A Case for Neuromorphic ISAs

Atif Hashmi Andrew Nere James Jamal Thomas Mikko Lipasti
Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI, USA
{ahashmi, nere, jjthomas3}@wisc.edu, mikko@engr.wisc.edu

Abstract

The desire to create novel computing systems, paired with recent advances in neuroscientific understanding of the brain, has led researchers to develop *neuromorphic architectures* that emulate the brain. To date, such models are developed, trained, and deployed on the same substrate. However, excessive co-dependence between the substrate and the algorithm prevents portability, or at the very least requires reconstructing and retraining the model whenever the substrate changes. This paper proposes a well-defined abstraction layer – the Neuromorphic instruction set architecture, or NISA – that separates a neural application's algorithmic specification from the underlying execution substrate, and describes the Aivo[1] framework, which demonstrates the concrete advantages of such an abstraction layer. Aivo consists of a NISA implementation for a rate-encoded neuromorphic system based on the cortical column abstraction, a state-of-the-art integrated development and runtime environment (IDE), and various profile-based optimization tools. Aivo's IDE generates code for emulating cortical networks on the host CPU, multiple GPGPUs, or as boolean functions. Its runtime system can deploy and adaptively optimize cortical networks in a manner similar to conventional just-in-time compilers in managed runtime systems (e.g. Java, C#).

We demonstrate the abilities of the NISA abstraction by constructing a cortical network model of the mammalian visual cortex, deploying on multiple execution substrates, and utilizing the various optimization tools we have created. For this hierarchical configuration, Aivo's profiling based network optimization tools reduce the memory footprint by 50% and improve the execution time by a factor of 3x on the host CPU. Deploying the same network on a single GPGPU results in a 30x speedup. We further demonstrate that a speedup of 480x can be achieved by deploying a massively scaled cortical network across three GPGPUs. Finally, converting a trained hierarchical network to C/C++ boolean constructs on the host CPU results in 44x speedup.

Categories and Subject Descriptors C.0 [*General*]: Instruction Set Design; C.0 [*General*]: Hardware/Software Interfaces; I.5.0 [*General*]

General Terms Algorithms, Design, Performance

Keywords Cortical Learning Algorithms, Neuromorphic Architectures, GPGPU

[1] Aivo is the Finnish word for *brain*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

1. Introduction

Biological and neuroscientific understanding of the structural and operational aspects of various components of the human brain has significantly increased over the past few decades. This has led to the development of a number of biologically-inspired software and hardware based computational models [5, 16, 43]. Most of these rely on the neocortex, the part of the brain that is evolutionarily the most recent and is unique to mammals, as their biological basis. These models implement some of the basic properties of the neocortex, including uniformity of structure, hierarchical arrangement, invariant representation of features, and feedback based prediction. Even more impressive is the fact that successful learning algorithms are now being deployed on specially designed hardwares which attempts to capture the physical properties of the brain [1, 11, 39].

Although these models show success at various learning tasks, they also suffer from a number of problems. First, the intrinsic complication of modeling the brain can make such models quite hard to understand. Second, many such models require explicit definition of network connectivity and hierarchical arrangements, often using low-level programming techniques. Third, given that many such models scale to large sizes, debugging can become quite cumbersome. Finally, as will be explained further in Section 5, many neuromorphic architectures strongly tie the proposed learning algorithm to the execution substrate. As such, these models can be quite difficult to port and are unable to take advantage of automated profile-driven and machine-specific optimizations to improve performance, reduced resource utilization, and improved robustness.

Historically, general-purpose Von Neumann computer systems suffered from a similar set of challenges until the introduction of standardized instruction set architectures (ISAs). An ISA creates a valuable layer of abstraction between an application's algorithmic specification (software) and it's execution substrate (hardware), enabling separate development in each domain, including automated tools for generating and optimizing machine code while allowing the same software to run on multiple generations of hardware. This paper advocates the adoption of a similar abstraction layer for neurally-inspired cortical models or neuromorphic computing systems and shows many of the same benefits follow.

Cortical models typically rely on some variant of Hebbian learning (rather than explicit programming) to train themselves to perform complex computational tasks. They encode the results of this training as a collection of synaptic weights, thresholds, connectivity, and other key parameters of neural components. Once the model has learned the values for these parameters, it can be deployed on an appropriate execution substrate. In the conventional approach to developing and deploying learning algorithms, the development and deployment substrates are the same, which leads to difficulties and/or compromises in the design and efficiency of both. Any changes to the execution substrate require reconfiguring, retraining, and possibly redesigning the entire network at the algorithmic level, while any attempts to optimize the network to better

match the characteristics of the execution substrate must be done manually, in a cumbersome and error-prone manner.

Instead, following the example of well-defined ISAs in conventional computing systems—which separate the algorithm from the execution hardware— this paper proposes adoption of a Neuromorphic Instruction Set Architecture (NISA), which forms an analogous implementation independent abstraction layer for neuromorphic systems.

As a case study of a NISA-based approach for developing neuromorphic systems, this paper introduces Aivo, a neuromorphic framework which consists of a well defined NISA, an integrated development environment (IDE), and several compelling optimization tools, as seen in Figure 1. The Aivo NISA defines an implementation of a cortically-inspired computational model proposed by Hashmi et al. [14, 15]. The key enabling mechanism that Aivo provides over prior approaches for modeling cortical networks is the NISA, which is used for specifying the structure, connectivity, semantics, state, and profile metadata of the cortical network. The NISA is implemented as a *virtual* ISA, and is deployed as an easily readable XML schema. This common intermediate representation allows the various components of the Aivo system to communicate and inter-operate smoothly, while providing a user-friendly, human-readable, self-documenting, and easily-extensible persistent store for all cortical network structure and state. Using the Aivo IDE, this paper shows how such learning models can be easily built, trained, and debugged through a user-friendly interface. Since the NISA approach allows a trained network to exist as a persistent contract between the learning algorithm and the execution substrate, we show how we are able to deploy our model on a single-core CPU, multiple GPGPUs, and even as functional boolean logic. Furthermore, the profiling information about the cortical network stored in the NISA, as well as knowledge of the deployment substrate, open up a number of optimizations for resource utilization and performance improvements. It should be noted that, as multiple ISAs exist in traditional computing systems, multiple NISAs may be developed to accommodate specific requirements of significantly different neuromorphic systems.

The contributions of this paper are as follows:

- We propose the NISA as a virtual ISA which serves as a meaningful layer of abstraction between cortically-inspired learning algorithms and available execution substrates.
- We implement Hashmi et al.'s cortical column model as the Aivo NISA to demonstrate the ability of this abstraction layer.
- We present the Aivo IDE as a useful tool for building, debugging, and deploying large neocortically-inspired networks.
- We demonstrate how the NISA allows for many algorithmic and substrate specific optimizations and guarantees that such translations are safe.
- Finally, we show how the NISA allows for easy deployment on CPU, GPGPU, or simple logic substrates.

The rest of the paper is organized as follows: Section 2 examines work related to learning models and development environments. Section 3 provides some basic background information about the structure and functionality of the neocortex. Section 4 details the biologically inspired computational model. Section 5 provides a detailed discussion of the proposed Neuromorphic ISA abstraction. Section 6 describes the Aivo integrated development environment. Section 7 demonstrates how the NISA abstraction allows us to deploy our learning algorithm on a host CPU, GPGPU, or as boolean logic. Section 8 presents some of the high level optimizations the NISA enables. Section 9 describes our experimentation methodology the results. Section 10 concludes the paper and proposes our future work.

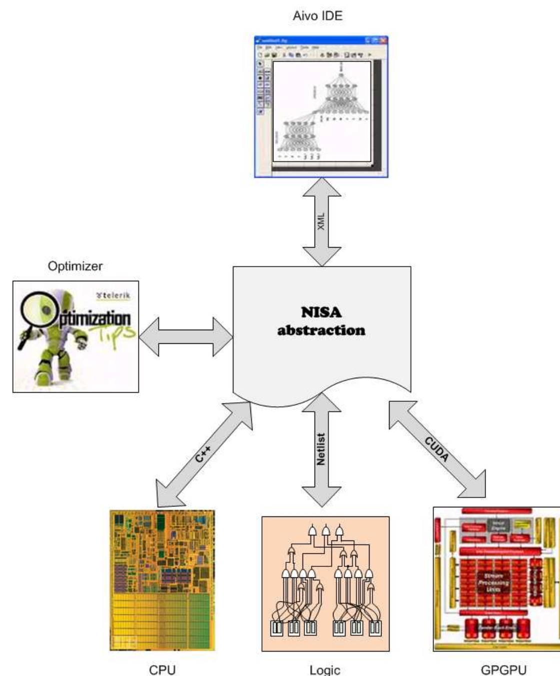

Figure 1. *Overview of how the NISA allows independent development and high level optimizations of the learning algorithm, capable of deployment across multiple available substrates.*

2. Related Work

Because the brain is so extraordinarily good at tasks such as image recognition, pattern classification, and motor skills, it has been the focus of artificial intelligence and neural network research for decades, and various learning models motivated by the properties of the brain have been proposed. Several models realize the properties of the brain at a very high level of abstraction. They do not model detailed neuron level behavior, but rather focus on how neurons as a group can realize functions like spatial and temporal pooling and developing invariant representations. Recently, Hierarchical temporal memories (HTM) [16] have gained recognition as a very high level computational model of the human brain. HTMs mainly construct a hierarchy of various nodes that learn to associate various spatial and temporal patterns with each other, thus learning complex patterns.

Convolutional neural networks trained with back-propagation are a well studied class of learning algorithm [25, 33] and have shown a degree of success in several applications. Spiking neural network models are another variation, and these models can be even further subdivided according to the biophysiological properties they emulate. One class of spiking neural model may simply incorporate the integrate-and-fire behavior of biological neurons, while another class may include the neural conductances and ion channels in their model [20].

Such learning algorithms have also been deployed on a variety of substrates. Both neural networks and deep belief networks have benefited from the parallel computational power of GPGPUs [21, 35]. FPGAs have even been used for deployment of learning algorithms, as their reconfigurability properties provide the flexibility to more closely resemble the structural properties of the brain [11, 23]. There have even been numerous hardware implementations of artificial neural networks, both past designs such as the Intel ETANN [17], or more recently the FACETS project [39]. Recent developments in memristor technology bring with them the

hope of creating online, on-chip plasticity for future neuromorphic hardware designs [9, 34].

Furthermore, there have been several attempts to create user-friendly development environments for developing, connecting, and testing various neurally inspired learning models. Matlab's Neural Network Toolbox [2] is a highly developed software application that has been used for both research and business applications, and Neuroph [3] is another similar tool with an open source development community. Both feature a user-friendly GUI interface, allow user's to import new training data sets, and feature a host of built-in neural network types.

These are just a sample of the ongoing research relating to learning algorithms, their deployment substrates, and the set of tools available for developing, optimizing, and deploying neurally inspired models. However, to the best of our knowledge we believe this paper is the first to advocate a complete abstraction of the underlying learning model. The Aivo development framework uses the NISA abstraction to independently develop cortical learning algorithms, deploy the algorithm across multiple substrates, and utilize various optimizations, while guaranteeing these translations are safe by using the NISA abstraction. In Section 5, we will provide more detail in regard to the benefits of using the NISA.

3. The Neocortex

The human brain can be divided into two main parts: the old brain and the new brain. The old brain constitutes the parts that developed early in evolution, including pathways from sensory modalities to the new brain, spinal cord, and other parts that deal with instinctual behavior. The new brain, also referred to as the *neocortex*, is unique to mammals and is highly developed for humans; it accounts for about 77% of the human brain (in volume). The neocortex is responsible for perception, language, mathematics, planning, and all the other aspects necessary for an intelligent system. It contains virtually all our memories, knowledge, skills, and experiences.

A very intriguing property of the neocortex is its apparent *structural and functional uniformity* [31]. Because of this property, the regions of the neocortex that process auditory inputs, for instance, appear very similar to the regions that handle visual processing. This uniformity suggests that even though different regions specialize in different tasks, they employ the same underlying algorithm. In essence, the neocortex is a hierarchy of millions of seemingly-identical functional units known as *cortical columns*. The concept of cortical columns was introduced by the neuroscientist Mountcastle in his seminal paper in 1978 [30]. Since then, this concept has been widely accepted and studied. Later studies showed that cortical columns could further be classified into *minicolumns* and *hypercolumns* [8]. A hypercolumn contains about 50 to 100 minicolumns, and each of these minicolumns consists of around 200 to 300 neurons. The minicolumns within the same hypercolumn share the same input and output connections and are strongly connected with each other through *inhibitory lateral connections*. Studies [18] hypothesize that the minicolumns use these connections to aid in learning unique and independent features from set of inputs they are exposed to. Hypercolumns are arranged in the form of a hierarchy throughout the neocortex. Information flows up this hierarchy via *excitatory feedforward paths* and flows down the hierarchy through *feedback paths*.

The arrangement and functionality of the hypercolumns and minicolumns has been studied in detail in the visual cortex [7, 18, 31]. These studies suggest that minicolumns at the lower levels of the hierarchy learn to identify very basic features (like edges of different orientation) and communicate their responses to minicolumns at the upper levels. Cortical regions operate by progressively abstracting and manipulating increasingly complex notions throughout the neural hierarchy. For instance, the visual cortex hierarchy will first identify segments, then elementary shapes such as angles and intersections, and increasingly complex combinations, such as objects found in our environment [13]. This automatic abstraction capability for various inputs (visual, auditory, olfactory, etc.) partly explains why the neocortex still outperforms traditional computers for a number of tasks, such as face recognition, language learning, and motor control. Emulating such capability is thus a major step in building computing systems that can compete with the processing characteristics of the brain.

4. A Biologically Plausible Learning Model

While the structures and functions of the brain have been investigated for a long time, quantitative models consistent with physiological data and capable of accounting for complex tasks have been proposed only recently [40, 41]. In this section, we describe the learning model which uses hyercolumns as the basic computational unit. Later, we describe how the NISA abstraction captures the important semantics and components of this algorithm.

This cortical network model draws inspiration directly from the organization and structures of the primate visual cortex. First, our model implements the preprocessing transformations that affect the visual input as it propagates from the retina to the primary visual cortex through the optical pathways. Second, we create a competitive learning based hierarchical network that uses the preprocessed visual data as input. This results in a biologically plausible system that learns to recognize various visual stimuli and shows partial rotation and scale invariance, as observed in mammals. Other competitive learning models [6, 10, 28] have been proposed in the past, but these models either ignore the important properties of the hypercolumns (discussed in Section 3) or have very high computational requirements.

Figure 2 shows the architecture of the basic functional unit in our competitive learning model, the hypercolumn (left side), in comparison to a biological hypercolumn (right side). We see in biology as well as our model, each hypercolumn contains multiple minicolumns that share the same receptive field. These minicolumns are strongly connected to neighboring minicolumns via inhibitory connections (solid lines in Figure 2).

4.1 Input and Receptive Field

In mammals, visual scenes are projected onto the retina. The activations of the retinal cells in response to the visual scene are transfered via the optical nerve to the Lateral Geniculate Nucleus (LGN) cells [24]. The LGN cells are contrast sensitive i.e. they react strongly to an illuminated point surrounded by darkness (known as on-off cells) or conversely to a dark point surrounded by light (off-on) cells. These cells are spatially distributed in such a way that on-off and off-on cells are intertwined [37] and receive inputs from neighboring retinal cells referred to as the *receptive field* of the LGN cells. Cells respond only to input changes that occur within their receptive field. Finally, the activations of the LGN cells become the input to the primary visual cortex.

In our model, all the inputs are preprocessed using the LGN transform [37] as well. For preprocessing, we consider a regular spatial distribution of LGN cells (one on-off and one off-on per pixel), but have also experimented with more random distributions without noticeable differences.

4.2 Random Activations and Initial Learning

In our cortical model, all the minicolumns within a hypercolumn are initialized with very weak random weights, thus showing no initial preference for any particular pattern. A key feature of our learning model is random neocortical firing behavior [12, 38]. The minicolumns exhibit high activations over random intervals, mimicking the stochastic nature of biological neurons in the presence of

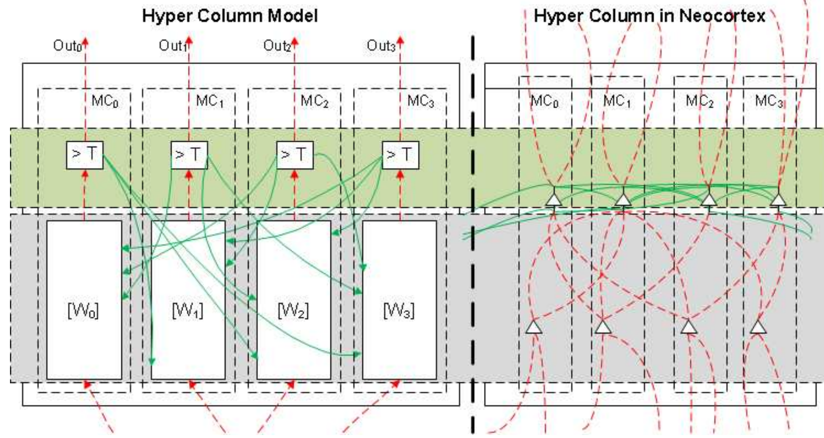

Figure 2. *Left: The hypercolumn model as defined by the cortical learning algorithm. Right: The structure of a typical biological hyppercolumn. MC=Minicolumn, T=Threshold of Activation Function. Dashed lines are feedforward paths, while solid lines are lateral inhibitory paths.*

noise. When the random activation of a specific minicolumn coincides frequently with various occurrences of the same pattern, the minicolumn adjusts its synaptic weights to correlate with the input pattern (described in detail in Section 4.4). Thus over time, that minicolumn develops a preference for that specific pattern. While this random activation of minicolumns may not intuitively seem productive, this behavior is harnessed to make the model fault-tolerant, improves the model's training time, and mimics the behavior of its biological inspirations.

4.3 Execution of Minicolumns

During each training epoch, each of the minicolumns evaluates the dot-product $DP = \sum_{i=1}^{N} X_i.W_i$ between its weights $\vec{W}$ and the input $\vec{X}$. The result of the dot-product becomes the input to the activation function given by,

$$\frac{1.0}{1.0 + e^{(-\frac{DP - cutoff}{\beta})}} + \alpha \times \sum |W_i| \quad (1)$$

Here, $cutoff = \phi \times \sum |W_i|$. ϕ determines the error tolerance of the minicolumn. β defines the sharpness of the activation function while α controls the effect of weight strength of a minicolumn on its output. The minicolumn is said to be active or to fire if the value of its activation function is greater than a determined threshold. At the same time, each minicolumn inhibits neighboring minicolumns from firing for the pattern it has learned to recognize via lateral inhibitory connections. Minicolumns also form a competitive learning network, and when multiple minicolumns fire at the same time, the one with the strongest response inhibits the ones with weaker responses. The inhibited minicolumns then weaken their weights corresponding to active X_i. As a result of this process, the hypercolumn network is able to recognize unique patterns without supervision.

4.4 Weight Update Rules

When a minicolumn fires, it strengthens its weights to increase correlation with the current pattern. Weights of a minicolumn corresponding to active inputs are strengthened with the following update rule"

$$W_i = \left(W_i + \left(C_1 + \gamma \times \frac{1.0}{1.0 + e^{(-\frac{W_i - C_2}{\beta})}} \right) \right) \quad (2)$$

Here, C_1 defines the minimum amount of strength added to the current W_i, and C_2 defines how the present W_i will affect the weight update. The update added to W_i is dependent upon the present value of W_i as well. This means that if W_i is strong it will get a higher update value, as observed in biological data [38]. When a minicolumn is inhibited, it modifies the weights corresponding to active inputs using the following update rule.

$$W_i = (W_i - \delta) \quad (3)$$

Here, δ defines the weight update rate in the presence of inhibition. It should be noted that other complex update rules for inhibition could also be used here.

4.5 Hierarchy of Hypercolumns

Much of the brain's ability comes from its hierarchical organization, which uses different processing levels to perform complicated tasks. Similarly, the modeled hypercolumns can be arranged in a multilevel hierarchy, which we refer to as a cortical network (see Figure 3). Lower hierarchical levels identify simple features and communicate their output to the higher levels which learn progressively more complex features [13]. In this manner the activations flow up the cortical network and the minicolumns in the top-level hypercolumns learn to identify the full complex input pattern.

For pattern recognition tasks, we modeled our hierarchy according to the known properties of the mammalian visual cortex. Once the input images are preprocessed using the LGN/Log-polar transform, they are exposed to a layer of hypercolumns with properties similar to Gabor filter banks [22], which models the organization of the first level of the visual cortex hierarchy. Each of the minicolumns within the Gabor hypercolumns learns to identify edges of different orientations and shows a rotational invariance of 12 degrees, which is in accordance with the neuroscientific experiments on monkeys [42]. The outputs of the Gabor hypercolumns propagate to the cortical network hierarchy, which learns progressively more complex features at each higher level.

4.6 Supervised Feedback and Pooling

The feedforward learning process enables our hierarchical cortical network to learn unique features from the input patterns in a completely unsupervised environment. Minicolumns consistently fire for small variations in their learned patterns, depending on their error tolerance parameter. However, if a single pattern can be represented by two very different variations, it is likely that a hypercolumn (and ultimately an entire cortical network) will recognize these as two different patterns. To resolve this issue and generate in-

variant representation for variations of the same pattern, we make use of a supervised feedback processing algorithm.

To illustrate how this feedback processing is useful, we consider a single handwritten digit from two individuals with very different handwriting styles. From the unsupervised feedforward processing, the cortical network will distinguish the differences between these different input patterns. As a result, the top level hypercolumn will have two trained minicolumns, each of which represents the same digit but are composed of different lower level features. However, for training sets such as handwritten digits, we want to be able to classify objects into categories rather than have the top level hypercolumn dedicate a minicolumn to every variation of a pattern. Here, the supervisory feedback signal of the model is used to notify the top level hypercolumn that it should pool together two variations of a single pattern. The minicolumn receiving excitatory feedback adjusts its weights so that it fires for both variation and inhibits any minicolumn firing for the new variation. The inhibited minicolumn changes its weights so that it does not fire for that input pattern, freeing itself to recognize a truly novel pattern. Thus over multiple exposures, the minicolumn firing for the original pattern will also consistently fire for the new variation.

This feedback pooling process will continue down through the cortical network's hierarchy. Once the top level minicolumn starts to give a stable activation for both variations, it will send a feedback signal down so that lower level minicolumns can also create invariant representations. The amount of feedback sent to each of the lower level minicolumns is proportional to its firing history, with more active columns receiving stronger feedback. The intuition of this proportional feedback is that objects in the visual field likely exhibit some amount of object permanence, so temporally related patterns are more likely variations of a single pattern. Thus, over time the most active minicolumns pool the inputs received from lower level minicolumns which results in invariant representations of an object as well as significant resource optimization. We note that this feedback process differs from back-propagation in neural networks since, rather than altering weights to reduce the error of the classification function, we are simply teaching the cortical network which features can be considered similar. Furthermore, the feedback process in our algorithm relies on spatial locality of the features and synaptic weights being pooled, where synaptic connections in most neural networks do not contain spatial information.

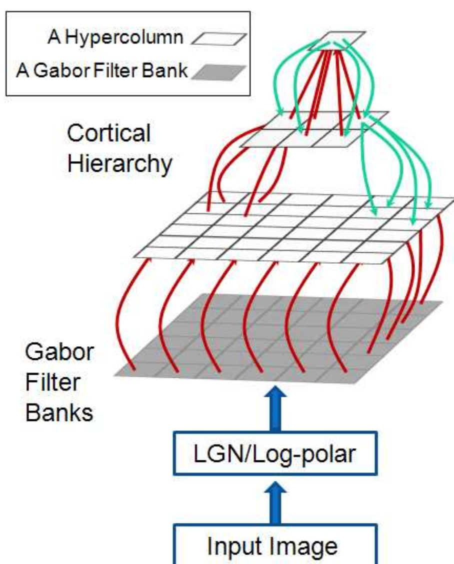

Figure 3. *A hierarchical organization of hypercolumns along with the gabor filter banks.*

The process of generating invariant representations within a minicolumn using feedback is explained in the pseudo-code provided in Algorithm 1. It invokes code to update synaptic weights as described in Equation 2 and 3.

Algorithm 1 Pseudo code for generating invariant representations within a minicolumn using supervised feedback.

```
if feedback > 0 then
  if hasNotFired then
    if hasMaxFiringHistory then
      UpdateSynapticWtsExcitatory(feedback)
    end if
  else
    if hasMaxFiringHistory then
      UpdateSynapticWtsExcitatory(feedback)
      if isStable then
        for i = 1 to N do
          if IsActive(child[i]) then
            SendFBToChild(i, feedback)
          end if
        end for
      end if
    else
      UpdateSynapticWtsInhibitory(feedback)
    end if
  end if
end if
```

5. Neuromorphic ISA

Neuromorphic architectures attempt to emulate the function, organization, and processing power of the the brain by combining a learning algorithm together with some type of execution substrate. The learning algorithms themselves may vary quite drastically in terms of the level of detail at which they model the brain, whether they are integrate-and-fire neurons that simply mimic spiking behavior, or highly complex algorithms such as the Hodgkin-Huxley model which includes details about the chemistry and conductances of neurons. The execution substrates on which neuromorphic architectures are deployed also vary widely from model to model. While traditional CPUs, GPGPUs, and even supercomputers have been utilized as the execution substrate for many of these models, research has also explored creating silicon chips that more closely resemble neurons and synapses in the brain [1, 19, 39]. While such projects are both interesting and ambitious, a number of challenges arise from the way these current neuromorphic architectures are designed.

First of all, present neuromorphic architectures strongly tie the learning algorithm to the execution substrate. The Blue Brain Project, which emulates the brain with the highly detailed and complicated Hodgkin-Huxley model, simulates on a Blue Gene supercomputer using the MPI programming interface. While not all researchers may wish to use such a detailed learning algorithm, the nature of tying the implementation of their algorithm so closely with deployment on a supercomputer makes it highly difficult for researchers without such resources to implement or expand on their work. Integrate-and-fire neural networks can be finely tuned for execution on modern day GPGPUs [21], though to achieve fast processing performance, considerable effort may be required to alter or tune the learning algorithms themselves to properly take advantage of the graphics processor architecture. Finally, special-purpose neuromorphic hardware may attempt to model the structure and plasticity of the brain, but limitations in such hardware

designs may limit the number of properties that can be presently implemented. These are just some examples of how hardware limitations in turn influence the development of the learning algorithms, which ultimately can defeat the purpose of developing computers that resemble the unique properties of the brain.

Furthermore, strictly tying a learning algorithm to an execution substrate limits the portability of the algorithm across platforms. For example, a learning algorithm like a neural network may be highly tuned to take advantage of GPGPU optimizations, though the same code will not be easily, or at least efficiently, executed on a commodity CPU. Tuning algorithms to optimally execute on novel neuromorphic hardwares may only exacerbate such problems.

Considering the history of the Von Neumann computer architecture, we are able to easily relate to such challenges. Originally, computer architectures and the programs that were executed on them were developed in concert, sacrificing portability and making independent development of hardware and software impossible. Such problems persisted until the introduction of the instruction set architecture (ISA) with the IBM System 360 [4]. The ISA was introduced as a persistent contract between the computer hardware and the software that would execute on it. Essentially, the ISA separated the algorithm from the deployment substrate, allowing each to be developed independently, as well as guaranteeing portability between hardware generations. The introduction of the ISA abstraction furthermore allowed the development of automated tools to allow the production and optimization of machine code, letting software take advantage of the specific benefits of different hardware generations or designs.

Borrowing from the lesson of the ISA, we advocate a Neuromorphic ISA (NISA) as a persistent contract between the learning algorithm and execution substrate. In our current implementation, the NISA serves as a virtual ISA which forms a level of abstraction between these two layers, allowing the independent development of both.

Traditional ISAs specify the *state*, *structure*, and *semantics* of the abstract machine they represent, including register and memory maps, supported data types, and details of operations for data handling, memory access, logic and arithmetic computation, control flow, and so on. Similarly, a neuromorphic ISA should precisely specify state, structure, and semantics for the abstract machine it represents. In most neuromorphic systems, *state* is present in at least two forms: the current level of activation of a component (e.g. is a neuron firing, or how much time has elapsed since it last fired) and the magnitude of the connections that influence its firing (e.g. a neuron's synaptic weights). This state is distributed across the elements (each neuron has local state), rather than being stored in a centralized memory or register file. As a consequence, the aggregate amount of state grows as the system grows, and there is no direct, memory-like, way of accessing this state. The *structure* of a neural system is determined by connections between components (dendritic and axonal projections that intersect), as well as any higher-level structural abstractions included in a particular model (for example, the cortical minicolumns and hypercolumns described in Section 4). Some models may also require representation of physical structure, rather than just logical connectivity, to represent entities such as cortical feature maps. Finally, the *semantics* of a neural system are determined by the rules that govern the activation and plasticity of the components of the system. Activation is typically determined by a thresholding function following a spatial and/or temporal integration of activity at synaptic inputs, while plasticity most often follows Hebbian learning rules to modify synaptic weights. A neuromorphic ISA must then provide primitives that are capable of expressing these characteristics at a level of abstraction that suits the particular neuromorphic system being developed. As is true for traditional ISAs, if neuromorphic system specifications significantly differ from each other, new NISAs may be developed to express them. The benefit of using the NISA abstraction are quite intuitive and clear. By separating the algorithm from the deployment substrate, each can be developed independently without one placing restrictions or limitations on the other. We can, in effect, write *neuromorphic programs* using an existing NISA, and can expect those programs to operate correctly on any current or future execution substrate that is compatible with our NISA.

```
<Hierarchy inputColumns="78" inputRows="24">
    <Grouping ID="6" Nodes="10">
        <Grouping ID="4" Nodes="10">
            <Grouping ID="0" Nodes="10"/>
                <Node>
                        <Status firing="1"/>
                        <Weights>0.093 0.078 ... </Weights>
                        <Threshold limit=".57"/>
                </Node>
                        .
                        .
                        .
            <Grouping ID="1" Nodes="10"/>
        </Grouping>
        <Grouping ID="5" Nodes="10">
                        .
                        .
                        .
        </Grouping>
        <Node>
                <Status firing="0"/>
                <Weights>0.091 0.079 ... </Weights>
                <Threshold limit=".84"/>
        </Node>
    </Grouping>
</Hierarchy>
```

Figure 4. *The XML-based Aivo NISA describes a cortical network.*

Given the characteristics of our cortical network model, we have developed a NISA that meets these requirements. Using this NSIA, we can develop a neuromorphic program today and deploy it on a GPGPU cluster, but as future neuromorphic hardwares becomes available, we can directly retarget the neuromorphic program to them by simply adding the new primitives to our NISA abstraction. A simplified example of a cortical network defined by the Aivo NISA, in XML, can be seen in Figure 4.

6. Aivo IDE

Throughout the development of the cortical learning model, it was necessary to have a method to create and train large networks, debug their behavior, and further develop the underlying learning algorithm. With these goals in mind, we developed the Aivo integrated development environment (IDE). The Aivo IDE streamlines the process of creating cortical networks by allowing networks to be built in two different ways. First, a network can be built by dragging and dropping hypercolumns onto the main screen of the GUI (see Figure 5). From the GUI, the user can then simply connect the network they desire for the task of interest. Second, a cortical network can also be imported from a previously created XML file, which also adheres to the syntax of the Aivo NISA. From the Aivo IDE, the user can select to deploy the network on either the CPU or the available CUDA enabled GPGPUs.

At any point in creation, debugging, or training, the Aivo IDE allows a cortical network design to be checkpointed to an XML file using the NISA. XML was chosen for this task due to its hierarchical format and the ability to be easily read or hand-coded by the designer. Furthermore, the NISA abstraction allows us to connect components such as the Aivo IDE and Cortical Network Optimizer without creating unnecessary dependencies between such modules.

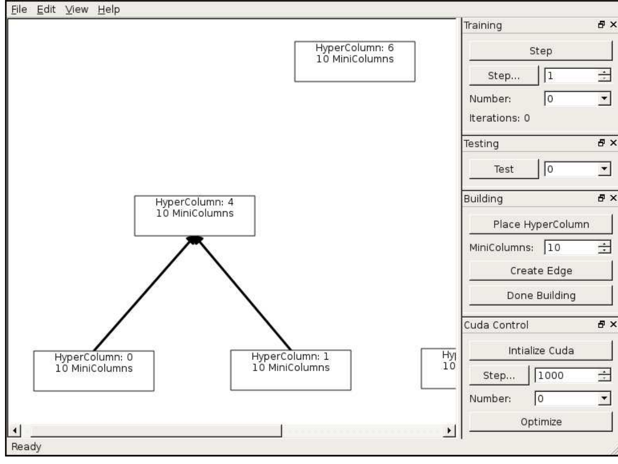

Figure 5. *Building a cortical network using the Aivo IDE.*

This file is useful for check-pointing network state and metadata that can be examined for specific information, reloaded for further training, or used for optimization purposes.

The Aivo IDE also features several options we have found useful for training, executing, and debugging a cortical network. Deployment of a cortical network on the host CPU allows a more precise control over the training process. From the Aivo IDE, the user can single-step through each training iteration or batch-train several chosen inputs for longer periods of time. Training on the CPU also allows the user to visually debug a cortical network by simply clicking on hypercolumns and minicolumns to display information should as synaptic weights and firing histories. While having the ability to interactively train and debug a network has its advantages, the Aivo IDE also allows for deployment and training on a CUDA enabled GPGPU substrate. Deployment on the GPGPU lends to some significant speedups in training times, though it loses the ease of user interaction. Section 7 describes the deployment substrates we have investigated in more detail.

7. Substrates and Code Generation

In conventional computer systems, a major advantage of having a well-defined ISA is that it allows the same software programs to run correctly on multiple generations of hardware, even with radically different implementations. In this section, we show how the NISA enables the same benefit, making the investigated cortical learning algorithm easily deployable on a commodity CPU, a GPGPU, and even as boolean logic.

7.1 Deployment on Commodity CPUs

The initial version of Aivo software was developed for a commodity CPU, and that remains the default deployment substrate. The Aivo IDE also runs on the CPU substrate to develop, initialize, and debug new cortical networks, as well as make changes to the specifics of the learning algorithm. Furthermore, as described in Section 8, execution on the CPU substrate allows us to create high-level optimizations that enhance the model, whether in terms of execution time or robustness. Presently, the learning algorithm is deployed as a single thread on the CPU. However in future work, we plan to utilize multicore processors to take advantage of the performance benefits of parallel execution.

7.2 Deployment on GPGPUs

In order to build a neocortical model that can perform comparably to the neocortex, the scale of these models must be significantly large. In fact, many researchers have proposed using supercomputers to affectively scale neocortical models to interesting problems [27, 36]. However, modern GPUs have gained considerable popularity as commodity available hardware accelerators for highly parallel applications. GPGPUs have even been used to speed up biologically-inspired computing models like cortical architectures and neural networks [21, 32]. In particular, Nvidia's CUDA framework has become a popular and affordable method for application acceleration and we have employed it to accelerate cortical networks.

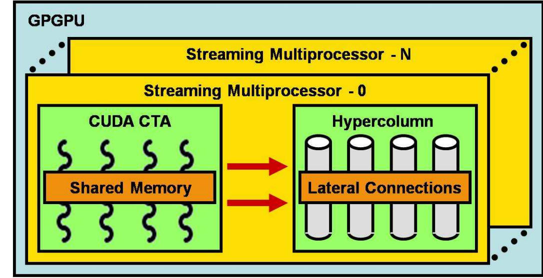

Figure 6. *Mapping a hypercolumn to a CUDA CTA.*

To deploy a network on a GPGPU, the Aivo IDE generates CUDA code to be executed on available Nvidia GPGPUs. Nvidia's CUDA framework is built around several hierarchically organized components which can be configured by the programmer. The CUDA-thread is the basic unit of execution, organized into thread-blocks, or *Cooperative Thread-Arrays* (CTAs). Within a CTA, the threads are able to both quickly synchronize using hardware primitives as well as communicate via a fast-access shared memory space. CTAs are then grouped into kernel-launches, or grids, for concurrent execution on the GPGPU.

From the Aivo IDE, the user can choose to deploy a created cortical network on the GPGPU. To these ends, we have created a template CUDA version of the learning model. The Aivo NISA is used to generate the complete CUDA code/binary using this template.

This CUDA-enabled version of our learning model takes advantage of the inherent parallelism of the cortical network. In Section 4, we described the cortical network as having different hierarchically organized components, composed of minicolumns, hypercolumns, and cortical networks. The GPU-accelerated code translates the components of the cortical network to the CUDA framework, modeling each minicolumn as a CUDA-thread and each hypercolumn as a CTA, as seen in Figure 6. With such an organization on CUDA, the minicolumns within a hypercolumn can take advantage of a CTA's synchronization and fast memory sharing abilities. As an example of machine-specific optimization, the Aivo code generator organizes the synaptic weights of each minicolumn into a strided pattern that maximizes memory coalescing on the GPGPU, leading to significant performance improvement.

From the Aivo IDE, the user can easily train a cortical hierarchy on any available CUDA-enabled GPUs. Because the cost of transferring the cortical network's weights and inputs can become quite expensive in terms of performance, utilizing GPU acceleration is most beneficial for batch training a large amount of different inputs for many training iterations. After training on the GPU, the cortical network's state information will be copied back to the GUI and saved in an XML file using the proper NISA syntax.

7.3 Logic Generation

The NISA abstraction provided by Aivo also supports deployment of a cortical network in the form of logic functions that can later be converted to netlists. For now, the logic level representation of

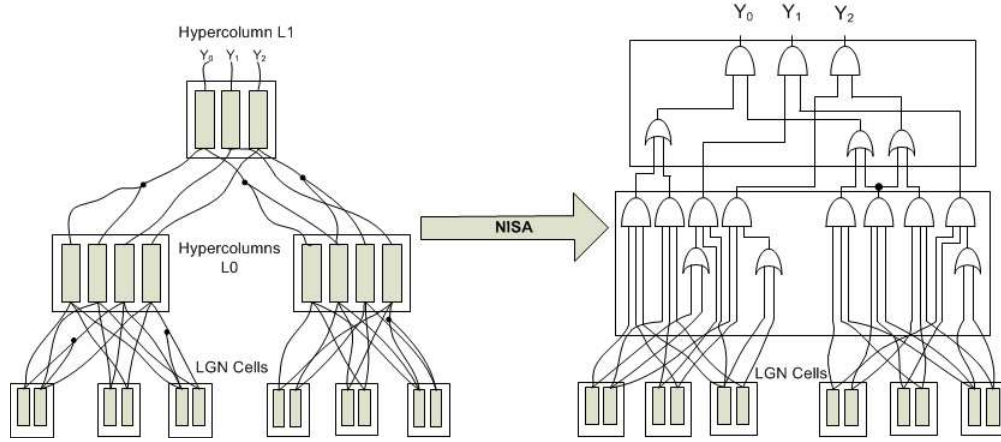

Figure 7. *A simple fully trained cortical network and corresponding logic network.*

a cortical network is generated as functional logic blocks in C++. AND- and OR-operations are hierarchically connected to represent the structure of the corresponding cortical network. Once a network is fully trained (i.e. 100% recognition rate on the training dataset), it can be converted to a logic representation for efficient execution. To achieve this, the Aivo IDE exports the state of the hypercolumns and minicolumns of the fully trained network using the Aivo NISA syntax. The Aivo NISA code is then processed to generate the logic representation of the cortical network.

Once a minicolumn has concretely learned a particular feature, its weights can be considered as binary synapses, i.e. it has a strong synaptic connection or no synaptic connection to a particular input. In terms of boolean logic, the output 'Y' of such a minicolumn can be represented as:

$$Y_i = \forall_{k \in S} AND(k)$$

Here, 'S' is the set of inputs to the minicolumn corresponding to high weights.

If a minicolumn has pooled different variations of an input as described in Section 4, then its output can be represented as:

$$Y_i = \forall_j AND(O_j)$$
$$O_j = \forall_{k \in M} OR(k)$$

Here, 'M' is the set of inputs corresponding to high weights that pool different variations of the same pattern.

Figure 7 illustrates the logic generation process using a simple trained cortical network as an example. In this example, the two levels of hypercolumns are replaced with logic equations that perform the equivalent detection or classification function. However, since the LGN cells in this circuit perform a type of analog-to-digital conversion of the input image, they are not simplified to boolean logic.

Even though converting the cortical network to boolean representation results in significant reduction in execution time for a learned task, it comes with a trade-off: this boolean network cannot learn new tasks or features. Rather, it can only detect the features it has already learned, and will not respond to new features appearing in the input. To address this shortcoming, The Aivo runtime monitor is used to detect when a boolean logic equivalent network is not sufficient for the learning task at hand. This runtime monitor relies on a simple property of the competitive learning-based cortical column model: a fully-trained cortical network should evoke a single winning response for every input (i.e. one minicolumn in each hypercolumn should fire). Aivo monitors the firing rate of each boolean circuit hypercolumn, and once it falls below a given threshold, Aivo's runtime will revert the boolean circuit back to a computational model that is able to learn the new features in the input. After the cortical network learns the new features, Aivo can regenerate a new boolean circuit and regain the efficiencies of this approach. Again, all conversions between hypercolumn models and boolean logic circuits occur as an offline process. This process is very similar to profile-driven re-optimization of machine code in managed runtime systems with just-in-time compilers (e.g. Java, C#).

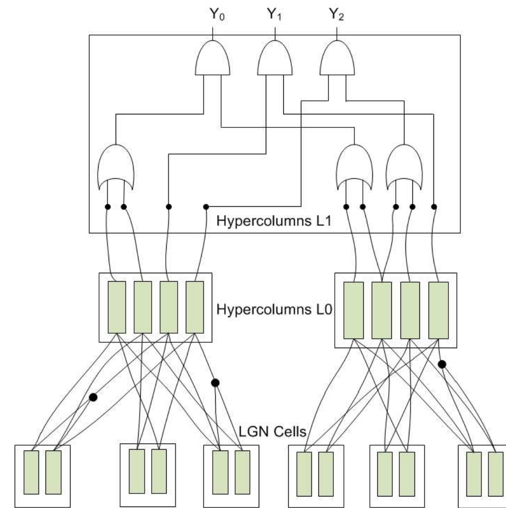

Figure 8. *An example of a hybrid network created using the NISA abstraction.*

7.4 Code/Logic Hybrid Generation

The NISA abstraction also supports generation of cortical networks using a code/logic hybrid approach. As describe in Section 7.3, once a network is fully converted to logic, it is unable to learn new features. Therefore, the cortical network should achieve 100% recognition rate on the training dataset before it may be converted into an equivalent functional logic representation. This means that during the training period, the cortical network cannot benefit from the logic generation capability of the NISA abstraction, as recognition has not stabilized. To avoid this dilemma, we extend the NISA abstraction to allow cortical networks to be partially converted as they stabilize, which we refer to as Code/Logic hybrid networks. This addition lets the NISA abstraction to partially convert a cortical network into logic (i.e. some of the hypercolumns are converted into their functional logic representation while others are not). To achieve this, the NISA abstraction can store the state activity of

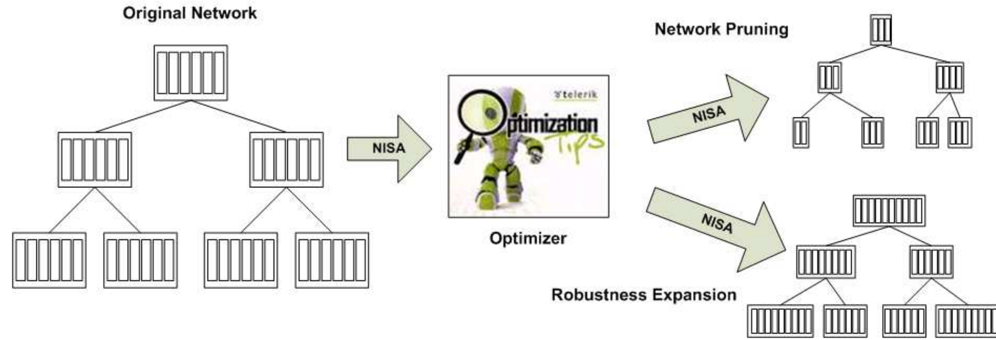

Figure 9. *The Cortical Network Optimizer optimizes a trained cortical network for resources utilization or for robustness.*

the hypercolumns in the network. If after a significant number of training epochs no new minicolumns within a hypercolumn learn to recognize any new features, the hypercolumn may be considered stabilized and can be safely converted to a functional logic representation.

The NISA abstraction also allows the programmer to explicitly control the conversion of a hypercolumns to a logic function (i.e. the programmer can mark certain hypercolumns so that they are not converted to logic). For example, the program may configure an explicit hybrid network where the upper levels of the hierarchy are converted to logic functions while the hypercolumns in the lower levels are not. Typically, this type of hybrid conversion is useful for robust recognition of input patterns in the presence of noise. Since the lower levels execute the hypercolumn learning algorithm, they exhibit more resilience to noise or slight variations present in the inputs while the logic converted upper level hypercolumns identify complex objects with computational efficiency. Figure 8 demonstrates this hybrid approach.

8. High Level Optimizations

Because the NISA provides a well described abstraction of the cortical learning algorithm we are investigating, we are able to create high level optimizations to improve performance, enhance the robustness of the network being trained, and reduce resource utilization. In this section we will describe some of the optimizations developed thus far.

8.1 Cortical Network Optimization

The Aivo Cortical Network Optimizer (CNO) is a high level optimization tool developed to improve the structure of the cortical network, either by expanding the structure to improve learning robustness, or reducing the structure to reduce required processing time. When a user initially creates a cortical network using the Aivo IDE, the correct number of resources (hypercolumns and minicolumns) required to achieve a particular task (e.g. robust recognition of the entire feature set) is probably unknown. Thus, it is likely that resources have been either over allocated or under allocated for the network to learn a particular task. When the resources are over allocated, the cortical network requires more computation than is necessary for each learning iteration. On the other hand, an under-allocated cortical network may not contain enough minicolumns to learn the full number of features in the dataset.

In the case when the cortical network resources are over allocated, all of the unique features of the dataset will be fully recognized after a sufficient amount of learning epochs. However, there will be a number of minicolumns that will not perform useful work, even though they must still evaluate at each learning iteration. While the evaluation of these minicolumns will not affect the activations propagated to the next level of the network, they still contribute to the total execution time. In such cases, the Aivo CNO can be invoked to perform network pruning and remove unnecessary minicolumns. First, the state and structure of the trained cortical network is exported using the Aivo NISA in the form of XML files. Then, the Aivo CNO parses these files and deletes the unnecessary minicolumns. A minicolumn is declared unnecessary if all its synaptic weights are close to zero, which suggests that it has not learned any interesting features from the training dataset; thus it is not required to robustly perform the learned task. Along with pruning unnecessary minicolumns, the CNO will regenerate the netlist that defines the connections between the minicolumns at various levels in the hierarchy to account for the deleted minicolumns and their connections.

Conversely, an under-allocated cortical network may not possess enough minicolumns to robustly recognize all of the features of the dataset. After a significant amount of learning epochs, the Aivo CNO may be invoked to perform a robustness expansion of the cortical network. The trained network's state is again exported in the Aivo NISA format, which the CNO parses, and then allocates more resources for under-allocated hypercolumns, as determined by a defined threshold (i.e. if 90% of minicolumns within a hypercolumns are doing useful computations, CNO allocates more minicolumns to this hypercolumn). Minicolumns are useful if they contain strong weights corresponding to the input activations. This feature is quite useful because the CNO will add minicolumns only to the necessary hypercolumns. Thus, using multiple invocations, Aivo generates cortical network that is sufficient in terms of resource allocation and execution time for the given input dataset. Figure 9 provides a pictorial representation of the CNO's operations.

We use an offline optimization approach for the CNO rather than performing such major structural changes to cortical network during runtime. Such offline structural changes can be considered biologically plausible as well, since there is evidence that memory-consolidations and translations that occur in the brain during sleep have a significant impact on how and where memories are stored [29]. In Aivo, changing the cortical network structure during runtime would result in both code complications and a drastic increase in the execution time for each learning epoch. Furthermore, the structure does not significantly change on an iteration by iteration basis, so it makes sense to optimize after a large number of training epochs.

8.2 Optimized Networks on GPGPUs

The Aivo CNO is also able to optimize cortical networks deployed for execution on a GPGPU. However, one limitation of CUDA-enabled GPGPUs is the *warp size*. The scalar multiprocessors in current generation Nvidia GPGPUs issue instructions in SIMD style for groups of 32 threads, known as a warp. The warp size essentially sets a minimum limit to the number of threads (and for

the cortical network algorithm, the number of minicolumns) that will execute in parallel in a hypercolumn. With such a limitation, a hypercolumn with only 3 minicolumns will not necessarily execute any faster than a hypercolumn with 32 minicolumns, though it may save some memory space. For this reason, when optimizing the cortical network for the GPGPU, resource allocation or resource recovery is only performed with a granularity of 32 minicolumns.

8.3 Utilizing Multi-GPU Systems

For systems with multiple CUDA-enabled devices, Aivo allows cortical networks to be distributed and trained across multiple GPGPUs in parallel. While these sub-hierarchies must synchronize and communicate data at points of convergence, this overhead can be minimized by properly partitioning the hierarchy. Since a system may have a heterogeneous collection of CUDA-enabled devices, we have employed a heuristic to split a cortical hierarchy based on each device's ability, specifically the number of Streaming-Multiprocessors. If multiple GPUs are selected to train a network, the Aivo IDE performs a device query (using a CUDA API call) to determine the number of SMs for each device. Each sub-network is then sized in proportion to each device's SM count, as in Figure 10. When the upper levels of the cortical hierarchy are executed, the GPGPUs must synchronize and copy data to a single GPGPU, though with large-scale networks this is typically negligible when compared to the overall execution time. In practice, this heuristic has shown to be quite effective for the GPGPUs we have available. We leave it to future work to investigate other heuristics for distributing cortical networks across multiple GPGPUs.

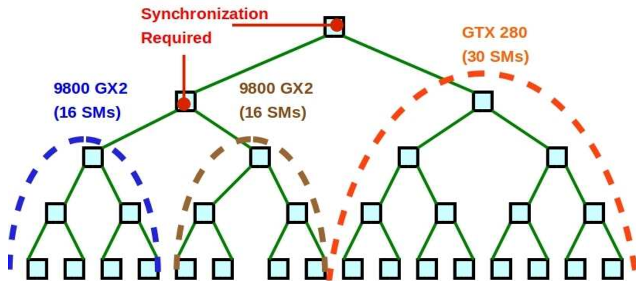

Figure 10. *A cortical network is proportionally split across two GPUs.*

9. Experimental Case Study

In this section, we perform several case studies to validate the various components of Aivo. We examine using the NISA abstraction for deployment on the CPU, GPGPUs, and logic substrates, as well as Aivo's various optimizations. We tested our model with a sample of handwritten digits (0-9) obtained from the MNIST Database (http://yann.lecun.com/exdb/mnist) [26].

9.1 Hardware

The performance results for the experiments are gathered from a AMD Phenom Quad-Core at 2.39 GHz and 8 GB of DRAM. Three CUDA-enabled GPGPUs are also used in these experiments. The first is an NVIDIA GeForce GTX 280, with 30 SMs operating at 1.46 GHz and 1 GB of global memory available. We also use an NVIDIA GeForce 9800 GX2, which contains two GPGPUs, each with 16 SMs operating at 1.5 GHz and 512 MB of memory each.

9.2 Recognition Performance

While it is beyond the scope of this paper to rigorously compare the cortical network to other learning algorithms, we include some performance recognition results to demonstrate how our biologically-inspired cortical network compares with more traditional but less

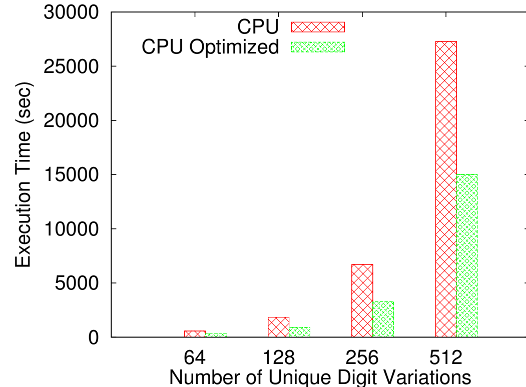

Figure 11. *Performance of original and CNO optimized networks on CPU.*

biologically-plausible implementations. Furthermore, since our cortical model is constantly changing due to the addition of new features, it is unfair to compare it with current state-of-the-art neural network models. However, we still compare the cortical network with an existing state-of-the-art convolutional neural network (CNN) based digit recognition system implemented by O'Neil [33] and proposed by LeCun et. al. [25]. O'Neil's implementation is able to achieve a recognition rate of 99.26% on a 10,000 handwritten digit test set, though to achieve such performance the network is trained using multiple distortions of a 60,000 character training set. However, it seems improbable that the human brain requires this many training examples to recognize digits 0 to 9. Requiring such a large number of training inputs seems like over-fitting the problem of digit recognition.

To compare our cortical network with O'Neil's implementation, we trained both the networks with just 1000 training digits (100 variations per digit chosen randomly) and tested it with the full 10,000 digit test set. With 100 variations of each digit, our model achieves a recognition rate of 87.5% which is slightly better than O'Reily's CNN implementation (86%). Unlike the CNN, our model does not use any carefully-tuned parameters, and is arguably more generalizable than the CNN. Overall we see that a competitive recognition can be achieved using a model that much more closely resembles the biological visual cortex.

9.3 Network Optimization using CNO

In our second experiment, we create four variations of cortical network to be deployed on the CPU and trained on images of handwritten digits from the MNIST database. Each of these networks consist of 47 hypercolumns, but the number of initial minicolumns is varied depending on the number of training digits. The network trained with 64 images is initialized with 64 minicolumns per hypercolumn, the network learning 128 images began with 128 minicolumns, and so on. In this experiment, we do not provide the supervisory feedback signal, so features are not pooled throughout the hierarchy. Figure 11 shows the execution time of each of the network configurations deployed on the CPU. As can be seen, the execution time grows substantially as the number of minicolumns is increased in each hypercolumn. After Aivo profiles and optimizes the network with CNO, we can clearly see the performance benefit of removing the unused minicolumns whose evaluation adds to the overall execution time, but not to the recognition of the dataset. These optimizations result in nearly 2x speedup for each of the network configurations tested.

These same four variations of the 47-hypercolumn cortical network are also deployed and trained on the GTX 280. In Figure 12, we see that executing the cortical network on the GPGPU can ren-

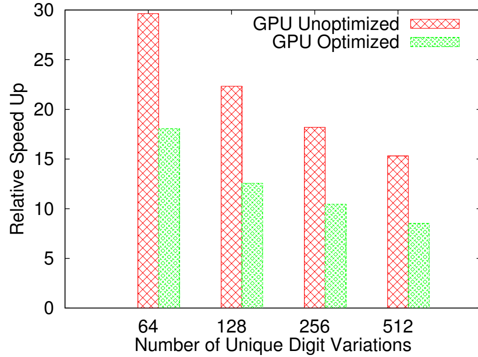

Figure 12. *Speedup of GPU vs. CPU cortical networks.*

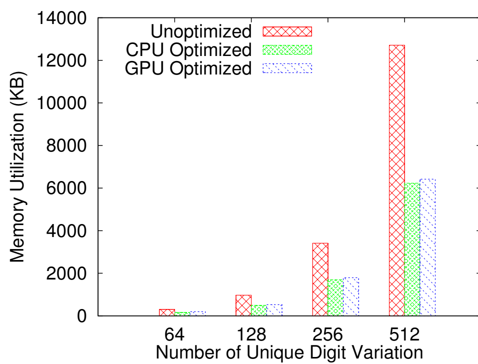

Figure 13. *Memory footprint of minicolumns before and after optimization.*

der significant speedups for both baseline and resource-optimized configurations. Executing large and unoptimized cortical network on the GPGPU intuitively exhibits a greater performance benefit since evaluating useless minicolumns does not affect the execution time if all minicolumns are evaluated in parallel. For the cortical network configurations examined, a maximum speedup of nearly 30x can be achieved when comparing an unoptimized cortical network on the CPU vs. the GTX 280 GPGPU. We also notice that the speedup for the GPGPU is maximal for the cortical network initialized with fewer minicolumns per hypercolumn. Since the minicolumns in a hypercolumn rely on the GPGPU's shared memory space for lateral communication and data sharing, smaller hypercolumns require less of this resource bottleneck. As a result, multiple hypercolumns can concurrently execute on each multiprocessor, resulting in more parallelism and better overall performance.

We also examine the amount of resources recovered by the CNO in terms of system memory in Figure 13. When the number of minicolumns is reduced across various hypercolumns, the receptive field size of higher level hypercolumns is also reduced. As this receptive field size is reduced, the number of synaptic weights needed for these upper level hypercolumns is in turn minimized as well. This results in nearly halving the amount of memory required to store the state of the overall cortical network. The optimized network deployed on the CPU recovers slightly more memory resources than the network deployed on the GPGPU. Again, we note that this is because the granularity of optimization is a result of hardware limitations of the GPGPU, and no performance benefit is achieved by reducing minicolumns at granularity less than 32 minicolumns per hypercolumn.

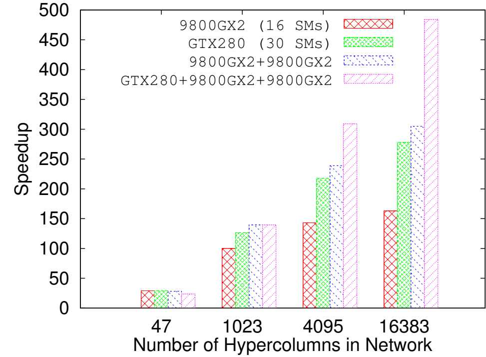

Figure 14. *Speedup of multi-GPU configurations vs. an unoptimized single-thread CPU implementation.*

	Original Network	Optimized Network	Logic Network
Execution Time / Iteration	3000 us	1080 us	68 us
Speedup	1x	2.78x	44.12x

Table 1. *The NISA abstraction provides performance benefits for equivalent optimized networks and boolean logic conversion.*

9.4 Scaling Across Multiple GPGPUs

In our third experiment, we examine the speedups achieved by deploying the learning model on the GPGPU. The number of minicolumns per hypercolumn fixed at 64 and the number of hypercolumns in the cortical network is varied. We compare the performance of the serialized cortical network deployed on the CPU to the same network deployed on a single 9800 GX2 (16 SMs), a single GTX 280 (30 SMs), a network split across both GPGPUs in the 9800 GX2s, and finally across all 3 GPUs. In Figure 14 we see that the cortical network's inherent parallelism fits well to the GPU architecture, and the achievable speedups improve with the addition of more GPU resources. The resulting speedups range from 30x to 480x for a massive cortical network proportionally split across three CUDA-enabled GPUs. We also note that the GTX 280 and the paired 9800 GX2s perform very similarly across the configurations tested, as they have a comparable number of total SMs (30 SMs compared to two GPUs with 16 SMs each). For the 47-hypercolumn cortical network, we notice a degradation in performance for the multiple GPU configuration relative to the single GPUs. This is because synchronization accounts for a larger portion of the execution time in smaller networks. It should be noted that these speedups are only achievable when batch training a set of inputs for many iterations, as frequently transferring data to and from the GPU quickly increases execution time.

9.5 Boolean Logic Generation Performance

In this experiment, we examine Aivo's ability to interpret a trained cortical network exported using the NISA abstraction and translate it into a boolean logic equivalent. Currently, the boolean logic equivalent cortical networks are simply deployed on the CPU as highly efficient C++ code, though in the future we plan to investigate these translations further, possibly creating equivalent netlists to be deployed on an FPGA or another substrate. We initialize a cortical network with 47 hypercolumns containing 20 minicolumns each, and train the network with 10 variations of a single digit until 100% recognition rate is achieved. We also provide the supervisory feedback signal to the top level of the cortical network, so features are robustly pooled throughout the hierarchy. Aivo then performs the necessary translations to convert the trained network

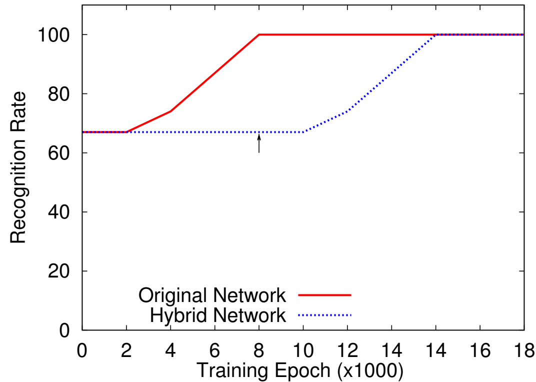

Figure 15. *Recognition performance of the original cortical network and its logic based equivalent. At 8000 learning epochs (arrow), Aivo recognizes the logic network is not firing and replaces it with a hypercolumn network.*

to its equivalent boolean logic equations, and also invokes the CNO to create an optimized version of the cortical network. Table 1 compares the performance of the optimized cortical network deployed on the CPU with the translated logic equivalent (also deployed on the CPU). We see that removing useless minicolumns provides a nearly 3x speedup, and deploying an equivalent network using boolean logic results in a 44x speedup for this 47 hypercolumn network.

9.6 Aivo's Runtime Monitoring

In this final experiment, we study Aivo's ability to revert the highly optimized boolean logic network back to a network of hypercolumns when new features are introduced into the learning dataset. If the boolean logic network deployed on the CPU does not exhibit any activity for a large number of epochs, Aivo's monitoring system detects it and reverts back to a network of hypercolumns in an effort to learn new features. For this experiment, we create a similar hierarchy as described in Section 9.5 and expose it to 10 variations of a single digit until it achieves 100% recognition rate. Once Aivo generates the optimized boolean logic network for the trained cortical network, we introduce five new variations of the same digit to the original network and optimized boolean logic network. For the sake of this experiment, we run both the hypercolumn network and boolean logic translated network alongside to observe their corresponding recognition rates.

In Figure 15, we see that initially, the new variations are not recognized by either networks. Thus, both the original and the converted logic network exhibit a recognition rate of 67%, i.e. 10 out of 15 digit variations are recognized. Since the original network of hypercolumns has enough resources available to learn new features, after 3000 training epochs, it starts to learn the new variations added to the training dataset. On the other hand, the optimized logic network does not show any improvement in the recognition rate since it can only perform the initial task it was trained to do. Aivo's runtime monitoring eventually recognizes that the logic network has not been active for a large number of epochs. As a result, Aivo translates the boolean logic network to a trainable hypercolumn network. Once the logic network is replaced with a hypercolumn network (at around 8000 epochs of no activity), we see improvement in the recognition rate of the hybrid network.

10. Conclusions and Future Work

This paper introduces the concept of a Neuromorphic Instruction Set Architecture (NISA), which separates the algorithmic description, learning, and optimization of cortical networks from their execution substrate. As a case study to demonstrate the abilities and usefulness of the NISA abstraction for neuromorphic architectures, we presented the Aivo framework. Aivo describes a NISA implementation for a rate-encoded neuromorphic systems based on modeled cortical columns, an integrated development and runtime environment, and several optimizations made possible by the NISA abstraction. The Aivo IDE simplifies the task of developing cortical networks by providing an interface to create, debug, train, checkpoint, profile, optimize, and deploy using the NISA abstraction, coded as a virtual ISA in XML. We demonstrated how the Aivo NISA is able to deploy cortical networks on general purpose CPUs as well as multiple CUDA-enabled Nvidia GPGPUs. We also showed how a trained network can be deployed as a third execution substrate, boolean logic, which is presently executed on the host CPU, though future work will investigate exporting a trained networks netlist for deployment on an FPGA or other appropriate hardware substrate. Finally, we demonstrated how the Aivo Cortical Network Optimizer performs high level optimizations to either enhance performance by removing unused modules or expand feature recognition robustness by adding more cortical modules.

The Aivo CNO showed significant performance and resource utilization improvements, resulting in a hypercolumn based cortical network with a 50% memory footprint reduction and up to a 3x speedup. Deploying a trained network as its boolean logic equivalent exhibited even more impressive runtime performance, resulting in a 44x speedup over the original hypercolumn based network. The same hypercolumn based cortical network exhibited a 30x speedup when executed on a GPGPU. Finally, a similarly structured but massively scaled cortical network exhibited up to a 480x speedup when properly distributed over three GPGPUs.

We plan to extend this work to much larger scales of networks that are capable of tackling complex classification problems like image and object recognition, and we expect deployment on GPGPU accelerators to provide the impressive speedups as evidenced in the results. As previously mentioned, we also plan to extend the role of translating a fully trained cortical network to its boolean logic equivalent, as well as hybrid networks, for deployment on an FPGA or other appropriate substrate. Finally, we plan on enhancing the NISA abstraction so other learning algorithms, such as spiking neural networks, can take advantage of the same benefits we have achieved for the cortical column model using the Aivo NISA.

Acknowledgments

We wish to thank our collaborators Olivier Temam and Hugues Berry for many fruitful discussions on cortical models, as well as the paper's anonymous reviewers and shepherds for their helpful comments. This work was supported in part by National Science Foundation award CCF-0702272, as well as equipment donations from Hewlett Packard.

References

[1] Systems of neuromorphic adaptive plastic scalable electronics (synapse). 2008. URL `http://www.darpa.mil/dso/solicitations/baa08-28.htm`.

[2] Matlab neural network toolbox, July 2010. URL `http://www.mathworks. com/ products/ neuralnet/`.

[3] Java neuroph, July 2010. URL `http://neuroph. sourceforge. net/ index.html`.

[4] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, 8(2):87 –101, 1964. ISSN 0018-8646. doi: 10.1147/rd.82.0087.

[5] J. Arthur and K. Boahen. Learning in silicon: Timing is everything. In *Proceedings of Advances in Neural Information Processing Systems*, volume 18, pages 75–82. Advances in Neural Information Processing Systems, 2006.

[6] R. K. B. Awerbuch. Competitive collaborative learning. *Journal of Computer and System Sciences*, 74(8):1271 –1288, 2008.

[7] T. Binzegger, R. Douglas, and K. Martin. A quantitative map of the circuit of cat primary visual cortex. *J. Neurosci.*, 24(39):8441–8453, Sep 2004.

[8] W. Calvin. Cortical columns, modules, and hebbian cell assemblies. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 269–272. MIT Press, Cambridge, MA, 1998.

[9] L. Chua. Memristor–the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[10] D. DeSieno. Adding a conscience to competitive learning. In *International Conference on Neural Networks, ICNN*, volume 1, pages 117 –124, 1988.

[11] F. Folowosele, R. Vogelstein, and R. Etienne-Cummings. Real-time silicon implementation of v1 in hierarchical visual information processing. In *Biomedical Circuits and Systems Conference, 2008. BioCAS 2008. IEEE*, pages 181 –184, 2008. doi: 10.1109/BIOCAS.2008.4696904.

[12] W. Freeman. Random activity at the microscopic neural level in cortex ("noise") sustains and is regulated by low-dimensional dynamics of macroscopic activity ("chaos"). *International Journal of Neural Systems*, 7(4):473–480, 1996.

[13] K. Grill-Spector, T. Kushnir, T. Hendler, S. Edelman, Y. Itzchak, and R. Malach. A sequence of object-processing stages revealed by fmri in the human occipital lobe. *Hum. Brain Map.*, 6:316–328, 1998.

[14] A. Hashmi and M. Lipasti. Discovering cortical algorithms. In *Proceedings of the International Conference on Neural Computation (ICNC 2010)*, 2010.

[15] A. Hashmi, H. Berry, O. Temam, and M. H. Lipasti. Leveraging progress in neurobiology for computing systems. In *Proceedings of the Workshop on New Directions in Computer Architecture held in Conjunction with 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[16] J. Hawkins and D. George. Hierarchical temporal memory. 2006. URL `www.numenta.com/Numenta_HTM_Concepts.pdf`.

[17] M. Holler, S. Tam, H. Castro, and R. Benson. An electrically trainable artificial neural network (etann) with 10240 'floating gate' synapses. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 191 –196 vol.2, June 1989. doi: 10.1109/IJCNN.1989.118698.

[18] D. Hubel and T. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, 195:215–243, 1968.

[19] K. Hynna and K. Boahen. Silicon neurons that burst when primed. *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 3363–3366, May 2007. doi: 10.1109/ISCAS.2007.378288.

[20] E. Izhikevich. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on*, 15(5):1063 –1070, 2004. ISSN 1045-9227. doi: 10.1109/TNN.2004.832719.

[21] H. Jang, A. Park, and K. Jung. Neural network implementation using cuda and openmp. In *DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pages 155–161, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3456-5. doi: http://dx.doi.org/10.1109/DICTA.2008.82.

[22] J. Jones and L. Palmer. An evaluation of the two-dimensional gabor filter model of simple receptive fields in cat striate cortex. *Journal of Neurophysiology*, 58(6):1233–1258, December 1987.

[23] S. Jung and S. su Kim. Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems. *Industrial Electronics, IEEE Transactions on*, 54(1):265 –271, 2007. ISSN 0278-0046. doi: 10.1109/TIE.2006.888791.

[24] E. Kandel, J. Schwartz, and T. Jessell. *Principles of Neural Science*. McGraw-Hill, 4 edition, 2000.

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

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

[27] H. Markram. The blue brain project. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 53, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: http://doi.acm.org/10.1145/1188455.1188511.

[28] T. Martinetz. Competitive hebbian learning rule forms perfectly topology preserving maps. In *International Conference on Artificial Neural Networks, ICANN*, pages 427 –434, 1993.

[29] M. Matthias and J. Born. Hippocampus whispering in deep sleep to prefrontal cortex for good memories? *Neuron*, 61:496–498, 2009.

[30] V. Mountcastle. An organizing principle for cerebral function: The unit model and the distributed system. In G. Edelman and V. Mountcastle, editors, *The Mindful Brain*. MIT Press, Cambridge, Mass., 1978.

[31] V. Mountcastle. The columnar organization of the neocortex. *Brain*, 120:701–722, 1997.

[32] A. Nere and M. Lipasti. Cortical architectures on a gpgpu. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 12–18, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-935-0. doi: http://doi.acm.org/10.1145/1735688.1735693.

[33] M. O'Neil. Neural network for recognition of handwritten digits, October 2010. URL `http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx`.

[34] Y. V. Pershin, S. La Fontaine, and M. Di Ventra. Memristive model of amoeba learning. *Phys. Rev. E*, 80(2):021926, Aug 2009. doi: 10.1103/PhysRevE.80.021926.

[35] R. Raina, A. Madhavan, and A. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880. ACM, 2009. ISBN 978-1-60558-516-1.

[36] K. L. Rice, T. M. Taha, and C. N. Vutsinas. Scaling analysis of a neocortex inspired cognitive model on the cray xd1. *J. Supercomput.*, 47(1):21–43, 2009. ISSN 0920-8542. doi: http://dx.doi.org/10.1007/s11227-008-0195-z.

[37] D. Ringach. Haphazard wiring of simple receptive fields and orientation columns in visual cortex. *J. Neurophysiol.*, 92(1):468–476, Jul 2004.

[38] U. Rokni, A. Richardson, E. Bizzi, and H. Seung. Motor learning with unstable neural representations. *Neuron*, 64:653–666, 2007.

[39] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431 –438, 2008. doi: 10.1109/IJCNN.2008.4633828.

[40] T. Serre, A. Oliva, and T. Poggio. A feedforward architecture accounts for rapid categorization. *Proc. Natl. Acad. Sci. USA*, 104(15):6424–6429, Apr 2007. doi: 10.1073/pnas.0700622104.

[41] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio. Robust object recognition with cortex-like mechanisms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(3):411–426, Mar 2007. doi: 10.1109/TPAMI.2007.56.

[42] R. Vogels and G. Orban. How well do response changes of striate neurons signal difference in orientation: a study in the discriminating monkey. *Journal of Neuroscience*, 10(11):3543–3558, 1990.

[43] H. Wersing and E. Korner. Learning optimized features for hierarchical models of invariant object recognition. *Neural Computation*, 15: 1559–1588, 2003.