Engineering 90 Project Proposal
Computing Hardware for Accelerated Training of Cascade-Correlation Neural Networks

David German '08

December 4, 2007

1 Introduction

This document proposes an Engineering 90 Senior Design Project. In this project, a hardware implementation of Fahlman's cascade-correlation algorithm for training an artificial neural network [9] will be produced, and device drivers enabling a Linux host PC to interface with the hardware will be developed. User-level libraries providing an abstract interface to the hardware for C and Python programmers will also be written. Prototype hardware will be produced on a field–programmable gate array (FPGA) using the Verilog hardware description language (HDL). If the prototype is finished ahead of schedule, VLSI layout generated from the Verilog code will be optimized for maximum network size and performance, and prepared for fabrication as an application-specific integrated circuit (ASIC).

2 Technical Discussion

2.1 Neural Networks

This section presents a conceptual overview of neural network theory that is common knowledge in the field of artificial intelligence. Artificial intelligence (AI) is a class of technologies that allow computers to learn by experience and pursue objectives in ways not specifically prescribed or anticipated by the programmer. This section does not present the mathematics in detail, since such discussion is readily found in reference material [5].

An artificial neural network is a function approximator composed of nodes that mimic the function of biological neurons. Each node receives some input stimulus, p_{net}, and generates an output stimulus determined by its *activation function*, $f_a(p_{net})$. The input stimulus is the weighted sum of the output of nodes that *feed forward* into it. If the function to be approximated has m inputs and n outputs, the neural network will have m special input nodes that output an element of the input vector, and n special output nodes that do not feed forward to any nodes. Nodes that are neither input nor output are *hidden*.

The process of adjusting a neural network to approximate the desired function is called *training*. Training requires a set of tuples $(\vec{i}, \vec{o})$ that the researcher samples from the function to be approximated. Input $\vec{i}$ is applied to the network as an input, the output is computed, and the error between the output and $\vec{o}$ is used to modify the feed–forward weights by gradient descent. This process is repeated with each tuple in the training set until the change in the network falls below an arbitrary threshold. Each cycle through the training set is called an *epoch*. Under some models, the network may incrementally train and grow. If the network has learned the function of interest well, the application of an input that was not in the training set will yield a good approximation of the desired output. Because the function is entirely learned by example, it need not have a convenient mathematical form; a neural network could, for instance, be trained to output a 1 for inputs representing images that contain a pizza, and a 0 for all other inputs. Continuous output from 0 to 1 could reflect the network's level of certainty in the classification.

2.2 The Cascade–Correlation Algorithm

2.2.1 Overview

In 1991, Fahlman and Lebiere proposed the cascade–correlation algorithm for training a neural network [9]. They give a thorough explanation of the motivation and mathematical details, but the summary here will suffice as background for a reader of this proposal.

In cascade–correlation learning, 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.

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.

2.2.2 Example

i_0	i_1	o
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: The two–input XOR.

Consider the two–input, one–output XOR function shown in Table 1. Training a neural network to implement XOR would have no practical value, since the function has a known closed form, but it will serve as a simple example. Let the training set consist of all four input–output tuples. Training would proceed as follows. In the initial output training, output weights would be adjusted to minimize error on the inputs alone. This has two possible stable outcomes, in which the network learns high weight for one input relative to the other. Which input is favored will be determined by the random initial weights; without loss of generality, suppose i_0 is favored. The network will then correctly produce a value near 0 for inputs (0, 0), and will correctly produce a value near 1 for (1, 0). However, it will incorrectly output near 0 for input (0, 1) and near 1 for input (1, 1). Qualitatively, the configuration is detecting the parity condition where i_1 is fixed low. Error is 50%.

Assuming such large error does not meet the arbitrary threshold, training will proceed to a candidate node phase. Recall that candidate nodes search for high–magnitude correlation with the remaining error. Since there is effectively no remaining error for training inputs (0, 0) and (1, 0), these elements of the training set will have minimal effect on the candidate nodes. The candidates will therefore learn to produce a high output for input (0, 1) and a low output for input (1, 1) by assigning large negative weight to input i_0 and smaller positive weight to input i_1. The inverse weight assignment is also possible, since the sign of the correlation is unimportant. One of the randomly–initialized candidates will achieve the highest correlation and be installed as a hidden node, with its new input weights frozen. This node is detecting the parity condition where i_1 is fixed high.

Another round of output weight retraining on the input nodes and the new hidden node will then begin. The equilibrium output weights will average the information about parity given high i_1 that comes from the hidden node with the information about parity given low i_1 that can be computed directly from the input nodes. The resulting configuration will have sufficient information to produce the correct output on all four training cases, so training will cease.

2.2.3 Advantages of Specialized Hardware

General-purpose microprocessors are designed to perform computations in series. In the past few years, CPUs with two or four parallel execution cores have been commercialized, but even so the bulk of their computational power rests in completing each operation quickly, not in completing many operations at once. This is appropriate for most software applications, in which an input to one instruction is very often the output of its predecessor. Massively parallel execution of such code is impractical: the overhead cost of hardware to check for dependencies and schedule parallel execution correctly is prohibitive.

The training of a cascade–correlation neural network, however, involves many computations that are guaranteed to be free of dependency. Computing the input stimulus to node j requires a multiply–accumulate operation; whenever a node generates an output, every node fed by it can multiply that output by a weight and accumulate it into the net stimulus simulataneously. All candidate nodes are completely dependency–free, since their outputs do not feed any other node, so they may calculate activation and train input weights fully in parallel. Similarly, all output nodes may retrain their weights for each hidden and input connection simultaneously, since final output, by definition, does not feed forward.

Let n be the number of nodes in the network, d be the number of hidden layers in the network, c be the number of candidate nodes, o be the number of output nodes, and t be the number of items in a training set. Table 2 presents the big-O time bound for each network operation under series and parallel operation. The gains to parallelism in evaluation are slight, since d grows as n in casacade–correlation. In training, however, the gains are substantial: time to train becomes *independent of network size*. Of course, these gains are only attainable for network sizes below the maximum capacity of the hardware.

Operation	Series	Parallel
Evaluation	$O(n)$	$O(d)$
Candidate Training	$O(ct)$	$O(t)$
Output Weight Training	$O(ot)$	$O(t)$

Table 2: Big-O time for cascade–correlation algorithm operations.

2.3 Previous Work

Hoehfeld and Fahlman showed in 1992 that cascade–correlation with fixed–point arithmetic finds minimum–depth solutions consistently with 13 or more bits of precision [11]. At moderately lower precision, the algorithm degrades gracefully, compensating for the loss by increasing network depth to meet a given threshold of error. At precision below 7 bits, however, training typically fails to converge on any error minimum.

Various work has also studied the possibility of decreasing connectivity or network depth in cascade–correlation, and thus increasing gains to parallelism in network evaluation [1][6][13]. One straightforward strategy simply allows the algorithm to install a candidate node as a sibling rather than a successor if doing so reduces error further. Obviously, this strategy at least matches cascade–correlation in network size and error reduction.

Since large VLSI process sizes severely curtailed the digital computing power available in early 1990s, there was interest during that period in analog training of neural networks [4][7][10]. The proposed circuits typically use programmable resistors to select weights; inputs are applied to the programmable resistor containing the corresponding weight, and summed using an op–amp. Unfortunately, the weighting circuit is limited in resolution by the precision of the programmable resistor: the largest resistor must vary by less than half the value of the smallest. In practice, this represents a cap on precision at five or six bits. Analog computing in this fashion clearly cannot compete with the capabilities of digital logic today.

2.4 Features of Proposed Hardware

The existing literature contains theoretical and simulation work relevant to hardware implementations of cascade–correlation, but it appears not to contain any actual attempt at hardware implementation. Therefore, the proposed project is believed to be original research. From the work discussed in Section 2.3, it is clear that a digital system is preferable to an analog one. It is also clear that at least 13 bits of arithmetic precision are desirable. [11] For simplicity, the scope of the project will be constrained to cascade–correlation as originally proposed by Fahlman [9], even though there are potential gains to pruning connectivity or allowing sibling nodes [1][6][13]. The following list formally specifies the proposed features.

- The device will implement the cascade–correlation algorithm. The implementation will
  - Support at least four input nodes, at least one output node, and at least sixteen hidden nodes.
  - Automatically evaluate the output that a network generates in response to a given input.
  - Automatically train a new network node in response to a set of input-output tuples.
  - Perform fixed–point arithmetic with at least 16–bit precision.
  - Parallelize both the training of candidate nodes and the retraining of output weights.
- The device will interface with a Linux host PC, and provide libraries that allow a C or Python programmer to conveniently
  - Specify an activation function for each node.
  - Switch between training and evaluation modes.
  - Supply an input in evaluation mode and receive an output.
  - Supply training tuples and execute a training epoch.
  - Inspect the weight of each network connection.
  - Inspect the correlation of candidate nodes with error.

Due to budget constraints, the device will be implemented on a low–cost FPGA, as will be discussed in Sections 3.1 and 5.2.2. Therefore, the neural networks it trains will contain a fairly small number of nodes, and it is not expected to outperform a contemporary PC on a comparable cascade–correlation training task. It is, however, expected to outperform a PC operating at comparable clockspeed on a comparable task. It will thus show that a similar design implemented on a faster, larger, more expensive FPGA or by fabrication as an ASIC would provide superior performance to a PC.

Even given the economical choice of FPGA, the network size specified is somewhat conservative. In particular, it would be desirable to implement more than one output node so that the network can undertake classification tasks with more than two possible values. Section 3.14 discusses a planned project activity to consider expansion of the network beyond the proposed minimum.

3 Project Plan

A Gantt chart presenting the plan for project completion by the customer's deadlines is appended. The tasks shown on the chart are discussed in this section. The chart anticipates 40 hours per week of work on the project from January 22 to May 8, 2008, excluding the spring vacation scheduled from March 7 to March 17. Bold arrows indicate the critical path.

3.1 Justify Component Purchases

Objective Secure funding in excess of the $200 nominal budget.

Approach Present a bill of materials to advisor, and to department chair where applicable. Articulate necessity of materials for project, particularly an Altera Development and Education (DE2) rapid prototyping board based on a Cyclone II EP2C35 FPGA. Identify benefits to customer of owning these materials after the completion of the project.

Output Approved funding for entire bill of materials.

3.2 Order and Receive Components

Objective Procure components required to begin work on project.

Approach Following customer processes, place orders with vendors.

Output Vendor components delivered.

3.3 Specify Host-Peripheral Interface

Objective Define protocol for communication between the PC host and the device.

Approach Select a peripheral interface technology. Enumerate the commands the device must accept. Enumerate the data the host must provide the device. Enumerate the data the device must send to the host. Determine encodings for commands and data. Document the protocol decisions.

Output An interface control document fully specifying the protocol.

3.4 Implement Device Control Unit

Objective Provide hardware or firmware support in the device for interfacing with the host.

Approach A Use the Verilog language to specify a control module that

- Manages the communications interface.
- Accepts input from the communications interface and translates it to internal signals.
- Accepts input from internal signals and translates it to output on the communications interface.

Validate this module in simulation by applying representative inputs on the communications and internal interfaces under both nominal and error conditions.

Approach B Use the C programming language to write a microprocessor program that

- Manages the communications interface.
- Accepts input from the communications interface and translates it to signals to the FPGA.
- Accepts input from the FPGA and translates it to output on the communications interface.

Validate this module in simulation by applying representative inputs on the communications and internal interfaces under both nominal and error conditions.

Output Verilog or C code specifying the control unit. An informal document summarizing the internal signals behavior expected by the control unit. A document containing test results that prove the module works as required by the interface.

3.5 Learn to Use Board Programming Apparatus

Objective Learn to program an Altera Cyclone II EP2C35 FPGA and Nios–II embedded processor.

Approach Install the Altera tools on the project workstation. Specify a simple digital logic function in Verilog. Program the FPGA with this function, mapping the input and output to pins that are physically accessible on the board. Test the function by applying logic voltages to the input pins and observing the output voltage. Write a simple test program in C for the Nios–II microprocessor that applies a square wave to an output pin. Connect an LED to that pin to observe the output.

Output None.

3.6 Implement Host Drivers

Objective Enable a Linux host PC to communicate with the hardware.

Approach Use the C programming language to write a Linux kernel module that

- Provides host support for issuing the device commands specified in the interface control document.
- Manages the data transactions specified in the interface control document.
- Allows userspace programs to interact with the device.

Compile the module and install it in the project workstation kernel. Validate behavior by connecting the driver to a virtual device. The virtual device shall be scripted to present representative inputs for both nominal and error conditions to the driver. The script input and driver output shall be captured to a text file.

Output Source code for a Linux kernel module providing device support. Annotated data from a virtual test run illustrating correct driver behavior.

3.7 Implement Host Libraries

Objective Provide high–level programming interfaces to the device.

Approach Enumerate the high–level tasks for which artificial intelligence researchers will use the device. Codify this enumeration in a library interface specification that will be immediately comprehensible to an AI expert unfamiliar with the device. Write a C library that satisfies this specification by combining sequences of system calls to the driver. Use the Python API to make this C library available as a Python class. Validate the libraries in test runs showing by printed messages that the library functions make appropriate system calls.

Output Source code for C and Python libraries. Documentation of the functions provided by the libraries. Test output illustrating correct library behavior.

3.8 Implement Input Nodes

Objective Provide hardware implementation of an input node to a cascade–correlation neural network.

Approach Use Verilog to specify a module that

- Captures an input activation on a control bus when signaled by the control unit to do so.
- Broadcasts its activation on an internal bus when signaled by the control unit to do so.

Validate the module in simulation by applying representative nominal and erroneous control signals.

Output Verilog code specifying the input node module. Simulation results documenting correct behavior.

3.9 Implement Hidden Nodes

Objective Provide hardware implementation of a hidden node in a cascade–correlation network.

Approach Use Verilog to specify a module that

- Stores input weights for each node that feeds into it.
- Captures input activations on an internal bus when signaled by the control unit to do so.
- Multiplies input activations by the corresponding weight and sums to find the net activation.
- Looks up the corresponding output activation in a table.
- Broadcasts the output activation on an internal bus when signaled by the control unit to do so.

Validate the module in simulation by applying representative nominal and erroneous input sequences.

Output Verilog code specifying the hidden node module. Simulation results documenting correct behavior.

3.10 Implement Output Nodes

Objective Provide hardware implementation of an output node in a cascade–correlation network.

Approach Use Verilog to specify a module that

- Stores input weights for all the input and hidden nodes.
- Captures input activations on an internal bus when signaled by the control unit to do so.
- Multiplies input activations by the corresponding weight and sums to find the net activation.
- Looks up the corresponding output activation in a table.
- Broadcasts the output activation to the control unit when signaled to do so.
- Computes and stores weight updates during the output weight retraining phase.

Validate the module in simulation by applying representative nominal and erroneous input sequences.

Output Verilog code specifying the output node module. Simulation results documenting correct behavior.

3.11 Implement Network Evaluation

Objective Synthesize node and control modules into a static network that, given weight values, determines network outputs for a given input.

Approach Use Verilog to specify a system that links input, output, and hidden nodes together in a cascade–correlation architecture and connects all node control inputs to the control unit. Validate the design by applying both nominal and erroneous simulated host PC input to the control unit. Under the nominal conditions, confirm correct performance by comparing results with the results generated by a software implementation of the same network.

Output Verilog code specifying a complete cascade–correlation network evaluator. Simulation results documenting correct behavior.

3.12 Implement Candidate Training

Objective Provide hardware implementation of candidate nodes that can be trained to maximize correlation with remaining error betwen target and network outputs over a training set.

Approach Use Verilog to specify a candidate node module that

- Stores input weights for all the input and hidden nodes.
- Captures input activations on an internal bus when signaled by the control unit to do so.
- Multiplies input activations by the corresponding weight and sums to find the net activation.
- Looks up the corresponding output activation in a table.
- Accumulates the correlation between output activation and target output until the control unit signals the end of the training set.
- Adjusts its input weights by gradient descent to improve the correlation.
- Reports the correlation to the control unit on an internal bus when signaled to do so.

Add to the control unit the ability to initialize candidate nodes, to administer a training epoch supplied by the host to the candidates in parallel, and to copy the parameters of the candidate node that maximizes correlation into a new hidden node. Validate these capabilities in a full-system test by programming the FPGA and establishing actual communication between the hardware and the host PC. Use device drivers and libraries developed to apply training inputs and inspect the candidate unit correlations. Confirm correct performance by comparing results with the results generated by a software implementation of the same network.

Output A hardware cascade–correlation system that can statically evaluate a network and can train candidate nodes. Test logs generated on the host PC that document correct behavior of the system.

3.13 Implement Output Retraining

Objective Provide hardware implementation for training weights in output nodes by single–layer gradient descent.

Approach Add to the control unit the ability to mode output nodes to retraining and administer a training epoch, allowing all output nodes to compute a weight update in parallel. Validate this capability in another full-system test in which the host PC applies training inputs and inspects the updated output weights. Confirm correct perofrmance by comparing results with the results generated by a software implementation of the same network.

Output A hardware cascade–correlation system that can statically evaluate a network and can retrain output weights. Test logs generated on the host PC that document correct behavior of the system.

3.14 Optimize FPGA Resource Utilization

Objective Decrease overall FPGA resource utilization, and if possible increase network size above the minimum specifications.

Approach Manually study the assignment of FPGA resources to Verilog modules generated by the compiler. Inspect this assignment for wasteful allocation of arithmetic, logic, or memory blocks. Refine the Verilog code to improve utilization wherever possible.

Output A more compact, larger scale implementation of the hardware cascade–correlation system.

3.15 Final System Integration and Validation

Objective Provide a complete host–peripheral system that can perform both phases of cascade–correlation training autonomously until the hardware support for hidden nodes is exhausted.

Approach Integrate all the modules and capabilities previously developed. Validate by using high–level libraries on host PC to train networks that approximate a variety of interesting nonlinear functions. Compare results to results of a software cascade correlation system trained on the same data. Compare performance of hardware system to a software cascade correlation system on the standard two–spirals classification problem. Invite students and faculty of the College who pursue AI research to evaluate the usability of the system for their work.

Output A fully–functioning system of parallel computation hardware and supporting PC host software for training and evaluating cascade–correlation neural networks. A document that benchmarks the system on standard tests and presents preliminary observations on the product's marketability.

3.16 Compile System Documentation

Objective Provide technical documentation for the product.

Approach Using standard open–source project documentation utilities, assemble documents generated while implementing host drivers and libraries into an HTML-based API manual. Include links to well–commented sample code. Compile a datasheet containing interface specifications, block diagrams, flowcharts, schematics, and pin mappings that explain the behavior of the hardware. Compose a brief guide to deploying the system.

Output A users' guide including installation instructions and a hypertext API manual. A PDF engineering datasheet summarizing the hardware configuration.

3.17 Prepare Mid-Semester Presentation

Objective Inform the customer of project status as of March 25, 2008.

Approach Prepare a brief presentation. Give an overview of the host–peripheral interface, and the functionality of the completed control unit module, node module, and host drivers. Report the current state of the candidate and output weight training implementations.

Output A well–rehearsed oral presentation using projected visuals where appropriate.

3.18 Prepare Report and Presentation

Objective Inform the customer of project outcome.

Approach Satisfy customer requirement for compliance with ABET Criterion 3 by generating a report that

- Details the mathematical theory underpinning the cascade–correlation algorithm.
- Presents a brief summary of the system design, referring to technical documentation for details.
- Explains the validation and debugging strategy followed.
- Reports areas in which the product does not meet proposal specifications and explains the causes.
- Produces test data that illustrates correct system performance.
- Identifies the potential market for the target, explains possible applications, and discusses preliminary findings on market reaction.
- Notes improvements required for marketability, and estimates costs of improvements.
- Discusses manufacturability and cost issues in both present implementation on an FPGA and future implementation as an ASIC.
- Estimates scale of commerical production and discuss lifecycle environmental effects.

Prepare a brief presentation that will give the customer an overview of the report's contents, and structure the customer's approach to reading the text itself.

Output A text report of approximately 20 pages, and a well–formatted version of the system documentation. A well–rehearsed oral presentation using projected visuals where appropriate.

3.19 Extensions

The additional activities discussed in this section will be undertaken only if the rest of the project is completed ahead of schedule.

3.19.1 Plan ASIC Implementation

Objective Ready device for fabrication as an application–specific integrated circuit.

Approach Identify a fabrication vendor that provides a process meeting the size and performance needs of the design. Compile design to VLSI layout using Mentor Graphics tools and design libraries provided by the vendor. Select packaging and plan pinout. Design a printed circuit board (PCB) for the device, power supply, and host interface connector. Select an appropriate casing for the PCB.

Output A complete proposal for ASIC production of the design.

3.19.2 Optimize VLSI Layout

Objective Improve performance and network size attainable by ASIC implementation of the device.

Approach Modify the compiled VLSI layout for critical–path performance enhancements using non–CMOS logic. Expand the design to include additional nodes, taking full advantage of the area provided by the fabricaton vendor. Verify correctness of modified design. Update interface control protocol, pinouts, and PCB as required by the layout changes.

Output A higher–performance design for ASIC production.

4 Project Qualifications

I am a double major in Engineering and Computer Science with a particular interest in embedded systems and digital hardware. My past and present coursework includes Digital Systems, Computer Architecture, Compiler Design and Construction, Electronic Circuit Applications, VLSI Design, and Operating Systems, a strong academic background for this project. I have also attended Prof. Lisa Meeden's lectures on neural networks in the Artificial Intelligence course. My practical experience with embedded systems includes a three–month internship with The Boeing Company in 2006, where I developed customer–requested modifications to the embedded software used for command and control of the International Space Station.

5 Project Cost

5.1 Labor

I estimate this project will consume 650 hours of my own time, including the approximately 50 hours already invested. I will request about 5 hours of Ed Jaoudi's time in small increments for assistance finding components and deploying tools. Prof. Tali Moreshet has agreed to advise my project. The amount of her time I will request depends largely on the magnitude of unforseen difficulties, but is unlikely to exceed 4% of the time I invest myself. I may occasionally request input of other professors of the College, especially Prof. Erik Cheever, Prof. Lisa Meeden, and Prof. Tia Newhall, in their areas of expertise. This input is not a required resource and may be provided at their discretion and convenience.

5.2 Materials

5.2.1 Facilities

The project will require 24–hour access to the space Prof. Moreshet has agreed to supply in her Hicks 212 lab. The project will also require superuser privileges on the Linux workstation supplied in that space, and the flexibility to reconfigure it, install new software on it, and reboot it without notice.

The project may intermittently take advantage of other computing facilities and lab resources owned by the Department of Engineering, including public area computers running Windows or Mac OS X, software licenses for MATLAB, software licenses for the Mentor Graphics VLSI design tools, and electronics prototyping tools. Usage will be consistent with the privileges normally granted any engineering student.

5.2.2 Components

The project will require the use of standard USB and serial cables for connectivity. The department owns these in substantial numbers, so the costs of temporarily reserving one USB and one serial cable for the project's duration are negligible.

The project's only budget item is the Altera Development and Education (DE2) board, a prototyping board incorporating a Cyclone II EP2C35 FPGA and a Nios-II microprocessor. This board is manufactured by Altera and is available to educational customers for $265. Since this amount exceeds the nominal $200 budget of this project, special authorization from the department will be sought as discussed in section 3.1.

6 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 development of this proposal.

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] 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.

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

[13] 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.

[14] 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.

[15] 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.

#	Title	Expected Start	Expected End	Expected Work
0	**ENGR90**	11/19/07	5/8/08	7.8m
1	Revise project proposal	11/19/07	11/30/07	1d
3	Justify component purchases to department	12/3/07	12/14/07	1d
5	Order and receive components	12/17/07	1/30/08	1d
7	Specify host-peripheral interface	1/22/08	2/5/08	2w
9	Learn to use board programming apparatus	1/31/08	2/1/08	1d
11	Implement device control unit	2/5/08	2/26/08	3w
13	Implement host drivers	2/26/08	3/27/08	3w
15	Implement host libraries	3/27/08	4/10/08	2w
17	Implement input nodes	2/1/08	2/6/08	3d
19	Implement hidden nodes	2/6/08	2/13/08	1w
21	Implement output nodes	2/13/08	2/20/08	1w
23	Implement network evaluation	2/20/08	3/5/08	2w
25	Implement candidate training	3/5/08	4/10/08	3.8w
27	Implement output retraining	3/5/08	4/10/08	3.8w
29	Optimize FPGA resource utilization	4/10/08	4/16/08	4d
31	Compile system documentation	4/16/08	5/8/08	1.6w
33	Final integration and validation	4/16/08	5/2/08	2.4w
35	Prepare draft report	4/11/08	4/18/08	1w
37	Draft report deadline	4/18/08	4/18/08	
39	Finalize report	5/2/08	5/8/08	4d
41	Final Report due	5/8/08	5/8/08	
43	Prepare mid-semester presentation	3/19/08	3/24/08	3d
45	Mid-Semester Presentation	3/24/08	3/25/08	
47	Prepare final presentation	4/30/08	5/5/08	4d
49	Final Presentation	5/5/08	5/5/08