Computing Hardware for Accelerated Training of Cascade-Correlation Neural Networks

ENGR 90: Senior Design Project

David German
Swarthmore College
Department of Engineering
`dgerman1@swarthmore.edu`

May 7, 2008

Abstract

This report presents an architecture for a specialized digital processor to accelerate evaluation and training of cascade-correlation neural networks by parallelization of multiply-accumulate operations. Design details are given for a prototype with 4 input, 1 output, 16 hidden, and 4 candidate nodes, implemented on an FPGA in communication with a host PC. Table 1 summarizes the performance attained for P training patterns. Technical documentation, issue tracking, and source code for the project is available to the public via a web-based project management tool.[1]

Operation	Clock Cycles
Evaluation	379
Candidate Training Epoch	$978P + 303$
Output Retraining Epoch	$783P + 2$

Table 1: Performance of the accelerator peripheral.

[1] `http://130.58.84.125/dgerman1_e90`

1 Introduction

When trained to solve standard benchmark problems, Fahlman's cascade-correlation neural networks converge in many fewer epochs than conventional back-propagation networks [9]. Nonetheless, training is often a bottleneck in the development cycle for artificial intelligence (AI) systems containing software implementations of large cascade-correlation networks. Research in the area would benefit from faster performance than general-purpose computing can presently provide.

As Fahlman's seminal paper notes, cascade-correlation is "attractive for parallel computation" because candidate nodes can be trained independently [9]. Later research investigates simplifications of cascade-correlation for parallel hardware, but gives only theoretical or simulated results [6, 1, 14, 12]. Eldredge and Hutchings report an actual hardware implementation of back-propagation [8], but the literature contains no similar work for cascade-correlation.

This report presents a scalable computing architecture for highly parallel evaluation and training of cascade-correlation networks. It documents a proof-of-concept implementation of the architecture for a small network and reports the performance attained. Project management strategies, product cost, and marketability are discussed. Complete source code is available via the Internet.[2]

2 Theory

This section gives the theory of the cascade-correlation algorithm, with an emphasis on deriving mathematical results important to the hardware implementation. Readers requiring a conceptual introduction to neural networks in general should review the proposal for this project [11], and consult a textbook such as Coppin's for more detail as needed [5].

2.1 The Cascade-Correlation Algorithm

In 1991, Fahlman and Lebiere proposed the cascade-correlation algorithm for training a neural network [9]. Under this algorithm, the dimensions of the network are not determined in advance. Rather, when training begins, a network consists solely of input and output nodes. Gradient descent is used on this two-layer network to select output weights that minimize error with the training set. When change in error per training epoch declines below an arbitrary threshold, this initial training ceases.

A pool of *candidate units* is then created. Each candidate unit is assigned a random weight for each input. The candidate units may also have diverse activation functions specified by the operator. Training epochs are performed, in which each candidate unit performs gradient descent on its input weights to maximize the correlation of its output with the *remaining error* between the target output and the network output as previously trained. Only the candidate units are modified during this phase; the rest of the network is frozen.

When change in candidate correlations per epoch declines below an arbitrary threshold, the candidate node with highest correlation to the error is installed as a hidden node. Its input weights are permanently frozen. Output weights of the input nodes and the new hidden node are then retrained to minimize error. A new pool of candidate units is then created, fed by all the input nodes and the hidden node, and the process repeats. Figure 1 is Fahlman's original schematic showing the incremental growth process.

Thus, cascade–correlation learning oscillates between two phases: the output weight retraining phase and the candidate unit input weight training phase. Each hidden layer contains a single node, fed by all inputs and all preceding nodes. All inputs and hidden layers feed the output. Qualitatively, each hidden unit specializes in detecting a "feature" of the function to be approximated that no previous hidden unit has specialized in, and then freezes its input weights so that no future training can disrupt its focus on that feature. Training stops once total error is satisfactorily low.

The rest of this section will rigorously derive some mathematical results important to the hardware implementation of cascade-correlation.

[2] `http://130.58.84.125/dgerman1_e90/browser`

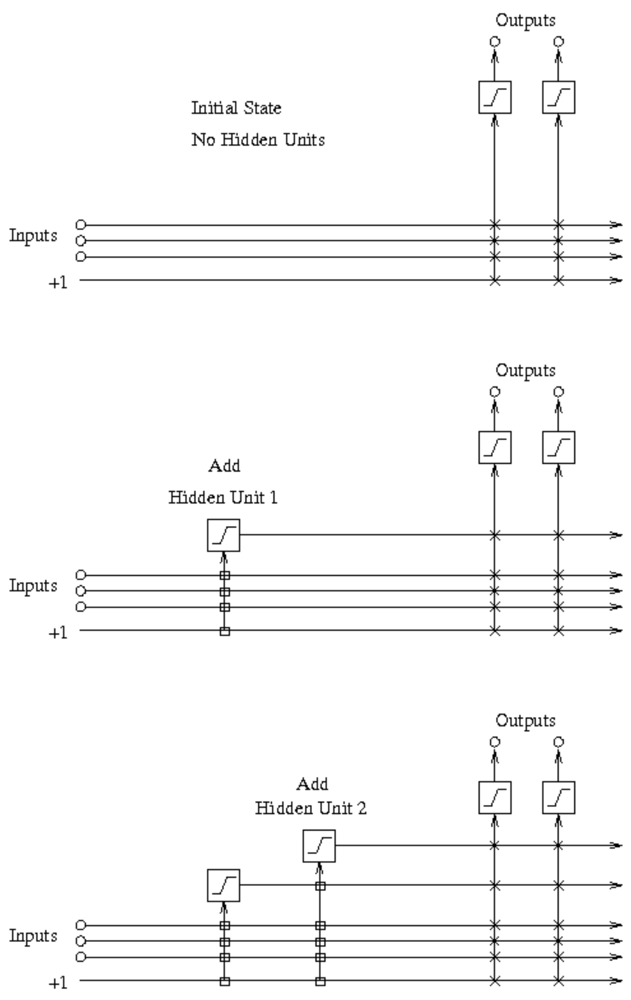


Figure 1: Fahlman's original cascade-correlation diagram: "The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly." [9]

2.1.1 Evaluation

Let $\vec{i}$ be the input vector of I elements and $\vec{o}$ be the output vector of U elements. Let f_n be the activation function for node n. Let w_{ij} be the weight that node j assigns to the output of node i. Let h_k be the hidden node in the k^{th} of H hidden layers. Then the output activation α_{h_k} of h_k is

$$\alpha_{h_k} = f_{h_k}\left(\sum_{s=0}^{I}(w_{i_s h_k} \cdot i_s) + \sum_{t=0}^{k-1}(w_{h_t h_k} \cdot \alpha_{h_t})\right) = f_{h_k}(a_{\text{net},h_k})$$

Similarly, a network output o_m is

$$o_m = f_{o_l}\left(\sum_{s=0}^{I}(w_{i_s o_m} \cdot i_s) + \sum_{t=0}^{H}(w_{h_t o_m} \cdot \alpha_{h_t})\right) = f_{o_m}(a_{\text{net},o_m})$$

This recursive formulation defines the feed-forward evaluation of a cascade-correlation network. In general, hidden node h_k cannot compute its output activation until hidden node h_{k-1} has activated; hidden node h_0 can activate given just the input vector. Network outputs cannot be computed until all hidden nodes have activated, at which point they can all be computed without dependence.

2.1.2 Activation Functions

For reasons that will become clear in the following subsections, the training algorithms require that the activation functions $f : \mathbb{R} \to \mathbb{R}$ are continuous and piecewise differentiable. Typical applications also require that f is monotonically increasing.

If some g meets these requirements, a piecewise linear function interpolated between sample points from g also meets them. The following function h, which samples g at the integers, lends itself particularly well to hardware implementation:

$$h(a_{\text{net}}) = g(\lfloor a_{\text{net}} \rfloor) + (g(\lceil a_{\text{net}} \rceil) - g(\lfloor a_{\text{net}} \rfloor)) \cdot (a_{\text{net}} - \lfloor a_{\text{net}} \rfloor)$$

2.1.3 Candidate Node Training

In candidate node training, the entire network is frozen. A pool of candidate nodes adjust their weights by gradient ascent on their output correlation S with the network's remaining error E over the training data. When that correlation ceases to improve, the highest-correlated candidate node is installed as a new hidden layer. Let $E_{o_m} = (d_m - o_m)$, where d is the desired value of o_m, and let p be the set of training patterns. Fahlman gives the following definition for S at a candidate node c [9]:

$$S = \sum_o \left| \sum_p (\alpha_{c,p} - \overline{\alpha_c})(E_{p,o_m} - \overline{E_{o_m}}) \right|$$

Qualitatively, S measures the tendency of the candidate node's activation α to be extreme on input patterns for which the network error is extreme. To perform gradient traversal on S, we must differentiate it with respect to w_{nc} for all input and hidden nodes n feeding c. Temporarily ignoring the absolute value, we find

$$\frac{\partial S}{\partial w_{nc}} = \sum_o \sum_p \left(\frac{\partial S}{\partial \alpha_{c,p}} \cdot \frac{\partial \alpha_{c,p}}{\partial a_{\text{net},c}} \cdot \frac{\partial a_{\text{net}}}{\partial w_{nc}} \right)$$

Expanding each partial derivative from the definition of S and the discussion in 2.1.1,

$$\frac{\partial S}{\partial w_{nc}} = \sum_o \sum_p \left((E_{p,o} - \overline{E_o}) \cdot f'(a_{\text{net},c}) \cdot \alpha_n \right)$$

For a differentiable function $y(x)$, $\frac{d}{dx}|y|$ is $\frac{dy}{dx}$ for $x > 0$ and $-\frac{dy}{dx}$ otherwise. Let σ be -1 if S is negative and 1 if S is positive. By the chain rule we arrive at Fahlman's result [9]:

$$\frac{\partial S}{\partial w_{nc}} = \sum_o \sum_p \left(\sigma \cdot (E_{p,o} - \overline{E_o}) \cdot f'(a_{\text{net},c}) \cdot \alpha_n \right)$$

Gradient ascent is then performed, adjusting w_{nc} in the direction of increasing S. Fahlman uses the quickprop algorithm for fast convergence, but a hardware quickprop implementation is beyond the scope of this project. Instead, we simply let η be a small, constant learning rate and compute a weight update using the well-known backprop rule:

$$\Delta w_{nc} = \eta \cdot \frac{\partial S}{\partial w_{nc}}$$

If the activation function f is h from Section 2.1.2, its derivative takes on a convenient form:

$$f'(a_{\text{net}}) = g(\lceil a_{\text{net},c} \rceil) - g(\lfloor a_{\text{net},c} \rfloor)$$

2.1.4 Output Node Retraining

In output node retraining, each output node adjusts its weights for all input and hidden nodes, including the newly installed one, by gradient descent on training pattern error. The network is evaluated for each training pattern. Then, for every output o_m, for every hidden or input node n,

$$\frac{\partial E_{o_m}}{\partial w_{no_m}} = \frac{\partial E_{o_m}}{\partial o_m} \cdot \frac{\partial o_m}{\partial a_{\text{net},o_m}} \cdot \frac{\partial a_{\text{net},o_m}}{\partial w_{no_m}}$$

Since we want to perform gradient descent on error E_{o_m}, we simply find the partial derivatives as before and invert the sign. Applying the backprop rule again, we arrive at the following weight update:

$$\Delta w_{no_m} = \eta \cdot E_{o_m} \cdot f'(a_{\text{net},o_m}) \cdot \alpha_n$$

2.2 Asymptotic Time Analysis

Consider the evaluation of a cascade-correlation network in series. Hidden node 0 must perform $O(I)$ multiply-accumulate operations to weight each input and sum it into a_{net}. It must then perform a constant-time operation to compute $f(a_{\text{net}})$. Hidden node k must perform $O(I + k)$ multiply-accumulates before activating. Thus, activating all the hidden nodes takes linear time in I and quadratic time in H. Each output node must then multiply-accumulate the stimulus from every input and hidden node, adding an additional factor of $O\left(U \cdot (I + H)\right)$. In a parallel implementation, all nodes fed by some node n can simultaneously multiply-accumulate its output when it activates. This eliminates asymptotic time dependence on U. It also reduces complexity in H from quadratic to linear, since hidden node k must perform just one multiply-accumulate after hidden node $k - 1$ activates.

In a candidate node training epoch, the network must be evaluated for each training pattern. Each candidate node must also activate, and accumulate the contributions of every output node into $\frac{\partial S}{\partial w_{nc}}$. This introduces an additional complexity factor of $O\left(C \cdot (U + I + H)\right)$ for series computation. After all patterns have been presented, actually computing weight updates is $O(CI + CH)$, which does not increase the bound. Since the training of each candidate node is independent, parallelization can eliminate the factor of C.

Analysis for an output node training epoch is similar. The network must be evaluated for each training pattern. Each output rule must then compute a backprop rule update to the weight for every input and hidden node, adding a complexity factor of $O(I + H)$. Parallelization eliminates this factor by allowing all output nodes to learn simultaneously.

Table 2 summarizes the "asymptotic" time bounds for each cascade-correlation phase. Hardware parallelization cannot affect the asymptotic behavior of an algorithm in the true mathematical sense. The size of the hardware impose a constant cap on the size of the problem; given constrained size, the problem is $O(1)$ with or without the accelerator. Table 2 thus invokes the fictional ability to make arbitrarily large hardware.

	Evaluation	Candidate Training Epoch	Output Retraining Epoch
Series	$I + H^2 + U \cdot (I + H)$	$P \cdot (CU + CI + CH + H^2)$	$P \cdot (UI + UH + H^2)$
Parallel	$I + H$	$P \cdot (I + H)$	$P \cdot (I + H)$

(a) Complexity

I	number of input nodes
H	number of hidden nodes (depth of network)
C	number of candidate nodes
U	number of output nodes
P	number of training patterns

(b) Parameters

Table 2: big-O running times for the cascade-correlation phases.

3 Design

Three parallelization opportunities are discussed in Section 2.2: accumulation of input, training of candidate nodes, and retraining of output nodes. This section will describe the design and implementation of a system to exploit these opportunities. The system consists of two components: a specialized processor that actually performs the parallel computation, and software that enables a general-purpose host PC to control the processor as a peripheral. The system supports cascade-correlation networks with 4 input, 1 output, 4 candidate, and up to 16 hidden nodes, with training on up to 256 patterns. It uses 16-bit, 2s complement fixed-point arithmetic with the radix point between bits 7 and 8.

The hardware accelerator is implemented on an Altera Cyclone II EP2C35 field-programmable gate array (FPGA). The host PC is a commonplace Intel x86-64 desktop with a Linux operating system. The two components communicate over an RS-232 serial link at 115.2 kbps with no flow control or error checking. The Host-Peripheral Interface Specification (Appendix A) lists the bit-level format and semantics of all legal messages between the host and the accelerator; both components abide by this document strictly. One could think of the accelerator as a CPU and the Specification as its instruction set; alternatively, one could abstract the accelerator as an object, and think of the Specification as the public methods. This section details the organization of both components and describes execution flow for evaluation.

3.1 Hardware

The organization of the processor can be divided along conventional lines into a control unit and a datapath. The control unit incorporates the serial manager (Section 3.1.1) and the decoder (Section 3.1.2). The serial manager parallelizes input from and serializes output to the RS-232 interface. The decoder interprets input, distributes control and data signals to the datapath, and synthesizes output. The datapath is highly specialized to the application domain: it consists of input nodes (Section 3.1.3), hidden nodes (Section 3.1.4), candidate nodes (Section 3.1.6), and output nodes (Section 3.1.5) assembled to model a cascade-correlation network. Figure 2 gives a module-level schematic of the processor.

Since the datapath bears little resemblance to the datapath in a general-purpose CPU, it will hereafter be referred to as the *network*. The network is highly stateful, incorporating nearly 140 kilobits of RAM and hundreds of registers, and this state is largely preserved between operations. A few of the interface operations, such as `dumpWeightTable` or `evaluate`, are performed simply to obtain a result, but the purpose of most is to perturb the state of the network. Incremental perturbation of the network's persistent state is the mechanism that enables the device to learn.

3.1.1 Serial Manager

Figure 3 gives state diagrams for the reception (Rx) and transmission (Tx) state machines that comprise the serial manager. The Rx machine monitors RS-232 pin 3 for a start (low) bit. Following a start bit, it samples the input on pin 3 every 8.606μs until it has collected a byte of input. It then raises a signal to the decoder indicating that the input byte is valid.

Conversely, when the decoder signals that an output byte is valid, the Tx machine latches it, raises an acknowledgment signal, and transmits a start bit on RS-232 pin 2. It then transmits each bit in the output byte, followed by a stop (high) bit. After sending a message, the host will not provide further input until some response is received, and the processor will not generate output unless it is occasioned by input from the host, so no buffering is required.

3.1.2 Decoder

The decoder is a finite state automaton that implements the Host-Peripheral Interface Specification. There are 99 named states in the decoder, but the behavior in many is influenced by other state data the decoder maintains, producing thousands of possible state configurations. As shown in Figure 2, the decoder outputs a 4-bit control bus to every node individually, an 8-bit address bus to all nodes, and an 8-bit pattern index

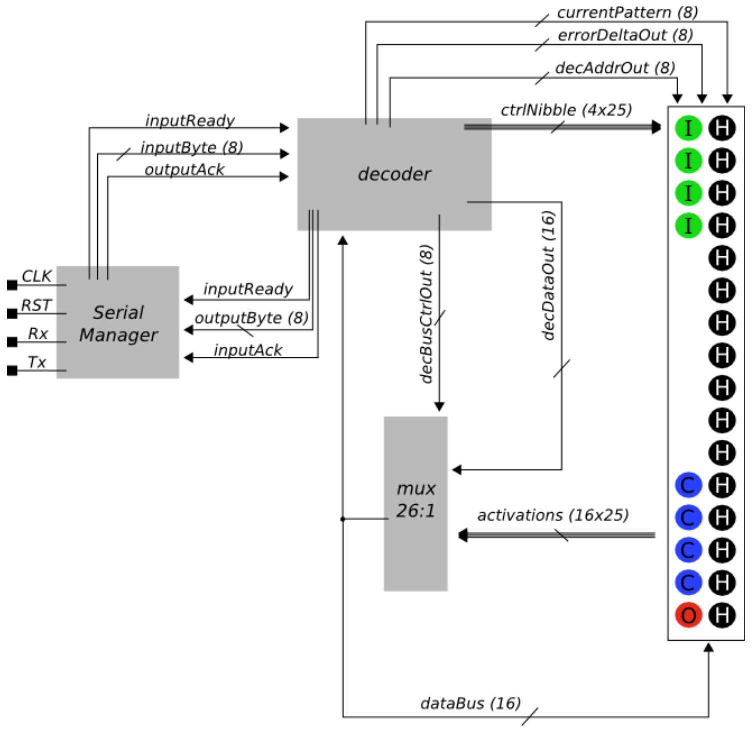
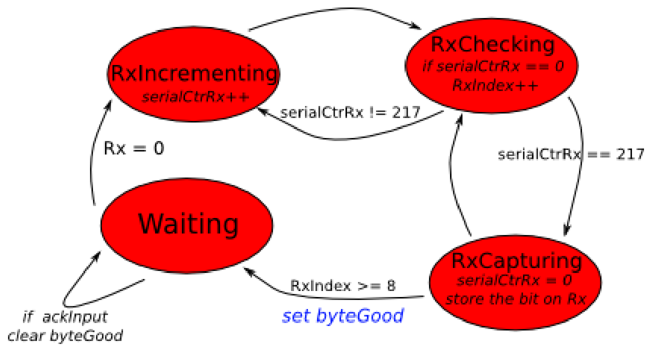
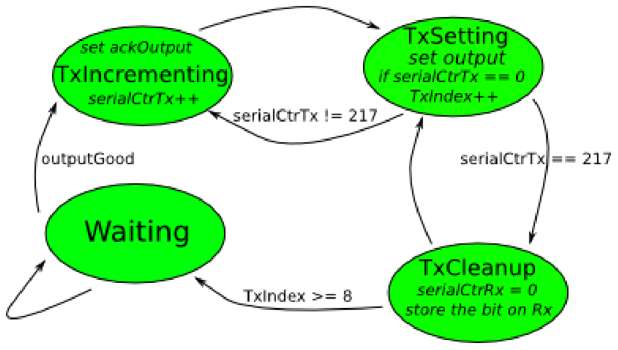


Figure 2: Module-level circuit schematic. The circles represent network nodes.



(a) Receiving



(b) Transmitting

Figure 3: State diagrams for the serial manager.

and 16-bit error bus to candidate nodes for training. The decoder also mediates write access to a 16-bit bidirectional data bus serving it and all the nodes. Thus, the decoder can supply inputs to the nodes, command them to act upon it, and direct resulting output to other nodes as appropriate. The decoder also incorporates RAM blocks for caching the training patterns. Figure 4 gives a circuit schematic.

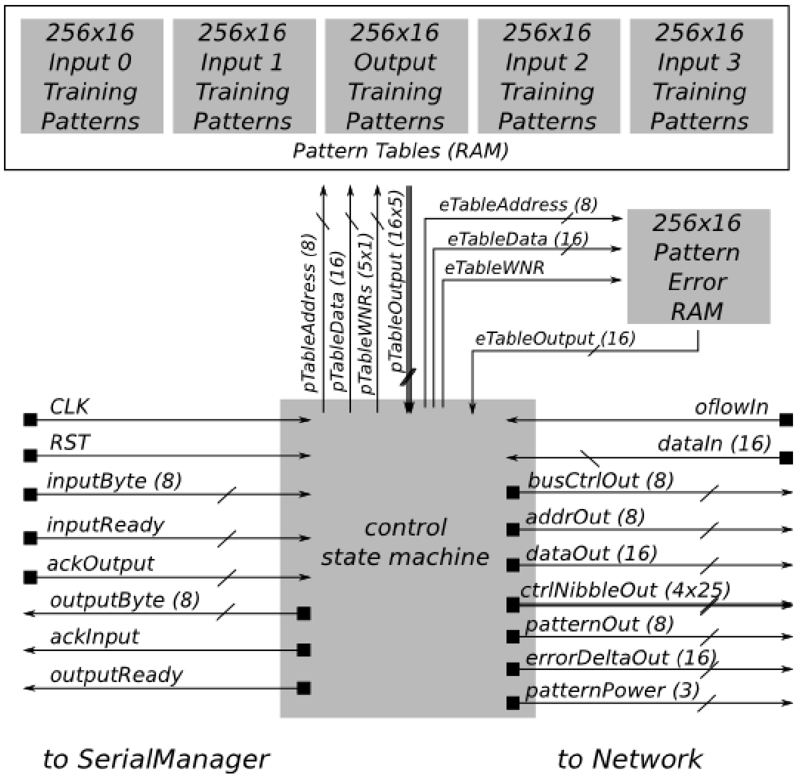


Figure 4: Simplified circuit schematic for the decoder.

The decoder initializes in the `Waiting` state, and remains in that state until the serial manager signals that an input byte is available. The first input byte must be the one of the message codes listed in Appendix A; if it is not, the decoder immediately signals the serial manager to transmit an error code and returns to `Waiting`. If the opcode is legitimate, the decoder classifies it as a *moding*, *command*, or *data* message, and transitions to an appropriate state for handling each. Moding codes consist of a single byte, and simply switch the processor between *training* and *evaluation* modes. While in one mode, it rejects all command or data messages reserved for the other by replying with an error code. Command messages instruct the peripheral to perform some network operation, or prime it to receive a certain type of data. Data messages contain numerical values for the peripheral to load directly into network state.

Command codes are always followed by a target byte. For commands that affect a single node in the network, the target byte identifies that node using the addressing scheme given in the specification. In a `setLearningRate` command, the target byte also encodes a new value for η. In a `loadTrainingData` command, the target byte gives the number of training item data messages to follow. Several other command codes prime the processor for data messages that the host must send immediately after the command, but in no other case is the number of those data messages variable. If the processor is primed for data, the decoder will respond to a command opcode with an error and return to `Waiting`. Table 3 summarizes the behavior of each command message.

If a command message is both valid and expected, the decoder transitions to a submachine of states dedicated to handling that message. Commands that prime for data simply output a code indicating that the peripheral is ready to receive data, set a state flag accordingly, and return to `Waiting`. The sub-machines for all other commands directly modify the network.

Data codes are always followed by a size byte; if N is the number of words of substantive data that follow, the size byte is $N-1$. If the processor is not primed for data, the decoder will immediately respond to data messages with an error code. Otherwise, it will latch the N subsequent words of input and write them into appropriate storage locations, informed where applicable by the target byte of the priming command.

The `weightTable` data message specifies the weight a hidden, candidate, or output node should assign to each possible feeding node. The `activationTable` message specifies the activation function for a hidden, candidate, or output node at each of the 256 integers. Each word in an `activationTable` or `weightTable` message is output to the network on the data bus, the address bus is set to the word's ordinal index, and the node specified in the target byte is given a control code to store it. Similarly, each word in an `evaluationTable` message is placed on the data bus, and the input node corresponding to the word's ordinal index is given a control code to latch the data. The contents of `trainingItem` messages are not sent to the network; instead, each word is stored in RAM internal to the decoder. As shown in Figure 4, there are five RAM blocks for storing training patterns. Each pattern occupies one line in all five blocks, so that blocks 0-3 store all the training inputs for a given input node, and block 4 stores all the expected outputs.

3.1.3 Input Node

The input node is simply a bank of 16 D flip-flops. On every clock edge, the flip-flops capture the contents of the data bus if the capture signal from the decoder is high. The decoder can connect the output of the flip-flops to the data bus via the 26:1 mux, as shown in Figure 2.

3.1.4 Hidden Node

The hidden node is a machine with 18 states, as diagrammed in Figure 6. The state machine initializes in the `Waiting` state; while in that state, it checks the control nibble on every clock cycle to determine the next transition. The decoder may set the control nibble to any of the values shown in Figure 6, which have the semantics listed in Table 4.

Figure 5 gives a simplified circuit schematic. Like the input node, the hidden node receives its inputs from the decoder, and outputs to the data bus via the 26:1 mux. It additionally has a one-bit overflow output connected directly back to the decoder; if any arithmetic overflow is detected internally, this line is set high until a `Clear` control code is received.

The RAM blocks are single-port with an $R/\overline{W}$ line not shown; they activate on the inverse clock edge, since the combinational logic driving their inputs is fast enough to permit it. The weight table stores the weight assigned by the hidden node to the nodes that feed it: lines 0-3 store weights for the four input nodes, and lines 4-19 store weights for up to 16 hidden nodes. Some lines at the end of the RAM will always be zero for nodes in the middle of the cascade. The activation table stores the activation function's value at $a_{\text{net}} = L$, where L is the integer address of the line of memory.

The linear interpolation function presented in Section 2.1.2 is easy to compute in hardware given the activation table. Recall that 16-bit arithmetic with a fixed point between bits 7 and 8 is used. Let $a_{15:8}$ be the 8 most significant bits of a_{net}. After an `Activating` control code, the hidden node subtracts the table entry at line $a_{15:8}+1$ from the entry at $a_{15:8}$ to obtain the slope, multiplies that slope by the 8 least significant bits to interpolate, and sums the resulting delta with the entry at $a_{15:8}$ to find $f(a_{\text{net}})$.

The linear interpolation process accounts for one of the multiplication operations shown in Figure 5. The other is used after an `Accumulating` control code to multiply the input data by the stored weight for the input address. The Cyclone II FPGA provides only 35 16-bit array multipliers, and multipliers implemented with general-purpose logic elements cannot meet timing specifications. Since accumulation and activation never happen simultaneously in a given hidden node, the same multiplier is used for both in the actual circuit. The relatively large number of states required to activate are spent clearing the accumulator, using its multiplier to interpolate, and then clearing it again. This expense is worthwhile, however, in view of the resource utilization results discussed in section 5.1.2.

Command	Mode	Ignores Target Byte	Primes for Data
`loadWeightTable`	Any	No	Yes
`loadActivationTable`	Any	No	Yes
`dumpWeightTable`	Any	No	No
`clear`	Evaluation	No	No
`evaluate`	Evaluation	Yes	Yes
`setLearningRate`	Training	No	No
`loadTrainingData`	Training	No	Yes
`candidateTrainingEpoch`	Training	Yes	No
`install`	Training	No	No
`outputRetrainingEpoch`	Training	Yes	No

Table 3: Summary of command message behavior.

Code	Name	Semantics
0	`Idle`	Remain in the `Waiting` state.
1	`Accumulate`	$a_{\text{net}} += w_{i_s} \cdot i_s$ for current address s and data i_s.
2	`Store Weight`	Set w_{i_s} for the current address s to the current data.
3	`Store Activation`	Set $f(a)$ for the current address a to the current data.
4	`Activate`	Compute and output $\alpha = f(a_{\text{net}})$ by linear interpolation on stored activation values.
5	`Clear`	Set the accumulated a_{net} to 0.
8	`Output Weight`	Output the stored value of w_{i_s} for the current address s.

Table 4: Hidden node control codes.

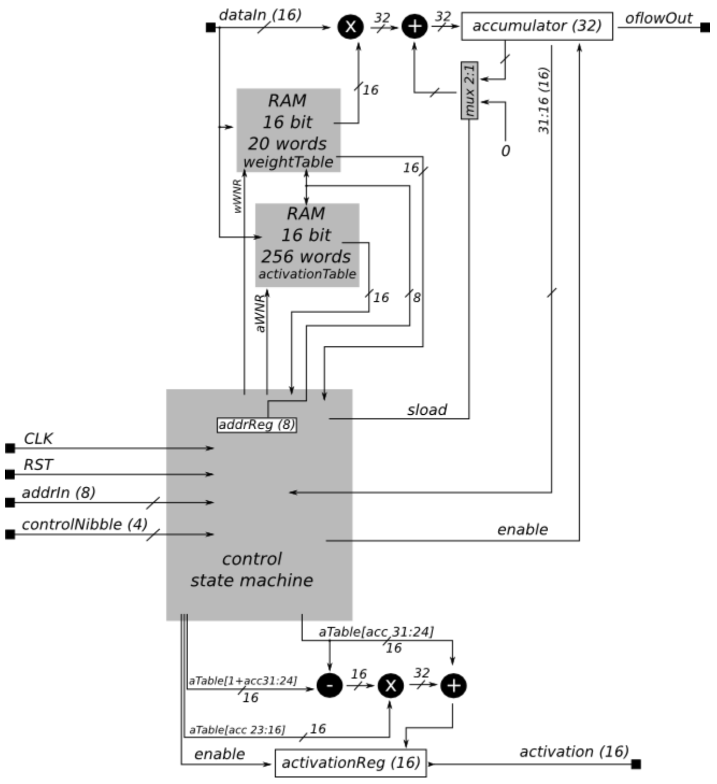


Figure 5: Simplified circuit schematic for the hidden node.

3.1.5 Output Node

The output node is a machine with 32 states, as diagrammed in Figure 7. Its functionality is a superset of the hidden node's supporting the control codes listed in Table 5 as well as the codes in Table 4.

The circuit is similar to the hidden node (Figure 5). There is an additional RAM block of 20 one-word lines for storing the most recent input α_n from each input and hidden node. These are then recalled for computing the perceptron rule weight updates after a `Learn` control code is received. Thus, if output retraining is being performed, the `Learn` control code should immediately follow an `Activate` control code.

Since η is constrained to be a negative power of 2, the learning rate can be applied by a right shift on the absolute value of $E \cdot f'(a_{\text{net}}) \cdot \alpha_n$. That product is easily and cheaply done in series, requiring just one additional multiplier. The derivative term $f'(a_{\text{net}})$ is found by subtraction on the activation table, just as is done for linear interpolation during activation. Sampling at intervals of 1 avoids performing division.

Code	Name	Semantics
9	`Set Learning Rate`	Set η to 2^{-k}, where k is the 3 LSBs of the current data.
F	`Learn`	Add $\eta \cdot (t - f(a_{\text{net}})) \cdot f'(a_{\text{net}}) \cdot \alpha_n$ to the weights for all feeding nodes n.

Table 5: Output node control codes. t is the current input, interpreted as the target output.

3.1.6 Candidate Node

The candidate node is a machine with 55 states, as diagrammed in Figure 8. Its functionality is a superset of the hidden node's, supporting the control codes listed in Table 6 as well as the codes in Table 4. Note that, by necessity, these codes reuse the namespace for output node control codes (Table 5), but an effort has been made to preserve semantic similarity.

Recall that a candidate node is trained to maximize the correlation S of its output with the error. As was shown in Section 2.1.3, the weight updates to the candidate nodes involve a sum over all patterns for each input or hidden node, so RAM is required to store those running sums. These sums are updated when the node receives a `Learn` control code; since the most recent activations and inputs are used, as in the output node, this should be done immediately after an `Activate`.

Furthermore, the product inside the sum includes $\Delta E = (E_p - \overline{E})$, which incorporates as a term the existing average error in the network. Thus, before any candidate node training begins, the existing network must be evaluated on all training patterns, and the error averaged. This necessitates the "Pattern Error RAM" seen in Figure 4: the decoder performs an evaluation pass over the training patterns, computes the difference between target and actual output for each pattern, and stores the result in this memory. It then averages the result and stores it in a register. Since the number of patterns P is constrained to be a power of 2, averaging is simply a division. The candidate nodes receive ΔE for the current pattern as a input from the decoder.

As yet another complication, the sign of S is a term in the first partials of $|S|$, so the candidate node must recompute its actual correlation with the error after each pass through the training patterns. This requires remembering its own output α_q for each pattern q, averaging them in the same fashion as the decoder averages the error, and then multiply-accumulating the $\Delta\alpha$ with the ΔE for every pattern. This computation has additional value as output to the host, which can use it to judge whether training has stagnated.

3.2 Software

The software running on the host PC consists of low-level drivers implemented in C, and a high-level Python API module. Since a standard serial link connects the peripheral to the host, the drivers are not actually a loadable kernel module; they are simply a relocatable library of functions that use the existing `serial` module to read from and write to the serial port. The library provides a function to send each message of the

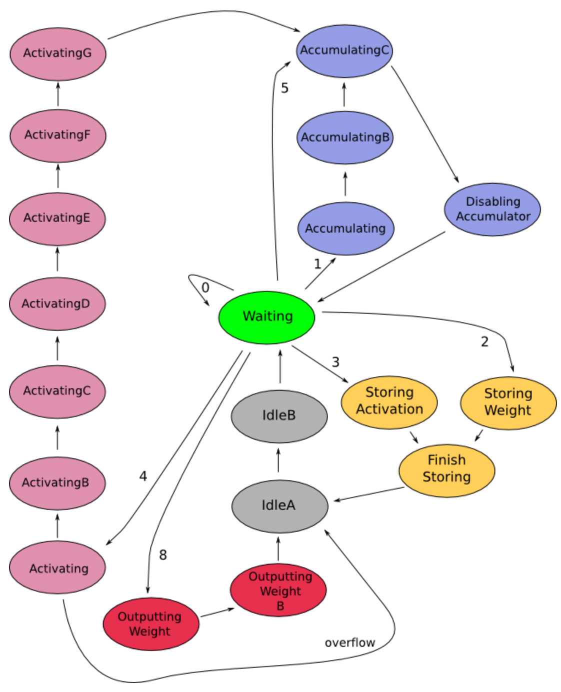


Figure 6: State diagram for the hidden node.

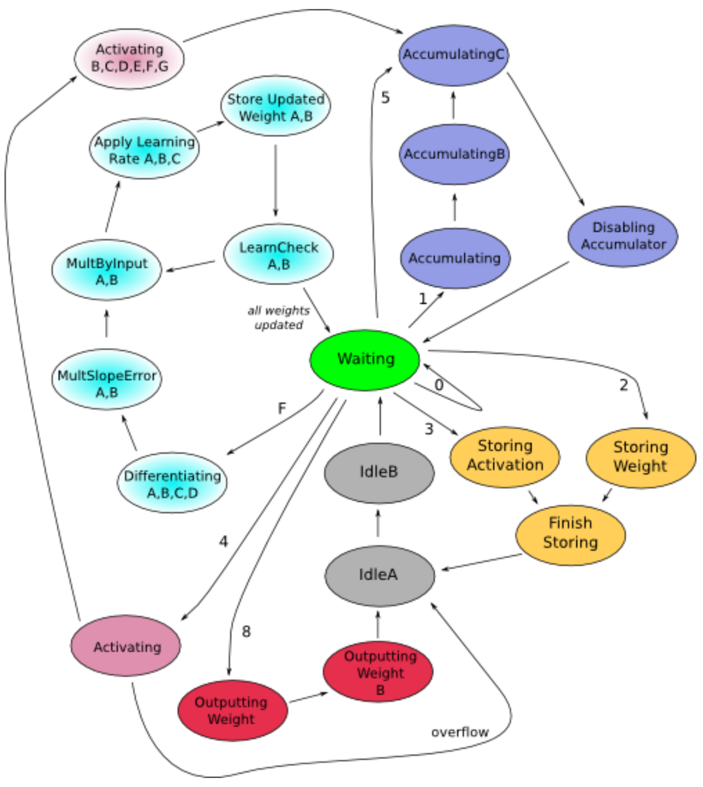


Figure 7: State diagram for the output node.

Code	Name	Semantics
6	`Clear Correlation`	Set the accumulated value of S to 0.
9	`Set Learning Rate`	Set η to 2^{-k}, where k is the 3 LSBs of the current data.
A	`Output Correlation`	Output the current value of S.
C	`Average`	Compute and store $\overline{\alpha}$ over the most recent α_q for all patterns q.
D	`Accumulate Correlation`	$S+=(\alpha_p-\overline{\alpha})\Delta E$ for all feeding nodes n.
E	`Learn`	$\frac{\partial S}{\partial w_n}+=\sigma\cdot\Delta E\cdot\alpha_n\cdot f'(a_{\text{net}})$.
F	`Absorb`	Add $\eta\cdot\frac{\partial S}{\partial w_n}$ to the weights for all feeding nodes n.

Table 6: Candidate node control codes. p is the current pattern index. ΔE is the current error delta.

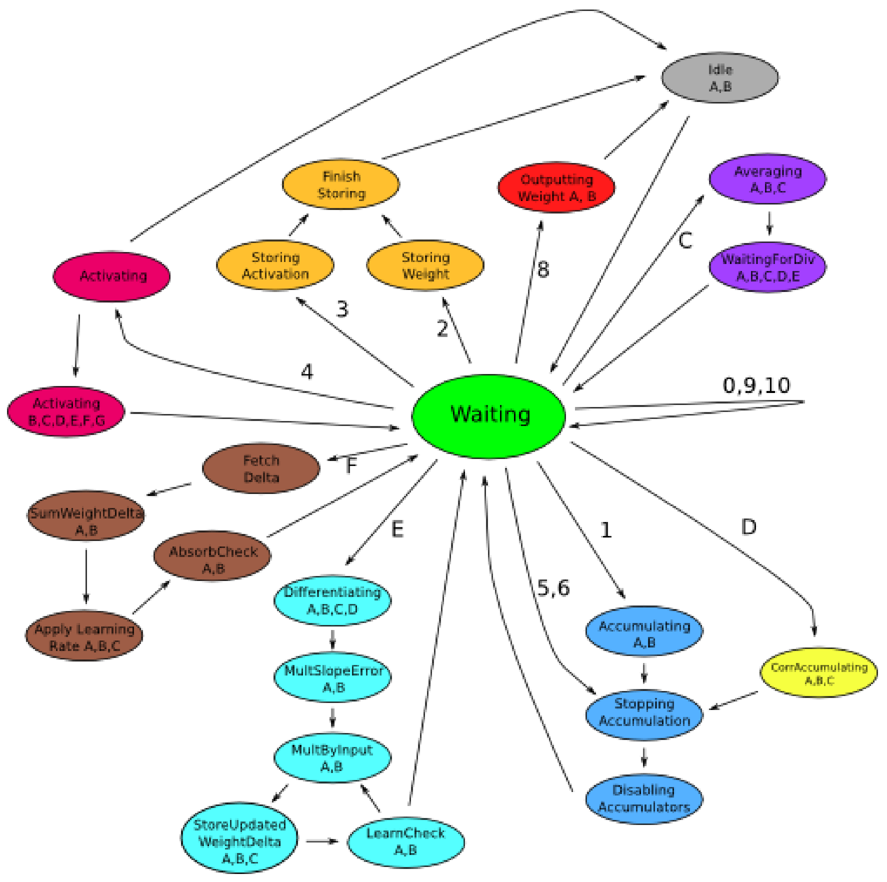


Figure 8: State diagram for the candidate node.

Host-Peripheral Interface Specification (Appendix A) and collect the response. If so inclined, the end-user could fully utilize the device from a C program that makes these calls directly.

However, the API offers a more convenient interface for Python users. It provides intermediate-level support for each message by calling the driver functions using SWiG wrappers (see Section 4.3). It then provides high-level methods that automate common evaluation and training tasks by issuing the correct series of calls, and returning only the results of interest to the user.

3.2.1 Drivers

Complete documentation for every function in the driver library is available via the Internet.[3] The driver supports an ordinary mode, in which it interacts with an actual peripheral via a serial port, and a test mode in which it writes output to a file and does not attempt to read responses. A driver session begins with a call to `openSerialDevice` or `openOrdinaryFile`, depending on the desired mode, and ends with the corresponding `closeSerialDevice` or `closeOrdinaryFile` call. When a serial device is used, the driver configures the port to disable all flow control and parity checking.

Each driver function takes the arguments appropriate to the corresponding message. For commands, these arguments are characters giving the contents of the target byte; for data, these arguments are arrays of `short int` giving each substance word of the message. It also receives a pointer to allocated memory large enough to accommodate the largest expected output.

The function assembles the arguments into an appropriate byte stream, with the opcode prepended, and passes it to the `sendMessageGetResponse` function along with the buffer pointer. `sendMessageGetResponse` writes the output, reads the peripheral's response into the buffer, and returns the number of bytes read. This number is then returned by the message function.

3.2.2 API

The API consists of the Python module `cascCorrAccelSession`. Complete documentation of the module is available via the Internet.[4] The `cascCorrAccelSession` object opens a new driver session upon instantiation, manages the session state, and provides API methods to interface with the session device.

High-level methods that issue commands take node types and indices as arguments, and construct the raw target bytes to be passed to the driver. Similarly, high-level methods that issue data messages, such as `setTrainingPatterns`, accept sequences of floating-point numbers, and construct the tables of `short int` that the target byte requires. Inputs are checked for validity before any communication with the peripheral; software exceptions are raised if the input is incorrect, so that the peripheral will not accidentally be put into an unknown state. For additional robustness, parts of the peripheral state are cached in the API. For instance, the session object will throw an exception rather than attempt a command for which the device is in the wrong mode. It also duplicates all the training patterns in host memory when setting them on the device, so that the `computeError` function can easily test the network's sum-squared error over all patterns during output retraining.

Inversely, data output from the peripheral is reassembled from little-endian byte streams into floating-point sequences that are ultimately returned to the user. If an unexpected amount of data is read, or the peripheral response indicates an error condition, an appropriate exception is generated. The user will be responsible for catching these exceptions if some response other than program termination is desired.

3.2.3 Test Case Utility

The test case utility is a simple tool for sending the contents of a binary file to the serial port and printing the response. The binary file encodes the total number of message bytes and the expected number of response bytes in its first two words. Documentation of the utility is available via the Internet.[5]

[3]`http://130.58.84.125/~dgerman1/cascCorrAccelDrivers/html/`
[4]`http://130.58.84.125/~dgerman1/API/`
[5]`http://130.58.84.125/~dgerman1/TestCaseUtility/html/`

3.3 Example: Network Evaluation

Suppose that the hidden and output node weights and activation functions for some desired cascade are already known, and the user wishes to find the network output for a particular input. Appendix B gives a detailed account of every execution step performed by the host and the peripheral in this use case. A summary follows.

To set the network up for evaluation, the host sends weight and activation tables for each node. The decoder puts each value on the bus as it arrives, and commands the target node to store it. The host then sends an `evaluate` command, and `evaluationInput` data. As each input arrives, the decoder loads it to the corresponding input node. The decoder then gives each input node the bus in turn and commands it to activate. In parallel, each hidden and output node multiplies that activation by stored weight, and accumulates the product into its net input stimulus. Now, every node that feeds hidden node 0 has activated, so its net stimulus is fully accumulated. The decoder gives it the bus and commands it to activate, and all the deeper nodes can weight and accumulate that output. Hidden nodes 1-15 can sequentially activate in the same way, and then the output node can activate. The decoder latches that output and transmits it back to the host.

3.4 Example: Training

Given an understanding of the execution flow for evaluation, the execution flows for output retraining and candidate training are intuitive. Both training cycles require that training patterns be loaded to the decoder's cache, and assume that the learning rate, activation functions, and initial weights have been set to desired values.

In an output training epoch, the decoder loads a pattern from cache to the input nodes, evaluates the network, provides the target output on the data bus, and instructs the output node to learn. If there were multiple output nodes, they could learn in parallel, as discussed in Section 2.2. In the API, stagnation detection may optionally be enabled, to terminate output retraining if total error over the training patterns ceases to decline. Since error is not explicitly computed in the training process proper, using this option slows down each epoch: the host switches the device back to evaluation mode after each training pass, and evaluates it on all patterns. It may nonetheless save the user time in comparison to a conservatively large guess at the appropriate number of epochs required.

In a candidate training epoch, the decoder evaluates the network for all patterns and computes the average error. It then replays each pattern while outputting the correct ΔE and giving the `Learn` control code to all candidates in parallel. After all patterns have been displayed, the candidate nodes have completely summed $\frac{\partial S}{\partial w_n}$, so the decoder applies the `Absorb` control code to them in parallel. It then applies the `Average` control code so that each candidate computes its $\overline{\alpha}$, again in parallel. Finally, correlation is computed: the decoder replays the ΔE for each pattern once more, with a parallel `Accumulate Correlation` control to all candidates after each. It then collects each candidate's response to `Output Correlation` and transmits the correlations back to the host. Since correlation, the function to be maximized, is an explicit byproduct of candidate training, API-level stagnation detection is inexpensive in this case. Though optional and off by default, it should be enabled in most practical situations.

4 Procedure

The design described in Section 3 was implemented, documented, and delivered on schedule. The development process was managed using conventional software engineering strategies and tools, which are easily extended to hardware design in an HDL.

4.1 Planning

Before the beginning of work, the project was planned using the critical path method (CPM). The *objective*, *approach*, and *output* for each atomic task in the project were enumerated, and the corresponding hours of work were estimated. The tasks were then assembled into a dependency graph and scheduled in the project time frame using the Merlin[6] planning tool. 40 hours of work per week were anticipated. Details of the plan, including the Gantt chart giving the final schedule, are provided in the project proposal [11].

4.2 Project Management

Trac[7], a "wiki and issue tracking system for software development projects", was used for lightweight, web-based project management. All project tasks were entered into the Trac system as *tickets*. Each ticket, in turn, was associated with a *milestone* corresponding to a major internal deadline established by the planning process. Progress on each ticket was noted in the associated comments. Technical documentation and meeting notes were entered into the Trac *wiki* for convenient access and editing. The tool was accessible to the project advisor and interested third parties for easy review of project status.[8]

4.2.1 Code Management

Subversion[9] was used for code management and version control. Every substantial change to the hardware, software, or documentation, including this report, was committed into the SVN repository along with a descriptive comment. Trac was configured to read the repository, allowing the cross-referencing of code with tickets and wiki pages, and providing a convenient interface for viewing revision diffs.

4.3 Tools

All development for the project was done in Ubuntu Linux on a typical Intel PC. The hardware was developed using Altera's Quartus II development environment, which incorporates a proprietary compiler, fitter, timing analyzer, and programmer for targeting Altera FPGAs such as the Cyclone II. The Quartus II package also includes a SPICE-type simulation tool and waveform editor that was used for unit test cases. Documentation of the hardware interfaces was manually maintained in the Trac wiki.

The host PC software was developed in the Eclipse[10] IDE, using the CDT[11] and PyDev[12] plugins. The standard GNU tool chain was used to compile the C drivers and link them into a relocatable library. SWiG[13] wrappers were generated to make the C drivers available to the Python API, which was run in the standard Python interpreter. Doxygen[14] and Epydoc[15] comments were used to automatically maintain HTML documentation of the drivers and API.

4.4 Testing

The design was subjected to *unit*, *integration* and *regression* tests, following standard practice. Test inputs and expected outputs were stored in version control, along with notes where needed. After design changes, the tests for affected components were repeated to check for unexpected side effects. All test scripts can be accessed for review via the Trac repository browser.

Each hardware component was unit tested under representative nominal and off-nominal conditions by simulation in Quartus; the ability to examine simulated register contents is a highly valuable debugging tool. The serial interface manager is prohibitively slow to simulate, since 87μs of simulation time are required to cover the transmission of one serial byte. Therefore, supplementary unit testing of interface logic was done by manually interacting with it via a serial console on the host PC. To achieve tractable simulation times for

[6] `http://www.projectwizards.net/en/merlin`
[7] `http://trac.edgewall.org/`
[8] `http://130.58.84.125/dgerman1_e90/` (may cease to be available after June 1, 2008)
[9] `http://subversion.tigris.org/`
[10] `http://www.eclipse.org/`
[11] `http://www.eclipse.org/cdt/`
[12] `http://pydev.sourceforge.net/`
[13] `http://www.swig.org/`
[14] `http://www.stack.nl/~dimitri/doxygen/index.html`
[15] `http://epydoc.sourceforge.net/`

non-interface control logic, a dummy high-speed interface controller was substituted for the serial manager. As for software, each driver function was unit tested by dumping its output to a file for examination with a hex editor. This process was extended to the Python API as it was developed.

At the Network Evaluation and Network Training milestones, the entire hardware-software system was integration tested. Inputs were developed to cover error conditions and a variety of nominal use cases. These test vectors were supplied to the hardware via the API, and the corresponding outputs as reported by the API were recorded. Expected outputs were independently computed by hand or with the FANN[16] library to validate the results.

After all development work was completed, a sampling of key tests from the entire suite were repeated as regression tests for final verification. The comprehensive demonstrations presented in Section 5 provide additional evidence that the design works as expected.

5 Results

5.1 Performance

Tables 7, 8, and 9 summarize the clock cycle consumption of evaluation, output node retraining, and candidate node training. These tables only count clock cycles spent doing actual computation. Most device time is spent waiting for I/O, since transmitting or receiving a single byte takes approximately $8 \cdot (50\text{MHz})/(115.2\text{kbps}) \approx 3472$ clock cycles. The issue of I/O speed will be discussed further in Section 6.4.

Comparing the results to general-purpose computing is not straightforward, since the details of contemporary CPU implementations are unpublished and proprietary. Such essential information as the number of clock cycles required to perform a fixed-point multiply (`MUL`) operation on Intel or AMD's latest offerings is not available. Superscalar designs will likely capture at least some of the parallelization opportunities this project exploits by detecting them in the execution buffer, and pipelining may also mitigate the losses of series computation.

However, detailed documentation of the Intel 486 instruction set is readily available[17]; this project's 50 MHz clock speed was a typical speed for a 486, so comparing the two is in some sense fair. On the 486, which was not superscalar, the `MUL` operation takes a minimum of 13 clock cycles. Clearly, it multiplies by successive addition, rather than using array multipliers.

Serialized, network evaluation requires a minimum of $4 + 5 + ... + 20 = 204$ multiply-accumulates, and 17 additional multiplications for linear interpolation. Those 221 `MUL` operations alone would take the 486 at least 2873 clock cycles. Additionally, the 486 would have to perform a variety of supporting arithmetic and memory accesses to complete the computation. The specialized processor thus evaluates the cascade in no more than 15% of the cycles that the 486 would require. Since evaluation is a step in both training epochs for every training pattern, faster evaluation directly gives faster training.

Furthermore, in output retraining, the 402 clock cycles required per pattern for learning are in principle constant for arbitrarily many output nodes, though the prototype uses only one. In candidate training, the 202 cycles per pattern for derivative accumulation are also constant for arbitrarily large pools, as are the 263 cycles for absorbing the backprop rule updates. Thus, parallel learning conserves an additional 606 cycles per pattern and 789 cycles per epoch during candidate training, a roughly 41.5% savings relative to naive serial implementation on the same hardware.

5.1.1 Timing

As reported by the Quartus II timing analyzer, the device as implemented can run safely at 50 MHz. This result was unattainable without restricting the number of training patterns to a power of 2; the division unit required to average in the general case could not be fit onto the device without creating combinational

[16]`http://leenissen.dk/fann/`
[17]`http://home.comcast.net/~fbui/intel.html`

Evaluation Step	Cycles	Repetitions	Total Cycles
Acknowledge Command	2	1	2
Parse Data Header	1	1	1
Store Inputs to Input Nodes	1	4	4
Activate an Input Node	3	4	12
Accumulate an Input Node	5	4	20
Activate a Hidden Node	15	16	240
Accumulate a Hidden Node	5	16	80
Activate the Output Node	15	1	15
Ready Output For Transmission	2	1	5
Total			379

Table 7: Accelerator performance for network evaluation, all hidden nodes in use, excluding I/O.

Output Node Retraining Step	Cycles	Repetitions	Total Cycles
Initialize	1	1	1
Load Pattern From Cache	7	P	$7P$
Evaluate The Network	370	P	$370P$
Compute Error	3	P	$3P$
Compute Backprop Rule Updates for All Weights	402	P	$402P$
Proceed to Next Pattern	1	P	P
Prepare to Transmit Response	1	1	1
Total			$783P + 2$

Table 8: Accelerator performance for one output node retraining epoch on P patterns, excluding I/O.

Candidate Node Retraining Step	Cycles	Repetitions	Total Cycles
Load Pattern From Cache	7	P	$7P$
Evaluate The Network	370	P	$370P$
Compute and Store Pattern Error	5	P	$5P$
Compute Average Error	5	1	5
Evaluate The Network	370	P	$370P$
Learn Updates to $\frac{\partial S}{\partial w_n}$	202	P	$202P$
Absorb New Weight Values	263	1	263
Clear Stored Correlations	3	1	3
Average Candidate Outputs	$8P + 16$	1	$8P + 16$
Accumulate Correlations	16	P	$16P$
Output Correlations	4	4	16
Total			$978P + 303$

Table 9: Accelerator performance for one candidate node training epoch on P patterns, excluding I/O.

propagation delays in excess of 100 ns. The timing analyzer report indicates that the slowest register-to-register propagation in the circuit is between the decoder output that controls the 26:1 mux and a register used to latch input in the fourth candidate node. This 8.961 ns delay would admit a safe clock speed as high as 54.48 MHz.

5.1.2 Resource Utilization

The Quartus II fitter successfully maps the design to the Cyclone II device. The design uses 36% of the LUT blocks in the FPGA for 10,863 combinational logic functions and 5,419 registers. 142,336 bits of integrated RAM are used, consuming just 29% of the total RAM resources. The embedded 9-bit multipliers, of which the EP2C35 series has 70, were the only resource that constrained development of the accelerator. Early project designs used 64 9-bit (32 16-bit) multipliers in the hidden nodes alone. Extensive re-engineering was undertaken to share multipliers among functions, and the entire project now fits in just 60 multipliers, for 86% utilization.

5.2 Learning Demonstrations

5.2.1 XOR

The system was successfully trained to learn the 2-bit XOR problem. Let $Y = A \bigoplus B$. In this experiment, input 0 corresponded to A, input 1 corresponded to B, input 2 was always 0, input 3 was always the bias value 1, and the network output corresponded to Y. Boolean false was represented as -1, and boolean true was represented as +1. Table 10 gives the four training patterns.

Input 0	Input 1	Input 2	Input 3	Expected Output
-1	-1	0	1	-1
-1	1	0	1	1
1	-1	0	1	1
1	1	0	1	-1

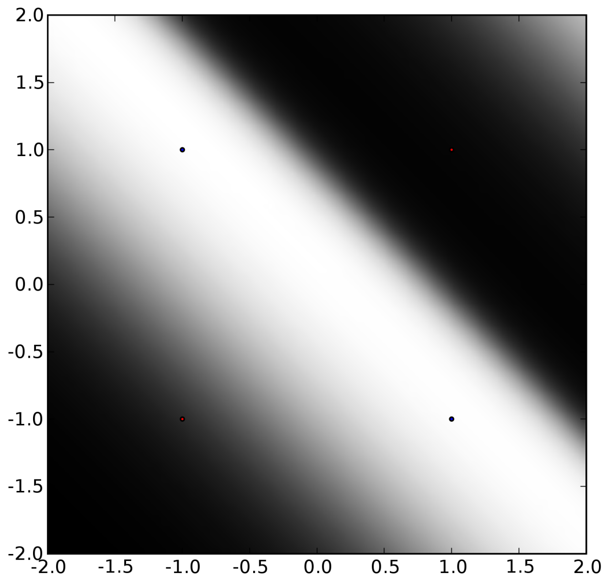
Table 10: Training patterns for two-bit XOR.

All candidate node activation functions were symmetric sigmoids, with the range $(0, 4)$. The output node activation function was a symmetric sigmoid with the range $(-2, 2)$. Initial candidate node weights were randomly selected from the interval $(-1, 1)$. The learning rate in all nodes was 2^{-3}. These parameters were manually tuned for reliable, fast training. Many unsuccessful pattern combinations were attempted before this one was discovered. The results of learning are highly dependent on initial parameters, but the reasons that some parameter choices fail to find global optima in the gradient search are seldom obvious. This weakness is intrinsic to the algorithm, not a property of the hardware implementation.

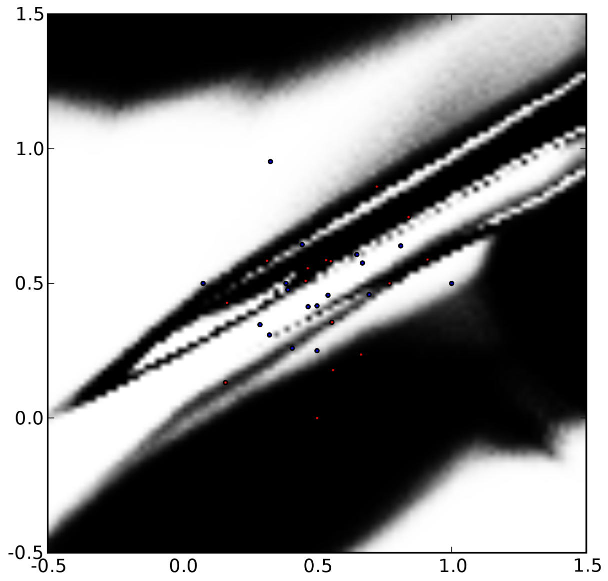
Since there is no linear separability in the 2-bit XOR problem, initial perceptron training is unproductive, and stagnates quickly. With the final parameters, about 200 epochs of candidate training to produce a single hidden node, followed by about 200 epochs of output retraining, are typically required. The exact duration of training depends unpredictably upon the initial candidate weights. The problem is considered solved when all test patterns expecting +1 produce outputs greater than +0.5, and all test patterns expecting -1 produce outputs less than -0.5.

Typical results are shown in Figure 9a. Input 0 is along the x-axis, and Input 1 is along the y-axis. Red dots indicate a training pattern for which the false output is expected, while blue dots indicate a training pattern for which the true output is expected. The greyscale background image indicates the learned behavior of the network, evaluated at each point on a fine grid. The lighter a pixel is, the closer to +1 is the network's output at those coordinates.

Similar results were obtained for the 3-bit XOR problem. With the same parameters, the network learned to classify all eight training patterns using a single hidden node, in about 200 epochs of each type of training.



(a) 2-bit XOR



(b) 32 points from the two spirals problem

Figure 9: Learning Results

5.2.2 Two Spirals Problem

Figure 10 is a plot of the two spirals problem, a classic neural network benchmark.[18] Each point marks a training datum; true outputs are expected for one spiral, false outputs for the other. The network must therefore learn to draw fine, sharp lines between the two. Fahlman reports that his cascade correlation software outperformed the best known traditional layered network by a factor of 23 in total multiplications, using between 12 and 19 hidden nodes [9].

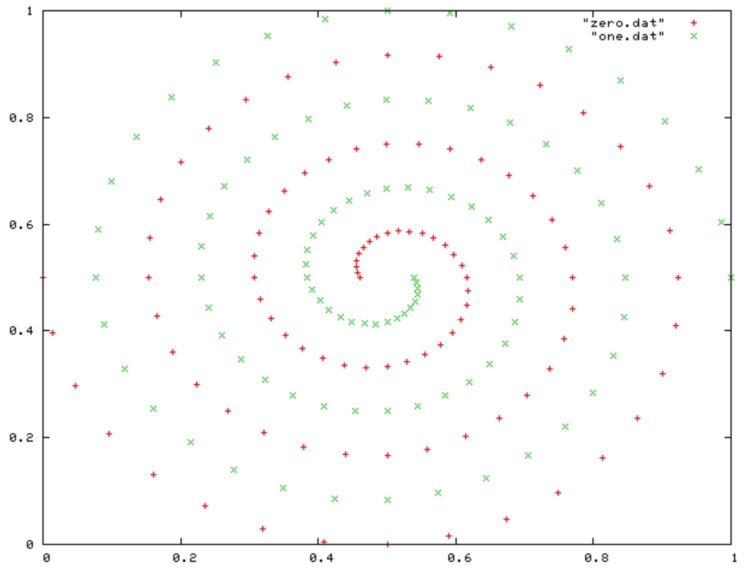


Figure 10: Plot of the complete 194-point two spirals problem.

Parameters that replicate this success have not yet been found for the project system. In every attempt to learn with a sample of 128 points from the spiral, both output training and candidate retraining stagnate after just a few epochs, typically on the order of 10^1. Accordingly, the sum squared error achieved is little better than random guessing. With -1 representing boolean false and +1 representing boolean true, no SSE below 110 has been attained. Since dozens of attempts have been made with random initial weights, the behavior suggests that in most parts of the input space, the problem's error and correlation surfaces are nearly flat. Ramifications will be discussed further in Section 6.3.

Successful learning has been accomplished on smaller subsets of the two spirals. Figure 5.2.2 shows successful learning of 32 randomly sampled points. All hidden, candidate, and output units are using symmetric sigmoid activation with range $(-1, 1)$ and $\eta = 2^{-3}$. All 16 hidden units are used. Interestingly, the network solves this problem without generalizing to the spiral. Enough high-frequency information has been removed that opportunistically drawing wedges around neighboring points of the same classification suffices to minimize error.

6 Conclusions

The system presented in Section 3 successfully evaluates and trains a cascade-correlation neural network to solve several test problems. Moreover, the specialized parallel processor presented in Section 3.1 requires substantially fewer clock cycles than a general-purpose CPU to solve the same cascade-correlation problems. The project is therefore a successful proof of concept for the architecture and organization used.

[18]Image from the Pyro Robotics project at `http://pyrorobotics.org/?page=TwoSpiralsProblem`

6.1 Costs

The following discussion is restricted to costs incurred solely by this project. It excludes Engineering department activities that support ENGR 90 in general.

6.1.1 Material

An Altera DE2 prototyping board, based on the Cyclone II FPGA, was the only cash expenditure. It was purchased directly from Altera for $269, exceeding the initial project budget of $200 by 34.8%. As is discussed in the project proposal, the overrun was approved in advance by the project advisor and by the Engineering department chair. Section 6.2 notes some auxiliary returns to the department from the investment.

The project also consumed various resources of unknown cash value. For the 16 weeks of active project development, it required exclusive use of a laboratory space, a high-end computing workstation, and an RS-232 cable. The workstation, DE2 board, laboratory lighting, and laboratory climate control consumed some unknown amount of electricity. Network resources were used both to publish project materials and to look up relevant documentation on the Internet. Approximately 200 pages of meeting notes, draft reports, checklists, handouts, and other paper articles were printed on the department laser printer in connection with the project, and about 150 pages of photocopies were made on library copiers.

6.1.2 Human

The project's human resource costs were approximately as follows:

- 550 hours of primary developer time for
  - proposal writing and specification development, 50 hours
  - design and implementation, 200 hours
  - testing and debugging, 200 hours
  - documentation, 75 hours
  - meetings and presentations, 25 hours
- 25 hours of Prof. Tali Moreshet's time for meetings, presentations, debugging assistance, and review.
- 4 hours of Prof. Lisa Meeden's time for discussion of neural network theory and current practice.
- 2 hours of Edmond Jaoudi's time for procurement and workstation configuration.

6.1.3 Environmental

The electricity and printing costs noted in Section 6.1.1 entail environmental costs. These costs were mitigated as much as possible by using only half the light bulbs in the laboratory, turning off the DE2 board when it was not in use, and reusing printouts as scratch paper before recycling them. Unfortunately, powering off the workstation itself was not an option, since it was hosting the Trac web interface for public access.

FPGAs are reprogrammable indefinitely. Barring an improbable failure or accident, the Altera DE2 board will likely be reusable by the department for many years. Since the department is currently using FPGA prototyping boards from 1994 in its curriculum, it seems unlikely that the DE2, a model that dates to 2001, will be retired before 2015. Even when the department is through with it, the board may well pass into other hands for ongoing reuse. By the time it is ultimately discarded as trash, it is difficult to predict what the conventional disposal method for electronics will be. Encouragingly, the FPGA on the DE2 is lead-free, but further information about materials used in the board is not readily available.

If a device based on the architecture presented in this report entered mass production as an ASIC, its environmental footprint would be comparable to any other integrated circuit device. Despite the attractiveness of the architecture, this situation is unlikely in the near future, as argued in Section 6.4.

6.2 Auxiliary Benefits

As mentioned in Section 6.1.3, the Altera educational FPGA prototyping boards presently in curricular use by the Engineering department are obsolete. In at least two instances known to this author, final projects for an engineering course have only been tested in simulation because they could not fit onto the department's devices. Engineering faculty have expressed interest in procuring newer kits. This project has therefore served as a low-cost dry run for a larger-quantity DE2 purchase by the department.

Such a purchase is unequivocally recommended. Since the Cyclone II EP2C35 could fit this project, albeit tightly (see Section 5.1.2), it is likely capacious enough for any ENGR 15: Digital Systems, ENGR 25: Computer Architecture, ENGR 71: Digital Signal Processing, or ENGR 72: Electronic Circuit Applications project in the foreseeable future. The DE2 also integrates a number of potentially valuable features not used in this project, such as 8 MB of SDRAM, an SD card slot, USB ports, A/V jacks, and an LCD display.

The DE2 is attractive for reliability as well as features. Programming the DE2 by USB is much faster and less failure-prone than the parallel-port JTAG connectors used by the legacy hardware. Most importantly, the Quartus II 7.0 development environment, at least when used in Linux, does not share Quartus II 5.0's habit of crashing with opaque error dialogs. Officially, Altera only supports Quartus for Windows and Red Hat Linux, but it was used under Ubuntu Linux throughout this project with just a few initial configuration hurdles. Quartus II 7.0 is supported by the College's existing Altera license server for any platform.

6.3 Known and Suspected Issues

As noted in Section 5.2.2, parameters that allow the system to solve the full two-spirals problem have not yet been found. It is suspected that the error and correlation surfaces are nearly flat over most of the input space, making successful parameters exceptionally difficult to find. Nonetheless, Fahlman uses cascade correlation to solve the problem reliably [9]. It is possible that the parameter tuning is just highly sensitive, and this research simply has not hit upon the right combination of learning rates, initial weights, input and output scaling, and activation functions. However, there are two other hypotheses explaining the discrepancy that should be investigated in future work.

Hypothesis 1: Training Overflow

The hardware checks the accumulation of a_{net} for overflow, but it does not check the various multiplications or accumulations used in training. During output retraining, if the error on a training pattern is large, and some α_n is also large, and $f'(a_{net}) > 1$, the multiplication of those three 16-bit numbers into a 32-bit register could overflow. In most tests, however, symmetric sigmoid activation functions were used without large scaling. Thus, it was usually the case that $f'(a_{net}) < 1$ for any a_{net}. Failure by overflow is a more plausible scenario during candidate training. If several successive training patterns generated large components of the same sign, the accumulated value for $\frac{\partial S}{\partial w_{nc}}$ could certainly experience intermediate overflow of the 16-bit storage. Both conditions should be checked, and the peripheral should generate error messages if they arise. If overflow is determined to be a problem, widening the affected memories should compensate.

Hypothesis 2: Search Rule Selection

The backprop rule used in this project computes weight updates using only the first derivatives of error and correlation with respect to each weight. Although he claims any search rule will work, Fahlman himself uses the quickprop rule, which incorporates information from higher-order derivatives into the weight update [9]. Sensitivity to acceleration may be critical for traversing shallow slopes quickly, and thus avoiding apparent stagnation. When considering the effects of limited precision on cascade correlation, Fahlman gives a quickprop algorithm that is convenient for hardware implementation [12]. Future revisions of the project should use it.

6.4 Marketability and Future Work

The selection of tools and technologies for this project was tailored for low-cost rapid prototyping. Unsurprisingly, these design choices have also ensured that the current product is not useful for serious AI research. Researchers who use cascade correlation have indicated that the most serious shortcomings are in network size, training set size, and real speed. Notably, the requirement that learning rates and training set sizes be powers of 2, the linear interpolation of the activation function, and the use of fixed-point arithmetic are not viewed as problematic.

Given these observations, there is a clear path for future development that will mature the prototype into a viable product without dramatic architectural changes. Firstly, the obsolete serial interface should be replaced with a modern bus technology such as PCI-X or HyperTransport. The chip itself could be updated to use a new interface simply by replacing the serial manager. Of course, moving away from the DE2 board would entail additional engineering effort for PCB layout, JTAG support, and power regulation.

Secondly, the network could be scaled up massively if the design were implemented as an ASIC. Some reworking of the design files would be required to support additional hidden nodes, output nodes, and training set capacity. The width of address buses and associated registers would change; delays would be adjusted; target and size information would no longer fit in a byte; Altera megafunctions for RAM and DSP units would be replaced with custom design. Much smaller process sizes and correspondingly higher clock speeds would be attainable, but additional engineering effort would be invested in timing, since the development tools would not be able to analyze it automatically. Importantly, none of these issues affect the fundamental architecture, which can scale as large as the fabrication technology will permit.

The project's results suggest that a scaled-up ASIC implementation, on a fast bus, running at clock speeds comparable to modern CPUs, would dramatically outperform a general-purpose computer at cascade-correlation training. It is an open question whether the demand for fast cascade-correlation, likely limited to academic and industrial AI laboratories, is sufficient to justify the engineering and production costs of ASIC deployment. Market research on that point should certainly precede any serious work in this direction.

As an alternative, the design could be scaled up and clocked faster on a more modern FPGA. The Cyclone II was a mature design in 2001. Altera's highest-end Stratix III FPGAs use a 65 nm process and have up to 896 18-bit multipliers, along with much-increased RAM and LUT resources. At a very conservative guess, the computational network could be 10 times as large, clocked to 1 GHz, have 2^10 inputs, and support 2^{12} training patterns. A bus upgrade would still be necessary, of course, but the total engineering effort is much reduced by remaining on an Altera platform. Moreover, the device would remain reconfigurable; network size could be traded off for speed, for training set size, or for precision simply by reprogramming. Also, the large startup costs of ASIC production would be avoided. This avenue lends itself to commercialization in a consulting model: the vendor produces the accelerator, deploys it at the client's site, and is available to assist with modifications and updates as the client's needs change. This model is likely more suitable for serving the small but well-funded market of institutional labs.

7 Acknowledgments

I thank Prof. Tali Moreshet for advising this project, and the Swarthmore College Department of Engineering for approving, supporting, and funding it. I am also grateful to Prof. Lisa Meeden for sharing her expertise in artificial intelligence, to George Dahl for suggesting the project topic, and to Jennifer Barry, Jeffrey Kaufman, and Lucas Sanders for enlightening conversation that contributed to the project's development.

References

[1] Shumeet Baluja and Scott E. Fahlman. Reducing Network Depth in the Cascade–Correlation Learning Architecture. Technical Report CMU–CS–94–209, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1994.

[2] Seth Bridges, Miguel Figueroa, David Hsu, and Chris Diorio. Field-Programmable Learning Arrays. In *Proceedings of the 2002 Neural Information Processing Systems Conference*, pages 1179–1186, 2002.

[3] Seth Bridges, Miguel Figueroa, David Hsu, and Chris Diorio. A Reconfigurable VLSI Learning Array. In *Proceedings of ESSCIRC 2005 31st European Solid-State Circuits Conference*, pages 117–120, 2005.

[4] T.G. Clarkson, Chi Kwong Ng, and Yelin Guan. The pRAM: An Adaptive VLSI Chip. *IEEE Transactions on Neural Networks*, 4(3):408–411, May 1993.

[5] Ben Coppin. *Artificial Intelligence Illuminated*. Jones & Bartlett, 1 edition, April 2004.

[6] T.A. Duong. Cascade Error Projection: An Efficient Hardware Learning Algorithm. In *IEEE International Conference on Neural Networks*, pages 175–178, 1995.

[7] Silvio P. Eberhardt. Analog Hardware for Delta–Backpropagation Neural Networks. U.S. Patent Application, August 1989. NASA - Jet Propulsion Laboratory.

[8] James G. Eldredge and Brad L. Hutchings. RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs. In *IEEE International Conference on Neural Networks: IEEE World Congress on Computational Intelligence*, pages 2097–2102, 1994.

[9] Scott E. Fahlman and Christian Lebiere. The Cascade–Correlation Learning Architecture. Technical Report CMU–CS–90–100, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, August 1991.

[10] William A. Fisher, Robert J. Fujimoto, and Robert C. Smithson. A Programmable Analog Neural Network Processor. *IEEE Transactions on Neural Networks*, 2(2):222–228, March 1991.

[11] David German. Engineering 90 Project Proposal: Computing Hardware for Accelerated Training of Cascade-Correlation Neural Networks, 2007.

[12] Markus Hoehfeld and Scott E. Fahlman. Learning with Limited Numerical Precision using the Cascade–Correlation Algorithm. *IEEE Transactions on Neural Networks*, 3(4):602–610, July 1992.

[13] Genevieve B. Orr. Error Backpropagation. `http://www.willamette.edu/~gorr/classes/cs449/backprop.html`.

[14] D.S. Phatak and I. Koren. Connectivity and Performance Tradeoffs in the Cascade Correlation Learning Architecture. *IEEE Transactions on Neural Networks*, 5(6):930–934, November 1994.

[15] Shigeo Sakaue, Toshiyuki Kohda, Hiroshi Yamamoto, Susumu Maruno, and Yasuharu Shimeki. Reduction of Required Precision Bits for Back–Propoagation Applied to Pattern Recognition. *IEEE Transactions on Neural Networks*, 4(2):270–274, March 1993.

[16] Takao Watanabe, Katsutaka Kimura, Masakazu Aoki, Takeshi Sakata, and Kiyoo Ito. A Single 1.5–V Digital Chip for a 10^6 Synapse Neural Network. *IEEE Transactions on Neural Networks*, 4(3):387–392, May 1993.

Appendices

A Host-Peripheral Interface Specification

This document governs communications between the network training accelerator and the PC host. It defines the communications technology, and the format and semantics of all messages.

Technology

A standard serial link will be used for communications between the peripheral and the host PC. The link will operate at 115.2 kbps = 8.68 μs/bit. All multi-byte encodings are little-endian.

Format

Type Bits	Type
00	Moding
01	Command
10	Data
11	*Reserved*

(a) Type Codes

Node Bits	Node
00	Hidden
01	Candidate
10	Output
11	*Reserved*

(b) Node Codes

Moding messages are a single byte. Command messages are two bytes long. Data messages may have an arbitrary length in whole bytes. The first two bits of a message indicate its type, as presented in Table 11a.

The least significant six bits of the first byte of a Moding or Command message specify an opcode. The second byte of a Command message specifies a target. If, semantically, the target is a node, the type of node will be specified by the most significant two bits of the target byte as presented in Table 11b.

The least significant four bits will be used to address a hidden node. The least significant two bits will be used to address a candidate node. The single output node requires no addressing.

The least significant six bits of the first byte of a Data message specify the type. The following byte has unsigned value one less than the number N of words to follow. The subsequent $2N$ bytes, hereafter called the *substance* bytes, encode the data in the manner specified for that particular message.

Moding Messages

`beginEvaluation`

opcode 000000. Enter network evaluation mode.

`beginTraining`

opcode 000001. Enter network training mode.

`forceEvaluation`

opcode 111111. Forces a return to evaluation mode regardless of any other conditions. Removes any expectation of data. The peripheral must respond with resultCode 000000.

Command Messages

`clear`

opcode 000000. Unfreeze all hidden nodes and reset internal state such that new training will install the first successful candidate as hidden node 0.

`dumpWeightTable`

opcode 000010. The target byte will specify a node. The peripheral responds with the 40 substance bytes of weightTable message, giving the contents of the weight table for the target node.

`evaluate`

opcode 111110. The peripheral must be in training mode. Compute the network output for a provided input, adjusting no weights. The peripheral must be in evaluate mode.

The peripheral will nominally reply with resultCode 001000. The host will then send an evaluationInput message. The peripheral will evaluate the network. If no error arises, it will send the two-byte network output.

`loadActivation`

opcode 000100. The target byte will specify a node. In nominal circumstances, the peripheral will acknowledge with resultCode 001000. The host must then send an activationTable message. IMPORTANT: After an activation, the input accumulation is cleared.

`loadWeightTable`

opcode 000101. The target byte will specify a node. In nominal circumstances, the peripheral will acknowledge with resultCode 001000. The host must then send a weightTable message.

`setLearningRate`

opcode 111000. The target byte will specify a candidate or output node with the two MSBs (7-6) and two LSBs(1-0); bits 4-2 will give an integer k. The learning rate for the targeted node will be set to 2^{-k}. In nominal circumstances, the peripheral will reply with resultCode 000000 or resultCode 000001.

`loadTrainingPatterns`

opcode 111001. Initiate the training of a new hidden node. The peripheral must be in training mode. For p training patterns, the target byte will be $\log_2 p$. The training set may have at most 256 elements, so the value of the target byte must not exceed 8.

In nominal circumstances, the peripheral will reply with resultCode 001000. For each of the patterns in the training set, the host will send a `trainingItem` message, and the peripheral will respond nominally with resultCode 001000. When the last element of the training set is transmitted, the peripheral will respond with resultCode 000001.

`candidateTrainingEpoch`

opcode 111100. The peripheral must be in training mode. Instruct the peripheral to perform one epoch of candidate training using the loaded training patterns. The peripheral will reply with four words giving the correlation to the error achieved by each candidate node.

`install`

opcode 111101. The peripheral must be in training mode. Indicate to the peripheral that the candidate node specified in the target byte should be installed as a hidden node. The weight table of that node will be copied into the next unused hidden node. In nominal circumstances, the peripheral will reply with result code 001000. The host will typically then send a `outputRetrainingEpoch` message.

`outputRetrainingEpoch`

opcode 111111. The peripheral must be in training mode. Instruct the peripheral to perform one epoch of output weight retraining using the loaded training patterns. The peripheral will reply with result code 000001 under nominal conditions.

Data Messages

Entry	Value
0	Input 0
1	Input 1
2	Input 2
3	Input 3

(a) `evaluationInput`

Entry	Value
0	Input 0
1	Input 1
2	Input 2
3	Input 3
4	Target

(b) `trainingItem`

Entry	Value
0	Input 0
1	Input 1
2	Input 2
3	Input 3
4	Hidden 0
5	Hidden 1
...	...
19	Hidden 15

(c) `weightTable`

Table 11: Data message formats.

`activationTable`

The opcode for an `activationTable` message is 000100. It has 512 substance bytes.

An activation table specifies the activation function of the target node. It has 256 signed entries. Entry 0xNN stores a 16-bit word that gives the value of the activation function at 0xNN00. Activation levels at net inputs between the listed values will be linearly interpolated.

`evaluationInput`

The opcode for an `evaluationInput` message is 111110. It has 8 substance bytes, consisting of 4 16-bit words, as shown in Table 11a.

`trainingItem`

The opcode for a `trainingItem` message is 111111. It has 10 substance bytes, consisting of 5 16-bit words, as shown in Table 11b.

`weightTable`

The opcode for a `weightTable` message is 000101. It has 40 substance bytes.

A weight table has 20 entries of 16 bits that record the weight assigned by a node to the output activation of the four input nodes and the sixteen hidden nodes. For the output node, all entries in the weight table will be relevant. For a hidden or candidate node, immaterial entries will always be 0. The entries are signed. Table 11c shows the organization.

Result Codes

The peripheral will not implement any message queueing. After sending a message to the peripheral, the host will not send another message until it receives a response from the peripheral. In cases that do not prompt a response with substantive data, the peripheral replies with a result code. Result codes are eight bits, with the semantics listed in the Table 12. Unenumerated values are reserved.

Code	Meaning
000000	Success. In Evaluation Mode. Any Moding or Command message legal.
000001	Success. In Training Mode. Any Moding or Command message legal.
001000	Success. Ready to receive data.
100000	Error: Undefined Message Code.
100001	Error: Training message not valid in evaluation mode.
110011	Error: Expected a data message.
100010	Error: Invalid Data - repeat.
100011	Error: Invalid Target Byte - repeat.
100100	Error: All Hidden Nodes In Use. In Evaluation Mode.
110000	Error: Arithmetic Error. In Evaluation Mode.
110001	Error: Overflow Error. In Evaluation Mode.
101000	Error: Default Error. In Evaluation Mode.

Table 12: Result codes.

B Evaluation Cycle Detail

The following list gives the series of calls made by a Python script for simple network evaluation, and the resulting activity in the accelerator. This procedure fully captures the parallelization opportunity discussed in Section 2.2.

1. The script instantiates a `cascCorrAccelSession` object with the path to the serial device.

2. For each hidden and output node, the script computes a sequence giving the activation function at all integers on the interval $[-128, 127]$, and passes that sequence to the API function `setActivation_Hidden` or `setActivation_Output`.

   (a) The API calls the driver function `Command_loadActivationTable`, which transmits that command's opcode followed by the appropriate target byte.

   (b) The decoder stores the opcode and target in registers, and responds with result code 001000: ready to receive data. This code is returned to the API, which notes that it is an expected result code, and therefore takes no action.

   (c) The API converts the sampled activation function to an array of `short int`, and passes it to the driver function `Data_activationTable`, which transmits that data message's opcode followed by the size byte 0xFF.

   (d) The decoder latches the opcode and size, moves to a state for collecting each word of the activation table, and sets its address output to 0x00.

   (e) Once a word is collected, the decoder gives itself control of the data bus, outputs the word, and sets the control code for the target node to `Store Activation`.

(f) The target node latches the activation table value from the data bus and writes it to internal RAM at the line specified by the address bus.

(g) The decoder decrements the stored size, increments the address output, and resumes waiting for the next word.

(h) Once the 256th activation table entry has been stored, the decoder transmits result code 000000 or 000001 and returns to waiting.

(i) The `Command_loadActivationTable` function returns that one byte is read. The decoder converts that byte to an integer, and checks whether it is an error code. Since it is not, it returns without raising an exception.

3. For each hidden and output node, the script passes a sequence giving the weight table value for all 20 possible feeding nodes to the API function `setWeights_Hidden` or `setWeights_Output`. Transmission and storage proceeds in exactly the same fashion as for the activation table, except the message size is 19.

4. The script calls the API function `evaluate`, passing it a sequence of 4 elements giving the input at inputs 0-3.

   (a) The API calls the driver function `Command_evaluate`, which transmits that command's opcode. The target byte's contents do not matter.

   (b) The decoder stores the opcode and target in registers, and responds with result code 001000: ready to receive data. This code is returned to the API, which notes that it is an expected result code, and therefore takes no action.

   (c) The API converts the inputs to an array of `short int`, and passes it to the driver function `Data_evaluationInput`, which transmits that data message's opcode followed by the size byte 0x03.

   (d) The decoder latches the opcode and size, and moves to a state for collecting each word of input.

   (e) Once a word is collected, the decoder gives itself control of the data bus, outputs the word, and raises the enable line to the input node corresponding to the word's position in the message.

   (f) The input node latches the value in its register.

   (g) The decoder decrements the stored size and resumes waiting for the next word.

   (h) Once all the inputs have been latched, the decoder gives the bus to the first input node, and sets the control codes for all hidden and output nodes to `Accumulate`.

   (i) After sufficient delay to allow accumulation, the decoder gives the bus to the each subsequent node and allows all the fed nodes to accumulate again.

   (j) Now the first hidden node is fully fed. The decoder sets its control code to `Activate` and gives it control of the bus.

   (k) Once the hidden node has had time to activate, the decoder sets the control code for all other hidden and output nodes to `Accumulate` once again.

   (l) Now the second hidden node is fully fed. It activates; all other nodes accumulate; then the third hidden node activates, and so on, until the output node itself activates.

   (m) The decoder gives the bus to the output node, latches the output, and transmits the word back to the host.

   (n) The `Command_evaluate` function returns that two bytes are read, as expected. The API converts the byte array giving the response to an array of `short int`, and thence to a sequence of floating-point numbers that is returned to the user's script.