works for 99% of cases
 and produces code
```

```
# that is easier to follow). Object is a blank object, and Work is
 the workhorse of
# the job_stream.inline library.
from job_stream.inline import Object, Work
import os
import sys
path = sys.argv[1] if len(sys.argv) > 1 else '.'

# Start by defining our Work as the files in the given directory
w = Work([ p for p in os.listdir(path)
      if os.path.isfile(p) ])

# For each file given, count the number of lines in the file and
 print
@w.job
def countLines(filename):
   count = len(list(open(filename)))
   print("{}: {} lines".format(filename, count))
   return count

# Join all of the prior line counts by summing them into an object'
 s "total" attribute
@w.reduce(store = lambda: Object(total = 0))
def sumDirectory(store, inputs, others):
   for count in inputs:
      store.total += count
   for o in others:
      store.total += o.total

# Now that we have the total, print it
```

```
@w.job
def printTotal(store):
    print("======")
    print("Total: {} lines".format(store.total))

# Execute the job stream
w.run()
```

job_stream lets developers write their code in an imperative style, and does all the heavy lifting behind the scenes. While there are a lot of task processing libraries out there, job_stream bends over backwards to make writing distributed processing tasks easy. What all is in the box?

- **Easy python interface** to keep coding in a way that you are comfortable
- **Jobs and reducers** to implement common map/reduce idioms. However, job_stream reducers also allow *recurrence*!
- **Frames** as a more powerful, recurrent addition to map/reduce. If the flow of your data depends on that data, for instance when running a calculation until the result fits a specified tolerance, frames are a powerful tool to get the job done.
- **Automatic checkpointing** so that you don't lose all of your progress if a multi-day computations crashes on the second day
- **Intelligent job distribution** including job stealing, so that overloaded machines receive less work than idle ones
- **Execution Statistics** so that you know exactly how effectively your code parallelizes

A.2 Requirements

- boost (filesystem, mpi, python, regex, serialization, system, thread)
- mpi (perhaps OpenMPI)

Note that job_stream also uses yaml-cpp, but for convenience it is packaged with job_stream.

A.3 Building job_stream

A.3.1 Building and Installing the Python Module

The python module job_stream can be built and installed via:

```
pip install job_stream
```

or locally:

```
python setup.py install
```

Note: You may need to specify custom include or library paths:

```
CPLUS_INCLUDE_PATH=~/my/path/to/boost/ \
   LD_LIBRARY_PATH=~/my/path/to/boost/stage/lib/ \
   pip install job_stream
```

Different mpicxx: If you want to use an mpicxx other than your system's default, you may also specify MPICXX=... as an environment variable.

A.3.2 Building the C++ Shared Library

Create a build/ folder, cd into it, and run:

```
cmake .. && make -j8 test
```

Note: You may need to tell the compiler where boost's libraries or include files are located. If they are not in the system's default paths, extra paths may be specified with e.g. environment variables like this:

```
CPLUS_INCLUDE_PATH=~/my/path/to/boost/ \
   LD_LIBRARY_PATH=~/my/path/to/boost/stage/lib/ \
   bash -c "cmake␣..␣&&␣make␣-j8␣test"
```

A.3.3 Build Paths

Since job_stream uses some of the compiled boost libraries, know your platform's mechanisms of amending default build and run paths:

Linux

- CPLUS_INCLUDE_PATH=... - Colon-delimited paths to include directories
- LIBRARY_PATH=... - Colon-delimited paths to static libraries for linking only
- LD_LIBRARY_PATH=... - Colon-delimited paths to shared libraries for linking and running binaries

A.4 Python

A.4.1 The Inline Module

The primary (user-friendly) way to use job_stream in python is via the inline module, which provides the objects `Work`, `Object`, and `Multiple`. Usually, only the `Work` object is required:

```
from job_stream.inline import Work
```

inline.Work

The main element used by `job_stream.inline` is the `Work` object. `Work` is initialized with a list (or generator). Each element of this initial list enters the system as a piece of work within the job stream.

Similar to traditional imperative coding practices, `job_stream.inline` passes work in the same direction as the source file. In other words, if the system starts with:

```
w = Work([ 1, 2, 3 ])
```

Then the numbers 1, 2, and 3 will be distributed into the system. Once work is in the system, we typically deal with them using decorated methods. *The ordering of the decorated methods matters!* `job_stream.inline` is designed so that your work flows in the same direction as your code. For instance, running:

```
w = Work([ 1, 2 ])
@w.job
def first(w):
    print("a: {}".format(w))
    return w

@w.job
def second(w):
    print("b: {}".format(w))
    return w
```

```
w.run()
```

will always print "a: ..." before "b: ..." for any given piece of work that enters the system. More can be learned about `inline.Work.job` in the corresponding section.

Multiprocessing - Python has the GIL in the default implementation, which typically limits pure-python code to a single thread. To get around this, the `job_stream` module by default uses multiprocessing for all jobs - that is, your python code will run in parallel on all cores, in different processes.

If this behavior is not desired, particularly if your application loads a lot of data in memory that you would rather not duplicate, passing useMultiprocessing = False to the `Work()` object's initializer will force all job_stream activity to happen within the original process:

```
w = Work([ 1, 2 ], useMultiprocessing = False)
```

inline.Work.init In practical systems, the initial work often might be generated by some initial code. If you distribute your code to multiple machines, then all code outside of `Work`'s methods will be executed N times, where N is the number of machines that you run your script on. For example, running:

```
print("Init!")
w = Work([ 1 ])
w.run()
```

on four machines, like this:

```
$ mpirun -host a,b,c,d python script.py
Init!
Init!
Init!
Init!
```

will print, "Init!", four times. The work element, `1`, will be be constructed and put into four different lists (one on each machine). However, as a piece of work, `1` will only go into the system once.

If it is important that setup code only be run once, for instance if a results file needs to be initialized, or some debug information is printed, then the init function is useful. For instance, the above code might be refactored as this:

```
w = Work()
@w.init
def generateWork():
   print("Init!")
   return 1
w.run()
```

Now, no matter how many machines the code is parallelized on, "Init!" will only be printed once, and the initial work `1` is only generated on one machine. Since it is just an integer in this case, that's not so bad, but for more complicated initial work it might make a difference.

The final work passed into the system will be the union of anything passed to `Work`'s initializer, and anything returned from an `@Work.init` decorated function. Returning `None` from a function will result in no work being added. To emit multiple pieces of work, look at the `inline.Multiple` object.

Note: `Work.init` is special in that it does not matter where in your source code it appears. Any functions declared with `Work.init` are *always* executed exactly one time, before any work is processed.

inline.Work.jobA job is the main workhorse of `job_stream`. It takes as input a single piece of work, processes it, and in turn emits zero or more pieces of work that flow to the next element of the pipeline. For instance, to add one to a list of integers:

```
from job_stream.inline import Work
w = Work([ 1, 2, 3 ])

# Now that we have 1, 2, and 3 as pieces of work in the system,
 this next
# function will be called once with each value (possibly in
 parallel).

@w.job
def addOne(w):
   return w + 1

# addOne will have been called 3 times, and have emitted 3 more
 pieces of work
# to the next element in the job stream.

w.run()
```

I/O Safety: It is not safe to write external i/o (such as a file) within a job. This is because jobs have no parallelism guards - that is, two jobs executing concurrently might open and append to a file at the same time. On some filesystems, this results

in e.g. two lines of a csv being combined into a single, invalid line. To work around this, see `inline.Work.result`.

inline.Work.finish Decorates a method that only runs on the main host, and only after all work has finished. Since MPI common code (outside of job_stream, that is) runs on all machines, it is occasionally useful to run code only once to finish a calculation. For instance, maybe the final results should be pretty-printed through `pandas`:

```
import pandas
from job_stream.inline import Work
w = Work([ 1, 2, 3 ])
@w.job
def addOne(w):
   return w + 1

@w.finish
def pandasPrintResults(results):
   print(pandas.DataFrame(results))
```

Note that this function is similar to inline.Work.result, but less efficient as it requires keeping all results leaving the job stream in memory. On the other hand, `finish` has access to all results at once, unlike `result`.

inline.Work.frame Frames (and their cousins Reducers) are the most complicated feature in `job_stream`. A frame is appropriate if:

- A while loop would be used in non-parallelizable code
- Individual pieces of work need fan-out and fan-in

Frames have three parts - an "all outstanding work is finished" handler, an aggregator, and everything in between, which is used to process recurred work.

For example, suppose we want to sum all digits between 1 and our work, and report the result. The best way to design this type of system is with a Frame, implemented in `inline` through `Work.frame` and `Work.frameEnd`. The easiest way to think of these is as the two ends of a **`while`** loop - `frame` is evaluated as a termination condition, and is also evaluated before anything happens. `frameEnd` exists to aggregate logic from within the **`while`** loop into something that `frame` can look at.

```
from job_stream.inline import Work, Multiple
w = Work([ 4, 5, 8 ])

@w.frame
def sumThrough(store, first):
   # Remember, this is called like the header of a while statement:
     once at
   # the beginning, and each time our recurred work finishes.
    Anything
   # returned from this function will keep the loop running.
   if not hasattr(store, 'value'):
      # Store hasn't been initialized yet, meaning that this is the
        first
      # evaluation
      store.first = first
      store.value = 0
      return first
```

```
    # If we reach here, we're done. By not returning anything,
      job_stream knows
    # to exit the loop (finish the reduction). The default behavior
      of frame is
    # to emit the store object itself, which is fine.

# Anything between an @frame decorated function and @frameEnd will
  be executed
# for anything returned by the @frame or @frameEnd functions. We
  could have
# returned multiple from @frame as well, but this is a little more
  fun

@w.job
def countTo(w):
    # Generate and emit as work all integers ranging from 1 to w,
      inclusive
    return Multiple(range(1, w + 1))

@w.frameEnd
def handleNext(store, next):
    # next is any work that made it through the stream between
      @frame and
    # @frameEnd. In our case, it is one of the integers between 1
      and our
    # initial work.
    store.value += next

@w.result
def printMatchup(w):
```

```
    print("{}: {}".format(w.first, w.value))

w.run()
```

Running the above code will print:

```
$ python script.py
4: 10
8: 36
5: 15
```

Note that the original work is out of order, but the sums line up. This is because a frame starts a new reduction for each individual piece of work entering the `@frame` decorated function.

inline.Work.reduceTODO. Almost always, programs won't need a reducer. Frames and the `Work.result` decorator replace them. However, if the aggregation of a calculation is resource intensive, `Work.reduce` can help since it can be distributed.

inline.Work.result Since jobs are not I/O safe, `job_stream.inline.Work` provides the `result` decorator. The `result` decorator must be the last element in your job stream, and decorates a function that takes as input a single piece of work. The decorated function will be called exactly once for each piece of work exiting the stream, and is always handled on the main host.

For example, here is some code that takes a few objects, increments their `b` member, and dumps them to a csv:

```
from job_stream.inline import Work
w = Work([ { 'name': 'yodel', 'b': 1 }, { 'name': 'casper', 'b': 99
  } ])
```

```
@w.job
def addOne(w):
   w['b'] += 1
   return w

# Note that @w.init is special, and can be declared immediately
 before the
# output job, regardless of jobs before it. It will always be
 executed first.
@w.init
def makeCsv():
   with open('out.csv', 'w') as f:
      f.write("name,b\n")

# @w.result is also special, as it is not allowed to be anywhere
 except for
# the end of your job stream.
@w.result
def handleResult(w):
   with open('out.csv', 'a') as f:
      f.write("{},{}\n".format(w['name'], w['b']))

w.run()
```

Return values from `Work.result` are ignored.

inline.Work.run After all elements in the job stream are specified, calling `Work.run()` will execute the stream. If your stream takes a long time to execute, it might be worth turning on checkpointing. `run()` takes the following kwargs:

- **checkpointFile** (string) The file path to save checkpoints at. If specified, checkpoints are enabled. By default, a checkpoint will be taken every 10 minutes (even with 20 machines, checkpoints typically take around 10 seconds).
- **checkpointInterval** (float) The number of seconds between the completion of one checkpoint and the starting of the next. Defaults to 600.
- **checkpointSyncInterval** (float) Used for debugging only. This is the mandatory quiet period between the detection of all communication ceasing and the actual checkpointing.

Typically, `Work.run()` will return `None`. However, if your stream has no `Work.result` decorated function, then on the primary host, `Work.run()` will return a list of work that left the system. On other hosts, it will still return `None`.

inline.Object

`inline.Object` is just a basic object that can be used to store arbitrary attributes. As a bonus, its constructor can take kwargs to set. `Object` is typically used with frames and reducers:

```
from job_stream.inline import Work, Object
w = Work([ 1 ])

@w.frame(store = lambda: Object(init = False))
def handleFirst(store, obj):
   if not store.init:
      store.init = True
      # ...
```

```
# ...
```

inline.Multiple

The Zen of Python states that explicit is better than implicit. Since lists or list-like objects may be desired to float around a job stream, all of `job_stream.inline` assumes that return values are single pieces of work. If that is not the case, and a single job should emit multiple pieces of work, simply wrap a collection with the `Multiple` object:

```
from job_stream.inline import Work, Multiple
w = Work([ 1 ])


@w.job
def duplicate(w):
   return Multiple([ w, w ])
```

Now, whatever work flows into `duplicate` will flow out of it with an extra copy.

A.4.2 Running External Programs (job_stream.invoke)

It is tricky to launch another binary from an MPI process. Use `job_stream.invoke()` instead of e.g. `subprocess.Popen` to work around a lot of the issues caused by doing this. Example usage:

```
from job_stream import invoke
out, err = invoke([ '/bin/echo', 'hi', 'there' ])
# out == 'hi there\n'
# err == '' (contents of stderr)
```

`job_stream.invoke()` will raise a RuntimeError exception for any non-zero return value from the launched program. If some errors are transient, and those errors have a unique footprint in stderr, the strings specifying those errors may be passed as kwarg `transientErrors`. Example:

```
from job_stream import invoke
out, err = invoke([ '/bin/mkdir', 'test' ],
      transientErrors = [ 'Device not ready' ])
```

mkdir will be run up to kwarg `maxRetries` times (default 20), retrying until a non-zero result is given.

A.4.3 Recipes

for x in ...

To parallelize this:

```
for x in range(10):
   print x
```

Do this:

```
from job_stream.inline import Work
w = Work(range(10))

@w.job
def printer(x):
   print x
   # Any value returned (except for a list type) will be emitted
     from the job.
   # A list type will be unwrapped (emit multiple)
```

```
    return x

w.run()
```

Nested for i in x

To parallelize this:

```
for x in range(10):
    sum = 0
    for i in range(x):
        sum += i
    print("{}: {}".format(x, sum))
```

Write this:

```
from job_stream.inline import Work
w = Work(range(10))

# For each of our initial bits of work, we open a frame to further
 parallelize within
# each bit of work
@w.frame
def innerFor(store, first):
    """This function is called whenever everything in the frame is
     finished. Usually,
    that means it is called once when a frame should request more
     work, and once when
    all of that work is done.
```

```
    Any work returned by this function will be processed by the jobs
       within the frame,
    and finally aggregated into the 'store' variable at the frameEnd
       function."""

    if not hasattr(store, 'init'):
        # First run, uninitialized
        store.init = True
        store.value = 0
        # Anything returned from a frame or frameEnd function will
          recur to all of the
        # jobs between the frame and its corresponding frameEnd
        return list(range(first))

    # If we get here, we've already processed all of our earlier
      recurs. To mimic the
    # nested for loop above, that just means that we need to print
      our results
    print("{}: {}".format(first, store.value))

@w.frameEnd
def innerForEnd(store, next):
    store.value += next

w.run()
```

Aggregating outside of a for loop

To parallelize this:

```
results = []
```

```
for i in range(10):
   results.append(i * 2)
result = sum(results)
```

Write this:

```
from job_stream.inline import Object, Work
w = Work(range(10))

@w.job
def timesTwo(i):
   return i * 2

# reduce is
@w.reduce(store = lambda: Object(value = 0), emit = lambda store:
 store.value)
def gatherResults(store, inputs, others):
   for i in inputs:
      store.value += i
   for o in others:
      store.value += o.value

# Run the job stream and collect the first (and only) result into
 our sum
result, = w.run()
```

Aggregating multiple items outside of a for loop

To parallelize this:

```
results = []
```

```
for i in range(10):
   results.append(i)
   results.append(i * 2)
result = sum(results)
```

Write this:

```
from job_stream.inline import Multiple, Object, Work
w = Work(range(10))

@w.job
def timesTwo(i):
   return Multiple([ i, i * 2 ])

# reduce is
@w.reduce(store = lambda: Object(value = 0), emit = lambda store:
 store.value)
def gatherResults(store, inputs, others):
   for i in inputs:
      store.value += i
   for o in others:
      store.value += o.value

# Run the job stream and collect the first (and only) result into
 our sum
result, = w.run()
```

A.5 C++ Basics

job_stream works by allowing you to specify various "streams" through your application's logic. The most basic unit of work in job_stream is the job, which takes some input work and transforms it into zero or more outputs:

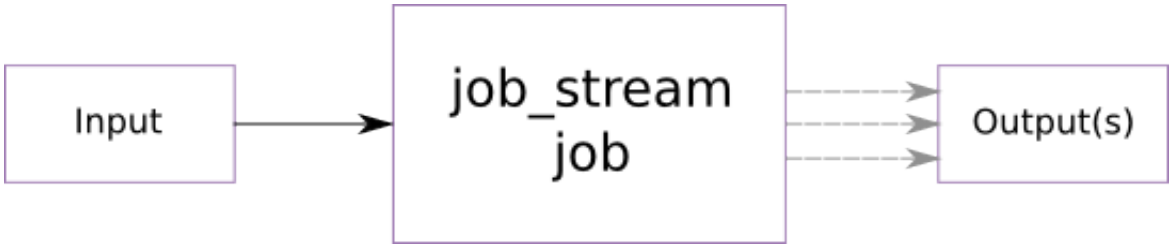

Figure A.1: A job_stream job takes some input, transforms it, and emits zero or more outputs

That is, some input work is required for a job to do anything. However, the job may choose to not pass anything forward (perhaps save something to a file instead), or it might apply some transformation(s) to the input and then output the changed data. For our first job, supppose we wanted to make a basic job that takes an integer and increments it, forwarding on the result:

Figure A.2: A job that adds one to the input and emits it

The corresponding code for this job follows:

```
#include <job_stream/job_stream.h>
//All work comes into job_stream jobs as a unique_ptr; this can be
 used
```

```
//to optimize memory bandwidth locally.
using std::unique_ptr;

/** Add one to the integer input and forward it. */
class AddOneJob : public job_stream::Job<AddOneJob, int> {
public:
   /** The name used to describe this job in a YAML file */
   static const char* NAME() { return "addOne"; }
   void handleWork(unique_ptr<int> work) {
      this->emit(*work + 1);
   }
} addOneJob;
```

The parts of note are:

- Template arguments to job_stream::Job - the class being defined, and the expected type of input,
- NAME() method, which returns a string that we'll use to refer to this type of job,
- handleWork() method, which is called for each input work generated,
- this->emit() call, which is used to pass some serializable object forward as output, and
- this->emit() can take any type of argument - the output's type and content do not need to have any relation to the input.
- There MUST be a global instance allocated after the class definition. This instance is not ever used in code, but C++ requires a instance for certain templated code to be generated.

NOTE - all methods in a job_stream job must be thread-safe!

In order to use this job, we would need to define a simple `adder.yaml` file:

```
jobs:
 - type: addOne
```

Running this with some input produces the expected result:

```
local$ pwd
/.../dev/job_stream
local$ cd build
local$ cmake .. && make -j8 example
...
# Any arguments after the YAML file and any flags mean to run the
 job stream
# with precisely one input, interpreted from the arguments
local$ example/job_stream_example ../example/adder.yaml 1
2
(some stats will be printed on termination)
# If no arguments exist, then stdin will be used.
local$ example/job_stream_example ../example/adder.yaml <<!
3
8
!
# Results - note that when you run this, the 9 might print before
 the 4!
# This depends on how the thread scheduling works out.
4
9
(some stats will be printed on termination)
local$
```

A.6 Reducers and Frames

Of course, if we could only transform and potentially duplicate input then job_stream wouldn't be very powerful. job_stream has two mechanisms that make it much more useful - reducers, which allow several independently processed work streams to be merged, and recursion, which allows a reducer to pass work back into itself. Frames are a job_stream idiom to make the combination of reducers and recursion more natural.

To see how this fits, we'll calculate pi experimentally to a desired precision. We'll be using the area calculation - since A = R*pi^2, pi = sqrt(A / R).
Randomly distributing points in a 1x1 grid and testing if they lie within the unit circle, we can estimate the area:

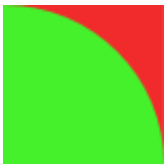

Figure A.3: Estimating pi

The job_stream part of this will take as its input a floating point number which is the percentage of error that we want to reach, and will emit the number of experimental points evaluated in order to reach that accuracy. The network looks like this:

As an aside, the "literally anything" that the piCalculator needs to feed to piEstimate is because we'll have piEstimate decide which point to evaluate.
This is an important part of designing a job_stream pipeline - generality. If, for instance, we were to pass the point that needs evaluating to piEstimate, then we have locked our piCalculator into working with only one method of evaluating pi.

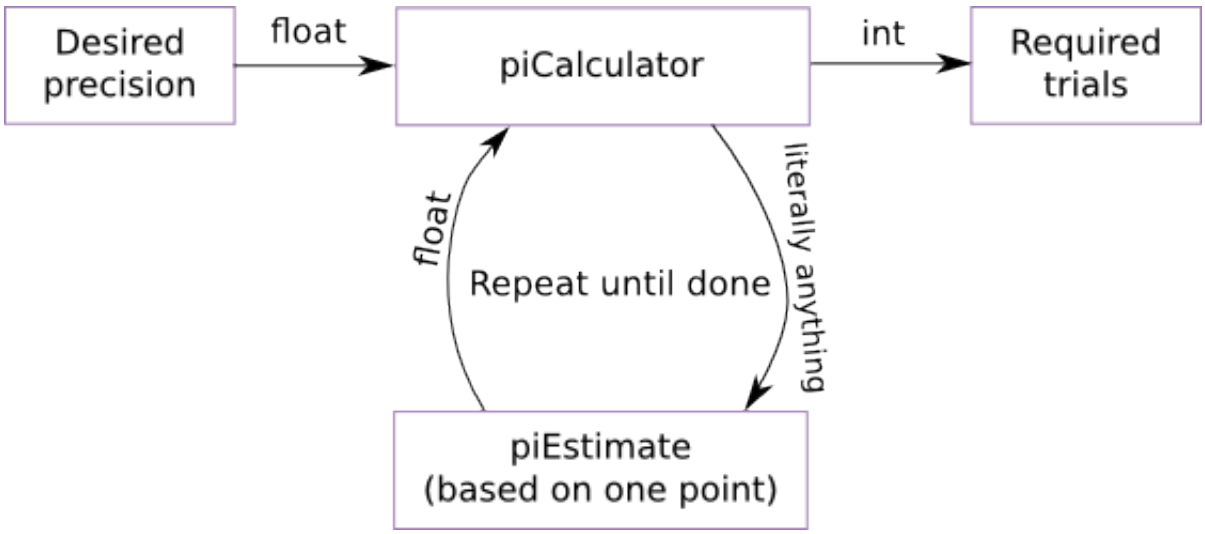

Figure A.4: Estimating pi

With the architecture shown, we can substitute any number of pi estimators and compare their relative efficiencies.

Before coding our jobs, let's set up the YAML file `pi.yaml`:

```
jobs:
 - frame:
     type: piCalculator
   jobs:
     - type: piEstimate
```

This means that our pipe will consist of one top-level job, which itself has no type and a stream of "jobs" it will use to transform data. Wrapped around its stream is a "frame" of type piCalculator. This corresponds to our above diagram.

piCalculator being a frame means that it will take an initial work, recur into itself, and then aggregate results (which may be of a different type than the initial work) until it stops recurring. The code for it looks like this:

```
struct PiCalculatorState {
   float precision;
   float piSum;
   int trials;

private:
   //All structures used for storage or emit()'d␣must␣be␣
    serializable
␣␣␣␣friend␣class␣boost::serialization::access;
␣␣␣␣template<class␣Archive>
␣␣␣␣void␣serialize(Archive&␣ar,␣const␣unsigned␣int␣version)␣{
␣␣␣␣␣␣␣␣ar␣&␣precision␣&␣piSum␣&␣trials;
␣␣␣␣}
};

/**␣Calculates␣pi␣to␣the␣precision␣passed␣as␣the␣first␣work.␣␣The␣
 template
␣␣␣␣arguments␣for␣a␣Frame␣are:␣the␣Frame's class, the storage type,
  the
   first work's␣type,␣and␣subsequent␣(recurred)␣work's type. */
class PiCalculator : public job_stream::Frame<PiCalculator,
      PiCalculatorState, float, float> {
public:
   static const char* NAME() { return "piCalculator"; }

   void handleFirst(PiCalculatorState& current, unique_ptr<float>
    work) {
      current.precision = *work * 0.01;
      current.piSum = 0.0f;
      current.trials = 0;
```

```
      //Put work back into this Frame. This will trigger whatever
       method
      //of pi approximation is defined in our YAML. We'll␣pass␣the
␣␣␣␣␣␣␣␣//current␣trial␣index␣as␣debug␣information.
␣␣␣␣␣␣␣␣this->recur(current.trials++);
␣␣␣␣}

␣␣␣␣void␣handleWork(PiCalculatorState&␣current,␣unique_ptr<float>␣
 work)␣{
␣␣␣␣␣␣␣␣current.piSum␣+=␣*work;
␣␣␣␣}

␣␣␣␣void␣handleDone(PiCalculatorState&␣current)␣{
␣␣␣␣␣␣␣␣//Are␣we␣done?
␣␣␣␣␣␣␣␣float␣piCurrent␣=␣current.piSum␣/␣current.trials;
␣␣␣␣␣␣␣␣if␣(fabsf((piCurrent␣-␣M_PI)␣/␣M_PI)␣<␣current.precision)␣{
␣␣␣␣␣␣␣␣␣␣␣␣//We're within desired precision, emit trials count
         fprintf(stderr, "Pi␣found␣to␣be␣%f,␣+-␣%.1f%%\n",
          piCurrent,
               current.precision * 100.f);
         this->emit(current.trials);
      }
      else {
         //We need more iterations. Double our trial count
         for (int i = 0, m = current.trials; i < m; i++) {
            this->recur(current.trials++);
         }
      }
   }
} piCalculator;
```

Similar to our first addOne job, but we've added a few extra methods - handleFirst and handleDone. handleFirst is called for the work that starts a reduction and should initialize the state of the current reduction.
handleWork is called whenever a recur'd work finishes its loop and ends up back at the Frame. Its result should be integrated into the current state somehow. handleDone is called when there is no more pending work in the frame, at which point the frame may either emit its current result or recur more work. If nothing is recur'd, the reduction is terminated.

Our piEstimate job is much simpler:

```
class PiEstimate : public job_stream::Job<PiEstimate, int> {
public:
   static const char* NAME() { return "piEstimate"; }
   void handleWork(unique_ptr<int> work) {
      float x = rand() / (float)RAND_MAX;
      float y = rand() / (float)RAND_MAX;
      if (x * x + y * y <= 1.0) {
         //Estimate area as full circle
         this->emit(4.0f);
      }
      else {
         //Estimate area as nothing
         this->emit(0.0f);
      }
   }
} piEstimate;
```

So, let's try it!

```
local$ cd build
```

```
local$ cmake .. && make -j8 example
local$ example/job_stream_example ../example/pi.yaml 10
Pi found to be 3.000000, +- 10.0%
4
(debug info as well)
```

So, it took 4 samples to arrive at a pi estimation of 3.00, which is within 10% of 3.14. Hooray! We can also run several tests concurrently:

```
local$ example/job_stream_example ../example/pi.yaml <<!
10
1
0.1
!
Pi found to be 3.000000, +- 10.0%
4
Pi found to be 3.167969, +- 1.0%
Pi found to be 3.140625, +- 0.1%
1024
1024
0 4% user time (3% mpi), 1% user cpu, 977 messages (0% user)
C 4% user time, 0% user cpu, quality 0.00 cpus, ran 1.238s
```

The example works! Bear in mind that the efficiency ratings for a task like this are pretty poor. Since each job only does a few floating point operations, he communication overhead well outweighs the potential benefits of parallelism. However, once your jobs start to do even a little more work, job_stream quickly becomes beneficial. On our modest research cluster, I have jobs that routinely report a user-code quality of 200+ cpus.

A.7 Words of Warning

fork()ing a child process can be difficult in a threaded MPI application. To work around these difficulties, it is suggested that your application use job_stream::invoke (which forwards commands to a properly controlled libexecstream).

Job and reduction routines MUST be thread safe. Job_stream handles most of this for you. However, do NOT create a shared buffer in which to do your work as part of a job class. If you do, make sure you declare it thread_local (which requires static).

It is wrong to build a Reducer or Frame that simply appends new work into a list. Doing so will cause excessively large objects to be written to checkpoint files and cause the backups required to support checkpoints to bloat unnecessarily (backups meaning the copy of each store object that represents its non-mutated state before the work began. Without this, checkpointing would have to wait for all Work to finish before completing). This leads to very long-running de/serialization routines, which can cause very poor performance in some situations.

If you use checkpoints and your process crashes, it is possible that any activity *outside* of job_stream will be repeated. In other words, if one of your jobs appends content to a file, then that content might appear in the file multiple times. The recommended way to get around this is to have your work output to different files, with a unique, deterministic file name for each piece of work that outputs. Another approach is to use a reducer which gathers all completed work, and then dumps it all to a file at once in handleDone().

Sometimes, passing -bind-to-core to mpirun can have a profoundly positive impact on performance.

A.8 Appendix

A.9 Running the Tests

Making the "test" target (with optional ARGS passed to test executable) will make and run any tests packaged with job_stream:

```
cmake .. && make -j8 test [ARGS="[serialization]"]
```

Or to test the python library:

```
cmake .. && make -j8 test-python [ARGS="../python/job_stream/test/"
 ]
```

A.10 Running a job_stream C++ Application

A typical job_stream application would be run like this:

```
mpirun -host a,b,c my_application path/to/config.yaml [-c
 checkpointFile] [-t hoursBetweenCheckpoints] Initial work string (
 or int or float or whatever)
```

Note that -np to specify parallelism is not needed, as job_stream implicitly multi-threads your application. If a checkpointFile is provided, then the file will be used if it exists. If it does not exist, it will be created and updated periodically to allow resuming with a minimal loss of computation time. It is fairly simple to write a script that will execute the application until success:

```
RESULT=1
for i in `seq 1 100`; do
   mpirun my_application config.yaml -c checkpoint.chkpt blahblah
   RESULT=$?
```

```
    if [ $RESULT -eq 0 ]; then
       break
    fi
done

exit $RESULT
```

If -t is not specified, checkpoints will be taken every 10 minutes.

A.11 Running in Python

Python is much more straightforward:

```
LD_LIBRARY_PATH=... ipython
>>> import job_stream
>>> class T(job_stream.Job):
      def handleWork(self, w):
         self.emit(w * 2)
# Omit this next line to use stdin for initial work
>>> job_stream.work = [ 1, 2, 3 ]
>>> job_stream.run({ 'jobs': [ T ] })
```

Appendix B

git-results: Accountable Organization for Experiments

git-results is a plugin for the Git revision control system to help with running and cataloguing the results of experiments. The tool automatically handles the build process, creating tags to work together with the git diff command in order to see the differences between experiments, and the storing and organization of results files. This tool was used to organize the results for this thesis; the figures were all generated from scripts that reach into the appropriate results folder and pull information from the latest experiments. git-results was written in Python and is available on github.com: `https://github.com/wwoods/git-results`. The README file for this project follows, which shows its operation and features:

A helper script / git extension for cataloguing computation results

B.1 Installation

Put git-results somewhere on your PATH. Proper setup can be verified by running:

```
git results -h
```

to show the tool's help.

B.2 Usage

`git-results` is a tool for organizing and bookmarking experiments locally, so that the exact conditions for each experiment can be remembered and compared intuitively and authoritatively. In its most basic mode, running `git-results` executes the following steps:

1. Switch to a temporary branch,

   - Add all local source changes, and create a commit on the temporary branch with all of your code changes,
   - Clone that commit to a temporary folder,
   - Execute `git-results-build` within that folder,
   - Snapshot the folder's contents,
   - Execute `git-results-run` within that folder,
   - Diff the folder's contents against the original snapshot, moving any new files to the specified results directory.

   A basic invocation of `git-results` looks like this:

```
$ git results results/my/experiment
```

This will open your favorite text editor (via environment variables `VISUAL` or `EDITOR`, or fallback to `vi`) and prompt for a message further describing the experiment. After that, `git-results` will do its thing, moving any results files to `results/my/experiment/1` where they are archived. Note the `/1` at the end of the path! Every experiment ran through `git-results` is versioned, assisting with iterative development of a single experiment.

B.3 Special Files

`git-results` relies on a few special files in your repository; these files define behaviors and parameters specific to your application.

B.3.1 `git-results-build`

is a required file that describes what steps are required to build your project - this is separated from `git-results-run` so that e.g. compile errors can be separated from runtime errors. This file also helps to separate any intermediate files created as a part of the build process from viable results that should be archived.

B.3.2 `git-results-run`

is a required file describing what needs to be run to produce output files that need to be recorded.

B.3.3 `git-results-progress`

is an optional file that should look at the project's current state and return a single, monotonically increasing, floating-point number that describes how far along a process is. This file is required only for the `-r` flag, which flags an experiment as retry-able on events such as failure or sudden system shutdown.

Typically, this file might amount to checking the timestamp on a checkpoint file; if the checkpoint file does not get updated, then the process is not progressing and it's possible that a non-transient error is impeding progress. For example, on a Linux system, this file might contain:

```
stat -c %Y my_checkpoint.chkpt 2>/dev/null || echo -1
```

If you wish to use `git-results -r` experiments, then note that you will need to run `git results supervisor` in your crontab or equivalent, to periodically check if any experiments need to be restarted.

B.4 What does `git-results` put in the output folder and the folders above it?

B.4.1 Meta information

If your `git-results-run` file lives at `project/git-results-run` relative to your git repository root, then executing an experiment from the `project` folder as `git results results/a` does the following:

1. Establishes a results root in `project/results`.

The results root is the folder one deeper than the folder containing `git-results-run`; in this example, `project` contains `git-results-run`, so that folder is the results root. If this folder is not already treated as a results root, then `git-results` will prompt for confirmation of result root creation. Results roots are automatically added to the git repository's `.gitignore`.

- Creates a versioned instance of the experiment named `a` in the results root `project/results`.

Most experiments need to be iteratively refined; `git-results` helps you to manage this by automatically creating subfolders `1`, `2`, etc. for your experiments. When an experiment completes successfully, these folders will not have a suffix. However, if they are still running, or `git-results-run` returns non-zero (failure),

then these subfolders will be renamed to reflect the experiment's ultimate status (`1-run`, or `1-fail`).

- Adds the experiment version to the `INDEX` file.

Each experiment directory gets a special `INDEX` file that correlates the number of the experiment to the message that was typed in when it was executed.

- In the results root, symlinks `latest/experiment` to the last-run version.
- Creates a link to the experiment version in the `dated` folder of the results root, with the same name and version as the experiment but prefixed with today's date. E.g., `dated/2015/04/13-your/experiment/here/1`

Note that these steps are identical and the results will be the same as if `git results project/results/a` had been run from the git root. `git-results` always stores information relative to the results root, calculated based on where `git-results-run` last occurs in the specified path.

B.4.2 Experiment results

The versioned experiment folder will contain the following:

- stdout
- stderr
- A meta-information file git-results-message, containing:
  - The git tag that marks the experiment
  - The message entered when the experiment was ran
  - The contents of `git-results-run` and `git-results-build` (and, if it exists, `git-results-progress`

  - The starting timestamp, total duration, and whether or not the program exited successfully.

- Any files created during the execution of `git-results-run`

(can also be executed as `git results project/results/a` from the git root; the path to `git-results-run` determines the working directory)

B.5 Comparing code from two experiments

```
git diff path/to/results/experiment/version path/toresults/other/
 version --
```

Note the -- at the end - without this, git doesn't know what to do.

B.6 Resuming / Re-Entrant git-results-run files

Check out `git-results-progress` above.

B.7 Special Directories

git-results automatically makes "dated" and "latest" folders.

"dated" contains a folder hierarchy ending in symlinks to the results for experiments, organized by date. For instance, results/dated/2014/03/24-test/run/2 would be a symlink to results/test/run/2. It's a longer path of course, but it's indexed by date.

"latest" contains a folder hierarchy pointing the the most recent run of a test, including in progress runs.

B.8 Moving / Linking results

If you wish to rename a tag at a later date, you can do so with move:

```
$ git results move test/run test/run2
```

This may not be the wisest idea if you are pushing your results to a remote repository, as git tags are relatively immutable on remote machines. But, it will work locally anyway.

If you simply wish to link results into a new location, use link:

```
$ git results link test/run test/run2
```

It just uses symlinks, meaning the data will not be copied, but subsequent moves will break the links.