Rochester Institute of Technology
RIT Scholar Works

Theses | Thesis/Dissertation Collections

6-1-2010

Modification of an asynchronous dexterous hand movement decoder for hardware implementation

Adam Kevin Bosen

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Bosen, Adam Kevin, "Modification of an asynchronous dexterous hand movement decoder for hardware implementation" (2010). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Modification of an Asynchronous Dexterous Hand Movement Decoder for Hardware Implementation

by

Adam Kevin Bosen

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Associate Professor Daniel B. Phillips
Department of Electrical Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2010

Approved By:

Daniel B. Phillips
Associate Professor, Department of Electrical Engineering
Primary Adviser

Juan Carlos Cockburn
Associate Professor, Department of Computer Engineering

Marcin Lukowiak
Assistant Professor, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: Modification of an Asynchronous Dexterous Hand Movement Decoder for Hardware Implementation

I, Adam Kevin Bosen, hereby grant permission to the Wallace Memorial Library reporduce my thesis in whole or part.

Adam Kevin Bosen

Date

Dedication

I highly doubt I could get away with not dedicating my first major work to my mother, so, thanks mom, this one is for you.

Acknowledgments

Thanks to Marc H. Schieber, who generated the neural data analyzed here and in previous research on this topic(supported by R01 NS27686).

Thanks to Vikram Aggarwal, Soumyadipta Acharya, and Francesco Tenore for performing the research that lead to my work and for providing the guidance and advice to get me going.

Thanks to Daniel Phillips, Juan Cockburn, and Marcin Lukowiak for providing me with direction and support throughout this research and for happily enduring the many questions and scheduling conflicts I frequently ambushed them with.

Abstract

One of the goals of modern prosthetics research is to provide natural, neurologically driven control of a prosthetic device, preferably in a portable format. Previously, an algorithm for asynchronously decoding individuated finger and wrist movements from recordings of neural activity in the primary motor cortex was developed by Aggarwal *et al.*and implemented in software. The first objective of this work was to determine what effect simplifying Aggarwal's algorithm by using linear Artificial neural networks instead of nonlinear ones would have on movement detection and classification accuracy. The simplified algorithm developed in this work was demonstrated to achieve movement detection and classification accuracies of 99.7%, 89.9%, and 95.3% for an individuated movement decoding task across three subjects using neural recordings from 80 neurons. In comparison, the original algorithm demonstrated accuracies of 96.2%, 90.5%, and 99.8% for the same task and subjects using neural recordings from 40 neurons. Additionally, the simplified algorithm was demonstrated to have a detection and classification accuracy of 80.5% for a combined movement task, whereas the original algorithm achieved accuracy of 92.5%. Even though a greater input space size was required for the linear decoder, the computational intensity was reduced with a mean accuracy loss in the individuated movement task of only 0.53%. However, the 12% loss of accuracy observed in the combined movement task is considered unacceptable and suggests that the simplified algorithm is not appropriate for this task.

The second objective of this work was to create a digital hardware implementation of the simplified linear artificial neural network version of Aggarwal's algorithm. A scalable, fully parallel architecture was designed for implementation on a Xilinx Virtex-4 FX60

FPGA. This implementation could be realized with an input dimension of up to 60 neurons on this FPGA, although computations were performed on the order of 10^4 times faster than was necessary for realtime operation, indicating that there is an opportunity to reduce hardware size in exchange for computational speed. This work is an important exploration into the eventual goal of incorporating a hardware movement decoder in a prosthetic device and demonstrates that a hardware implementation is feasible using currently available technology.

Contents

Dedication . . . **iii**

Acknowledgments . . . **iv**

Abstract . . . **v**

1 Introduction . . . **1**
- 1.1 The Nervous System . . . 2
- 1.2 Recording Nervous Signals . . . 3
- 1.3 Machine Learning . . . 4
- 1.4 Asynchronous Dexterous Decoder . . . 8
- 1.5 Thesis Objective . . . 12

2 Algorithm Modifications . . . **14**
- 2.1 Replacement of Nonlinear Artificial Neural Networks . . . 15
- 2.2 Gating Logic Modification . . . 17
- 2.3 Elimination of Floating Point Multiplication . . . 22

3 Hardware Implementation . . . **26**
- 3.1 Architecture Selection and Requirements . . . 26
- 3.2 Choice of a Development Platform . . . 28
- 3.3 Hardware Implementation Overview . . . 30
- 3.4 Input Window . . . 31
- 3.5 Classifiers . . . 32
- 3.6 Output Logic . . . 33
- 3.7 IO Interface . . . 34
- 3.8 Hardware Analysis . . . 36

4 Test Methodology . . . **39**
- 4.1 Determination of gating classifier parameters . . . 40

4.2 Classifier testing 41
4.3 Comparing Hardware and Software Implementations 42

5 Results **43**

6 Discussion **49**
6.1 Algorithm Modification Analysis 49
6.2 Hardware Implementation Analysis 51

7 Conclusions **55**

Bibliography **58**

List of Figures

1.1 Example of a simple artificial neural network with three inputs, a single hidden layer, and one output. Each node, represented by circles, computes the weighted sum of each input, represented by arrows. 6

1.2 A diagram of the original asynchronous dexterous decoder[1]. The use of two classifiers allows for the detection and classification of movement from recorded nerual signals, $X(t_k)$, without any reliance on external cuing. The gating classifier output, $G(t_k)$, is a boolean value that corresponds to whether or not movement is occurring, with the heuristic check, $G_{track}(t_k)$, eliminating spurious classifications. The movement classifier output, $S(t_k)$, represents which type of movement, out of a predefined set, is occurring. Together, they allow for the asynchronous detection of a specified set of finger and wrist movements from the recorded firing rates of neurons in the primary motor cortex[1]. 11

1.3 Block diagram of a complete neurally controlled hand prosthesis. White text indicates preexisting structures, red text indicates the proposed work, and blue text indicates additional components necessary for a complete portable neurally controlled hand prosthesis. 13

2.1 Number of multiplications required per classification with respect to input space dimension, assuming the nonlinear classifier uses an average hidden layer size 1.5 times larger than the input space and a 12 movement output space. 16

2.2 Response of linear gating classifier to a set of four movements occurring once every two seconds using the entire input space. The blue solid line (desired output) represents the ideal signal that the gating classifier was trained to, and the red dashed line (actual output) represents the actual response of the gating classifier. Note that the peak level, duration, and location with respect to the desired output is not consistent from trial to trial. 17

2.3 The discrete derivative of the gating classifier response to a set of four movements using the entire input space, computed as $G(t_k) - G(t_k - \Delta)$, where Δ ranges from 1 to 5. Movements occurred at times 1, 3, 5, and 7 seconds, to which the negative signal peaks correspond (the peak observed near time 2 seconds is a false positive). Note that as Δ increases both the peak magnitude and delay between movement time and peak time increase. 19
2.4 Response of linear gating classifier to a set of four movements occurring once every two seconds using the entire input space with gating classifier derivative. The blue solid line represents the function that the gating classifier was trained to, the red dashed line represents the actual response of the gating classifier, and the green line represents the discrete time derivative of the gating classifier response. 20
2.5 Response of linear movement classifier to a sequence of trials occurring once every two seconds using the entire input space. The movements were presented every odd second in the following order: extension of each digit, in order from thumb to little finger (e1-e5), extension of the wrist(ew), flexion of each digit in the same order(f1-f5), and flexion of the wrist(fw). The diagonal row of spikes in the center of the diagram shows that the largest response during each trial is the expected movement type. 21
2.6 This graph shows the base-10 logarithm of a sample distribution of classifier weights after training to a full input space (1495 weights total). 22

3.1 Block diagram of input window 31
3.2 Block diagram of linear classifiers 32
3.3 Block diagram of gating logic 33
3.4 Block diagram of movement logic 34
3.5 Block diagram of output logic 35
3.6 Block diagram of asynchronous dexterous decoder hardware implementation with IO interface 36
3.7 Critical path delay with respect to input space dimension. Note that since the graph scale is on the order of nanoseconds the 20ms maximum is not an issue. 37
3.8 Number of FPGA Slices required to implement linear classifier with respect to input space dimension. The black dashed line is the total amount of available slices on the Virtex-4 FX60 FPGA. 38

5.1 Movement detection accuracy rate of the gating classifier with respect to the level threshold and derivative threshold parameters γ and δ in the region $0.2 \leq \gamma \leq 0.5$, $-0.1 \leq \delta \leq 0$. 44
5.2 Movement detection accuracy rate of the gating classifier with respect to the level threshold and derivative threshold parameters γ and δ in the region $0.25 \leq \gamma \leq 0.35$, $-0.06 \leq \delta \leq -0.01$. 44
5.3 False Positive rate of gating classifier with respect to the gating classifier refractory period parameter ρ. 44
5.4 Linear classifier decoding accuracy with respect to input space size. Note that accuracy starts to drop around approximately 60 neurons, indicating that a larger input space is necessary to attain a level of accuracy similar to that of the nonlinear classifier. Additionally, note that the accuracy of the combined movement trials is significantly worse than the nonlinear classifier. 46

List of Tables

5.1 Summary of gating classifier accuracy using the full input space of each subject. 45
5.2 Summary of movement classifier accuracy using the full input space of each subject. 45
5.3 Table of output differences between hardware and software implementation over 100 trials with an input dimension of 10. 46
5.4 Table of output differences between hardware and software implementation over 100 trials with an input dimension of 20. 47
5.5 Table of output differences between hardware and software implementation over 100 trials with an input dimension of 30. 47
5.6 Table of output differences between hardware and software implementation over 100 trials with an input dimension of 40. 48
5.7 Table of output differences between hardware and software implementation over 100 trials with an input dimension of 50. 48

6.1 Comparison of original nonlinear artificial neural network and modified linear artificial neural network algorithm accuracy with respect to input dimension. Information on original algorithm accuracy was obtained from [1], which only provided information up to 40 neurons. 50

Chapter 1

Introduction

One of the primary problems in prosthetics development is the need for a means of control that functions similarly to the biological system the prosthetic is intended to replace. Recent research has focused on brain-machine interfaces (BMI), which is the recording and decoding of electrophysiological measures of neural activity to control an electronic device[1]. The nervous system has several unique properties that must be accounted for when designing a BMI. One of these properties is the duplication of signals in multiple locations in the body, which has allowed researchers to explore multiple methods of recording nervous information[2, 3]. This also means that it is often very difficult to isolate a single signal responsible for any individual action, so machine learning techniques are typically employed to convert neural recordings to useful information. One method of performing this, the asynchronous dexterous decoder developed by Aggerwal *et al.*[1], is the foundation of the research presented in this paper. This algorithm has been previously demonstrated to be capable of asynchronously detecting and identifying hand movements in rhesus monkeys with as high as a 99.8% accuracy[1], indicating that it is potentially viable for use in a hand prosthesis. The goal of this research is to develop a simplified version of the asynchronous dexterous decoder that requires fewer computational resources per classification decision, quantify the accuracy of the simplified decoder, and design and implement a hardware realization of this decoder.

1.1 The Nervous System

In order to meet the objective of this research it is necessary to understand the structure of the nervous system and the nature of the signals it generates. The nervous system is one of the primary organ systems responsible for control and communication between all other systems in the body. Its functions include sensory integration and motor control, along with cognition. The functional units of the nervous system are nerves, which are bundles of specialized excitable cells called neurons that can accept and propagate signals through the body[4].

The primary means of signal propagation in an individual neuron is the action potential, a rapid depolarization and repolarization of the electrical potential between the interior and exterior of the neuron[5]. A depolarization in one part of the cell will cause depolarization of adjacent portions of the cell, effectively allowing for the action potential to travel down the cell. Action potentials typically have a amplitude of +100mV and can occur at a maximum rate of 200Hz[5]. Since the amplitude of an action potential is fixed, information is encoded in the rate at which the action potentials occur. The rate at which action potentials occur is limited by the presence of a refractory period following each action potential, during which another action potential cannot occur[4].

Individual neurons are bundled in groups called nerves which are responsible for transmitting information through the body. Any nerve not located in the brain or spinal cord are designated as part of the peripheral nervous system (PNS), while the brain and spinal cord are collectively known as the central nervous system (CNS). The PNS is primarily responsible for propagation of sensory and motor information, while the CNS performs most decision making and control signal generation tasks. The brain is divided into several distinct components, each of which is primarily responsible for different aspects of mental function, including sensory integration, cognition, and control of autonomic processes. The outer layer of neurons in the brain, the cerebral cortex, plays a significant role in perception, thought, and voluntary control[4].

The cerebral cortex can be divided into regions called Brodmann areas that are primarily responsible for specific functions[4]. Communication in these area is performed by ensembles of neurons firing together, so recording from any portion of one of these areas should give a good overall representation of the area's current activity. The function of an individual neuron in one of these areas may change over time, but the overall function of the area remains fairly constant[4].

One Brodmann area of particular interest for this research is the primary motor cortex, which is responsible for voluntary movement. Control of every portion of the body is mapped topographically to the primary motor cortex, so recording the activity of a portion of the primary motor cortex can provide relevant information about what the corresponding portion of the body is doing[4]. Research has demonstrated that sufficient information to decode the movement of individual fingers is distributed throughout the hand control region of the primary motor cortex, so neural recordings taken from any portion of this region can provide a good representation of overall hand activity[3].

1.2 Recording Nervous Signals

In order to make use of the information available in the nervous system a means of recording and interpreting its signals is necessary. Several methods of performing this task are currently being researched, one of which is electrode implantation. As the name implies, this method involves directly implanting electrodes into some section of the brain for the purpose of recording neural signals. Typically, recording is done using a cluster of electrodes arranged in an array suitable for implantation in one specific portion of the brain[6]. Research has shown that recording as few as 20 neural signals located anywhere within the hand area of the primary motor cortex is sufficient to provide a reliable estimate of hand activity[3], although the most common type of electrode array, the Utah Intracortical Electrode Array, is capable of recording significantly more[6]. Individual electrodes in the array may pick up signals from multiple neurons as well as additional background noise,

but several filtering and sorting algorithms known collectively as spike sorters have been developed to address this issue[7, 8]. Additionally, long term studies have been performed on electrode arrays that demonstrate they can maintain signal quality for a period of at least 1.5 years, which shows potential for use in long term applications[9]. The specific recording areas, significant quantity of data, and long term durability make this approach viable for prosthetic control.

However, electrode implantation does have a few significant limitations. The biggest barrier to this approach is the need to directly implant the electrodes into the brain, which requires significant surgery to remove the skin, bone, and meninges directly over the desired recording site [6]. This surgical procedure has been performed on cats[6] and monkeys[1], but obvious ethical considerations prevent much experimentation on humans. Additionally, the function of individual neurons is known to potentially shift over time[9], so systems designed to use information recorded from an electrode array will need to be capable of accounting for this shift.

1.3 Machine Learning

Once nervous signals are obtained from a subject, they can be processed using machine learning. Machine learning is the study of algorithms that can learn rules based on example data[10]. It is often used to create rules that relate data sets in situations where the relationship between those sets is not well understood. In the context of brain-machine interfacing, machine learning can be applied to create rules that relate nervous activity to observed body function. The ability to generate relationships between data sets is of particular value in this application due to the fact that the exact function of any individual nervous signal is often difficult to determine. In general, there is no guarantee that a relationship between a set of nervous signals and body function exists, but previous research has demonstrated that activity in the hand control region of the primary motor cortex can be mapped to a set of hand movements[3, 1]. Several different classification algorithms exist for the purpose of

creating a relationship between two data sets[11]. One approach, artificial neural networks, can be used to create mappings of arbitrary complexity from one data set to another and are capable of defining complex, nonlinear relationships.

An Artificial Neural Network (ANN) is a directed graph where each node computes a weighted sum of its inputs. An ANN consists of external inputs from one data set, weighted connections between nodes, and external outputs that represent the network's mapping into a different set. The complexity and linearity of the mapping performed by the ANN is controlled by the number of nodes and the way they are connected, while the exact shape is determined by the weight values. Mapping an input set to an output set is performed via training, which uses presented data to adjust network weights. Although ANNs can be constructed arbitrarily, in this work only layered feedforward networks will be considered in order to avoid introducing the increased complexity of time variant systems. A feedforward network consists of two or more discrete layers of neurons in which every node of one layer functions as an input to every node in the next layer. The first layer is the input layer, where system inputs are directly applied, and the last layer is the output layer, where the output values are read from. Additional layers between the input and output layers are referred to as hidden layers, because they are not directly controlled or visible outside the ANN. In hidden layers a nonlinear function, typically a sigmoid or a hyperbolic tangent is applied to each sum to allow for the ANN to produce nonlinear mappings[10]. Figure 1.1 shows a diagram of a simple ANN consisting of three inputs, a single hidden layer with two nodes, and a single output.

Each connection from one layer to the next can be viewed as a transform from one space to another. If no hidden layers are present in the ANN then the relationship between the inputs and an individual output can be expressed simply as a weighted sum and is therefore linear. Geometrically, this describes a hyperplane drawn through the input space. The distance from a point in the input space to this plane determines the value of the corresponding output of the ANN. Alternatively, if hidden layers are used then the transformation from input space to output space ceases to be as simple and is no longer linear, due to the fact

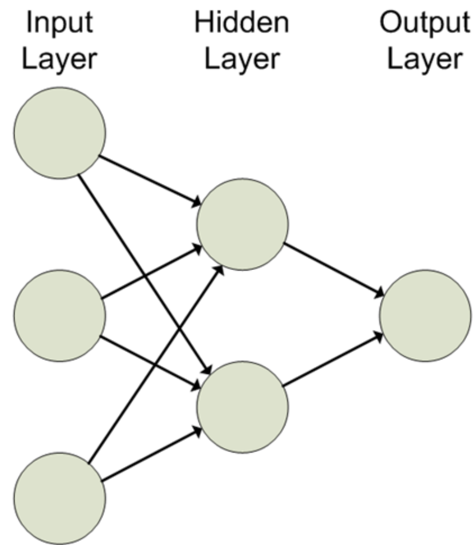

Figure 1.1: Example of a simple artificial neural network with three inputs, a single hidden layer, and one output. Each node, represented by circles, computes the weighted sum of each input, represented by arrows.

that a nonlinear function is applied to each sum. If this nonlinear function was not present the ANN would be several cascaded linear transforms, which could be simplified to a single linear transform and render it unable to perform higher order functions. Nonlinearity is advantageous because it allows for mappings that are not possible with a linear transform, although additional computation is required to perform this mapping. Each hidden layer increases the order of the mapping function, which can allow for mappings that would not be attainable with a lower order function[10].

For the sake of simplicity it is preferable to use the least computationally intensive ANN that meets performance requirements. The number of hidden layers and hidden nodes in an ANN determine the order of complexity of the functions it is capable of learning as well as the number of calculations required to compute a result[10]. Therefore, there is a tradeoff between accuracy and computational cost that is governed by hidden layer size and quantity. An ANN with no hidden layers is only capable of computing linear functions, but only requires a number of computations of order n, where n is the input dimension. In comparison, an ANN with hidden layers can compute a greater class of functions for

each hidden layer, but requires a number of computations of order n^m, where m is the number of hidden layers. Additionally, each hidden layer requires the computation of a nonlinear function for each sum. This tradeoff indicates that it is desirable to find the smallest ANN that is capable of estimating the desired function to a specified accuracy. If problem complexity is unknown, as is the case for many machine learning applications, then using an ANN with more hidden layers or nodes than necessary will not adversely affect performance, but will be more computationally intensive than necessary.

One of the objectives for any machine learning task is the minimization of output error. Typically, this requires that the system be trained using sample data. One of the common methods of training an ANN is gradient descent. Gradient descent is a supervised learning algorithm that minimizes error by descending the gradient of the mean square error between the actual and desired outputs. In an ANN with no hidden layers this is done by giving the system an input value, comparing the system output to the desired output, and modifying the weights in the weight vector by subtracting from it the partial derivative of the error with respect to that weight multiplied by a learning factor. In this method, weights that have a greater contribution to the incorrect value are modified more heavily. As the system is trained the learning factor is gradually decreased in order to converge on a single set of weights for the system. Equation 1.1 expresses this rule mathematically, where Δw_j^t is the amount added to weight w_j after training example t, η is the learning factor that controls the magnitude of weight adjustments, r^t is the desired output at training example t, y^t is the actual output at training example t, and x_j^t is the input corresponding to weight w_j[10].

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \tag{1.1}$$

For an ANN with hidden layers gradient descent is performed via backpropagation, which essentially performs the same weight modifications one layer at a time. The output layer is modified in a manner identical to gradient descent, and previous layers are iteratively modified after determining the amount of error each neuron in the previous layer contributed. Note that in order for this to be possible the output of every neuron needs to be differentiable, which is a constraint on the nonlinear function applied to each sum in the hidden

layer[10].

Another potential goal for a machine learning task that has a large number of inputs is to reduce the number of inputs to a smaller subset that still meet performance requirements. This is especially useful when it is suspected that some inputs provide information that is redundant or independent of the desired output function. Reduction of system complexity can be accomplished by incorporating a method of structural adaptation into the learning algorithm. This can be either constructive, where inputs are added to a very simple network as needed, or destructive, where inputs are removed from a complex network as they are determined to be unnecessary. One simple method of destructive reduction is weight decay. Weight decay replaces the error function used for gradient descent with an error function that penalizes output error *and* weight size. This gives each weight in the system a tendency to decay to zero. If a particular input associated with a weight is not necessary to provide accurate classification then its contribution to the system is removed without affecting overall accuracy. Input weights that are critical to correct classification are updated whenever an error occurs, so their weights are not removed. At the end of training, any inputs with weights of zero can be removed without affecting classifier accuracy[10].

1.4 Asynchronous Dexterous Decoder

The asynchronous dexterous decoder developed by Aggarwal *et al.*is a system for decoding a specific set of hand movements from neural signals recorded from the primary motor cortex. This decoder uses committees of artificial neural networks to detect and classify finger and wrist extension and flexion from intracortical neural signal recordings. It was demonstrated to be accurate over 90% of the time in three different subjects, indicating that it is a potentially viable for use in a neurally controlled multifingered hand prosthesis[1].

The data used for this work were intracortical neural signals recorded from three male rhesus monkeys, identified as monkeys C, G, and K. Each monkey was trained to perform the desired set of hand movements in response to a visual cue for a water reward. While

performing these movements, a recording was obtained from a single neuron in the primary motor cortex. Each subject had different numbers of recording sites: 312 neurons were recorded from in monkey C, 125 neurons were recorded from in monkey G, and 115 neurons were recorded from in monkey K. Data was collected from each neuron individually in a series of repeated trials for each movement type. For monkeys C and G up to seven trials were recorded for each neuron for each movement type, while for monkey K up to 15 trials were recorded per neuron per movement type. These recordings were then processed by a spike detector that quantified the time at which action potentials occurred with respect to time at which a movement was performed[1].

The feature that makes this decoder unique is its asynchronous nature. Most prior research focused on cued decoding, which relies on an external signal to indicate when decoding should be performed. Obviously, this dependence is undesirable when attempting to develop a useful prosthetic device. The asynchronous dexterous decoder overcomes this limitation by using two classifiers: one classifier determines whether or not a movement is occurring (gating classifier), and the other determines what type of movement out of a specified set is being performed (movement classifier). Input to both classifiers is provided as a count of the number of action potentials that occur during a 100ms window that shifts forward every 20ms. This sliding window produces a set of discrete time, discrete valued signals that describe the activity of each neuron in the system[1].

As shown in figure 1.2, the gating classifier was designed as a odd numbered committee of artificial neural networks trained to produce an output between 0 and 1 corresponding to the probability that movement was occurring. Each ANN possessed a single tan-sigmoidal hidden layer containing anywhere from 0.5 to 2.5 times the number of input neurons, a single output neuron, and a log-sigmoidal output function. The output of each ANN was thresholded at a value of γ to produce a boolean value of 0 or 1. A majority voting rule was used to determine the committee output. If the committee output was 1 more than β times in the past τ decisions then the gating classifier fired, indicating positive movement. Once the gating classifier fired it was prevented from firing again for a refractory period ρ

to guard against false positives[1]. This rule is expressed mathematically in equation 1.2, where $G(t_k)$ is the gating ANN committee output and $G_{out}(t_k)$ is the movement detection decision, with a 1 indicating movement and a 0 indicating no movement.

$$G_{out}(t) = \begin{cases} 1 & \text{if } \sum_{t=t_k-\tau}^{t_k} G(t_k) > \beta \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

Like the gating classifier, the movement classifier consists of an odd numbered committee of ANNs, but each ANN had a number of output neurons equal to the number of movement types decoded, with each neuron corresponding to a specific movement type. A set of twelve distinct movement types were performed in all subjects: flexion and extension of each individual finger and the wrist. In one subject, monkey K, additional data was recorded for an extra set of six movement types: combined flexion and extension of the thumb and forefinger, the forefinger and middle finger, and ring finger and little finger. Every movement type was treated as a binary decision, with no consideration given to partial movements. Each ANN was trained to output values between 0 and 1 corresponding to the probability that a given movement type was occurring. The movement type with the highest probability was chosen as the output of each network, and a majority voting rule was used to determine the movement classifier output[1]. Equation 1.3 expresses this rule mathematically, where $s(t)$ is the movement decision out of the set of i movements at time t and $P_i(t)$ is the output of movement ANN i at time t.

$$s(t) = \arg\max_i P_i(t) \tag{1.3}$$

The final classification decision was produced by multiplying the output of the gating classifier with the output of the movement classifier. Ideally, this produces an output of zero when no movement is detected and an integer corresponding to a specific movement type when that movement type is detected. A diagram of the decoder is shown in figure 1.2 All portions of the decoder were implemented and trained in software using Matlab[1]. The asynchronous dexterous decoder was demonstrated to have an accuracy of above 90%

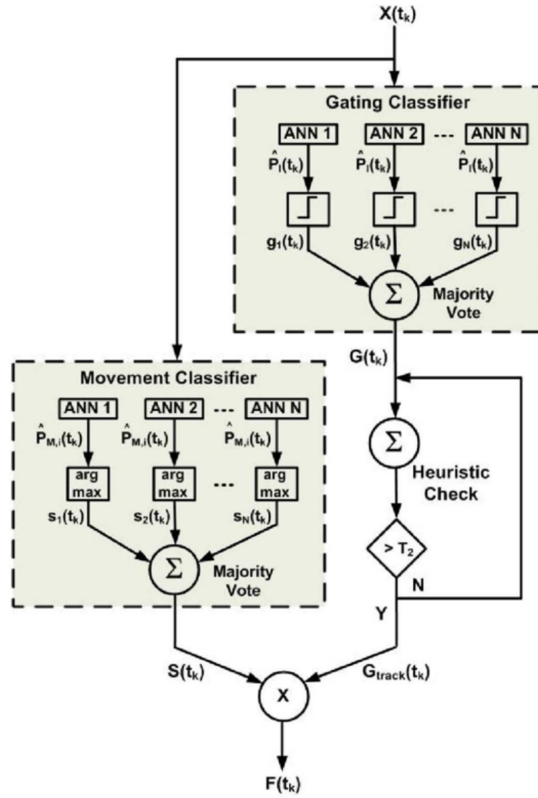

Figure 1.2: A diagram of the original asynchronous dexterous decoder[1]. The use of two classifiers allows for the detection and classification of movement from recorded nerual signals, $X(t_k)$, without any reliance on external cuing. The gating classifier output, $G(t_k)$, is a boolean value that corresponds to whether or not movement is occurring, with the heuristic check, $G_{track}(t_k)$, eliminating spurious classifications. The movement classifier output, $S(t_k)$, represents which type of movement, out of a predefined set, is occurring. Together, they allow for the asynchronous detection of a specified set of finger and wrist movements from the recorded firing rates of neurons in the primary motor cortex[1].

for all three subjects when as few as 40 neurons were used as input. Additionally, the decoder was shown to be as high as 99.8% accurate for monkey K in the individual finger movement task and 92.5% accurate for monkey K when combined movements were added to the output space. These results demonstrate that this algorithm is robust to decode hand movements regardless of the specific neural population recorded from, which indicates that it has potential application to the development of a neurally controlled hand prosthesis[1]. Additionally, research by Acharya *et al.*[3] demonstrates that movement can be accurately decoded by as little as 20 neurons, suggesting that recordings from a greater number of

neurons contain some redundant information. Since it is known that a good nonlinear approximation exists when 20 neurons are used adding information from other neurons can be viewed as a transformation to a higher dimensional space, which may allow classes to be separated linearly[10]. However, since the exact transformation that occurs when neurons are added is unknown, linearly separability cannot be guaranteed, but the possibility that linear separability exists for this problem was investigated because it would allow for the order of complexity of the ANNs in the asynchronous dexterous decoder to be reduced.

1.5 Thesis Objective

The purpose of this research is to determine if an asynchronous dexterous decoder that uses only linear ANNs can achieve accuracy comparable to that of the original decoder presented in [1] and, if so, implement the linear ANN decoder in digital hardware. Ideally, the linear ANN decoder will be as accurate as the nonlinear ANN decoder, although accuracy degredation is acceptable if the best case decoder accuracy does not fall below 90%, which is comparable to the lowest best case performance demonstrated by the original nonlinear ANN decoder. It is proposed that meeting this accuracy objective will require a larger input space than necessary for comparable accuracy in the nonlinear ANN decoder. This requirement is considered acceptable, because even if the linear ANN decoder requires a larger input space, the order of complexity will still be lower than that for the nonlinear ANN decoder. If this accuracy objective can be met, the linear ANN decoder would be adapted for a digital implementation capable of making classification decisions in real time, which was defined in the original algorithm as one decision every 20ms.

The implementation of a digital realization of the asynchronous dexterous decoder is an important step toward a practical prosthesis, but it is beyond the scope of this work to develop a completely independent prosthetic control system. All ANN training will be performed in software using MATLAB (Mathworks Inc., Natick, MA) using custom scripts and the MATLAB Neural Network Toolbox. Once the digital implementation is designed,

feasibility of implementing a hardware based machine learning system will be performed as a basis for potential future work. Additionally, the data used to train and test the system will be the discrete time, discrete valued integer counts of the number of action potentials(spikes) that occurred in the past 100ms, moving forward every 20ms, as described in section 1.4. In a complete neurally controlled prosthesis this information would need to be extracted from neural recordings by spike sorting and counting hardware, then provided to the asynchronous dexterous decoder. Several algorithms for the detection of action potentials that would be suitable for incorporation into a neurally controlled prosthesis have been developed previously[12, 13], indicating that this requirement is not a major limitation on future work. The other necessary component of a neurally controlled prosthesis, the end effector, will not be integrated with the asynchronous dexterous decoder in this work. Figure 1.5 shows a block diagram of a complete neurally controlled hand prosthesis and the way in which the research performed in this thesis is meant to be incorporated into a complete prosthetic system.

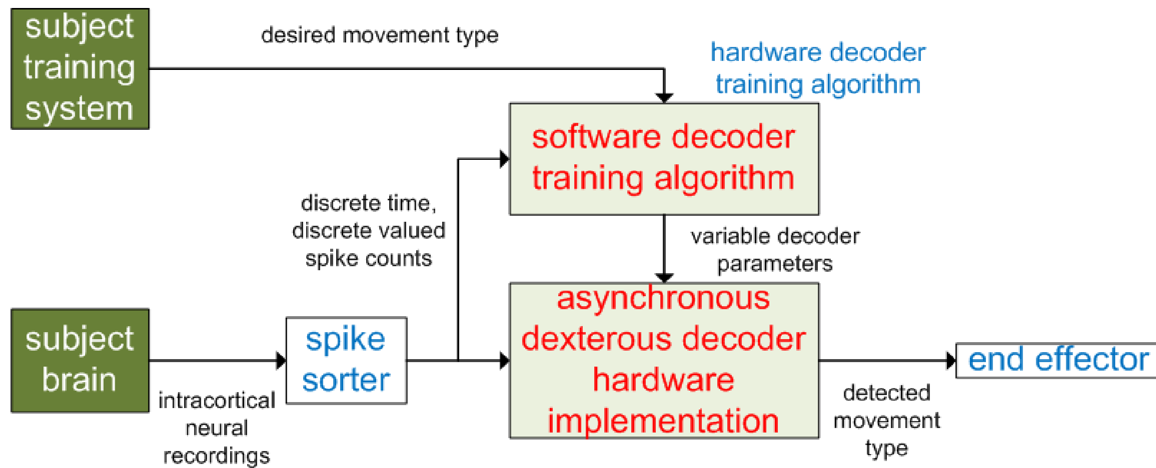

Figure 1.3: Block diagram of a complete neurally controlled hand prosthesis. White text indicates preexisting structures, red text indicates the proposed work, and blue text indicates additional components necessary for a complete portable neurally controlled hand prosthesis.

Chapter 2

Algorithm Modifications

The first objective of this research is to determine if the asynchronous dexterous decoder algorithm can be modified to use linear artificial neural networks while still meeting the accuracy requirements defined previously. This chapter describes the theoretical modifications made to the asynchronous dexterous decoder algorithm and their impact on the computational complexity of the system. For a digital hardware implementation of an ANN the number of floating point multiplications necessary to compute the output is the most intensive aspect of the algorithm[14], so computational complexity of the decoder will be primarily measured in terms of number of floating point multiplications. The four modifications made to the decoder algorithm were the replacement of the nonlinear ANNs with linear ANNs, the removal of the committee voting structure, a new gating decision algorithm, and an integer approximation of all floating point math. These modifications purposefully do not include any changes to the sliding input window or decision rate, so that any larger system can use the original decoder algorithm developed by Agarwal *et al.*or the new, modified algorithm interchangeably, and to simplify direct comparison between the two algorithms. All modifications were designed with the intention of providing the same movement classification accuracy as the original algorithm at a reduced computational cost. Since the functionality of the original algorithm is only approximated some error is introduced, although attempts were made to minimize its impact on movement classification accuracy. All training and testing computations described in this chapter were performed by scripts created for use with MATLAB.

2.1 Replacement of Nonlinear Artificial Neural Networks

The first modification performed on the decoder algorithm was the replacement of the non-linear, single hidden layer ANNs with linear ANNs. The purpose of this modification was to determine if the same classification accuracy can be achieved at a lower computational cost. The switch from a nonlinear ANN to a linear ANN can be viewed as simply removing the hidden layers from the ANN. Removing hidden layers imposes a significant constraint because it restricts the ANN to only linear transformations, but if linear separability can be demonstrated then this constraint will not adversely affect performance.

Switching from nonlinear ANNs to linear ANNs reduces the usefulness of the committee voting structure used in the original algorithm. An ANN with hidden layers inherently implements a nonlinear function, so more than one function that can approximate the desired output may exist. Additionally, the number of nodes in a hidden layer may be varied, so multiple ANNs, each with a different number of hidden layer nodes, can be combined in a committee structure to yield accuracy greater than each ANN individually[1]. In comparison, an ANN without hidden layers is only capable of implementing linear functions. This limitation means there will only be one error minimum towards which all stable learning algorithms will converge. Therefore, every member in a committee of linear ANNs would appear to be almost identical, which eliminates the utility of the committee. Based on this fact, it was decided a single linear ANN was sufficient to replace each committee.

In terms of computational complexity, the single most prevalent and expensive operation in the implementation of an ANN is floating point multiplication. A multiplication is needed for each connection from one layer to the next, from which the total number can be determined as the product of the number of neurons in each layer. For example, a typical implementation of the original asynchronous dexterous decoder uses about 40 input neurons, 3 committees for each classifier, and 12 movement types. Assuming the mean number of hidden layer neurons of 1.5 times the number of input neurons gives an estimated need for ((40 input neurons)(60 hidden neurons)+(60 hidden neurons)(12 output neurons))(3 ANNs in each committee) = 9360 floating point multipliers for the movement

classifier and ((40 input neurons)(60 hidden neurons)+(60 hidden neurons)(1 output neuron))(3 ANNs in each committee) = 7,380 floating point multipliers for the gating classifier. This gives a combined total of 16,740 floating point multiplications needed for the original algorithm for each classification decision. In comparison, the number of multipliers needed for a linear, committee-less algorithm is (40 input weights)(1 linear classifier) = 40 multipliers for the gating classifier and (40 input weights)(12 movement types)(1 linear classifier) = 480 multipliers. This gives a total of 520 multipliers, which is still about 35 times fewer than the neural network. Furthermore, the lack of a hidden layer means that the number of multiplications required to calculate a classification scales linearly with input dimension, unlike the nonlinear ANN approach, where the number of multiplications required increases as the square of the input dimension. Figure 2.1 compares the relative complexity (number of multiplications required per classification) of each classification algorithm with respect to the size of the input space. Note that, despite the change in the way computations are performed, the input and output, a count of the number of spikes for each neuron in the past 100ms and an integer corresponding to a classification decision, are untouched, so from an external perspective the linear and nonlinear classifiers are interchangable.

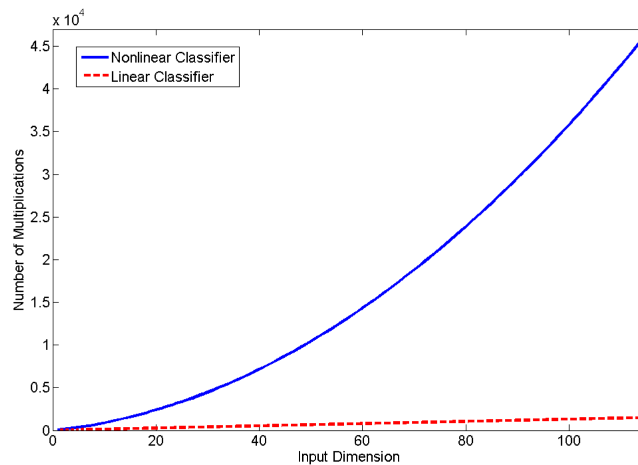

Figure 2.1: Number of multiplications required per classification with respect to input space dimension, assuming the nonlinear classifier uses an average hidden layer size 1.5 times larger than the input space and a 12 movement output space.

2.2 Gating Logic Modification

Ideally, the above changes should not have significantly impacted classifier performance. However, it was observed in early testing that this was not the case. Figure 2.2 shows the output of the new, linear gating classifier in response to a set of four movement trials occurring once every two seconds. The variation in the shape of this output was observed to be typical for the gating classifier. As shown, the gating classifier output does not closely follow the desired output, in one case rising only as high as 0.58 instead of the desired value of 1, but does demonstrate a rising and falling pattern that roughly follows the rise and fall of the desired output. The potential reason for these variations is discussed later.

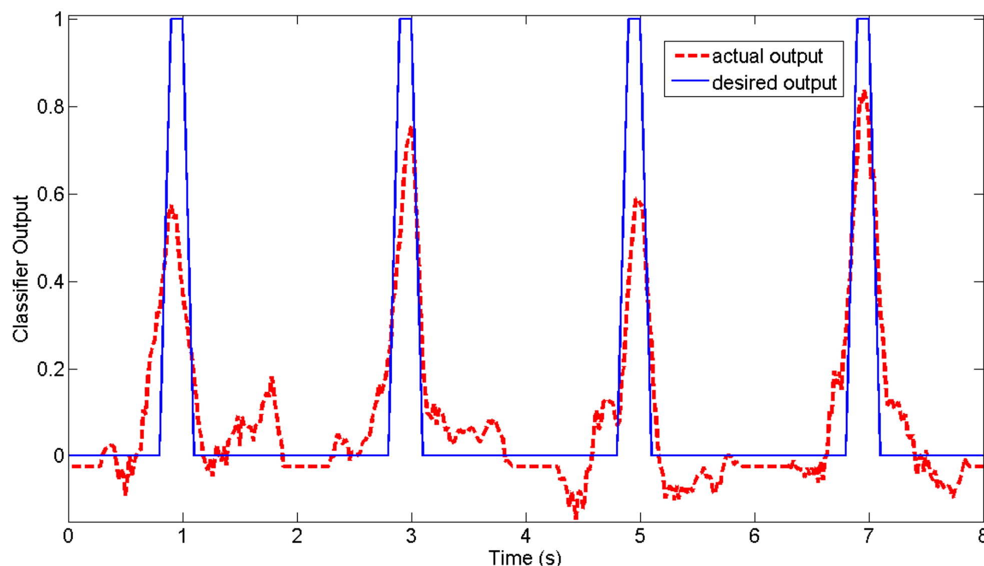

Figure 2.2: Response of linear gating classifier to a set of four movements occurring once every two seconds using the entire input space. The blue solid line (desired output) represents the ideal signal that the gating classifier was trained to, and the red dashed line (actual output) represents the actual response of the gating classifier. Note that the peak level, duration, and location with respect to the desired output is not consistent from trial to trial.

The error demonstrated in figure 2.2 by the gating classifier is undesirable. The peak level, time, and duration are not consistant from trial to trial, and the classifier output wanders (even taking negative values) when no activity is present. If the primary objective of the classifier was to match the desired response signal this classification scheme would

be discarded in favor of a different solution. However, the goal of this classifier is to provide enough information to allow a decision making scheme to accurately determine when a movement occurs. Based on the clear presence of output activity at the desired times an alternative decision making rule was designed.

As described previously, the original algorithm used a movement detection scheme that counted the number of times the gating classifier output was above a static threshold in a set window and fired if the output was above the threshold above a certain number of times. This approach was sufficient for the nonlinear classification scheme because the nonlinear gating classifier closely and predictably followed the desired output, but since the linear gating classifier did not consistently reach a specific peak value for a predictable duration this decision scheme was not as effective. A four dimensional search of the parameters γ, τ, β, and ρ was performed (values ranged between 0.15 to 1 in 0.05 increments for γ, 3 to 10 in integer increments for τ and β, and 2 to 10 in integer increments for ρ) to find parameter values that met the specified accuracy requirements, but a set of values that achieved at least 90% movement detection accuracy was not found. As a result, an alternative decision method was developed. Analysis of the problem showed that although the gating classifier output did not peak consistently, it did drop off rapidly following a movement. This lead to the decision to use the derivative of the gating classifier output as well as the output itself. Figure 2.3 shows the derivative of the actual response signal from figure 2.2, computed as $G(t_k) - G(t_k - \Delta)$, where Δ ranged from 1 to 5. The important aspect of these signals is the negative peaks, which correspond to a decrease in the gating classifier output, indicating that a movement occurred. As Δ increases the magnitude of the negative peaks increases, along with the delay between the actual movement time and peak time. This relationship creates a tradeoff between movement identifiability and temporal accuracy. To compromise, a value of $\Delta = 3$ was selected, creating a delay of three time samples (60ms) and negative peaks that are sufficiently large to detect movement (Determination of exact threshold levels is discussed later, in chapter 4). Further analysis of figure 2.3 shows that the derivative signal is insufficient to detect movement on its own as smaller negative

peaks, such as the one slightly before the two second mark, are present in the signal even when no movement is occurring. Because of this, the decision was made to use both the gating classifier output and its derivative to detect movement. Figure 2.4 shows the same movement response as in figure 2.2 with added information of the gating classifier output discreet derivative, computed as $G(t_k) - G(t_k - 3)$.

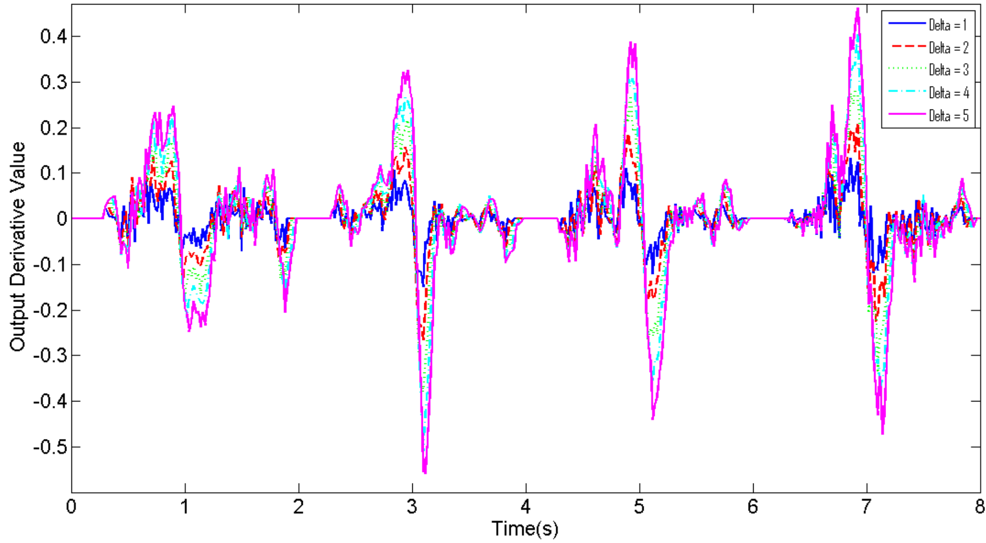

Figure 2.3: The discrete derivative of the gating classifier response to a set of four movements using the entire input space, computed as $G(t_k) - G(t_k - \Delta)$, where Δ ranges from 1 to 5. Movements occurred at times 1, 3, 5, and 7 seconds, to which the negative signal peaks correspond (the peak observed near time 2 seconds is a false positive). Note that as Δ increases both the peak magnitude and delay between movement time and peak time increase.

Based on this, the new movement detection decision scheme was designed to check if the output was above some static threshold γ and instead of counting the number of time windows during which the output was above this threshold also checked the derivative of the gating classifier output. If the difference between the current output and the output three time windows ago was less than the negative constant δ and the current output was above the threshold γ the gating classifier fired. Equation 2.1 expresses this rule mathematically, where γ and δ are the parameters described above, $G(t)$ is the gating classifier output and $G_{out}(t)$ is the movement detection decision, with a 1 indicating movement and a 0

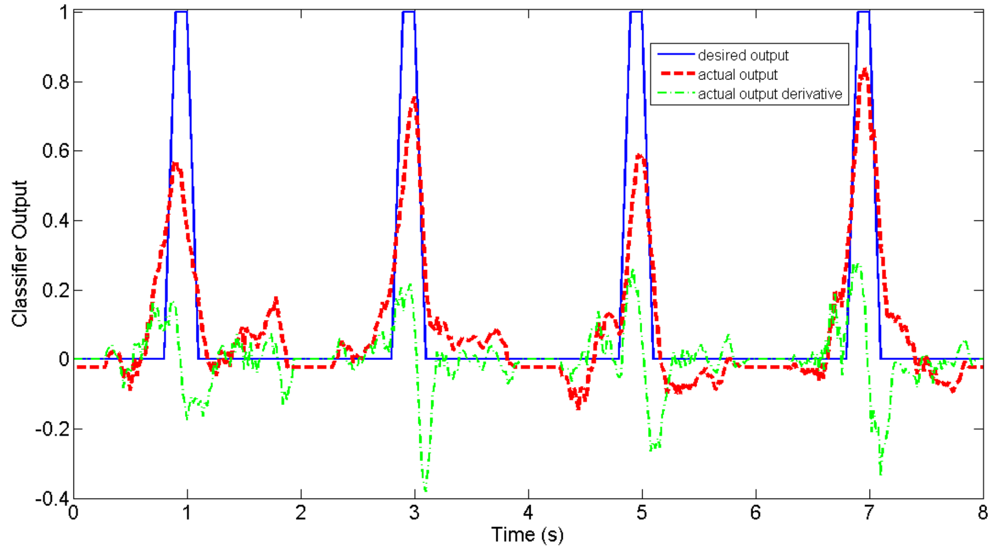

Figure 2.4: Response of linear gating classifier to a set of four movements occurring once every two seconds using the entire input space with gating classifier derivative. The blue solid line represents the function that the gating classifier was trained to, the red dashed line represents the actual response of the gating classifier, and the green line represents the discrete time derivative of the gating classifier response.

indicating no movement.

$$G_{out}(t) = \begin{cases} 1 & \text{if } G(t) > \gamma \text{ and } G(t) - G(t-3) < \delta \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

One additional issue highlighted by figure 2.4 is that the criteria for movement detection defined in equation 2.1 can be true for more than one classification decision during each movement trial. Every time the gating classifier fires it is considered a separate detection of movement, which could lead to the misclassification of a single movement as multiple movements. To reduce this occurrence a refractory period was added. Once the gating classifier output a positive movement decision, further positive decisions were ignored until at least ρ time steps had occurred.

The output of the movement classifier was also observed to determine if it still performed similarly to the original algorithm. Figure 2.5 shows the response of the movement classifier to a sequence of 12 movements occurring once every two seconds in the following order: extension of each digit, in order from thumb to little finger (e1-e5), extension of the

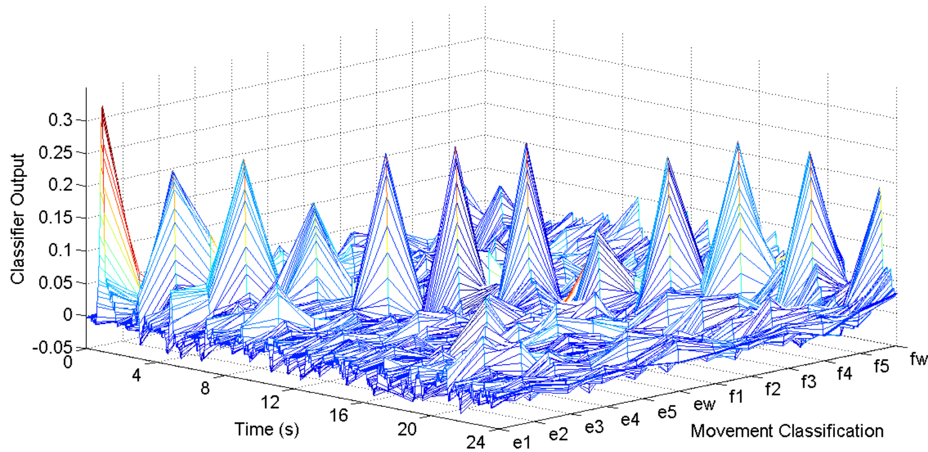

Figure 2.5: Response of linear movement classifier to a sequence of trials occurring once every two seconds using the entire input space. The movements were presented every odd second in the following order: extension of each digit, in order from thumb to little finger (e1-e5), extension of the wrist(ew), flexion of each digit in the same order(f1-f5), and flexion of the wrist(fw). The diagonal row of spikes in the center of the diagram shows that the largest response during each trial is the expected movement type.

wrist(ew), flexion of each digit in the same order(f1-f5), and flexion of the wrist(fw). The desired output for this trial is a series of output spikes that have a value of 1 when the corresponding movement is being performed and 0 all other times. As shown in the diagram, the response does not closely follow this desired output, instead peaking at a maximum of 0.32, but as in the gating classifier, enough information is still present to make a correct decision. As mentioned previously, the movement classifier output is only used when the gating classifier determines that a movement has occurred. As shown in this graph, at the time a movement occurs there is a clear spike from the desired movement type, while all other movement types show little activity. This demonstrates that choosing the movement type with the largest corresponding movement classifier output when the gating classifier fires will provide a correct movement classification. This rule is expressed mathematically in equation 2.2, where $P_i(t)$ is the output corresponding to each movement type i and $s(t)$ is the chosen movement type.

$$s(t) = \arg\max_i P_i(t) \tag{2.2}$$

2.3 Elimination of Floating Point Multiplication

Even with a reduced number of floating point multiplications this operation still represents the most computationally intensive portion of the algorithm. Fortunately, analysis of the decoder algorithm shows that floating point math can be completely replaced by integer math without introducing error sufficient to adversely affect classification decisions. Initial analysis of weights in the gating and movement classifier after training with a full input space showed that weights differed by several orders of magnitude, although the majority were less than 10^{-3}. Figure 2.6 shows the magnitude logarithm of an example set of post-training weights. Examination of this figure suggests that weight magnitudes are approximately log-normally distributed. The fact that the order of magnitude of the weights roughly falls in this distibution indicates that an approximation of these weights needs to primarily focus on preserving the accuracy of the weights closer in order to the mean. In this set of weights, the magnitude mean was $9.98 * 10^{-4}$ and the magnitude standard deviation was $1.2 * 10^{-3}$, with minimum and maximum magnitudes of $4.11 * 10^{-7}$ and $1.3 * 10^{-2}$. This range of about five orders of magnitude was typical, although exact values varied each time the system was trained.

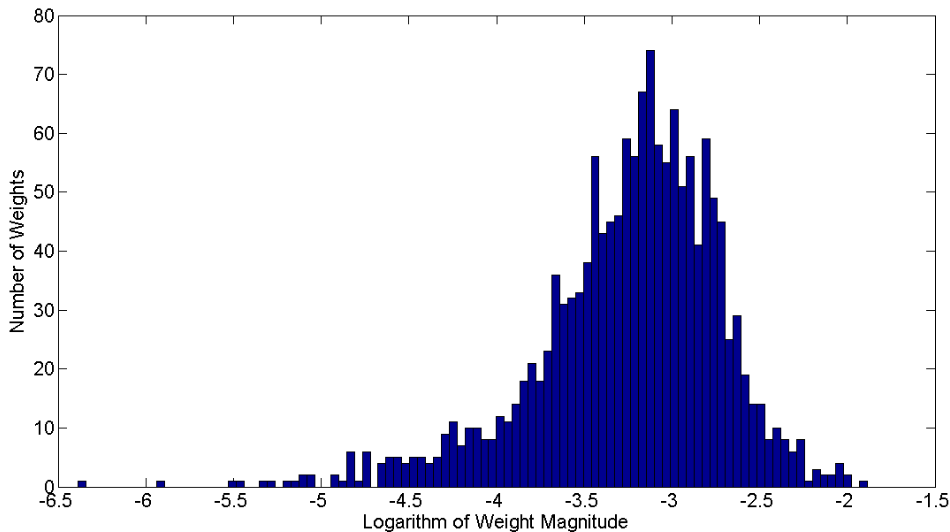

Figure 2.6: This graph shows the base-10 logarithm of a sample distribution of classifier weights after training to a full input space (1495 weights total).

With this distribution in mind a simplification that eliminated floating point multiplications was derived. Generally, a linear classifier can be described by equation 2.3, where n is the number of inputs, w is the constant weight multiplied by each input x, b is a constant offset, and y is the classifier output.

$$y = \sum_{i=0}^{n} w_i x_i + b \tag{2.3}$$

For this specific problem it is known that all inputs x are nonnegative integers with a maximum value of about 20, due to the limiting effect of refractory periods. The weights are floating point numbers and are on the order of 10^{-2} to 10^{-7}, which makes direct rounding them a poor idea. However, scaling by a large constant α (henceforth referred to as the rounding coefficient) reduces the error introduced by rounding.

$$\alpha y = \sum_{i=0}^{n} \alpha w_i x_i + \alpha b \tag{2.4}$$

Note that in this state αw_i is still a floating point number. To simplify, a new value $\hat{w}_i$ that is the nearest integer to αw_i is substituted into the equation.

$$\hat{w}_i = round(\alpha w_i) \tag{2.5}$$

$$\alpha \hat{y} = \sum_{i=0}^{n} \hat{w}_i x_i + \alpha b \tag{2.6}$$

This substitution effectively eliminates the need for floating point multiplication. To prove that this introduces negligible error rounding can be modeled as a uniform error between -0.5 and 0.5, assuming that the value of the rounding coefficient is large enough that every weight has a magnitude of at least 1, and the value of $\hat{y}$ can be compared to y.

$$\hat{w}_i = \alpha w_i + U(-0.5, 0.5) \tag{2.7}$$

Substituting this distribution into equation 2.6 gives

$$\alpha \hat{y} = \sum_{i=0}^{n} \alpha w_i x_i + \sum_{i=0}^{n} U(-0.5, 0.5) x_i + \alpha b \tag{2.8}$$

The products of uniform distributions and inputs are independent, identically distributed random variables, so their sum can be approximated as a normal distribution[15].

$$\sum_{i=0}^{n} U(-0.5, 0.5)x_i \approx N(0, \sigma^2) \tag{2.9}$$

Given that $U(-0.5, 0.5)$ has a variance of 0.0833 and a mean of 0, assuming that x_i is discretely uniformly distributed between 0 and 20 at integer intervals and therefore has a variance of 33.3 and a mean of 10, and that the number of inputs is 40, it is estimated that the variance of the normal distribution is approximately $\sigma^2 = \frac{0.0833*33.3+10^2*0.0833}{40}$ = 0.278.

$$\alpha\hat{y} = \sum_{i=0}^{n} \alpha w_i x_i + N(0, \sigma^2) + \alpha b \tag{2.10}$$

Dividing both sides by the rounding coefficient gives an equation for $\hat{y}$.

$$\hat{y} = \sum_{i=0}^{n} w_i x_i + b + \frac{N(0, \sigma^2)}{\alpha} \tag{2.11}$$

Substituting in the original equation for y gives the following

$$\hat{y} = y + \frac{N(0, \sigma^2)}{\alpha} \tag{2.12}$$

Note that as the size of the rounding coefficient increases $\hat{y} \to y$. In practice, the rounding coefficient was chosen to be the reciprocal of the smallest weight present in any classifier, and was observed to be between the order of 10^5 to 10^7. Therefore the error introduced is 99.7% likely to be less than $\pm 2.78 * 10^{-4}$, which is likely negligible when compared to the magnitude of the signals in figures 2.4 and 2.5.

This simplification is useful due to the fact that the weights are computed during training and do not change after that. Multiplying every weight by the rounding coefficient and then rounding introduces error on the order of 10^{-4} and eliminates the need for real time floating point multiplication. Furthermore, since the refractory period of a neuron prevents it from firing above a rate of about 200Hz it is very unlikely that any individual input to the decoder will ever be above 20 in any given 100ms input window. This means that inputs fall in a small range of values(0-20), which allows the integer multiplication required to be

implemented as a lookup table, greatly reducing hardware demand. Note that one floating point division is necessary if the value of $\hat{y}$ is needed, but if subsequent processing can use $\alpha\hat{y}$ instead this division is not necessary. In equation 2.1 $G(t)$, γ, and δ can be multiplied by the rounding coefficient α and rounded to the nearest integer in order to take advantage of the elimination of floating point math given in equation 2.6. By doing this the output of the gating classifier multiplied by α can be used directly in equation 2.1, eliminating the need to perform a floating point division. Additionally, since equation 2.2 chooses the maximum likely movement type out of each of the movement classifier outputs its behavior is unaffected if all movement classifier outputs $P_i(t)$ are multiplied by a constant, so no additional modifications need to be made to the movement classifier.

Chapter 3

Hardware Implementation

Once the modified asynchronous dexterous decoder algorithm was developed the next step was to design a hardware implementation. The first step in this task was to choose an architecture for the implementation and identify a target platform that could meet the requirements of the architecture. The target platform size and speed capabilities were analyzed to ensure that it could meet the requirements of the hardware implementation. Next, the digital design task was broken down into four independent stages that were designed individually and then combined to form the complete decoder. Once the digital implementation was designed the resulting hardware performance and size were analyzed.

3.1 Architecture Selection and Requirements

The first step in design of the hardware decoder was an analysis to determine an appropriate architecture for performing the required computations. The hardware decoder needed to be able to take the number of spikes that occurred in the past 20ms for each neuron and based on that data perform the necessary computations, as described above, to produce a movement classification. The necessary computations can be broken down into three stages: The first stage stores and sums the number of spikes that occurred over the sliding 100ms window, the second stage performs the linear classification by multiplying spike counts by weights and summing the result, and the third stage uses the classifier results to determine movement type. Input data is partially shared between each movement decision,

but, aside from refractory period suppression of positive movement classifications, each movement decision is independent of any other classification. Additionally, the computations performed in each classifier within a single movement period are independent of one another. It was decided that this independence should be exploited to create a hardware architecture where all classifiers and multipliers were implemented in parallel. This fully parallel approach closely mimics the theoretical algorithm structure, which is advantageous because it is easy to understand and compare to both the theoretical behavior and software implementation. Each hardware component can be easily compared to its theoretical counterpart, allowing for easy verification of proper functionality. Additionally, a fully parallel implementation should have a short critical path delay, making the 50Hz target clock rate easily achievable. Since the product of intermediate computations do not need to be stored if every computation is happening simultaneously the entire system can be clocked at the 50Hz rate, which also closely matches the theoretical algorithm structure. Optimizations undoubtedly exist, but since this work represents a first attempt at implementing this algorithm in hardware design simplicity was favored over speed and size.

Once a fully parallel approach was selected an analysis of required digital components was performed. The sliding input window requires five registers for each input to store the past five 20ms spike counts and a five input integer adder for each input. For the linear classifiers the most prevalent operations are the multiplications required to compute the weighted inputs to each classifier. Because the weights are constant when the decoder is not being trained and the input value falls in a set range of 0 to 20 each multiplier can be implemented as a lookup table, which is a basic digital design component. The number of lookup tables required to implement the decoder is equal to the number of classifier inputs multiplied by the number of movement types plus one. In order to complete each classifier an adder is necessary to sum the multiplication products. The number of adders required is equal to the number of movement types plus one, and each must have a number of inputs equal to the number of classifier inputs. For the gating logic two 2-input comparators and an adder are necessary for movement detection and one comparator with a number of

inputs equal to the number of movement types is necessary for movement classification. Additionally, three registers are needed for the derivative computation, and one register is needed for the refractory period. To give a concrete example of hardware requirements, implementing a 50 input, 12 movement type decoder in hardware would require 64 registers, 12 5-input adders, 650 lookup tables, 13 50-input adders, 2 2-input comparators, and 1 12-input comparator.

3.2 Choice of a Development Platform

The above analysis gives an indication of the resources the hardware platform had to be capable of providing to implement the decoder. For this work the ability to test the hardware was necessary, so the platform also had to have an external communication interface capable of sending data to and from a Matlab test script. It also had to be capable of computing a result at a minimum rate of 50Hz to allow for realtime movement classification. Because training was unique to each subject and the significance of recorded values could potentially change over time the linear classifier weights had to be modifiable. Based on these requirements, it was decided that a field programmable gate array would be the most appropriate platform for hardware development.

A Field Programmable Gate Array(FPGA) is an electronic device consisting of small digital memory and logic units that can be configured to form specific connections and implement arbitrary digital circuits. Typically, a hardware description is written using a hardware description language such as VHDL, synthesized to convert the hardware description to a set of physical connections, and programmed into the FPGA. The net result is a physical realization of a digital electronic circuit that can be reconfigured, which meets the requirements for a method of modifying classifier weights. Additionally, this reconfigurability also makes debugging a hardware design easier because corrections can be made and applied nondestructively. In a digital implementation of the decoder algorithm each stage could be computed in parallel rather than sequentially, which would easily allow

for the 50Hz rate to be met without issue. Finally, a single FPGA is capable of providing enough hardware to implement the dexterous decoder, as the hardware description and analysis below demonstrate.

The specific FPGA targeted for development was a Xilinx (Xilinx, Inc., San Jose, CA) Virtex-4 FX60 FPGA (Virtex-4 for short) mounted on a Xilinx ML410 embedded development platform. The Virtex-4 contains 25,280 logic slices that can be used to implement user defined functionality, 128 digital signal processing(DSP) slices that are designed for signal processing type applications, and two PowerPC processor cores that can run user defined software, among other features. Each logic slice contains two 4-bit lookup tables and two storage elements. Logic slices are organized into larger groups called configurable logic blocks(CLB), with four slices in every CLB[16]. Additionally, the ML410 platform provides 256 megabytes of DDR2 memory and a 512 megabyte flash card[17]. The moderate number of slices provided should be sufficient for a hardware implementation and the additional features are available for future development, although their use will be avoided in this research so that the hardware implementation will be portable to other FPGA platforms.

Use of the Xilinx ML410 can be justified by predicting the number of slices necessary to implement each decoder component. Each register is eight bits wide, so four logic slices will be needed for each. Each two-input adder and lookup table(5-bit input, 32-bit output) will require 16 slices to implement. Each two-input comparator will require 32 slices to implement. Components with greater than two inputs can be treated as the number of inputs minus one two-input components of the same type (i.e. a 16-input adder can be treated as 15 two-input adders). Using these estimates the 60 input, 12 movement type example given above would require 64*4 = 256 slices for registers, (48+637)*16 = 10960 slices for adders, 650*16 = 10400 slices for adders, and (2+11)*32 = 416 slices for comparators. This gives a total of 22,032 slices necessary to implement the decoder without including logic required for testing, which is less than the number of slices available on the Virtex-4 FPGA. This rough estimate does not guarantee that a 50 neuron input space will be realizable when

testing logic is incorporated, but suggests that it is likely. This estimate also suggests that a full input space (115 inputs, at minimum) will not be realizable on the Virtex-4.

3.3 Hardware Implementation Overview

From a hardware perspective, the decoder can be viewed as a series of data transformations, starting with the set of neuron spikes in the past 20ms and ending with a single number corresponding to a movement type. The three primary components are the input window, the linear classifiers, and the gating logic. The input window stores the number of spikes that occurred in the past five 20ms windows and outputs the sum of those windows, effectively creating a 100ms window that slides every 20ms. The classifiers take this data and perform linear classification, which creates a single output from each classifier. This data is then fed to the output logic, which performs the maximum and thresholding operations described above. In addition to these three components that form the decoder, a serial IO interface was also implemented to allow for hardware testing. This interface received spike counts from a Matlab program, sends those spikes to the input window, and sends back to the Matlab program the classification result created by the output logic.

VHDL `generate` statements were used extensively in the design to allow for easy synthesis of decoders with different numbers of inputs or movement types. Because of this, the component descriptions given below are in terms of various synthesis parameters. The number of input neurons is denoted n and the number of movement types is denoted m.

Algorithm training was performed offline in Matlab; as mentioned previously, hardware that can adapt online is restrictively complex and beyond the scope of this thesis. A Matlab script was created that can generate a VHDL constants file usable in hardware synthesis. Once the algorithm is synthesized and programmed into the FPGA the parameters cannot be changed without another synthesis, although, outside of testing purposes, the FPGA does not need to interact with PC once programmed and can perform movement classification

independent of any other system.

For use in a complete system it is assumed that input signals would be provided in real time, allowing the entire system to be clocked at a rate of 50Hz, which would create the 20ms time windows desired. However, because communication with test software running on a PC does not occur at a predictable rate the system clock rate was given to the IO interface, as described below.

3.4 Input Window

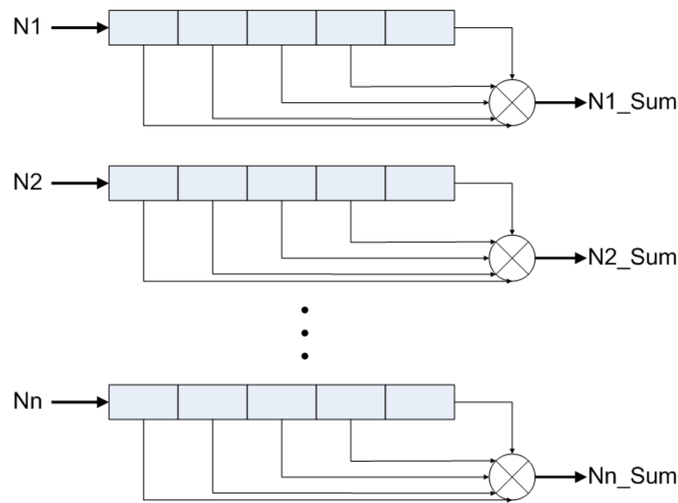

Figure 3.1: Block diagram of input window

The input window is a set of n five byte shift registers and a five input adder, as shown in figure 3.1. Input to this component is an array of unsigned 8 bit integers indicating how many spikes have occurred in each neuron since the input window was last clocked. When the input window is clocked each shift register captures its corresponding input and shifts all previous data by one, eliminating the oldest value. The contents of each shift register are summed and that sum, another unsigned eight bit integer, is sent to the classifier stage. If this stage is clocked at 50Hz it will effectively create a 100ms window that slides every 20ms. It is assumed that in a complete system the number of neural spikes in a 20ms period

would be accumulated by an asynchronous counter that would be reset by the controlling clock.

3.5 Classifiers

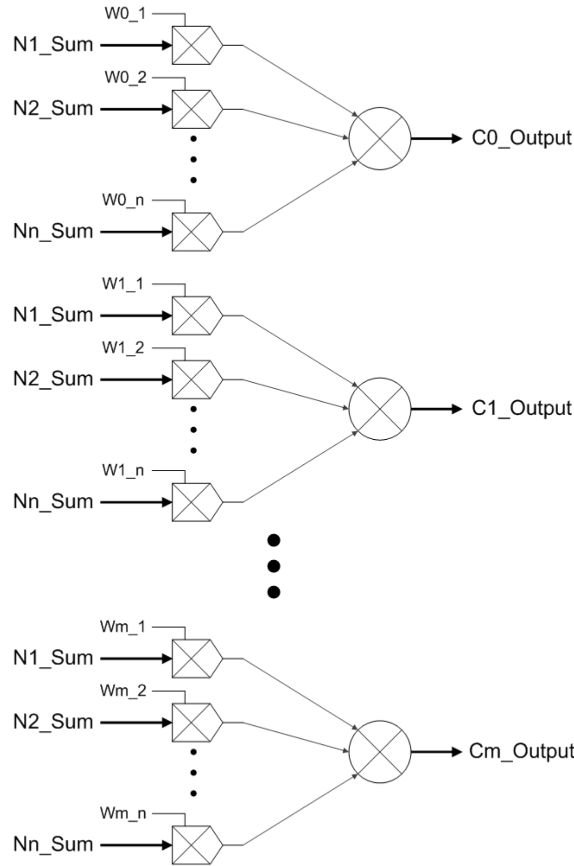

Figure 3.2: Block diagram of linear classifiers

Each classifier consists of n multipliers and one n input adder. Each multiplier, implemented as a lookup table, takes one of the summed inputs provided by the input window and converts it to a 32 bit signed integer based on the constant weight specific to that classifier and input. The n input adder, implemented as a linear array of two input adders, sums the 32 bit integers produced by the multipliers. A total of $m + 1$ classifiers were implemented: classifier zero is the gating classifier and classifiers 1 through m correspond to a

particular movement type. A block diagram of the classifiers is shown in figure 3.2. Note that this functionality consists solely of combinational logic, so no clock is necessary.

3.6 Output Logic

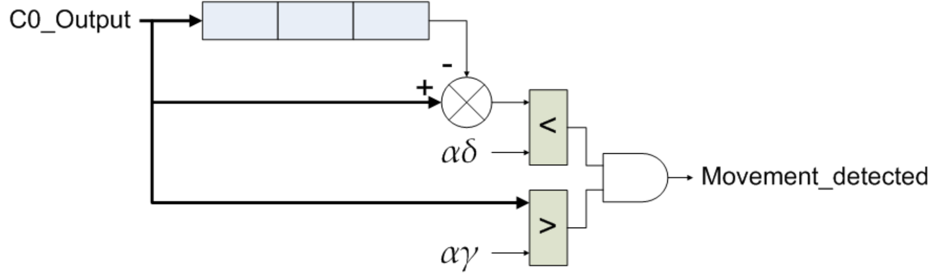

Figure 3.3: Block diagram of gating logic

The output decision logic consists of gating logic and movement logic, as in the modified algorithm described above. The gating logic takes input from the gating classifier and determines if movement has or has not occurred. The movement logic takes input from the m movement classifiers and determines the index of the largest movement value. Note that since the multipliers used in the classifiers use weights scaled by α the output logic constants must also be scaled by α.

The output of the gating classifier is sent to a three value 32 bit shift register that holds the past three outputs of the gating classifier. The difference of the current output and the output that occurred three clock cycles ago is then thresholded against the derivative constant $\alpha\delta$, with inputs less than $\alpha\delta$ generating a high output. Additionally, the current gating classifier output is also compared to a level threshold $\alpha\gamma$, with input greater than $\alpha\gamma$ generating a high output. The output of these two thresholds is anded together and used to gate the output of the movement classifier. Figure 3.3 shows the block diagram of the gating logic.

Movement type is determined by the classifier with the largest output. This determination is done by an array of two input signed comparators that accept as input two values

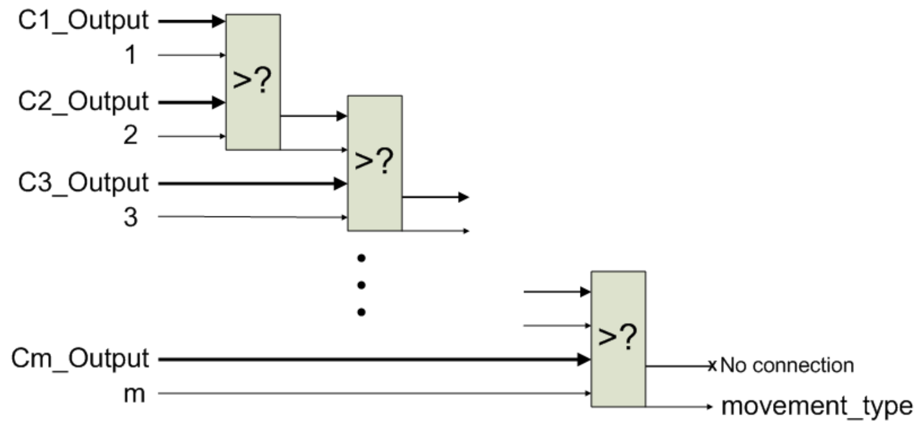

Figure 3.4: Block diagram of movement logic

and their corresponding indices and output the higher value and its corresponding index. The output of a comparator is connected to the input of another comparator, which is also connected to a different input. At the end of the line of comparators the highest value and its index is obtained, and the index of this value is the movement logic output. Each index and the output is an unsigned eight bit integer. Figure 3.4 shows the block digram of the movement logic.

The output of the gating logic anded with the output of the movement logic, effectively outputting a zero when no movement is detected and the movement type when movement is detected. Additionally, a counter is used to create the refractory period. The counter decrements every clock cycle when its value is nonzero and is set to a value of ρ every time a nonzero output is generated. If the counter is not zero when movement is detected it prevents the movement type from being sent. This prevents a second classification from occurring less than ρ time units after a classification, which helps reduce the false positive rate. Figure 3.5 shows the block diagram of the complete output logic circuit.

3.7 IO Interface

The IO interface was built solely to demonstrate hardware functionality and is not a component that would be present in a real world implementation. However, its design is important

for efficient testing of the hardware algorithm. It is responsible for receiving neuron spike counts from a serial port, storing those values for the input interface, clocking the algorithm implementation, and transmitting outputs over the same serial port. The IO interface is configured to communicate at a 115200 baud rate with an 8N1 data format. Its operation is controlled by the transmission of synchronization bytes(sync bytes) from the PC connected to its serial interface. When a sync byte, defined as 255, is received by the IO interface from the serial port it clocks the algorithm implementation, transmits the current movement classification, and resets its input index to prepare for another set of inputs. After the sync byte, it is expected that the IO interface receives the spike counts for each neuron in order. The IO interface stores each spike count in a register connected to the corresponding input window signal. After every register has been filled with a spike count the IO interface waits for another sync byte, at which point the currently held register values are clocked into the input window and the process repeats. Note that since the software used to verify functionality will be designed in Matlab no guarantees of data transmission rate will be made. The PC is expected to send data as fast as it can, but is not expected to send data fast enough to meet the 50Hz realtime requirement. This will not be a limitation in the implementation of this hardware in a prosthetic system since no PC will be present, so this issue does not violate the timing requirement. It is assumed that the system sending the data to the IO interface performs all verification functionality; the IO interface does not check the correctness of the data it sends and receives. Figure 3.6 shows the block diagram

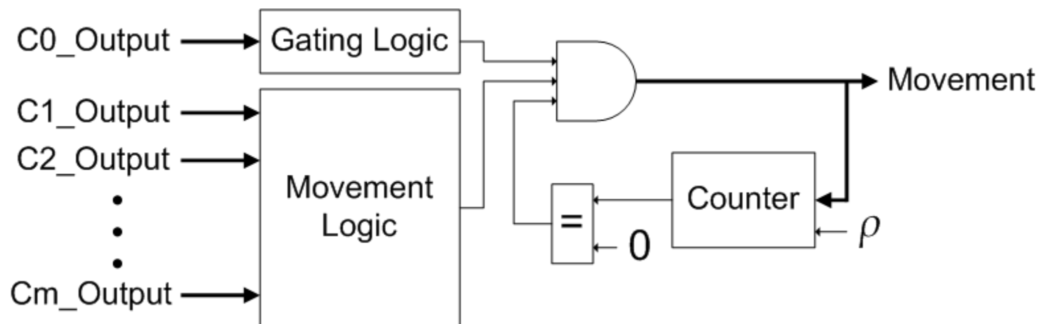

Figure 3.5: Block diagram of output logic

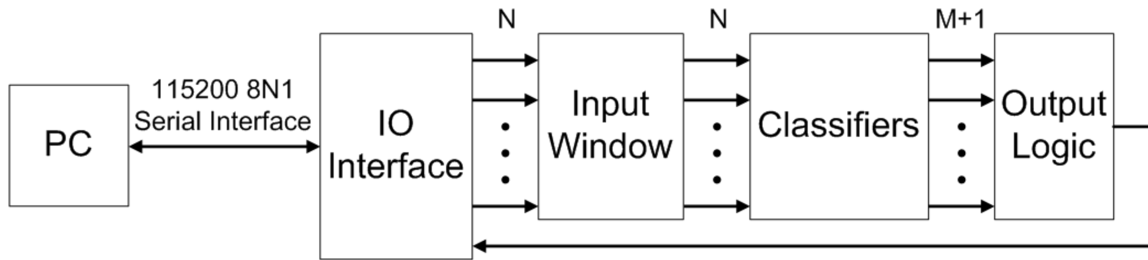

Figure 3.6: Block diagram of asynchronous dexterous decoder hardware implementation with IO interface

for the IO interface connected to the complete hardware implementation.

3.8 Hardware Analysis

After each component was designed and the decoder system was constructed limitations on computation speed and algorithm size were analyzed. The significant metrics obtained from this analysis were the maximum critical path delay and percent of available resources used with respect to input dimension.

Figure 3.7 shows a graph of the critical path delay of the hardware realization with respect to decoder input dimension. The maximum allowable delay for a realtime implementation of this algorithm is 20ms, which is never approached by this implementation. Therefore, this implementation meets the computation speed requirement.

Figure 3.8 shows a graph of the number of FPGA slices required to implement the linear classifier with a given input dimension. The synthesis results show slightly more efficient use of FPGA resources than predicted. The rough estimate performed earlier suggested that about 22,000 slices would be required to implement a decoder with an input space of 50 neurons, but the results show that less than 20,000 were actually required, even with the IO interface. This discrepancy is most likely due to the synthesis software performing automatic optimizations that were not accounted for in the original estimate. As predicted, the Virtex-4 FPGA used for this experiment does not have enough resources to utilize the full input space for any monkey, but can realize a synthesizable decoder with an input space

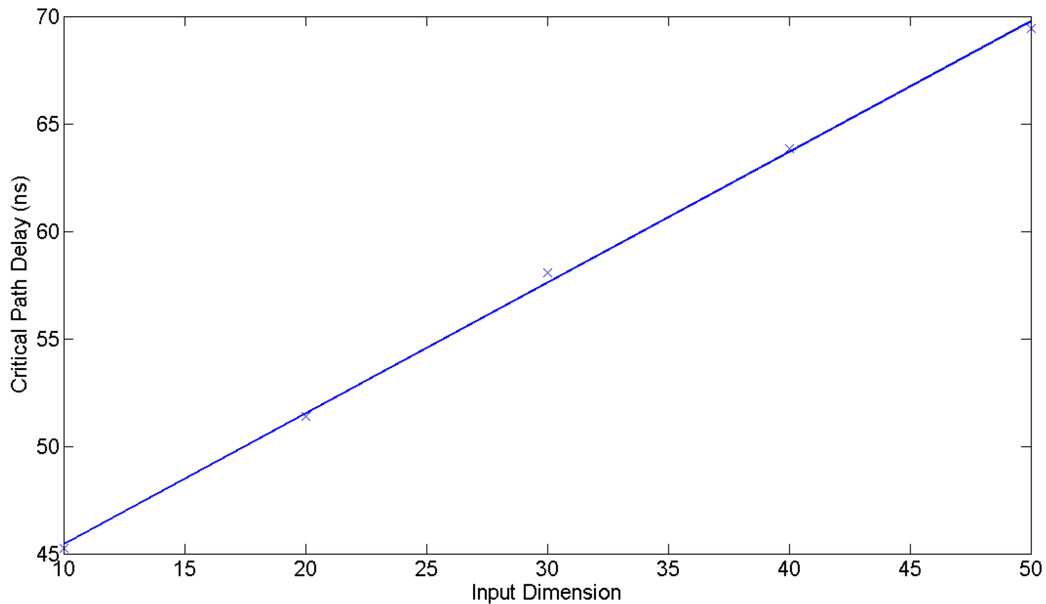

Figure 3.7: Critical path delay with respect to input space dimension. Note that since the graph scale is on the order of nanoseconds the 20ms maximum is not an issue.

of up to 60 neurons.

The results in figures 3.7 and 3.8 show that while it is possible to exceed the hardware resources available on a single FPGA the time required to compute a result is orders of magnitude less than the realtime computation requirement. Extrapolating from figure 3.7, it can be estimated that even with a full, 312 neuron input space, the largest for any subject, the critical path delay would be roughly 260ns, which, when compared to the 20ms realtime computation requirement, indicates that only $1.3 * 10^{-4}\%$ of the available time is being used to do work. Using a different hardware decoder architecture the extra time available could be substituted for space, which would result in a hardware implementation that could either do more work or be implemented on a smaller FPGA. However, the purpose of this work is simply to demonstrate that the viability of a hardware implementation, and for this purpose it was preferred to design an implementation that was as similar to the theoretical structure as possible. Optimizing the hardware implementation is a task that will be left to future work.

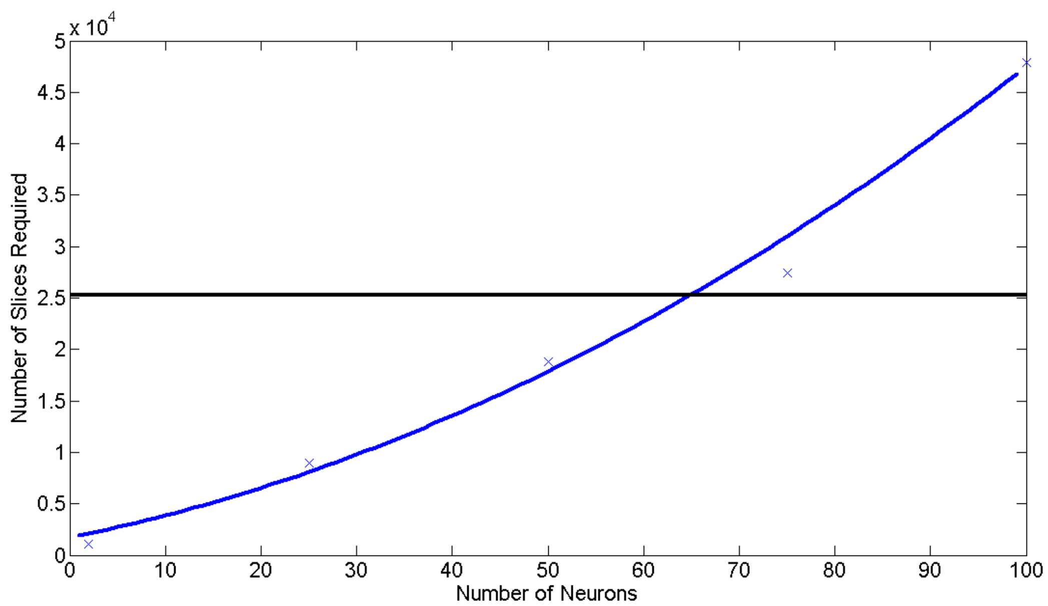

Figure 3.8: Number of FPGA Slices required to implement linear classifier with respect to input space dimension. The black dashed line is the total amount of available slices on the Virtex-4 FX60 FPGA.

Chapter 4

Test Methodology

The desired goal of this research is to demonstrate the viability of the hardware implementation described above. However, the significant modifications made to the underlying algorithm necessitated additional software level testing to verify that it could still provide adequate performance. As a result, preliminary testing in Matlab, using floating point multiplications, was performed prior to the testing of the hardware implementation. All testing was performed using the same primate data used by Aggarwal *et al.*to allow for comparison of new data with previous results.

The primate data used for all testing was obtained from a previous experiment involving three male rhesus monkeys, identified as C, G, and K. Each monkey was trained to either flex or extend a single digit or wrist in response to specific visual cues, for a total of 12 movement types. Additionally, monkey K was trained to perform six combined finger movements along with the 12 single-finger movements. If the subject successfully performed the desired action within a 700ms time limit it was given a water reward. Single unit recording from a surgically implanted recording unit contralateral to the target hand was used to capture the neural activity of the subject during movement trials. Individual trials took roughly two seconds to complete, and all recordings were centered so that movement occurred at the one second mark. Neural activity was recorded from 312 neurons in monkey C, 125 neurons in monkey G, and 115 neuron in monkey K[1]. The time at which action potentials occurred in each neural recording was quantified relative to movement time and was used to form counts of the number of action potentials in each 20ms period.

This data could then be used in the manner described previously. Testing was primarily performed using data from monkey K, although results from C and G were considered to ensure generalized performance.

4.1 Determination of gating classifier parameters

The first tests performed were to determine values for the gating classifier parameters δ, γ, and ρ that would maximize correct movement identifications while minimizing false positives. This was done in two steps. First, a point by point search was performed to determine the optimal combination of derivative threshold δ and level threshold γ. Since the values for δ and γ were both related to movement detection accuracy they had to be considered together. A script was created that would train a gating classifier and then step through values for both parameters incrementally. For each pair of parameter values the gating classifier was provided with a sequence of movements. The movement detection rule using the current pair of parameters was applied to the output of the gating classifier. If the rule produced a high output solely in a 160ms window around the expected movement time and not prior to or after this window the trial was considered a success. An exception was provided for this rule to allow for the effect of the refractory period: if the rule produced a high output during the expected window then the 100ms following the expected window were ignored and not classified as false positives. Once this experiment was performed the parameter combination that produced the highest number of successful trials was selected for use in all other experiments. Selection of ρ was performed similarly. A gating classifier was trained and the false positive rate was observed for a set of movement classifications with different values of ρ. The smallest value of ρ that minimized false positives caused by a second classification of a single movement was selected for use in all further experiments.

4.2 Classifier testing

The next step was to test the gating and movement classifiers individually. Two scripts was created that trained and then tested a classifier of each type with the full input space of a specified subject. For the gating classifier, data was collected on the number of trials that either produced no movement classification, produced more than one movement classification, produced a classification prior to the expected movement time window, or produced a classification after the expected movement time window. Any classification that occurred within the refractory period was ignored. For the movement classifier, data was collected on movement classification accuracy. A movement classification was considered correct if, during the expected movement time window, the most likely movement type that occurred the most matched the expected movement classification. Testing of both classifiers was performed for each of the three subjects.

Once accuracy analysis was performed for both the gating and movement classifiers, a script was developed to train and test the whole decoder with different neuron input sets. Additionally, this script could reduce the input space by iteratively training and then removing the neuron whose net weight magnitude was the least. It should be noted that in destructive input space reduction the standard practice is to train and test the network with every input removed one at a time and then chose the configuration that affects the accuracy the least. This approach was not used because computationally it is of order n^2, whereas the approach used was of order n but still took as long as 18 days to compute in the case of monkey C. Analysis of data presented in the results section included in the discussion section provided validation of this approach. To determine decoder accuracy in a number of different configurations, testing began with the full input space and iteratively trained, tested, and removed an input until the input space was null. Accuracy was recorded for each test, providing a measurement of decoder accuracy with respect to input space size. This test was performed for each of the three monkeys and the results were compared. Additionally, for monkey K, decoder accuracy was compared between the individual movement case (12 classes) and the individual and combined movement case (18 classes).

4.3 Comparing Hardware and Software Implementations

To demonstrate that the hardware decoder described in chapter 3 performed computation identical to the linear decoder implemented in Matlab software, both a gating and movement classifier were trained, the classifier weights were used to synthesize a hardware implementation for the Virtex-4 FPGA, and a script was used to feed identical trial data to both the software and hardware decoders. The software decoder used floating point computations to produce results, whereas the hardware decoder used the simplified computations described in section 2.3. Since the only visible product of the hardware decoder is a sequence of classification decisions, this was the metric used for comparing both implementations. A list of positive output classifications and the corresponding trial number and classification time were recorded for both the hardware and software decoders and compared. Since the objective of this experiment is to demonstrate that the hardware and software decoders behave identically a random input set was used and accuracy was not observed. This experiment was performed for multiple input spaces of different dimension (dimensions of 10, 20, 30, 40, and 50 were used) to demonstrate that the hardware implementation scaled properly with respect to input dimension.

Chapter 5

Results

The results of the test to determine optimal values for the gating classifier parameters is shown in figures 5.1, 5.2, and 5.3. The experiment to determine level threshold γ and derivative threshold δ parameters was performed twice, a decision made after analysis of the first experiment. Figure 5.1 shows the result of the first experiment, which tested a value range of -0.1 to 0 in 0.01 increments for δ and of 0.2 to 0.5 in 0.025 increments for γ, with 500 samples per point. A higher precision for both values was desired, so the experiment was repeated in the region of highest accuracy of the first experiment, from -0.06 to -0.01 in 0.005 increments for δ and from 0.25 to 0.35 in 0.01 increments for γ, with 750 trials per point. Figure 5.3 shows the percent of false positives that occurred for values of the refractory period from no period to a period of 10 windows. Every increment of ρ represents an additional 20ms delay after movement is detected during which no second classification is allowed. From these graphs, it was determined that gating classifier error was minimized when $\gamma = 0.31$ and $\delta = -0.055$. The false positive rate was found to reach its minimum when $\rho = 9$, so a refractory period of 180ms was selected.

Tables 5.1 and 5.2 summarize the gating and movement classifier accuracy results, respectively. For the gating classifier, statistics were recorded on the percent of trials in which no classification occurred, false positives occurred, early classification occurred, or late classification occurred. For the movement classifier only classifier correctness during the expected movement window was observed.

Figure 5.4 shows the accuracy of the complete decoder algorithm with respect to input

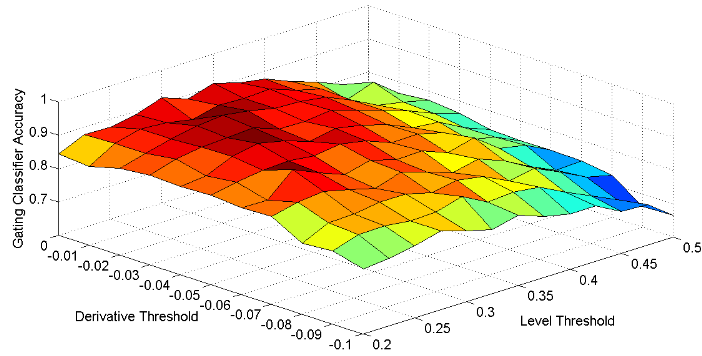

Figure 5.1: Movement detection accuracy rate of the gating classifier with respect to the level threshold and derivative threshold parameters γ and δ in the region $0.2 \leq \gamma \leq 0.5$, $-0.1 \leq \delta \leq 0$.

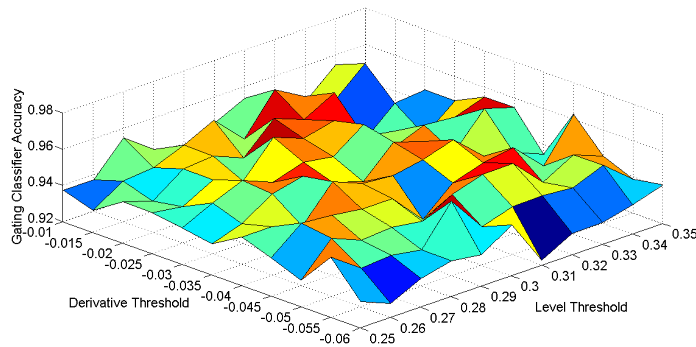

Figure 5.2: Movement detection accuracy rate of the gating classifier with respect to the level threshold and derivative threshold parameters γ and δ in the region $0.25 \leq \gamma \leq 0.35$, $-0.06 \leq \delta \leq -0.01$.

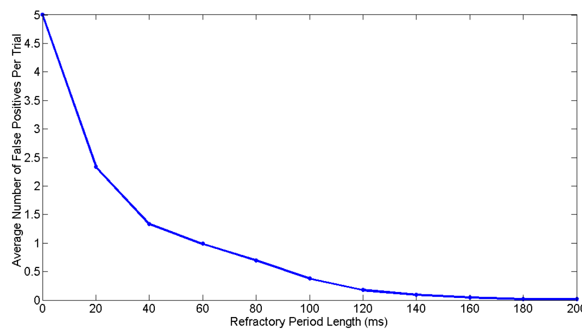

Figure 5.3: False Positive rate of gating classifier with respect to the gating classifier refractory period parameter ρ.

Subject Monkey	Input Dimension	Percent Missed	Percent False Positive	Percent Early	Percent Late
C	312	0.01	0	0	0.01
G	125	0.17	6.67	3.69	7.4
K(indiv)	115	2.24	0.14	0.08	0.47
K(comb)	115	1.29	4.91	0.22	2.51

Table 5.1: Summary of gating classifier accuracy using the full input space of each subject.

Subject Monkey	Input Dimension	Percent Correct
C	312	100
G	125	100
K(indiv)	115	99.88
K(comb)	115	93.3

Table 5.2: Summary of movement classifier accuracy using the full input space of each subject.

dimension for each subject, including the combined movement case for monkey K. Each point was obtained by performing 100 trials per movement type for each input dimension, starting from the full input space. For clarity, only input dimensions 0 through 80 are shown. Above an input dimension of 80 accuracy did not improve for any subject.

Comparison of hardware and software decoder implementations showed some variation between the two implementations. Tables 5.3, 5.4, 5.5, 5.6, and 5.7 show each case where hardware and software output differed during comparison of decoders with input dimension of 10, 20, 30, 40, and 50, respectively. For each trial the expected output is provided along with movement decisions made by the hardware and software decoders and the time at which each decision was made. All trials not presented here showed identical activity between hardware and software decoder.

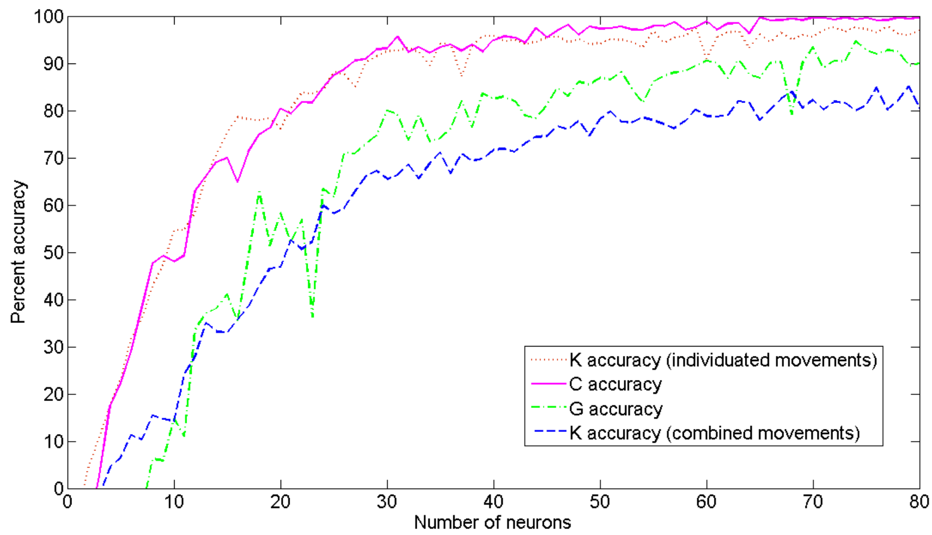

Figure 5.4: Linear classifier decoding accuracy with respect to input space size. Note that accuracy starts to drop around approximately 60 neurons, indicating that a larger input space is necessary to attain a level of accuracy similar to that of the nonlinear classifier. Additionally, note that the accuracy of the combined movement trials is significantly worse than the nonlinear classifier.

Trial Number	Expected Output	Software Decoder Output	Software Decoder Output Time(ms)	Hardware Decoder Output	Hardware Decoder Output Time(ms)
13	e1	0	x	e1	1100
15	e3	e3	1000	0	x
21	f3	f3	1080	0	x
27	e3	fw	1020	e3	1020
31	f1	fw	920	f1	920
32	f2	fw	1060	f2	1040
33	f3	f3	1060	f3	960
37	e1	e5	860	e1	860
39	e3	fw	940	e3	940
45	f3	ew ew	800 1120	ew	820
59	f5	fw	1040	0	x
61	e1	f5	980	0	x
65	e5	fw	1080	f4	1080
69	f3	fw	1100	f3	1100
70	f4	fw	1120	f4	1120
82	f4	f4	1060	0	x
94	f4	f5	1040	f4	1080
100	e4	e4	1040	f3	1040

Table 5.3: Table of output differences between hardware and software implementation over 100 trials with an input dimension of 10.

Trial Number	Expected Output	Software Decoder Output	Software Decoder Output Time(ms)	Hardware Decoder Output	Hardware Decoder Output Time(ms)
4	e4	fw	1120	f5	1120
5	e5	fw	1120	e5	1120
9	f3	fw	1000	f3	1000
10	f4	fw	1060	f4	1060
16	e4	fw	1080	e4	1080
21	f3	f5	1100	f3	1100
28	e4	fw	980	e4	980
33	f3	fw	1060	f3	1060
34	f4	f5	1060	f4	1060
46	f4	f3 f4	840 1060	f4 f5	840 1060
52	e4	0	x	e4	1020
71	f5	fw	1160	e3	1160
76	e4	e3	1040	e4	1040
88	e4	fw	1000	e4	1000
93	f3	f5	1080	f3	1080
94	f4	fw	1080	f4	1080

Table 5.4: Table of output differences between hardware and software implementation over 100 trials with an input dimension of 20.

Trial Number	Expected Output	Software Decoder Output	Software Decoder Output Time(ms)	Hardware Decoder Output	Hardware Decoder Output Time(ms)
1	e1	e1	980	0	x
5	e5	0	x	e5	1120
9	f3	fw	1060	f3	1060
18	ew	0	x	ew	1020
34	f4	0	x	f4	1020
35	f5	f5	1100	0	x
40	e4	e3	1100	e4	1100
45	f3	fw	1060	f3	1060
57	f3	fw	1000	f3	1040
69	f3	fw	1040	f3	1040
71	f5	fw	1120	0	x
81	f3	0	x	f3	1100
95	f5	0	x	f5	1140

Table 5.5: Table of output differences between hardware and software implementation over 100 trials with an input dimension of 30.

Trial Number	Expected Output	Software Decoder Output	Software Decoder Output Time(ms)	Hardware Decoder Output	Hardware Decoder Output Time(ms)
1	e1	0	x	e1	1100
5	e5	e5	1100	e5	1230
9	f3	f4	1040	f3	1040
10	f4	f4	1020	f4	920
14	e2	e5	1040	e5	1020
21	f3	0	x	f3	960
34	f4	f4	940	f4	880
45	f3	fw	1020	f3	1020
46	f4	f4	1020	f4	920
58	f4	f4	1000	f4	940
69	f3	f4	1060	f3	1060
70	f4	f4	1020	f4	1140
73	e1	e1	1100	f5	1100
76	e4	fw	1080	e4	1080
78	ew	0	x	ew	1000
96	fw	0	x	fw	1100

Table 5.6: Table of output differences between hardware and software implementation over 100 trials with an input dimension of 40.

Trial Number	Expected Output	Software Decoder Output	Software Decoder Output Time(ms)	Hardware Decoder Output	Hardware Decoder Output Time(ms)
14	e2	e2	1060	e2	1000
15	e3	e3	1040	e3	1000
57	f3	f5	1120	f3	1120
58	f4	f4	900	f4	1020

Table 5.7: Table of output differences between hardware and software implementation over 100 trials with an input dimension of 50.

Chapter 6

Discussion

6.1 Algorithm Modification Analysis

From the results presented in chapter 5 it was concluded that the goal of maintaining comparable accuracy (above 90%) for the individual movement cases was met. Table 6.1 shows a comparison of accuracy between the original nonlinear artificial neural network approach and the linear approach presented in this paper. As shown, for the linear ANN monkey C performs better than the original algorithm with an input space of 50 or greater, and comes within 0.1% of reaching the 99.8% accuracy of monkey K in the individual movement task, which was the highest documented accuracy of the original algorithm. For the linear ANN monkey G achieved similar levels of accuracy to the nonlinear ANN approach when an input space of over 60 neurons was used. Also, for the linear ANN monkey K performed above 95% accuracy for the individual movement task, although this was lower than the 99.8% accuracy achieved by the nonlinear ANN approach. It was expected that algorithm simplification would reduce accuracy to some extent, so the loss is considered acceptable for this work. The one loss of accuracy that was higher than expected and considered unacceptable was for monkey K in the combined movement task, which never got much higher than 80%, as shown in figure 5.4.

Data presented in tables 5.1 and 5.2 suggest that gating classifier is primarily responsible for decoder failures. The percent missed and percent false positive statistics show explicit failures of the gating classifier that lead to incorrect classifications. Future work on

subject monkey	classification algorithm	25 neurons	35 neurons	40 neurons	50 neurons	60 neurons	80 neurons
C	nonlinear	85.0%	92.9%	96.2%	-	-	-
C	linear	87.4%	93.4%	94.9%	97.3%	98.8%	99.7%
G	nonlinear	72.9%	78.3%	90.5%	-	-	-
G	linear	61.6%	74.2%	82.4%	86.9%	90.5%	89.9%
K (indiv)	nonlinear	95.4%	98.3%	99.8%	-	-	-
K (indiv)	linear	88.1%	94.2%	95.8%	94.3%	95.4%	95.3%
K (comb)	nonlinear	82.4%	86.7%	92.5%	-	-	-
K (comb)	linear	58.2%	71.2%	71.7%	78.2%	78.8%	80.5%

Table 6.1: Comparison of original nonlinear artificial neural network and modified linear artificial neural network algorithm accuracy with respect to input dimension. Information on original algorithm accuracy was obtained from [1], which only provided information up to 40 neurons.

this subject should focus on minimizing these failures in order to increase decoder accuracy. One potential solution would be to swap the linear gating classifier for the a nonlinear classifier by using a software coprocessor or an alternative hardware implementation to compute the required number of floating point operations. Additionally, the significant amount of early and late classifications, especially for monkey C, may cause a movement misclassification when the movement classifier made a correct selection during the expected movement time. This idea is also supported by the very high movement classifier accuracy, which demonstrates that, aside for the combined movement case, the movement classifier correctly classifies movement type for the majority of the expected movement window. Note that the movement classifier accuracy test did not account for the window in which movement was detected. Currently, the decoder only observes the movement classifier output at the time the gating classifier detects movement, which can allow for noise in movement classifier output to lead to incorrect classifications. In future work, this could potentially be reduced by changing the movement decision to incorporate movement classifications that occurred at previous times.

The reduced accuracy demonstrated by the movement decoder when combined movements were incorporated highlights an important limitation of this implementation. The lower accuracy of the movement classifier with combined movements as compared to single finger movements, shown in table 5.2, suggests that a linear ANN may be incapable

of completely distinguishing single finger movements from combined movements. This is most likely caused by the fact that some overlap occurs in the neural signals for individuated and combined movements, resulting in movement classifier confusion. Surprisingly, training with combined movements also decreased the accuracy of the gating classifier by 4.77%. Since the neuron input space was identical between the individuated and combined movement trials, this could be a product of the different forms of neural activity that occur during combined movements.

6.2 Hardware Implementation Analysis

The discrepancies between the behaviors of the FPGA hardware decoder implementation and the decoder implemented in Matlab software highlights a potential flaw in the design assumptions presented in sections 2.3 and 2.2. The transition from equation 2.8 to equation 2.9 assumes that the product of rounding errors and input values can be approximated as a Gaussian distribution. This approximation allows for the assumption that rounding error will become negligible as the rounding coefficient becomes sufficiently large. As shown in the results section, the error introduced by rounding is not negligible, due to the fact that it is capable of causing changes in performance. This incorrect approximation can potentially be attributed to one of three assumptions: that the input dimension is large enough for the central limit theorem to be applicable, that individual inputs can be approximated as stationary uniform distributions, and that the rounding coefficient is actually large enough to negate error introduced by rounding. Given that the input dimension ranged from 10 to 50 in the hardware-software comparison tests it is likely that the central limit theorem is actually not applicable, particularly at the smaller dimensions. Additionally, assuming that individual inputs are stationary and uniformly random is clearly incorrect, otherwise any attempt to extract time dependent information from these signals, such as in the gating and movement classifiers, would fail. However, it is unclear if this approximation is sufficiently inaccurate to cause a difference in system performance. In terms of rounding, it is possible

that since the rounding coefficient was chosen to be the multiplicative inverse of the smallest weight magnitude that the weights on the same order of magnitude were rounded in a non-negligible manner. Approximating rounding error as a uniform distribution fails to capture this possibility, and therefore may not have been the ideal model. The weight distribution shown in figure 2.6 demonstrates that the weights range across several orders of magnitude, which would cause rounding to have a greater impact on smaller weights, potentially changing their contribution to the output signal enough to change its classification. Note that it may be possible, if the assumptions that allow for the approximation in equation 2.12 are true, to reduce the error introduced by the approximation by simply increasing the rounding coefficient. One additional potential source of the discrepancy is the multiplication and rounding performed on the derivative and level thresholds. These fixed thresholds are multiplied by the rounding coefficient and then rounded to eliminate the need for any floating point math; it is possible that in some cases this rounding changes the thresholds in a non-negligible manner. Regardless of cause, these discrepancies do not adversely affect hardware performance (Out of all 67 observed discrepancies, 11 were solely timing related, 43 were caused by incorrect software classification only, 8 were caused by incorrect hardware classification only, and 5 were caused by both incorrect software and hardware), so the hardware decoder can still be considered as accurate as the software implementation of the decoder.

One of the limitations of this implementation is that it is only able to realize an input space of at most 60, although it was demonstrated in figure 5.4 that an input space of at least 80 is needed to reach peak decoder accuracy. Given the discrepancy between size and time utilization highlighted in section 3.8 it seems feasible that an alternative implementation to the one designed in this work would be capable of providing the required input dimension without exceeding size constraints. One potential alternative would be to perform the multiplications for each classifier serially, rather than in parallel. This could be done by multiplexing the spike count inputs and a lookup table of weight values for a classifier into a single multiply-accumulate unit. If the multiplexer and multiply-accumulate unit were

clocked at a rate equal to the input dimension times faster than the 50Hz clock this serial classifier implementation would function identically to the fully parallel implementation presented in this work. The serial implementation would scale in both space and time as input dimension was increased, but space requirements would grow at a slower rate because only the weight lookup table and multiplexers would increase in size. Alternatively, weights could be stored in random access memory outside of the FPGA, which would eliminate the growing space requirement for weights. Time requirements would increase at a higher linear rate, but given the current imbalance between space and time utilization this is not an issue. Additionally, since the number of multipliers in this approach would be constant, floating point arithmetic could be used without increasing the rate at which space requirements increased with respect to input dimension.

Recent trends have shown steady increases in FPGA size. At the time of publication, the most recently released commercially available Xilinx FPGA, The Virtex-6 SXT line of products, can incorporate as many as 74,400 logic slices, each of which provides 2 more lookup tables and six more flip flops than a Virtex-4 logic slice, and 2,016 DSP slices, each of which contains a 25 by 18 integer multiplier[18]. An increase in available resources could allow for implementation of decoder features that were restrictively resource intensive during development. One obvious result would be the ability to implement a decoder that used a larger input space than was feasible for the Virtex-4. A less trivial use of the added resources would be the implementation of a hardware learning system. A hardware learning system could be implemented if the roughly 1000 lookup tables required to implement a fixed parameter hardware decoder were replaced by integer multipliers that performed the product of classifier weights and input spike counts. This would allow for the classifier weights to be updated by a training component that could adjust each value based on the learning heuristic used. If training speed was not an issue, classifier weights could be multiplexed to the training system, allowing for efficient space utilization. Since each Virtex-6 DSP slice contains a multiplier large enough to perform one of the required multiplications the learning system would theoretically be implementable on a single Virtex-6

FPGA. Additionally, decoder accuracy could be improved by using an alternative to the linear gating classifier. Research performed on the subject of nonlinear ANN implementation on FPGAs has demonstrated that multiplier multiplexing techniques can maximize resource utilization on currently existing FPGAs[14], which may allow for the implementation of an nonlinear gating classifier.

Chapter 7

Conclusions

The goal of this research was to work toward a mobile, brain controlled prosthetic device by creating a hardware implementation of an asynchronous dexterous hand movement decoder that could theoretically be used to control a prosthetic hand. Previous research provided an algorithm that was capable of accurately decoding hand movements from neural recordings[1], but it required a high number of floating point operations. Modifications were made to reduce the computational complexity of the algorithm to the point where it could be implemented on a single Virtex-4 FX60 FPGA. A hardware implementation was designed and demonstrated to operate at accuracy comparable to the original algorithm in real time, except in the case of combined movements. The fully parallel hardware implementation was observed to be inefficient in its use of available computation time, suggesting that an alternative digital architecture may be better suited to this task.

The primary objective of this research, to create a hardware implementation that could decode movements in realtime with an accuracy of at least 90%, was met. The first phase of this work, reducing the complexity of the original decoder algorithm while maintaining comparable accuracy to it, was accomplished by using linear ANNs and by using integer multiplication instead of floating point multiplication. The use of linear ANNs changed the gating classifier signal in such a way that the movement detection heuristic in the origial algorithm was unable to meet accuracy requirements with any combination of parameter values, so an alternative detection scheme was created based on the gating classifier signal and its derivative that did meet the requirement. Additionally, the approximation used to

replace floating point math with integer math was observed to change the decoder output in some instances and was therefore flawed, although it did not adversely affect accuracy. The second phase of this work, the design and implementation of a hardware decoder, led to creation of a decoder that could use an input space up to 60 neurons on a single Virtex-4 FPGA. Accuracy analysis showed that this input space size limitation is not ideal, as maximum accuracy is reached at an input space of at least 80 neurons, so an alternative implementation would be preferable in future work. However, the hardware implementation does successfully demonstrate proof of concept and provides a baseline for future hardware implementations.

The ability to decode individuated hand movements from neural signals in a hardware system is a significant advancement toward the development of a next generation prosthesis. However, additional work must be performed before a neurally controlled prosthetic system can approach the functionality of a biological human hand. Two limitations demonstrated by this research significantly restrict the ability of a prosthesis to be truly dexterous: the difficulty in distinguishing between individuated and combined finger movements and the need for a refractory period following a movement classification to prevent false positives that lead to unintended prosthesis activity. Aside from some anatomical limitations, the average human is capable of moving each joint in each finger on a hand simultaneously and independently of one another. In contrast, the decoder presented in this research is capable of making an individual movement once ever 180ms, which would allow for only about 5.5 movements per second. While this is an important first step toward creating a hand prosthesis, it is still a long way from being truly natural.

That being said, this research does provide a good starting point and baseline accuracy measurements for future hardware implementations of movement decoder algorithms. Increases in FPGA complexity and alternative algorithm implementation methods will remove limitations on what is feasible in hardware, enabling improvements that may eventually lead to a prosthesis comparable to the human hand. Future implementations must focus on the problem of further increasing accuracy to the point where reliability is not an

issue for most subjects, the problem of allowing for accurate classification of simultaneous and rapid movements, and the constraints imposed by the need for portability.

Bibliography

[1] V. Aggarwal, S. Acharya, F. Tenore, H.-C. Shin, R. Etienne-Cummings, M. Schieber, and N. Thakor, "Asynchronous decoding of dexterous finger movements using m1 neurons," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 16, no. 1, pp. 3–14, Feb. 2008.

[2] P. Afshar and Y. Matsuoka, "Neural-based control of a robotic hand: evidence for distinct muscle strategies," *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 5, pp. 4633–4638 Vol.5, April-1 May 2004.

[3] S. Acharya, F. Tenore, V. Aggarwal, R. Etienne-Cummings, M. Schieber, and N. Thakor, "Decoding individuated finger movements using volume-constrained neuronal ensembles in the m1 hand area," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 16, no. 1, pp. 15–23, Feb. 2008.

[4] K. H. Elaine N. Marieb, *Human Anatomy & Physiology*, 7th ed. Pearson Education, Inc., 2007, ch. 12.

[5] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J Physiol*, vol. 117, no. 4, pp. 500–544, 1952. [Online]. Available: http://jp.physoc.org

[6] C. T. Nordhausen, E. M. Maynard, and R. A. Normann, "Single unit recording capabilities of a 100 microelectrode array," *Brain Research*, vol. 726, no. 1-2, pp. 129–140, Jul. 1996. [Online]. Available: http://www.sciencedirect.com/science/article/B6SYR-4C8P9C8-4/2/6dadcc11a236310d2deee315e225e9b1

[7] M. S. Fee, P. P. Mitra, and D. Kleinfeld, "Automatic sorting of multiple unit neuronal signals in the presence of anisotropic and non-gaussian variability," *Journal of Neuroscience Methods*, vol. 69, no. 2, pp. 175–188, Nov. 1996. [Online]. Available: http://www.sciencedirect.com/science/article/B6T04-3P8WW7J-7/2/0eef0f66ef3d0601bbbb52865e9f460a

[8] K. H. Kim and S. J. Kim, "Method for unsupervised classification of multiunit neural signal recording under low signal-to-noise ratio," *Biomedical Engineering, IEEE Transactions on*, vol. 50, no. 4, pp. 421–431, April 2003.

[9] S. Suner, M. Fellows, C. Vargas-Irwin, G. Nakata, and J. Donoghue, "Reliability of signals from a chronically implanted, silicon-based electrode array in non-human primate primary motor cortex," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, vol. 13, no. 4, pp. 524–541, Dec. 2005.

[10] E. Alpaydin, *Introduction To Machine Learning*. The MIT Press, 2004, ch. 10,11, pp. 197–273.

[11] A. B. Schwartz, D. M. Taylor, and S. I. H. Tillery, "Extraction algorithms for cortical control of arm prosthetics," *Current Opinion in Neurobiology*, vol. 11, no. 6, pp. 701–708, Dec. 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6VS3-44KHB41-C/2/536642e66763c61af1860dcfdd0508e8

[12] M. Sekerli, C. Del Negro, R. Lee, and R. Butera, "Estimating action potential thresholds from neuronal time-series: new metrics and evaluation of methodologies," *Biomedical Engineering, IEEE Transactions on*, vol. 51, no. 9, pp. 1665–1672, Sept. 2004.

[13] A. Tankus, Y. Yeshurun, and I. Fried, "An automatic measure for classifying clusters of suspected spikes into single cells versus multiunits," *Journal of Neural Engineering*, vol. 6, no. 5, p. 056001 (12pp), 2009. [Online]. Available: http://stacks.iop.org/1741-2552/6/056001

[14] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization," *Neural Networks, IEEE Transactions on*, vol. 18, no. 3, pp. 880–888, May 2007.

[15] J. L. Devore, *Probability and Statistics For Engineering and the Sciences*. Brooks/Cole - Thomson Learning, 2004, ch. 5, pp. 239–242.

[16] *Virtex-4 Family Overview*, Xilinx, Inc., 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[17] *Virtex-4 FX ML410 Embedded Development Platform*, Xilinx, Inc., 2009. [Online]. Available: http://www.xilinx.com/products/devkits/HW-V4-ML410-UNI-G.htm

[18] *Virtex-6 Family Overview*, Xilinx, Inc., 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf