AN ABSTRACT OF THE THESIS OF

Christopher Moore for the degree of Master of Science in Computer Science presented on October 23, 2009.

Title:

A Recurrent Neural Network Implementation Using the Graphics Processing Unit

Abstract approved: _____

Mike Bailey

We took the back-propagation algorithms of Werbos for recurrent and feed-forward neural networks and implemented them on machines with graphics processing units (GPU). The parallelism of these units gave our implementations a 10 to 100 fold increase in speed. For nets with less than 20 neurons the machine performed faster without using the GPU, but for larger nets use of the GPU always gave better times.

A Recurrent Neural Network Implementation Using the Graphics Processing Unit

by

Christopher Moore

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented October 23, 2009
Commencement June 2010

Master of Science thesis of Christopher Moore presented on October 23, 2009.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Christopher Moore, Author

ACKNOWLEDGEMENTS

I would like to thank my committee professors.

TABLE OF CONTENTS

Page

1 Introduction 1

2 Neural Networks 3
- 2.1 Biological Inspiration . . . 3
- 2.2 Mathematical Model . . . 5
- 2.3 Supervised Learning Error Functions . . . 11
- 2.4 Gradient Descent . . . 14
- 2.5 Back Propagation . . . 17
- 2.6 Gradient Descent Example: Logical NOT . . . 20

3 Recurrent Neural Networks 24
- 3.1 Feed Forward . . . 24
- 3.2 Back Propagation . . . 25

4 GPGPU 30
- 4.1 Current GPGPU Implementation Options . . . 30
- 4.2 NVIDIA Cg . . . 30
- 4.3 HLSL . . . 30
- 4.4 GLSL . . . 31
- 4.5 NVIDIA CUDA . . . 32
- 4.6 ATI Brook . . . 32
- 4.7 OpenCL . . . 32

5 Algorithm 33
- 5.1 Choosing GLSL . . . 33
- 5.2 Hardware Constraints . . . 34
- 5.3 Implementation . . . 36
- 5.4 Usage . . . 37
  - 5.4.1 RNN::process . . . 38

TABLE OF CONTENTS (Continued)

Page

5.4.2 RNN::cycleLayers 39
5.5 CPU Implementation 40
5.5.1 CPURNN::feedForward 40
5.5.2 CPURNN::calcDeDy 41
5.5.3 CPURNN::calcError 42
5.5.4 CPURNN::updateHistory 43
5.5.5 CPURNN::updateWeights 48
5.6 GPU Implementation 50
5.7 GPU Ping-Pong Implementation 52
5.7.1 GPURNN::prepare 53
5.7.2 GPURNN::feedForward 53
5.7.3 matVecRowScaleShader 59
5.7.4 matVecColToRowCombineShader 59
5.7.5 GPURNN::calcDeDy 61
5.7.6 calcDeDyShader 63
5.7.7 GPURNN::calcError 63
5.7.8 errorDifShader 64
5.7.9 rowReduce 65
5.7.10 matVecRowReduce16to1Shader 67
5.7.11 GPURNN::updateHistory 69
5.7.12 backpropAccumShader 75
5.7.13 backpropAccumDeltaShader 78
5.7.14 backpropScaleByDerivShader 78
5.7.15 GPURNN::updateWeights 79
5.7.16 updateWeightsAccumShader 83
5.8 GPU With Unravelled Loops 85
5.8.1 GPURNNTimeframe::feedForward 87
5.8.2 GPURNN::calcError 89
5.9 GPU With For Loops 90
5.10 $\frac{\partial E}{\partial net}$ Optimization 91
5.11 Future Optimization 91

TABLE OF CONTENTS (Continued)

	Page
6 Results	93
6.1 Accuracy	93
6.2 Performance	94
6.2.1 Hardware Trends	100
7 Conclusion	102
Bibliography	102

LIST OF FIGURES

Figure		Page
2.1	Example of $tanh(x)$ as an activation function. Note how the signal eventually tapers and levels off as x tends further from zero.	4
2.2	Diagram of a feed-forward calculation for a single-layered network. First input vector elements (x_j) are multiplied by weight coefficients ($w_{i,j}$) and summed into net vector elements (net_i). Next the activation function ($\sigma(x)$) is applied to each net vector element to produce the output vector elements (y_i).	5
2.3	Examples of lines (which are one-dimensional planes) dividing half-spaces of classified and unclassified points for the Logical AND and Logical OR problems. Input values of 0.3 are used to represent "true" while values of -0.3 are used to represent "false." An additional neuron is appended to the input layer: a "bias" signal which perpetually sends a value of 1. This allows for solving non-homogeneous systems and allows the classifying plane to move off of the origin. The transfer function is defined as $\sigma(x) = tanh(x)$. Positive output values denote a "true" output while negative output values denote a "false" output.	7
2.4	Examples of the the Logical XOR problem which cannot be classified by a single half-space. Two half-spaces are required to accurately define the classified and unclassified points for this problem.	8
2.5	An arbitrary possible topology of a neural network	9
2.6	Graphs of the error value (z-axis) vs. the weight of the input, x, vs. the weight of the bias signal for the Logical NOT problem. Left: $x = 0.3, d = -0.3$, Middle: $x = -0.3, d = 0.3$, Right: the average of the two. True boolean values are expressed as signal values of 0.3, while false values are expressed as signal values of -0.3.	13
2.7	Contour lines of the averaged error functions for the Logical NOT problem	14

LIST OF FIGURES (Continued)

Figure		Page
2.8	A visual description for back propagating the $\frac{\partial E}{\partial net_j^i}$ error term.	19
2.9	Graphs of the error value (z-axis) vs. the weight of the input, x, vs. the weight of the bias signal for the Logical NOT problem. Left: $x = 0.3, d = -0.3$, Right: $x = -0.3, d = 0.3$	22
2.10	A linear combination of the two input/desired set's error functions	22
2.11	Log scale plot of the convergence of a feed forward network training to the Logical NOT operation shown above. Error is scaled logarithmically. Fluctuations are due to randomness in the network training information. Near iteration 3300 the error levels off due to it reaching a hardware precision limit.	23
3.1	The top figure shows an example recurrent network. The single red line shows the weight for which the error gradient will be calculated with respect to. Below, the information propagation of the network is divided among various instances in time. The red lines show the dependencies of computation for the gradient with respect to the target weight.	26
3.2	For each weight, a copy of the network is stored. The edges of the copy contain tensors that store history. Unlike back propagation, the history is updated in a feed-forward method. This circumvents limitations to graphs with cycles which would cause gradient equation dependences.	29
5.1	Demonstration of a remedy to supporting vector sizes larger than the limiting size of textures on graphics cards.	35
5.2	Demonstration of a remedy to supporting incoming connections larger than the limiting number of texture units on graphics cards.	36
5.3	A demonstration of how a list of 2D textures contain the rank-3 history gradient tensor.	51

LIST OF FIGURES (Continued)

Figure | Page

5.4 An example of matrix multiplication. (1) First each row is multiplied with the input vector and stored in place. (2) Next the columns are summed together into a single column. (3) Last the remaining column is transposed into a compact representation of the result. 56

5.5 Visual example of the history tensor computation. The above shows the computation of the $\sum_{k,l} w_{j,l}^{i,p_{i,k}}(\tau)\xi_{s,t,l}^{m,p_{m,n},p_{i,k}}(\tau)$ component. 75

5.6 Example of the $\frac{\partial E}{\partial w}$ computation from the $\frac{\partial E}{\partial y}$ vector and the $\frac{\partial y}{\partial w^{m,p_{m,n}}}$ tensor. Each tensor slice is scaled by an associated component of the error vector and summed to produce the change in the $\partial w^{m,p_{m,n}}$ weight. 80

6.1 An example of a sequence for which the CPU and GPU values align closely 94

6.2 An example of a sequence for which the CPU and GPU values eventually diverge 95

6.3 Logarithmic plot of time versus bit size. Performance comparison for an Elman network on an 1.83 GHz Intel dual core with 2GB RAM and an ATI X1600 graphics card. Stair-stepping of performance as bit size increases exponentially is due to the textures being rounded up to the next power of two in size. . . 97

6.4 Logarithmic plot of time versus bit size. Performance comparison for an Elman network on an 3.20 GHz Intel dual core with 2GB RAM and a NVIDIA GeForce 8600 GTS graphics card. . 98

6.5 Logarithmic plot of time versus bit size. Performance comparison for a feedforward network between unoptimized computation and $\frac{\partial E}{\partial net}$ optimized computation on an 1.83 GHz Intel dual core with 2GB RAM and an ATI x1600 graphics card. 99

LIST OF FIGURES (Continued)

Figure		Page
6.6	Logarithmic plot of time versus bit size. Performance comparison for a feedforward network between unoptimized computation and $\frac{\partial E}{\partial net}$ optimized computation on an 3.20 GHz Intel dual core with 2GB RAM and a NVIDIA GeForce 8600 GTS graphics card.	100
6.7	Logarithmic plot of time versus bit size. Comparison between different machines of the GPU ping-pong implementation. . .	101

DEDICATION

To Hien, for dedicating his thesis to his fellow lab dwellers, of which I am one.

Chapter 1 – Introduction

One of the latest trends in commercial computing is a a shift towards parallelism [5]. In place of a single processor executing all facets of one task, a collection of processors will be used to attack individual portions of the task. While parallel programming has been around for several decades now, only recently has it been so available, thanks to commercial GPUs. This rise came about due to the encouragement of the video game industry. It was from this revolution that the first commercial programmable graphics processing unit was designed. It was also the video game industry that drove the capabilities of such chips at a rate far surpassing Moore's Law [13]. Now, several years into this recent boom, a variety of general purpose GPU (GPGPU) algorithms can be found which take advantage of the power of the GPU. The applications of these algorithms include physics simulation [7], bioinformatics [3], and database searches [6], among many others.

Artificial intelligence is among these fields to begin making use of general-purpose GPU programming. Several algorithms in this field have been migrated to GPGPU programming. Feed forward neural networks [15], genetic algorithms [11], and self-organizing maps [18] are among the list.

Another technology that has recently seen a recent increase of interest is neural network implementations. Neural networks were heavily researched during the mid 20'th century set atop discoveries like Hebb's Rule [8]. For a period of time the interest in these algorithms died down until the 80's when Werbos' error back-propagation algorithm for feed-forward neural networks gained attention [16]. Once again in the spotlight, and now with the advantage of better technology, a wide variety of neural network algorithms have since emerged on the scene. A branch of these algorithms includes recurrent neural networks, a category of networks capable of holding a sense of short-term memory. Jordan, Elman, and Juergen are all famous for their contributions in this area [10] [4] [9].

One such recurrent neural network is the Real Time Recurrent Learning (RTRL) neural network[17]. This employs the same algorithm as traditional feed forward models, with an added set of inputs from the prior frame in time. This allows for easier classification of time-dependent patterns, alluding to finite state machine classification [12]. However true gradient calculations of the RTRL's recurrent components turns out to be a computationally costly algorithm. For this reason CPU applications of RTRL networks can be limited by their slow computation time. To alleviate this problem, this paper will explore a GPGPU implementation of a generalization of the Real Time Recurrent Learning neural network algorithm.

Chapter 2 – Neural Networks

The operation of the brain has been pondered by humans for centuries. Humans recognized the brain as the thinking organ as early as the 4th century BC, even to a surprisingly accurate degree. Hippocrates was among the first to recognize the brain as the organ responsible for conscious thought, and to even distinguish between sensory and motor neurons. With further advances in the study of biology, mathematical models were constructed of their behavior. Hebb discovered the change in connectivity with relation to firing rate of neighboring neurons [8]. Further and greater advances in research on the properties of the human mind are being made to this day.

2.1 Biological Inspiration

Brains are made of neuron cell bodies. Neurons collect incoming signals through touching dendrites. They feed those signals forward to other touching neurons through their axion. When a neuron fires, it forwards its electrical signal to neighboring neurons. Those neighboring neurons in turn accumulate charge until they reach a certain threshold. At this point they stop accepting charge from their neighbors and their carried charge levels off. This threshold

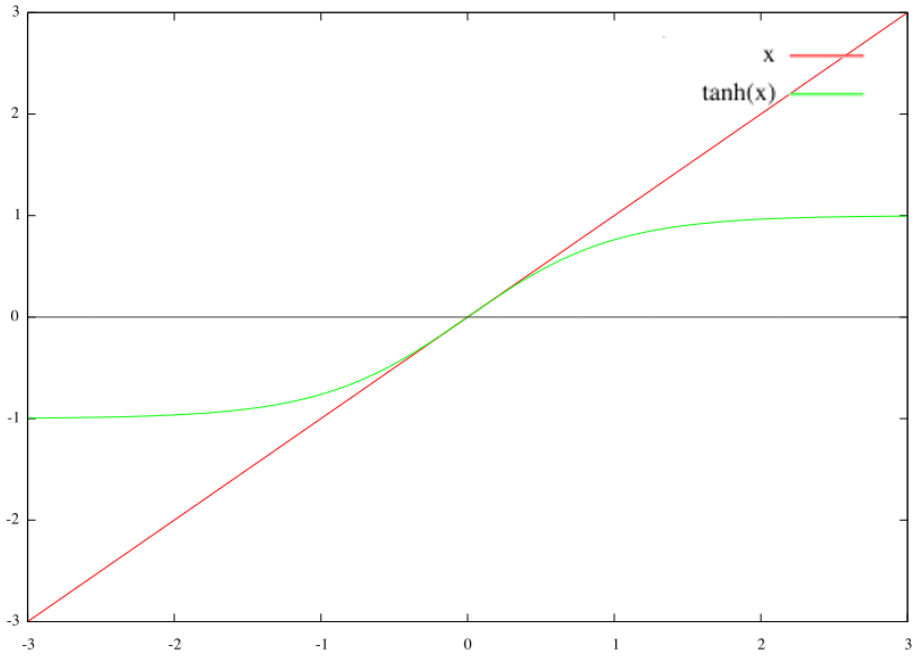

Figure 2.1: Example of $tanh(x)$ as an activation function. Note how the signal eventually tapers and levels off as x tends further from zero.

limit is mathematically represented by an activation function. An example of an activation function can be seen in Figure 2.1.

Neurons accepting charge from their neighbors also forward this charge to their subsequent neighbors. Neurons which cause each other to fire grow stronger connections. Hebb came up with a model on the firing rate of neurons with respect to their distance, and measured the rate at which their distance grew in proximity with relation to their firing [8].

2.2 Mathematical Model

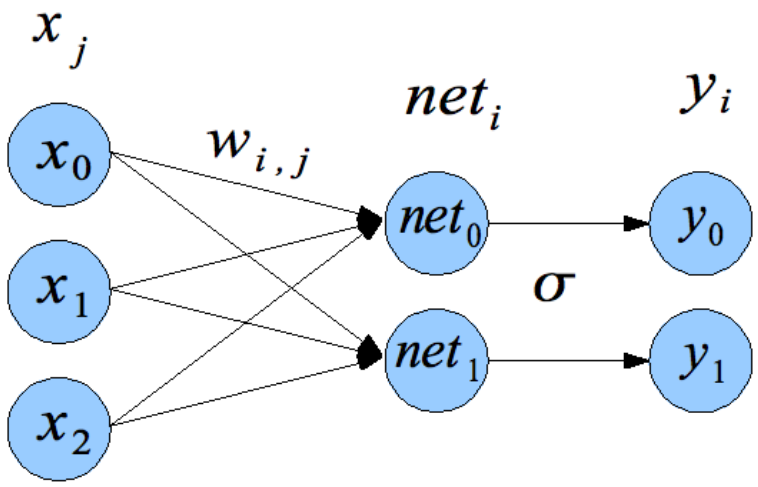

Figure 2.2: Diagram of a feed-forward calculation for a single-layered network. First input vector elements (x_j) are multiplied by weight coefficients ($w_{i,j}$) and summed into net vector elements (net_i). Next the activation function ($\sigma(x)$) is applied to each net vector element to produce the output vector elements (y_i).

A simple, single-layered neural network can be represented with the following components: Let x be the input layer, a vector composed of input signal values x_j. Let w be the weight matrix, a matrix storing the coefficients of relation between the components of the input and output vectors. Let net be the vector of the net sums of linear components of the input layer vector

and the weight coefficients, such that $net_i = \sum_j w_{i,j}$ This is analogous to the biological model of input neurons sending signals through their axions to be accumulated in the output layer's neurons. Let y be the output layer, a vector composed of output signal values y_i. Let the activation function of the neuron be represented as $\sigma(x)$. The value of the output signal after its transfer through the activation function is calculated as $y_i = \sigma(net_i)$. The process can be seen visually in Figure 2.2.

Popular implementations of $\sigma(x)$ are the function of $\frac{1}{1+e^{-x}}$, e^{-x^2}, or $tanh(x)$. These are chosen because, past some critical point, the function asymptotes at a fixed value. Using a boolean analogy for the neuron signal, signals are usually classified as being high or low depending on whether they are closer to the supremum or infemum of the range of the activation function.

The model of the single-layer neural network is as follows:

$$net_i = \sum_j w_{i,j} x_j \tag{2.1}$$

$$y_i = \sigma(net_i) \tag{2.2}$$

This works well for simple linear classification problems, such as the Logical AND and Logical OR problems (Figure 2.3). Each row of the weight matrix represents a distinct plane in the output space's dimension. These planes'

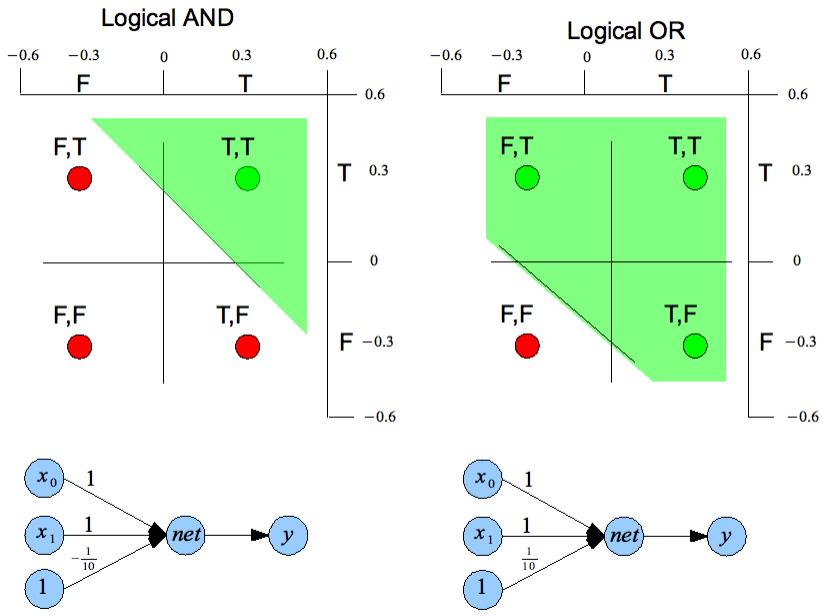

Figure 2.3: Examples of lines (which are one-dimensional planes) dividing half-spaces of classified and unclassified points for the Logical AND and Logical OR problems. Input values of 0.3 are used to represent "true" while values of -0.3 are used to represent "false." An additional neuron is appended to the input layer: a "bias" signal which perpetually sends a value of 1. This allows for solving non-homogeneous systems and allows the classifying plane to move off of the origin. The transfer function is defined as $\sigma(x) = tanh(x)$. Positive output values denote a "true" output while negative output values denote a "false" output.

half-spaces eventually adjust to a linear classification of data. These planes can be fittingly used to represent simple logical operations. However classified data can only fall in a certain half-space. Certain logical problems, such as the Logical XOR problem, cannot be classified by a single half-space and therefore

cannot be learned by a single-layer neural network (Figure 2.4).

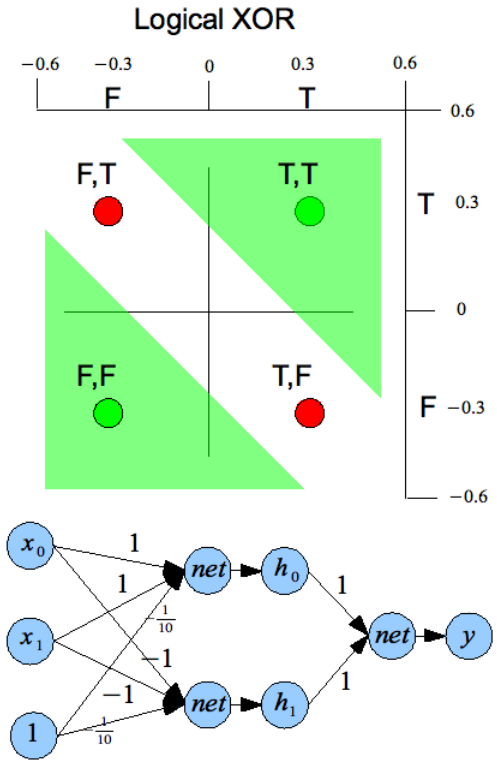

Figure 2.4: Examples of the the Logical XOR problem which cannot be classified by a single half-space. Two half-spaces are required to accurately define the classified and unclassified points for this problem.

To represent more complex classifications than half-spaces, we can take the signals of the output layer and treat them as inputs into a new single-layer network. Let a^i be the i'th layer of neurons in our neural network. Let a^i_j be

the individual elements of a^i. Let $w^{i,k}$ be the matrix of the linear coefficients of influence from layer a^k to layer net^i. Let $w^{i,k}_{j,l}$ be the coefficient that a^k_l influences net^i_j.

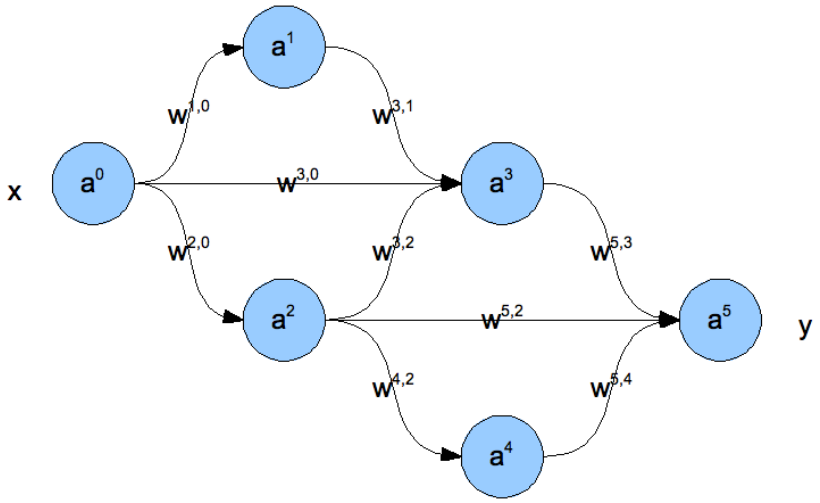

Figure 2.5: An arbitrary possible topology of a neural network

To preserve acyclicity of the connetivity, let p_i represent the set of indexes of vectors that feed into the i'th vector. We have to enforce the rule $\forall i, k, p_{i,k} < i$ to ensure no cycles will exist in our model. Enforcing this rule makes computation easier and also comes in handy when differentiating the variables. We will get to differentiation in the upcoming section on gradient descent. Our model can now be written as follows:

$$net_j^i = \sum_k \sum_l w_{j,l}^{i,p_{i,k}} a_l^{p_{i,k}} \tag{2.3}$$

$$a_j^i = \sigma(net_j^i) \tag{2.4}$$

Forward-propagating the signals through a network of n layers can be performed in $O(n)$ time. This can be achieved due to the acyclic connection constraint imposed on feed-forward networks – specifically – that $\forall i, k, p_{i,k} < i$. When calculating the i'th layer this constraint assures us that all previous layers l such that $l < i$ have already been calculated, therefore all input layers $a^{p_{i,k}}$ have already been calculated. The order of calculations can be performed as follows:

$$a_j^0 = x_j \tag{2.5}$$

$$net_j^1 = \sum_k \sum_l w_{j,l}^{1,p_{1,k}} a_l^{p_{i,k}} \tag{2.6}$$

$$a_j^1 = \sigma(net_j^1) \tag{2.7}$$

$$net_j^2 = \sum_k \sum_l w_{j,l}^{2,p_{1,k}} a_l^{p_{i,k}} \tag{2.8}$$

$$a_j^2 = \sigma(net_j^2) \tag{2.9}$$

$$\cdots$$

$$net_j^y = \sum_k \sum_l w_{j,l}^{y,p_{y,k}} a_l^{p_{i,k}} \tag{2.10}$$

$$y_j = a_j^y = \sigma(net_j^y) \tag{2.11}$$

The output of the final layer now holds the output of the network, y.

2.3 Supervised Learning Error Functions

We have just covered the task of calculating the output to a network for a given input and a given state of the weight matrices. A neural network will only accurately classify a function if the state of its weight matrices are properly set to represent that function. This brings us to the issue of how to correctly set the weight matrix values. There are several algorithms which can be used to set the weights of a neural network to represent a desired function.

One of the more popular methods is by supervised learning. Supervised learning involves training the neural network using a subset of correct outputs for specific inputs. The neural network then extrapolates a means to classify vectors of input signals that it had not yet been presented with.

A single iteration of training the network operates via the following steps: First the an arbitrary pair of inputs and associated known correct outputs are chosen. The inputs are copied into the neural network's input layer. The signals are then fed forward through the neural network to calculate the network's output. The network output values are then compared to the correct

output values associated with the inputs that the network was provided with. The vector of the difference between the produced output and desired output is then used to adjust the weights.

Let x be the vector of input signals into the neural network. Let d be the vector of known correct outputs. This is also known as the network's desired outputs. Let y be the vector of output signals produced by the network from forward propagating the input vector x.

An error function, E, is created and defined as the square distance between the vectors of the network's outputs and desired outputs. This function has a global minima at the point at which the desired output and network output are equal. The function is concave on either side. A scalar of $\frac{1}{2}$ is multiplied to the function so that the coefficient from the squared terms will be cancelled out when the function is differentiated. The function is described as follows:

$$E = \frac{1}{2} \sum_{i} (y_i - d_i)^2 \tag{2.12}$$

Note that the error function is dependent on the network output and the desired output. Recall that to train a network we act on sets of inputs and associated known outputs, and that a different set of input and known output is chosen for each training iteration. This causes the state of the weights of

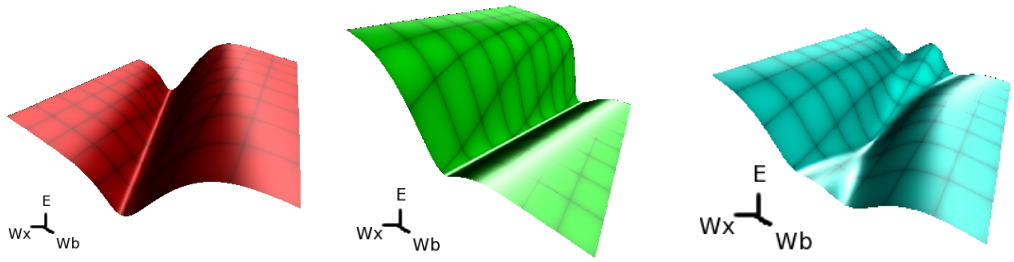

Figure 2.6: Graphs of the error value (z-axis) vs. the weight of the input, x, vs. the weight of the bias signal for the Logical NOT problem. Left: $x = 0.3, d = -0.3$, Middle: $x = -0.3, d = 0.3$, Right: the average of the two. True boolean values are expressed as signal values of 0.3, while false values are expressed as signal values of -0.3.

the network to converge along the average of each error function associated with each desired output (Figure 2.6). The end result is a classification that works approximately well for what exposure it had of the data that it trained to. Neural networks are not meant to recreate exact models of the system they observe.

The weights are adjusted by the gradient of the error function with respect to the weights. Each iteration the negative of the gradient is added to the weights in order to minimize the error. The weights are subsequently adjusted closer to the minima of the error function equation (Figure 2.7). This can be represented with the following equation:

$$\frac{\partial w_{s,t}^{m,p_{m,n}}}{\partial t} = -\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.13}$$

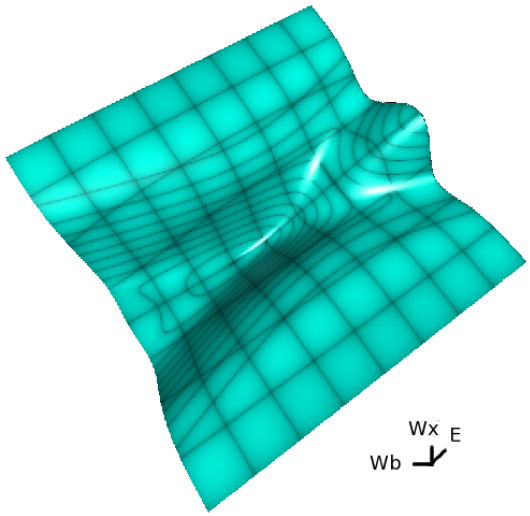

Figure 2.7: Contour lines of the averaged error functions for the Logical NOT problem

2.4 Gradient Descent

To find the value that each weight is adjusted with regard to the error function we must compute the partial term of the error with respect to each weight. Calculating the error gradient can first be broken down in the following terms:

$$\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.14}$$

First we will focus on the partial term $\frac{\partial E}{\partial y}$.

$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_k (y_k - d_k)^2\right] \tag{2.15}$$

$$\frac{\partial E}{\partial y_j} = \frac{1}{2}\sum_k \frac{\partial}{\partial y_j}\left[(y_k - d_k)^2\right] \tag{2.16}$$

$$\frac{\partial E}{\partial y_j} = \sum_k \left((y_k - d_k) \cdot \left(-\frac{\partial y_j}{\partial y_k}\right)\right) \tag{2.17}$$

$$\frac{\partial E}{\partial y_j} = \sum_k \left((y_k - d_k) \cdot -\delta_{j,k}\right) \tag{2.18}$$

$$\frac{\partial E}{\partial y_j} = d_j - y_j \tag{2.19}$$

Now we know $\frac{\partial E}{\partial y_j}$. We need only calculate $\frac{\partial y_j}{\partial w_{s,t}^{m,p_{m,n}}}$ and multiply the two to find the value of $\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}}$. To find this we can once again break $\frac{\partial a_j^y}{\partial w_{s,t}^{m,p_{m,n}}}$ into separate components via the chain rule. We will define this relation in terms of a_j^i due to the fact that the equation applies the output vectors of all layers in the network.

$$\frac{\partial a_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial a_j^i}{\partial net_j^i} \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.20}$$

Note that the indices of $\frac{\partial a_j^i}{\partial net_j^i}$ match. This is due to an implementation constraint that activation functions operate per-element on the vectors they are applied to. All derivatives $\frac{\partial a_j^i}{\partial net_{j'}^i} = 0$ for $j \neq j'$. For this reason the $j \neq j'$ indices are omitted, and j' is replaced by j.

Differentiating the activation function with respect to the net input of the linear combination gives us the following:

$$\frac{\partial a_j^i}{\partial net_j^i} = \sigma'(net_j^i) \tag{2.21}$$

Differentiating $\frac{\partial net}{\partial w}$ is calculated as follows:

$$net_j^i = \sum_k \sum_l w_{j,l}^{i,p_{i,k}} a_l^{p_{i,k}} \tag{2.22}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial}{\partial w_{s,t}^{m,p_{m,n}}} \left[\sum_k \sum_l w_{j,l}^{i,p_{i,k}} a_l^{p_{i,k}} \right] \tag{2.23}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \sum_k \sum_l \frac{\partial}{\partial w_{s,t}^{m,p_{m,n}}} \left[w_{j,l}^{i,p_{i,k}} a_l^{p_{i,k}} \right] \tag{2.24}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \sum_k \sum_l \left(\frac{\partial w_{j,l}^{i,p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} a_l^{p_{i,k}} + w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \right) \tag{2.25}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \sum_k \sum_l \left(a_l^{p_{i,k}} \delta_{i,m} \delta_{p_{i,k},p_{m,n}} \delta_{j,s} \delta_{l,t} + w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \right) \tag{2.26}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \sum_k \left(a_t^{p_{i,k}} \delta_{i,m} \delta_{p_{i,k},p_{m,n}} \delta_{j,s} + \sum_l \left(w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \right) \right) \tag{2.27}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = a_t^{p_{m,n}} \delta_{i,m} \delta_{j,s} + \sum_k \sum_l \left(w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \right) \tag{2.28}$$

If the weight which we are differentiating with respect to lies between layers a^i and $a^{p_{i,k}}$ then we can use the network topology's constraint of acyclicity to

assert that it will not appear in any previous connections' derivatives. The $\frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}}$ relation is zero. We can also assert that $i = m$, and therefore do not need to sum across p_i but rather only need to compute the gradient with respect to $p_{m,n}$. The equations simplify to the following:

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = a_t^{p_{m,n}} \delta_{i,m} \delta_{j,s} + \sum_k \sum_l \left(w_{j,l}^{i,p_{i,k}} \cdot 0 \right) \tag{2.29}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = a_t^{p_{m,n}} \delta_{i,m} \delta_{j,s} \tag{2.30}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{p_{i,n}}} = a_t^{p_{i,n}} \delta_{j,s} \tag{2.31}$$

If the weight we are differentiating with respect to does not lie between layers a^i and $a^{p_{i,k}}$ then the value of $\delta_{i,m}$ is zero. We are then left with a product in terms of the previous layer's gradient, $\frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}}$:

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = a_t^{p_{m,n}} \cdot 0 \cdot \delta_{j,s} + \sum_k \sum_l \left(w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \right) \tag{2.32}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} = \sum_k \sum_l w_{j,l}^{i,p_{i,k}} \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.33}$$

2.5 Back Propagation

Regardless of what calculation is used to compute $\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}}$, the $\frac{\partial E}{\partial a_j^i}$ and $\frac{\partial i_j}{\partial net_j^i}$ terms used to compute the error function will still remain the same. Combined

they form the $\frac{\partial E}{\partial net_j^i}$ term of the gradient (Eqn. 2.34). Due to the fact that several weights can feed into the same layer, it is possible that several weight updates will use the same $\frac{\partial E}{\partial net_j^i}$ calculation. For this reason it is useful to store this term of each layer's gradient calculations.

$$\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial E}{\partial a_j^i} \cdot \frac{\partial a_j^i}{\partial net_j^i} \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.34}$$

$$\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial E}{\partial net_j^i} \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.35}$$

Solving for $\frac{\partial E}{\partial net_j^i}$ in terms of the successive layer's networks yields Eqn. 2.36. A graphical depiction of the computations behind this process can be seen in Figure 2.8.

$$\frac{\partial E}{\partial net_l^k} = \frac{\partial a_l^k}{\partial net_l^k} \cdot \frac{\partial E}{\partial a_l^k} \tag{2.36}$$

$$\frac{\partial E}{\partial net_l^k} = \frac{\partial a_l^k}{\partial net_l^k} \cdot \sum_{i \in outputs(k)} \sum_j w_{j,l}^{i,k} \frac{\partial E}{\partial net_j^i} \tag{2.37}$$

Both the $\frac{\partial E}{\partial net^i}$ vector can be calculated and the weight can be adjusted in the same pass. Due to the acyclicity constraint imposed on $p_{i,k}$ all these calculations can be performed in linear time with respect to the number of layers provided we begin by calculating the last layer's values and finish calculating the first layer's values – in reverse-order that the feed-forward equations were

computed.

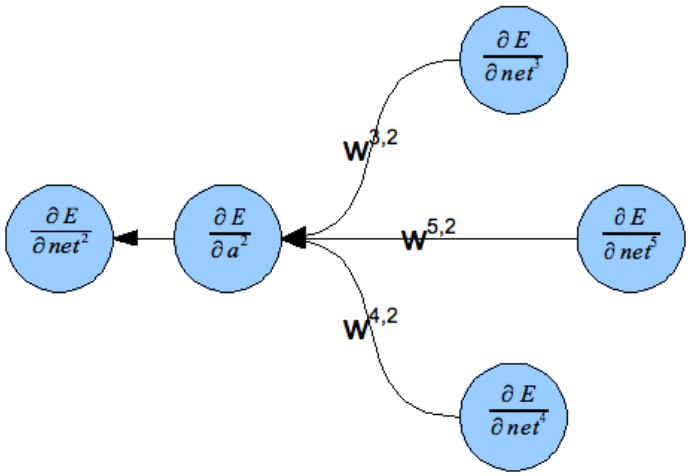

Figure 2.8: A visual description for back propagating the $\frac{\partial E}{\partial net_j^i}$ error term.

Combining the two products of the chain rule, $\frac{\partial E}{\partial net_j^i}$ and $\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}}$, produces the desired gradient of $\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}}$. If we substitute our prior calculations we find the following:

$$\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} = \frac{\partial E}{\partial net_j^i} \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \tag{2.38}$$

$$\frac{\partial E}{\partial w_{s,t}^{m,p_{m,n}}} = a_t^{p_{m,n}} \delta_{i,m} \delta_{j,s} \cdot \frac{\partial E}{\partial net_j^i} \tag{2.39}$$

$$\frac{\partial E}{\partial w_{j,t}^{i,p_{i,n}}} = a_t^{p_{i,n}} \cdot \frac{\partial E}{\partial net_j^i} \tag{2.40}$$

Renaming the $p_{i,n}$ index to k and the t index to l gives us the following:

$$\frac{\partial E}{\partial w_{j,l}^{i,k}} = a_l^k \cdot \frac{\partial E}{\partial net_j^i} \tag{2.41}$$

We now update the weights using a Euler integration method:

$$\frac{\partial w_{j,l}^{i,k}}{\partial t} = -\frac{\partial E}{\partial w_{j,l}^{i,k}} \tag{2.42}$$

$$\frac{\partial w_{j,l}^{i,k}}{\partial t} = -a_l^k \cdot \frac{\partial E}{\partial net_j^i} \tag{2.43}$$

2.6 Gradient Descent Example: Logical NOT

To provide a simple example, we will consider the convergence of weights trained to the logical function $y = NOT\ x$. For our σ activation function, we will use $tanh(x)$. When providing boolean inputs for training data to our neural network our values corresponding with a "true" will be encoded as 0.3. False will be -0.3. I chose this value because it is set between the midpoint and supremum of the range of $tanh(x)$. It also exists at a point where the derivative of $tanh(x)$ is still near its peak magnitude. The encoding of the outputs will be as follows: if the output signal is greater than zero it will be considered "true." If it is less than zero then it will be considered "false."

The symbols are as follows: y is the output corresponding to a_0^1, x the input corresponding to a_0^0. The net accumulation of the input layers is net and the weight of the input's contribution to the net is $w_{0,0}^{1,0}$. The bias' linear influence to net is given by $w_{0,1}^{1,0}$. Finally, d is the desired signal of the system.

$$net = x \cdot w_{0,0}^{1,0} + w_{0,1}^{1,0} \tag{2.44}$$

$$y = \sigma(net) = tanh(net) \tag{2.45}$$

$$E = \frac{1}{2}(d - y)^2 \tag{2.46}$$

Expanding the error function in terms of the varying two weights gives us the following:

$$E = \frac{1}{2}\left(d - tanh\left(x \cdot w_{0,0}^{1,0} + w_{0,1}^{1,0}\right)\right)^2 \tag{2.47}$$

The possible inputs and desired outputs for training this system are $d = -0.3$ when $x = 0.3$, corresponding to the expression "not true = false," and $d = 0.3$ when $x = -0.3$, corresponding to the expression "not false = true." These can be seen in Figure 2.9.

Since the differentiation is a linear operation we know it is distributive over addition. From there we can linearly combine the two error functions to see what basins the system will converge to. Figure 2.10 demonstrates this result-

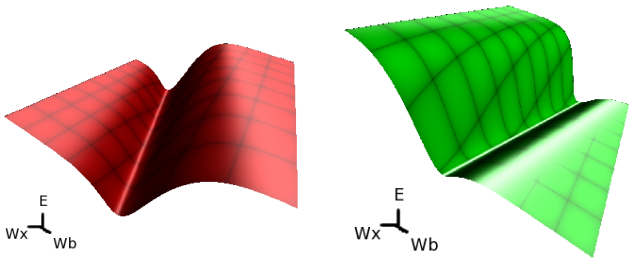

Figure 2.9: Graphs of the error value (z-axis) vs. the weight of the input, x, vs. the weight of the bias signal for the Logical NOT problem. Left: $x = 0.3, d = -0.3$, Right: $x = -0.3, d = 0.3$

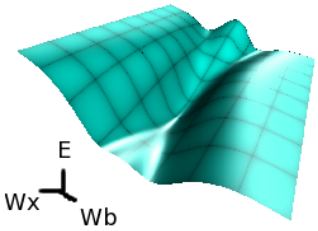

Figure 2.10: A linear combination of the two input/desired set's error functions

ing combination.

Once the weights reach that minimum the neural network is then capable of providing a numeric analogy to the Logical NOT problem. Figure 2.11 shows an example of the error level of a neural network converging on the Logical NOT problem. Note that as the network converges there is still some fluctuation in the descent. This is due to the fact that each iteration which the network is trained, it is trained to a potentially different set of data. The subtle errors of a converged network show the weights slightly changing as the

network tries to negotiate itself between all trained solutions, as is shown in Figure 2.10. Also note that at one point the graph immediately levels off. This is due to the network reaching the precision limit of the hardware which it is running on.

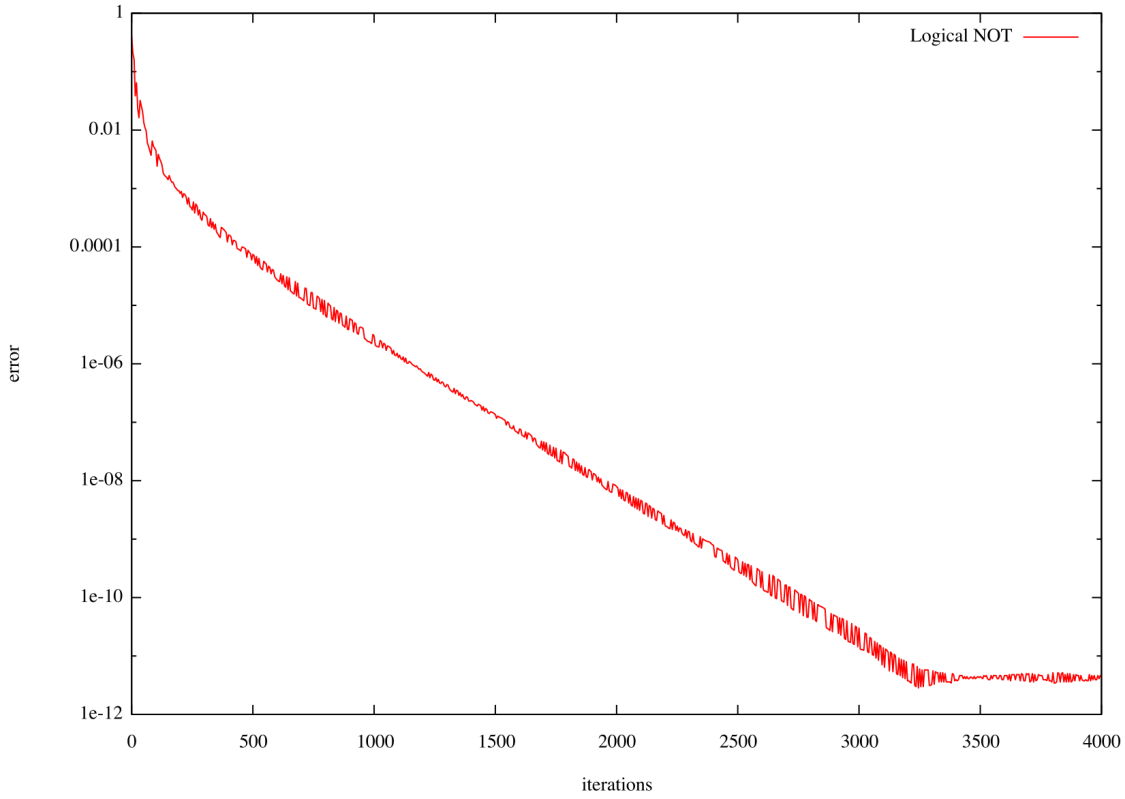

Figure 2.11: Log scale plot of the convergence of a feed forward network training to the Logical NOT operation shown above. Error is scaled logarithmically. Fluctuations are due to randomness in the network training information. Near iteration 3300 the error levels off due to it reaching a hardware precision limit.

Chapter 3 – Recurrent Neural Networks

Traditional artificial neural networks are useful for function approximation, however they are only good for classification of the immediate vector of input signals the system is presented with. To classify a pattern or a grammar it would be necessary to keep track of an observer's state. Recurrent neural networks allow this by introducing cycles to the topology of the network.

3.1 Feed Forward

To account for recurrent connectivity in our network we will start with our original equation for feeding forward our signal, equation 3.2. From there we will add an extra term to represent the influence of the previous iteration's state by means of recurrent connections. Let $\mu_{j,l}^{i,k}$ represent the j, l index of the matrix of weights connecting from layer k into layer i. Let $q_{i,k}$ denote the index of the layer in the network corresponding to the k'th incoming connection from the previous timeframe into layer i. Let τ represent the time at which the value is being referenced.

$$a_j^i(\tau) = \sigma(net_j^i(\tau)), \tag{3.1}$$

$$net_j^i(\tau) = \sum_k \sum_l w_{j,l}^{i,p_{i,k}}(\tau) a_l^{p_{i,k}}(\tau) + \sum_k \sum_l \mu_{j,l}^{i,q_{i,k}}(\tau) a_l^{p_{i,k}}(\tau - 1) \tag{3.2}$$

3.2 Back Propagation

Now that the feed forward aspect has been described we will move on to the amendments to updating the weight matrices. Unlike $p_{i,k}$'s constraint, $q_{i,k}$ contains elements that could be of any index of the network. This invalidates some of the previous simplifications taken when calculating the equations for back propagation. Due to the recurrence in the dependencies of equations, the gradient terms now possess references to every previously referenced index in time. This leaves us with an infinite number of paths to account for when describing our gradient function:

$$\frac{\partial a_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau = \sigma' \left(net_j^i (\tau) \right) \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau \tag{3.3}$$

$$\begin{aligned} \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau = & \sum_{k,l} \left(\frac{\partial w_{j,l}^{i,p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \cdot a_l^{p_{i,k}}(\tau) + w_{j,l}^{i,p_{i,k}} \cdot \frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau \right) + \\ & \sum_{k,l} \left(\frac{\partial \mu_{j,l}^{i,q_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau \cdot a_l^{q_{i,k}}(\tau - 1) + \mu_{j,l}^{i,q_{i,k}}(\tau) \cdot \frac{\partial a_l^{q_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \mid_{\tau - 1} \right) \end{aligned} \tag{3.4}$$

$$\frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau = \delta_{i,m}\delta_{j,s}a_t^{p_{m,n}}(\tau) + \sum_{k,l} w_{j,l}^{i,p_{i,k}}(\tau)\left(\frac{\partial a_l^{p_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau\right) + \sum_{k,l}\left(\mu_{j,l}^{i,q_{i,k}}(\tau)\frac{\partial a_l^{q_{i,k}}}{\partial w_{s,t}^{m,p_{m,n}}} \mid_{\tau-1}\right) \tag{3.5}$$

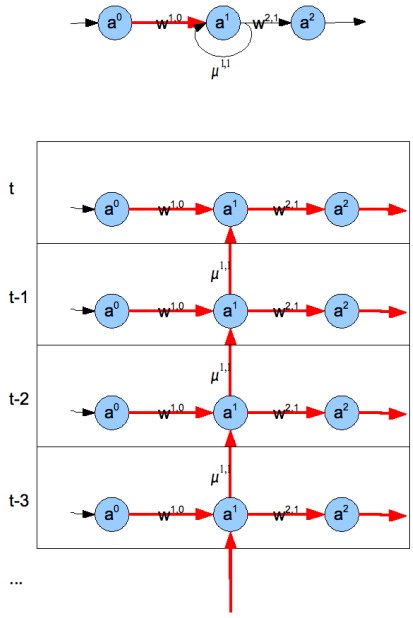

Figure 3.1: The top figure shows an example recurrent network. The single red line shows the weight for which the error gradient will be calculated with respect to. Below, the information propagation of the network is divided among various instances in time. The red lines show the dependencies of computation for the gradient with respect to the target weight.

The initial gradient calculation shown in Eqn. 3.5 leaves us with a circular definition. The derivative is written in terms of the derivative itself. A visual description of what is causing this can be seen in Figure 3.1. We can circumvent this as follows: The relation between the partial of each layer's output signal vector and the weight for which we are calculating the gradient of takes on the form of a rank-3 tensor. Storing this tensor and recalculating it each iteration will maintain the information for all previous instances in time of the gradient. The $\frac{\partial E}{\partial net}$ vector that was used in updating the weights of non-recurrent networks can no longer be used due to its dependency on acyclicity.

$$\xi_{s,t,j}^{m,p_{m,n},i}(\tau) = \frac{\partial a_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau = \sigma'\left(net_j^i(\tau)\right) \cdot \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau \tag{3.6}$$

$$\begin{aligned} \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau = \delta_{i,m}\delta_{j,s} a_t^{p_{m,n}}(\tau) + \sum_{k,l}\left(w_{j,l}^{i,p_{i,k}}(\tau)\xi_{s,t,l}^{m,p_{m,n},p_{i,k}}(\tau)\right) + \\ \sum_{k,l}\left(\mu_{j,l}^{i,q_{i,k}}(\tau)\xi_{s,t,l}^{m,p_{m,n},q_{i,k}}(\tau-1)\right) \end{aligned} \tag{3.7}$$

$$\zeta_{s,t,j}^{m,p_{m,n},i}(\tau) = \frac{\partial a_j^i}{\partial \mu_{s,t}^{m,q_{m,n}}} \mid_\tau = \sigma'\left(net_j^i(\tau)\right) \cdot \frac{\partial net_j^i}{\partial \mu_{s,t}^{m,q_{m,n}}} \mid_\tau \tag{3.8}$$

$$\frac{\partial net_j^i}{\partial \mu_{s,t}^{m,q_{m,n}}} |_\tau = \sum_{k,l} \left(w_{j,l}^{i,p_{i,k}}(\tau) \zeta_{s,t,l}^{m,q_{m,n},p_{i,k}}(\tau) \right) + \delta_{i,m}\delta_{j,s} a_t^{q_{m,n}}(\tau - 1) + \sum_{k,l} \left(\mu_{j,l}^{i,q_{i,k}}(\tau) \zeta_{s,t,l}^{m,q_{m,n},q_{i,k}}(\tau - 1) \right) \tag{3.9}$$

The change in weights, $\frac{\partial E}{\partial w}$, can be calculated by combining the terms of $\frac{\partial y}{\partial w}$ and $\frac{\partial E}{\partial y}$. To update each weight we then combine the last layer's gradient tensor, $\frac{\partial y}{\partial w}$, with the error function's provided $\frac{\partial E}{\partial y}$. This can be seen in Figure 3.2.

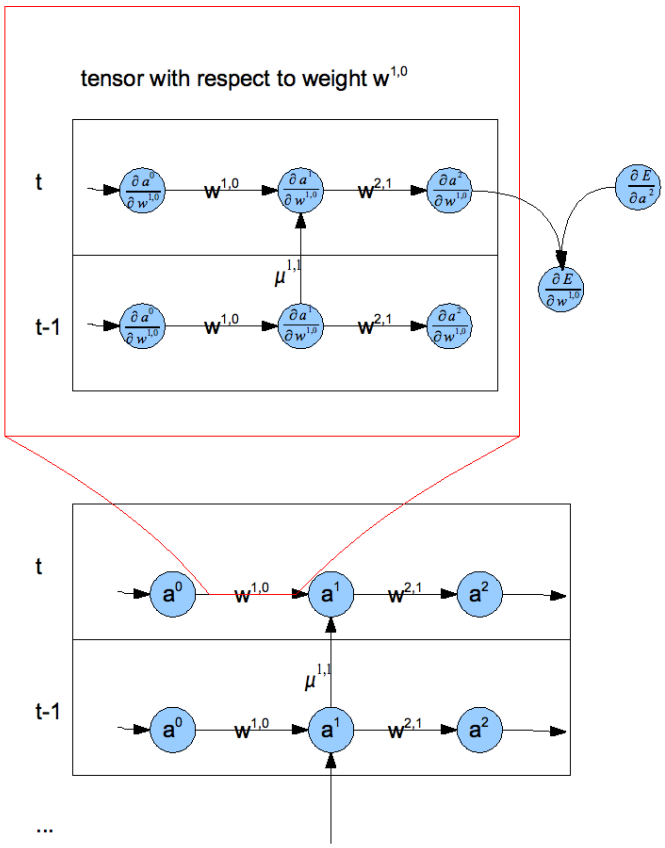

Figure 3.2: For each weight, a copy of the network is stored. The edges of the copy contain tensors that store history. Unlike back propagation, the history is updated in a feed-forward method. This circumvents limitations to graphs with cycles which would cause gradient equation dependences.

Chapter 4 – GPGPU

4.1 Current GPGPU Implementation Options

A need for higher level languages emerged from the many different chipsets and instruction sets of various GPUs. As different vendors made different attempts to remedy this situation, different GPU programming languages emerged. Developers of GPGPU code have several different options:

4.2 NVIDIA Cg

Cg was the first proposal of shading languages. It was developed by NVIDIA as they created the first commercial programmable GPU graphics card. It was originally designed to be easy to learn for programmers already familiar with C In terms of graphics card vendors, Cg was an exclusively-NVIDIA creation.

4.3 HLSL

The languages of Cg and HLSL were developed cooperatively by NVIDIA and Microsoft with the intention of being interchangeable for each platform. For this reason the two languages are near identical. These languages were

highly tailored to the fixed-functionality pipeline that graphics cards had been producing for the last decade.

4.4 GLSL

GLSL was the attempt by OpenGL to standardize all the vendor-specific GPU languages that had been developed. GLSL was similar to Cg in some respects. It differs in subtle ways. The shader functions in Cg operated on accepting input parameter lists and returning structures. Those of GLSL accept no parameters and return no values, however they pre-populate certain global variables (declared `varying`, `uniform`, etc) before the function's execution and require certain variables to be written to by its end (`gl_Position`, `gl_FragColor`, etc).

The design of the language was for the purpose of rendering, however its texture mapping and framebuffer rendering features can be manipulated to perform the parallel computations. This is the most cumbersome to use for our implementation due to repetitive manipulation of framebuffer objects and graphics contexts. In the end it is favorable for being the most cross platform. In some cases it utilizes the hardware to the greatest extent.

4.5 NVIDIA CUDA

NVIDIA bridged the gap of all the graphics programming required to perform non-graphics algorithms with CUDA. This is the most C-like, which makes it the easiest to integrate into applications. However, like Cg, it still fails to be provided by vendors other than NVIDIA. [1]

4.6 ATI Brook

Brook was developed at Stanford and adopted by ATI for competition with CUDA. Brook was designed around the notion of streams and kernels. Stream objects are allocated on GPU memory and read to or written from through CPU-GPU functions. GPU kernel functions then act on streams just as CPU functions act on CPU memory. [2]

4.7 OpenCL

OpenCL is the standardization for what GPGPU programming CUDA allowed to be done. OpenCL will do its best unify all the various attempts at GPGPU programming languages. It is being added to OpenGL 3.0 and will be seen in Mac OS X Snow Leopard 10.6. [14]

Chapter 5 – Algorithm

5.1 Choosing GLSL

I chose GLSL because it was the most platform-independet and backwards compatible choice. CUDA would have been a likely candidate, however it was a vendor-specific language. I decided to maintain independence of platform in my implementation. OpenCL is still budding technology and would not have as large of a backwards compatibility support as GLSL.

GLSL can perform kernel passes using OpenGL textures. Textures have certain restrictions on format, size, and resolution. Typical GPGPU programs operation on a fairly recent invention in the lifetime of graphic cards: floating point precision textures. These are textures whose channel elements (red, green, blue, alpha) are represented with floating point precision data.

The first implementation of the algorithm only stored one vector/matrix/tensor element per pixel. Rather than only use one channel for each texel the GLSL code was redesigned to use all four channels. This requires a bit more special planning, but was still feasible. See section 5.7.2 for an example of matrix multiplication using all four channels.

Another constraint on GLSL usage is texture size. Older cards have a restriction to only use power-of-two texture constraints. Newer cards can use non-power of two sizes via newer OpenGL API extensions. For flexibility I designed this for old power-of-two-constrained cards. To accomplish this I had to pad all 2D textures with unused pixels to round the width and height up to the next power of two.

5.2 Hardware Constraints

OpenGL graphics cards only support textures with dimensions up to a certain size. This can be found by querying the `GL_MAX_TEXTURE_SIZE` variable. On modern graphics cards this is typically a value of 1024, 2048, 4096, or 8192. Vectors used in the GPU version of the RNN algorithm must keep their size smaller than the texture maximum size. Networks which require larger sizes than their hardware allows can circumvent this by splitting oversized vectors into many smaller-sized vectors. At present this limitation is left to the implementer of the network, however a future development would be for the GPU algorithm to automatically perform this adjustment. Figure 5.1 gives a visual example of this transformation.

Another important constraint for different underlying hardware is its number of texture units. GPGPU kernels are limited by a maximum number of

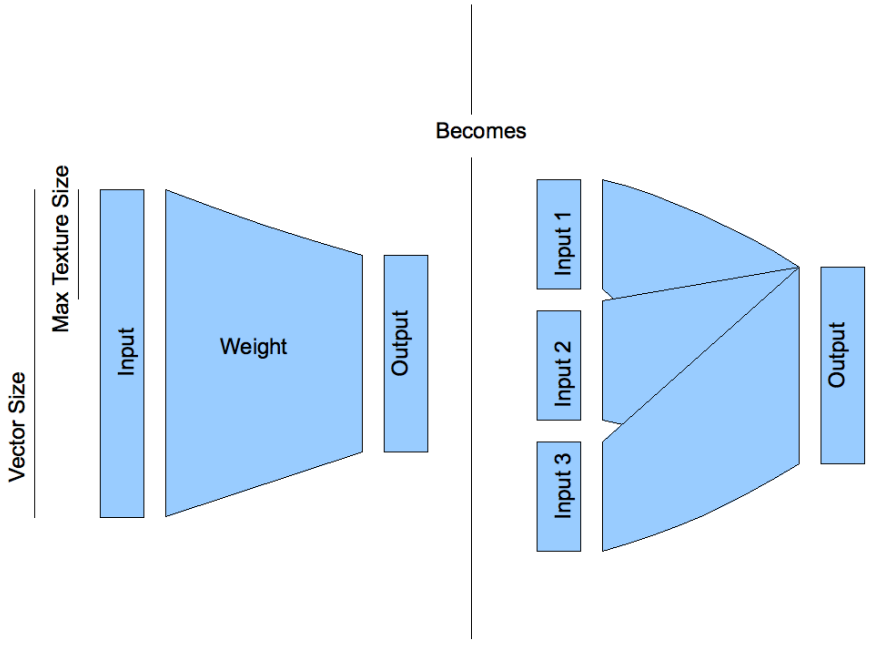

Figure 5.1: Demonstration of a remedy to supporting vector sizes larger than the limiting size of textures on graphics cards.

texture units. For certain implementations described below an arbitrary number of inputs is required. If a network layer calculation requires more units than hardware allows then layer accumulation can be deferred through additional layers that are connected through identity weight relations and identity activation functions. Figure 5.2 shows a visual example of this.

Despite it being a standard in the GLSL reference, older NVIDIA implementations do not support the operator[ ] non-constant integer access. For

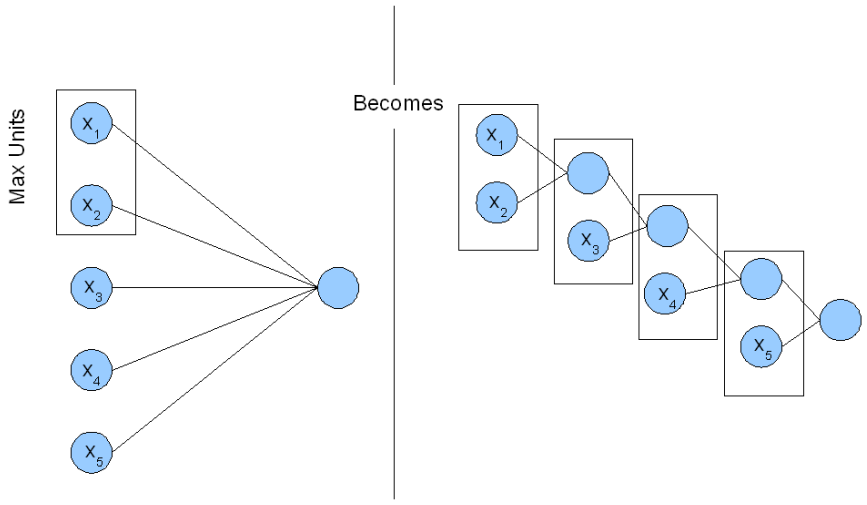

Figure 5.2: Demonstration of a remedy to supporting incoming connections larger than the limiting number of texture units on graphics cards.

that reason some special case code had to be implemented as a work-around.

5.3 Implementation

To accurately perform all the correct computations for the layer at time $\tau+1$, the τ and $\tau-1$ layers need to be stored. Each timeframe holds information pertaining to the layers, the weights, and the history tensor.

Each timeframe's ith layer contains the following: the activation function, σ^i; the signal before activation, net^i_j; the signal after activation, a^i_j; the deriva-

tive between the two, $\frac{\partial a_j^i}{\partial net_j^i}$. As stated in section 2.4, the current algorithm assumes that this function operates on each vector element individually – that is – that $\frac{\partial a_j^i}{\partial net_{j'}^i} = 0$ for $j \neq j'$. The timeframe's connection from layer $p_{i,k}$ to layer i holds the weight matrix, $w_{j,l}^{i,p_{i,k}}$. The timeframe's history gradient tensor is also stored. This tensor holds the content of $\xi_{s,t,j}^{m,p_{m,n},i}$ for non-recurrent connections and $\zeta_{s,t,j}^{m,q_{m,n},i}$ for recurrent connections.

5.4 Usage

RNN's are constructed by specifying the number of layers and connectivity between them. Optional information can be specified in addition to this, such as what activation functions are used at each layer and what connections should exist between layers.

Once created, an RNN then undergoes an iterative training procedure in which they adjust to fit to some unknown classifying function. First, inputs are filled according to the black box's inputs. Next, desired outputs are specified for the certain inputs. Last, the object's `RNN::process` method is called: the network's output is computed, the error is compared with the desired output, and the weights are adjusted accordingly via back propagation.

```
// set up your network:
rnnlib::GPURNN rnn;
```

```
rnn.setDirect(2, 2, 1);
rnn.prepare();

for (int i = 0; i < iterations; i++) {
 int a = rand() & 1;
 int b = rand() & 1;
 int d = a & b;
 // prepare inputs
 rnn.input[0] = rnn.getInputSignal()->to(a);
 rnn.input[1] = rnn.getInputSignal()->to(b);
 // prepare outputs
 rnn.desired[0] = rnn.getOutputSignal()->to(d);
 // evaluate and adjust weights
 double error = rnn.process();
}
```

5.4.1 RNN::process

The `RNN::process` method is broken into several steps which can be executed explicitly on their own:

```
float RNN::process() {
 feedForward();
 calcDeDy();
 float error = calcError();
```

```
 updateHistory();
 updateWeights();
 cycleLayers();
 return error;
}
```

5.4.2 RNN::cycleLayers

The last step in the iteration is to cycle the pointers of the layers. This moves the contents of our current timeframe to our last timeframe and recycles our last timeframe to be used as our next timeframe. This is a fairly straightforward operation:

```
void RNN::cycleLayers() {
  RNNTimeframe *lastPreviousFrame = frames[PREVIOUS_FRAME];
  frames[PREVIOUS_FRAME] = frames[CURRENT_FRAME];
  frames[CURRENT_FRAME] = lastPreviousFrame;
}
```

5.5 CPU Implementation

5.5.1 CPURNN::feedForward

The `CPURNN::feedforward` method works as follows: First the network input vector is copied into the first layer of the current frame. Next, for each layer, for each input into that layer, the input vector is transformed by the weight matrix and accumulated in the layer's `net` variable. The `net` is transformed by the activation function and stored in `x`. Finally the `net` and `x` are used to calculate the activation function's derivative, which is stored in `dXdNet`.

```
void CPURNN::feedForward() {
 frames[CURRENT_FRAME]->fillInput(input);
 frames[CURRENT_FRAME]->feedForward(frames[PREVIOUS_FRAME]);
}
```

```
void CPURNNTimeframe::feedForward(CPURNNTimeframe *lastframe) {
 for (int i = 1; i < layers.dim(); i++) {
  vecf &net = layers(i)->net;
  net.fill(0.f);

  for (int k = 0; k < i; k++) {
   if (!weights(i,k)) continue;
   matf &w = weights(i,k)->w;
   vecf &x = layers(k)->x;
```

```
    net += w * x;
   }

   vecf &y = layers(i)->x;
   y = layers(i)->signal->transfer(net);

   vecf &dYdNet = layers(i)->dXdNet;
   dYdNet = layers(i)->signal->deriv(net, y);
  }
}
```

5.5.2 CPURNN::calcDeDy

The network's $\frac{\partial E}{\partial y}$ placed in a distinct structure for flexibility of implementation. This function calculates the gradient of the error function with respect to last layer's signal vector. By default, the `CPURNN::calcDeDy` assumes the error function to be $E(y) = \frac{1}{2}\sum_j (y_j - d_j)^2$. This is a standard for supervised learning. However for certain tasks, such as unsupervised learning, it is useful to manually calculate and provide the $\frac{\partial E}{\partial y}$ vector.

```
void CPURNN::calcDeDy() {
 frames[CURRENT_FRAME]-> calcDeDy(desired, dEdY);
}
```

```
void CPURNNTimeframe::calcDeDy(const vecf &desired, vecf &dEdY) {
 CPURNNLayer *lastlayer = layers(layers.dim()-1);
 const vecf &y = lastlayer->x;
 dEdY = y - desired;
}
```

5.5.3 CPURNN::calcError

The CPURNN::calcError function is a purely cosmetic function. It performs the actual $E(y) = \frac{1}{2}\sum_j (y_j - d_j)^2$ calculation. This is useful for observing the convergence behavior of the network, but does not come into play in regards to the back-propagation.

```
void CPURNN::calcError() {
 float e = 0.f;
 for (int i = 0; i < dEdY.dim(); i++) {
  e += dEdY(i) * dEdY(i);
 }
 e *= 0.5;
 return e;
}
```

5.5.4 CPURNN::updateHistory

The CPURNN::updateHistory function updates the $\frac{\partial a}{\partial w}$ components of each layer with respect to each weight. Acyclic topology neural networks back-propagate the errors, first calculating the last layer error term and then using those values to calculate prior $\frac{\partial E}{\partial w}$ terms. The dependency of the $\frac{\partial a}{\partial w}$ is the opposite: it instead forward-propgates the partials, starting at the first layer and finishing at the last. This is done up each connection of the network until the $\frac{\partial y}{\partial w}$ partial is computed. Only then can the error partial be calculated with respect to the weights, and the weights can be updated (Figure 3.2).

```
void CPURNN::updateHistory() {
 frames[CURRENT_FRAME]->updateHistory( frames[PREVIOUS_FRAME],
     dEdY);
}


void CPURNNTimeframe::updateHistory(CPURNNTimeframe *lastframe,
    vecf &dEdY) {

 //for the weights feeding in from the current timeframe
 //forward-propogate the dX/dW calculations to find dY/dW for each
     W
 for (int m = 1; m < layers.dim(); m++) {
  const vecf &xm = layers(m)->x;
```

```
for (int n = 0; n < m; n++) {
 if (!weights(m,n)) continue;
 const vecf &xn = layers(n)->x;
 for (int i = 1; i < layers.dim(); i++) {
  const vecf &xi = layers(i)->x;
  const vecf &dXidNet = layers(i)->dXdNet;
  tensor3f &hmni = history(m,n,i)->dXdW;
  for (int j = 0; j < hmni.depth(); j++) {
   for (int s = 0; s < hmni.height(); s++) {
    for (int t = 0; t < hmni.width(); t++) {
     hmni(s,t,j) = 0.f;
     for (int k = 0; k < i; k++) {
      if (!weights(i,k)) continue;
      if (k) { //no weights feed into the input layer
       const vecf &xk = layers(k)->x;
       const matf &wik = weights(i,k)->w;
       const tensor3f &hmnk = history(m,n,k)->dXdW;
       for (int l = 0; l < xk.dim(); l++) {
        hmni(s,t,j) += wik(j,l) * hmnk(s,t,l);
       }
      }
      if (i == m && j == s && k == n) { //Kronecher deltas
       hmni(s,t,j) += xn(t);
      }
     }
     for (int k = 1; k < layers.dim(); k++) {
```

```
        if (!recurWeights(i,k)) continue;
        const vecf &xk = layers(k)->x;
        const matf &wik = recurWeights(i,k)->w;
        const tensor3f &hmnk = lastframe->history(m,n,k)->dXdW;
        for (int l = 0; l < xk.dim(); l++) {
         hmni(s,t,j) += wik(j,l) * hmnk(s,t,l);
        }
       }
      }
     }
    }
    for (int j = 0; j < hmni.depth(); j++) {
     for (int s = 0; s < hmni.height(); s++) {
      for (int t = 0; t < hmni.width(); t++) {
       hmni(s,t,j) *= dXidNet(j);
      }
     }
    }
   }
  }
}

//for the weights feeding in from the previous timeframe
//forward-propogate the dX/dW calculations to find dY/dW for each
    W
for (int m = 1; m < layers.dim(); m++) {
```

```
const vecf &xm = layers(m)->x;
for (int n = 0; n < m; n++) {
 if (!recurWeights(m,n)) continue;
 const vecf &prevXn = lastframe->layers(n)->x;
 for (int i = 1; i < layers.dim(); i++) {
  const vecf &xi = layers(i)->x;
  const vecf &dXidNet = layers(i)->dXdNet;
  tensor3f &hmni = recurHistory(m,n,i)->dXdW;
  for (int j = 0; j < hmni.depth(); j++) {
   for (int s = 0; s < hmni.height(); s++) {
    for (int t = 0; t < hmni.width(); t++) {
     hmni(s,t,j) = 0.f;
     for (int k = 1; k < i; k++) {
      if (!weights(i,k)) continue;
      const vecf &xk = layers(k)->x;
      const matf &wik = weights(i,k)->w;
      const tensor3f &hmnk = recurHistory(m,n,k)->dXdW;
      for (int l = 0; l < xk.dim(); l++) {
       hmni(s,t,j) += wik(j,l) * hmnk(s,t,l);
      }
     }
     for (int k = 1; k < layers.dim(); k++) {
      if (!recurWeights(i,k)) continue;
      if (k) {
       const vecf &xk = layers(k)->x; //dimensionality only
       const matf &wik = recurWeights(i,k)->w;
```

```
          const tensor3f &hmnk = lastframe->recurHistory(m,n,k)->
              dXdW;
          for (int l = 0; l < xk.dim(); l++) {
           hmni(s,t,j) += wik(j,l) * hmnk(s,t,l);
          }
         }
         if (i == m && j == s && k == n) {
          hmni(s,t,j) += prevXn(t);
         }
        }
       }
      }
      for (int j = 0; j < hmni.depth(); j++) {
       for (int s = 0; s < hmni.height(); s++) {
        for (int t = 0; t < hmni.width(); t++) {
         hmni(s,t,j) *= dXidNet(j);
        }
       }
      }
     }
    }
   }
}
```

5.5.5 CPURNN::updateWeights

After the $\frac{\partial x}{\partial w}$ is calculated for each layer, the gradient with respect to the last layer, $\frac{\partial y}{\partial w}$, can be multiplied with the $\frac{\partial E}{\partial y}$ to get the gradient of the error with respect to each weight, $\frac{\partial E}{\partial w}$. Take care to note that, because we are trying to minimize the error, we want to travel in the direction of the negative of the gradient rather than the positive.

```
void CPURNN::updateWeights() {
 frames[CURRENT_FRAME]->updateWeights(frames[PREVIOUS_FRAME], dEdY
     , timestep);
 frames[PREVIOUS_FRAME]->fillWeights(frames[CURRENT_FRAME]);
}

void CPURNNTimeframe::updateWeights(CPURNNTimeframe *lastframe,
    const vecf &dEdY, float stepSize) {
 //scale the dY/dW by the dE/dY to get each weight's dE/dW
 for (int m = 1; m < layers.dim(); m++) {
  const vecf &xm = layers(m)->x;
  for (int n = 0; n < m; n++) {
   if (!weights(m,n)) continue;
   const vecf &xn = layers(n)->x;
   matf &wmn = weights(m,n)->w;
   const vecf &y = layers(layers.dim()-1)->x;
   const tensor3f &hmny = history(m,n,layers.dim()-1)->dXdW;
   for (int i = 0; i < y.dim(); i++) {
```

```
    for (int s = 0; s < xm.dim(); s++) {
     for (int t = 0; t < xn.dim(); t++) {
      float dYdW = hmny(s,t,i);
      wmn(s,t) -= stepSize * dEdY(i) * dYdW;
     }
    }
   }
  }
}

//scale the dY/dW by the dE/dY to get each weight's dE/dW
for (int m = 1; m < layers.dim(); m++) {
 const vecf &xm = layers(m)->x;
 for (int n = 0; n < m; n++) {
  if (!recurWeights(m,n)) continue;
  const vecf &prevXn = layers(n)->x;
  matf &wmn = recurWeights(m,n)->w;
  const vecf &y = layers(layers.dim()-1)->x;
  const tensor3f &hmny = recurHistory(m,n,layers.dim()-1)->dXdW;
  for (int i = 0; i < y.dim(); i++) {
   for (int s = 0; s < xm.dim(); s++) {
    for (int t = 0; t < prevXn.dim(); t++) {
     float dYdW = hmny(s,t,i);
     wmn(s,t) -= stepSize * dEdY(i) * dYdW;
    }
   }
```

```
    }
   }
  }
}
```

5.6 GPU Implementation

For each layer, each of our *net*, a, and $\frac{\partial a}{\partial net}$ vectors is stored in its own 2D texture. I first considered using a single texture by using the red texel component for the *net* vector, the green for a, etc. However this proved to be a poor decision since some GPU texture kernel operations require reading to some and writing to other vectors in the same layer. Implementing two such vectors on the same texture would impede such an operation with at least the need for an additional temporary texture and a separate texture copy operation, so I chose otherwise.

The $w_{j,l}^{i,k}$ matrix is stored column-order, with every four column vector elements stored in a texel's RGBA components. Matrices are also allowed an optional bias row, equivalent in operation as extending the matrix height by one unit, and appending a "1" value to any multiplied vector. Matrices are also allowed an optional flag to set whether they are constant or not.

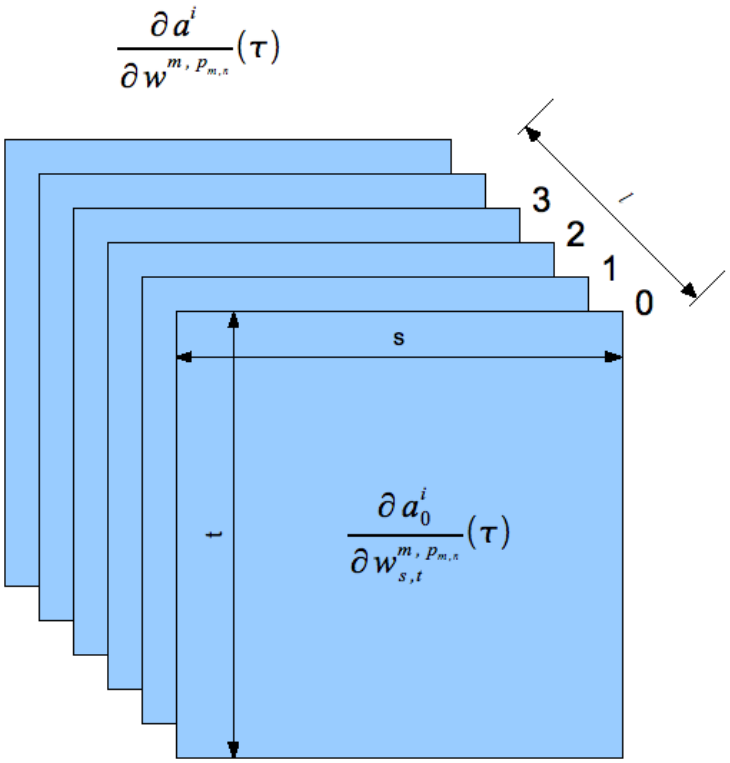

Figure 5.3: A demonstration of how a list of 2D textures contain the rank-3 history gradient tensor.

The history gradients, $\eta_{i,j,k} = \frac{\partial a_k}{\partial w_{i,j}}$, are stored as a distinct 2D texture per each ∂a_k, with the dimensions of this texture spanning the i, j components of $\partial w_{i,j}$. These tensors are accessed for their weight updates per-layer, so the two indices of the weight matrix go to the texture and the one of the layer go to

the array (Figure 5.3). This way, when we go through all our connections and forward pass the $\frac{\partial a}{\partial w}$ value to come up with each $\frac{\partial y}{\partial w}$, we only need to access the index in the array corresponding to what layer we are forward-passing our gradient along.

5.7 GPU Ping-Pong Implementation

The GPU implementation differs slightly from the CPU in several aspects. Tensor operations are organized by their two most predominantly-used dimensions. Copies between elements are limited as much as possible, favoring exchanges of references over values. Unique shaders also need to be created for each step of the operation.

A key component to the GPU implementation of the RNN is its `GPURNNAuxInfo` class. This information is shared between timeframes. It includes floating point textures to be used as temporary render targets. It also includes all the shaders that are compiled and referenced during the operation.

A staple technique for GPGPU programs is ping-ponging. GPU kernels do not allow reading and writing to the same location of memory. For this reason it is popular to set aside two textures for any dynamic GPGPU kernel: one for reading, one for writing. Once the kernel operation takes place the roles of the two textures are reversed before processing the system again. This prevents

a need to perform an extra copy routine, however it has the minor drawback that the memory location of the most current data is constantly changing.

5.7.1 GPURNN::prepare

Preparing the GPU RNN is a complicated process and varies greatly depending on the implementing algorithm. For the ping-pong model this involves allocating temporary floating-point textures and creating all shaders and activation function kernels.

Once the network is prepared it can be used for general RNN usage. While the basic `RNN::process` is still the same, the members that it calls vary in their implementation. Each is described below:

5.7.2 GPURNN::feedForward

This is the first component of the process step. The `RNN::input` and `RNN ::desired` vectors are filled before the `RNN::process` method call to provide a usage interface abstracting whether or not the GPU was used. The advantage to this is interchangeability and flexibility of implementing the RNN class in environments whose calculations are performed or stored outside of GPU memory. The downside is that CPU-to-GPU memory transforms are typically bottlenecks in GPU performance.

```
void GPURNN::feedForward() {
 frames[CURRENT_FRAME]->layers(0)->x.fillFromVec(input);
 desTex.fillFromVec(desired);
 frames[CURRENT_FRAME]->feedForward(frames[PREVIOUS_FRAME], aux);
}
```

To circumvent this bottleneck an alternative method was added to the GPU RNN for exposing the floating point textures of the input and desired vectors directly. Paired with it is an alternative to `GPURNN::feedForward` which does not need to copy the input and desired vectors from CPU memory.

```
FloatTex *GPURNN::getInputTex() {
 GPURNNTimeframe *currentFrame = frames[CURRENT_FRAME];
 if (!currentFrame->getLayers().dim()) return NULL;
 return &currentFrame->layers(0)->x;
}


FloatTex *GPURNN::getDesiredTex() {
 return &desiredTex;
}


void GPURNN::feedForwardWithoutFilling() {
 frames[CURRENT_FRAME]->feedForward(frames[PREVIOUS_FRAME]);
```

}

From here we discuss our GPU implementation of matrix multiplication. For the most primitive implementation this will make use of floating point texture ping-ponging. Accompanying shaders are designed to sum and reduce the channels in the textures at an exponential rate. After the matrix multiplication completes and writes its result to the output vector, a bias element is appended at the end of the vector. The `glColorMask` function is used to apply it only to the desired component of the RGBA texel of the vector.

Several shaders are used during the matrix multiplication process:

1. The `matVecRowScaleShader` performs a simple texture scale operation used as the first pass to the matrix-vector multiplication.

2. The results of the scale are sent through a `rowReduce` call to add like columns into a single column.

3. The `matVecColToRowCombineShader` then takes up the task of rotating the final column into a final row.

4. This final row is copied into the single row of the output *net* texture.

5. The layer's signal's `transferShader` calculates y from net.

6. `derivShader` uses both net and y to compute $\frac{\partial y}{\partial net}$.

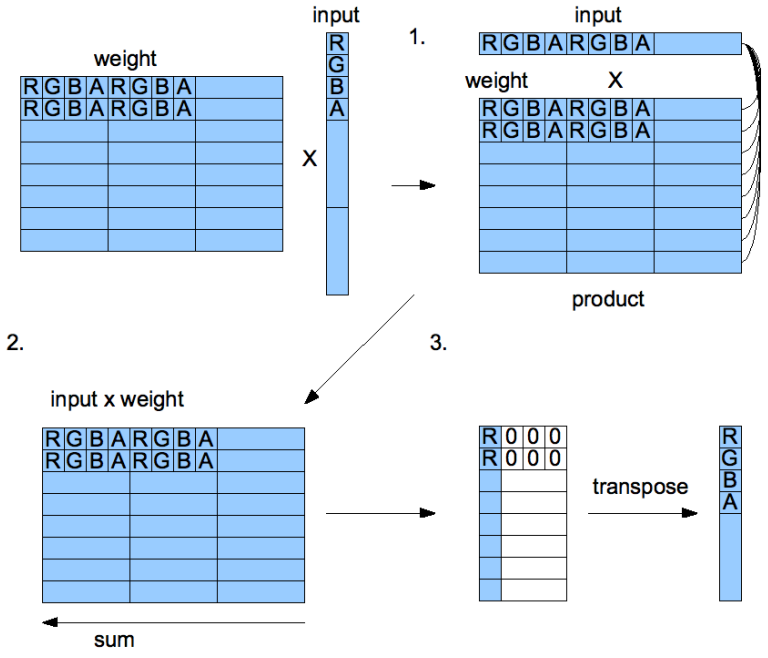

Figure 5.4: An example of matrix multiplication. (1) First each row is multiplied with the input vector and stored in place. (2) Next the columns are summed together into a single column. (3) Last the remaining column is transposed into a compact representation of the result.

Figure 5.4 gives a visual depiction of the process.

The `transferShader` and `derivShader` is scalar function applied to all components of the vector. It varies depending on what activation function is used for the accumulated values of the receiving layer.

```
void GPURNNTimeframe::feedForward(GPURNNTimeframe *lastframe,
    GPURNNAuxInfo &aux) {
 glColor3f(1,1,1);
 setUnitViewport();
 SetFBO fbo(aux.fboID);
 for (int i = 1; i < layers.dim(); i++) {
  GPURNNTimeframeLayerInfo &layerInfo = layerInfos[i];
  GPURNNLayer *y = layers[i];
  vec3i ri(0,1,2);
  for (int j = 0; j < layerInfos[i].inputs.size(); j++) {
   GPURNNWeight *w = layerInfos[i].inputs[j].w;
   GPURNNLayer *x = layerInfos[i].inputs[j].x;
   if (fbo.target(aux.regs[ri.x])) {
    glViewport(0,0,VEC2ELEM(aux.regs[ri.x].texSize));
    glClearColor(0,0,0,0);
    glClear(GL_COLOR_BUFFER_BIT);
    SetShader ss(aux.matVecRowScaleShader);
    SetTextures<2> st(w->tex, x->x);
    SetViewport sv(vec2i(w->tex.texSize.x, w->tex.matSize.y));
    st.draw(sv);
   }
   rowReduce(fbo, aux, ri, x->x.texSize.x, w->tex.matSize.y);
   bool lastpass = j == layerInfos[i].inputs.size()-1;
   const FloatTex &desttex = lastpass ? y->net : aux.regs[ri.y];
   if (fbo.target(desttex)) {
    SetShader ss(aux.matVecColToRowCombineShader);
```

```
  ss.setUniform<float>(0, y->net.texSize.x);
  ss.setUniform<float>(1, 1.f);
  ss.setUniform<float>(2, aux.regs[ri.x].texSize.x);
  ss.setUniform<float>(3, aux.regs[ri.x].texSize.y);
  ss.setUniform<float>(4, j>0);
  SetTextures<2> st(aux.regs[ri.x], aux.regs[ri.z]);
  SetViewport sv(vec2i(y->net.texSize.x, 1));
  drawScreenQuad();
 }
 swap(ri.y, ri.z);
}
if (fbo.target(y->x)) {
 {
  SetShader ss(y->signal->transferShader);
  SetTextures<1> st(y->net);
  SetViewport sv(vec2i(y->x.texSize.x, 1));
  drawScreenQuad();
 }
 glColorMask((y->size() & 3) == 0, (y->size() & 3) == 1, (y->
     size() & 3) == 2, (y->size() & 3) == 3);
 glBegin(GL_QUADS);
 int qsize = y->size() >> 2;
 float x1 = (float)qsize / (float)y->x.texSize.x;
 float x2 = (float)(qsize + 1) / (float)y->x.texSize.x;
 glVertex2f(x1, 0);
 glVertex2f(x1, 1);
```

```
    glVertex2f(x2, 1);
    glVertex2f(x2, 0);
    glEnd();
    glColorMask(1,1,1,1);
   }
   if (fbo.target(y->dXdNet)) {
    SetShader ss(y->signal->derivShader);
    SetTextures<2> st(y->net, y->x);
    SetViewport sv(vec2i(y->x.texSize.x, 1));
    drawScreenQuad();
   }
  }
}
```

5.7.3 matVecRowScaleShader

```
uniform sampler2D mattex;
uniform sampler2D rowtex;
void main() {
 vec4 rowcolor = texture2D(rowtex, gl_TexCoord[0].st);
 vec4 matcolor = texture2D(mattex, gl_TexCoord[0].st);
 gl_FragColor = rowcolor * matcolor;
}
```

5.7.4 matVecColToRowCombineShader

```
varying vec2 pos;
uniform sampler2D coltex;
uniform sampler2D accumtex;
uniform float coltexwidth;
uniform float coltexheight;
uniform float writewidth;
uniform float writeheight;
uniform float accum;

void main() {
 vec2 tc = vec2(pos.x * writewidth - .5, pos.y * writeheight - .5)
    ;
 vec2 accumtc = vec2((tc.x + .5) * coltexwidth, (tc.y + .5) *
    coltexheight);
 gl_FragColor = texture2D(accumtex, accumtc) * accum;
 vec2 coltc;
 float colr;
 coltc = vec2(.5 / coltexwidth, (4. * tc.x + .5) / coltexheight);
 colr = texture2D(coltex, coltc).r;
 gl_FragColor.r += colr;
 coltc = vec2(.5 / coltexwidth, (4. * tc.x + 1.5) / coltexheight);
 colr = texture2D(coltex, coltc).r;
 gl_FragColor.g += colr;
 coltc = vec2(.5 / coltexwidth, (4. * tc.x + 2.5) / coltexheight);
 colr = texture2D(coltex, coltc).r;
 gl_FragColor.b += colr;
```

```
 coltc = vec2(.5 / coltexwidth, (4. * tc.x + 3.5) / coltexheight);
 colr = texture2D(coltex, coltc).r;
 gl_FragColor.a += colr;
}
```

5.7.5 GPURNN::calcDeDy

The GPURNN::calcDeDy is a straightforward port from the CPU variant. It requires the current layer's output to have been generated, as is generated at the end of GPURNN::feedForward. It also requires the texture holding the desired output to be provided, as occurs at the beginning of GPURNN::feedForward. It relies on the calcDeDyShader for subtracting the network output and desired output vectors.

```
void GPURNN::calcDeDy() {
 frames[CURRENT_FRAME]->calcDeDy(aux, desTex, dEdYTex);
}

void GPURNNTimeframe::calcDeDy(GPURNNAuxInfo &aux, const FloatTex
    &desTex, FloatTex &dEdYTex) {
 GPURNNLayer *y = layers[layers.dim()-1];
 SetFBO fbo(aux.fboID);
 if (fbo.target(dEdYTex)) {
  SetShader ss(aux.calcDeDyShader);
  SetTextures<2> st(y->x, desTex);
  SetViewport sv(y->x.texSize);
```

```
    drawScreenQuad();
  }
}
```

5.7.6 calcDeDyShader

```
varying vec2 pos;
uniform sampler2D ytex;
uniform sampler2D destex;

void main() {
 vec4 ycolor = texture2D(ytex, pos);
 vec4 descolor = texture2D(destex, pos);
 gl_FragColor = ycolor - descolor;
}
```

5.7.7 GPURNN::calcError

The `GPURNN::calcError` function implementation is very close to that of the CPU version.. It relies on the `errorDifShader`, which subtracts the network output from desired output and squares the terms. Next it calls `rowReduce` to sum the vector into a single error value.

```
float GPURNN::calcError() {
 GPURNNTimeframe *frame = (GPURNNTimeframe*)getCurrentFrame();
 GPURNNLayer *outputLayer = (GPURNNLayer*)frame->getOutputLayer();
 SetFBO fbo(aux.fboID);
 vec3i ri(0,1,2);
 if (fbo.target(aux.regs[ri.y])) {
  glViewport(0,0,VEC2ELEM(aux.regs[ri.y].texSize));
```

```
  glClearColor(0,0,0,0);
  glClear(GL_COLOR_BUFFER_BIT);
  SetShader ss(aux.errorDifShader);
  SetTextures<2> st(outputLayer->x, desTex);
  glViewport(0,0, outputLayer->x.texSize.x, 1);
  drawScreenQuad();
 }
 swap(ri.x, ri.y);
 rowReduce(fbo, aux, ri, outputLayer->x.texSize.x, 1);
 float e = -fInf;
 glReadPixels(0,0,1,1, GL_RED, GL_FLOAT, &e);
 return e;
}
```

5.7.8 errorDifShader

```
varying vec2 pos;
uniform sampler2D ytex;
uniform sampler2D destex;

void main() {
 vec4 ycolor = texture2D(ytex, pos);
 vec4 descolor = texture2D(destex, pos);
 gl_FragColor = descolor - ycolor;
```

```
 gl_FragColor *= 0.5 * gl_FragColor;
}
```

5.7.9 rowReduce

The `rowReduce` function performs a sum across the x dimension of a texture. It makes use of the four-component dot product and several texture lookups in order to combine the rows at up to sixteen to one. Its viewport window of operation on the texture window shrinks by a factor of four – the number of texture lookups per kernel pass – per iteration. Its render target output is a fourth the size of the input, due to the four texture lookups per pass as well. It relies on the `matVecRotReduceNto1Shader`s to perform the reduction.

```
void rowReduce(SetFBO &fbo, GPURNNAuxInfo &aux, vec3i &ri, int
    baseWidth, int viewportHeight) {
 glClearColor(0,0,0,0);
 bool compressed = false;
 int curWidth = baseWidth;
 for (; curWidth >= 4; curWidth >>= 2) {
  int destWidth = curWidth >> 2;
  if (fbo.target(aux.regs[ri.y])) {
   if (!compressed) {
    glViewport(0,0,VEC2ELEM(aux.regs[ri.y].texSize));
    glClear(GL_COLOR_BUFFER_BIT);
   }
```

```
  SetShader ss(aux.matVecRowReduce16to1Shader);
  ss.setUniform<float>(0, destWidth);
  ss.setUniform<float>(1, aux.regs[ri.x].texSize.x);
  ss.setUniform<float>(2, (float)viewportHeight / (float)aux.regs
      [ri.x].texSize.y);
  SetTextures<1> st(aux.regs[ri.x]);
  SetViewport sv(vec2i(destWidth, viewportHeight));
  drawScreenQuad();
 }
 swap(ri.x, ri.y);
 compressed = true;
}
if (!compressed || curWidth == 2) {
 if (fbo.target(aux.regs[ri.y])) {
  glViewport(0,0,VEC2ELEM(aux.regs[ri.y].texSize));
  glClear(GL_COLOR_BUFFER_BIT);
  SetShader ss(curWidth == 2 ? aux.matVecRowReduce8to1Shader :
      aux.matVecRowReduce4to1Shader);
  ss.setUniform<float>(0, aux.regs[ri.x].texSize.x);
  ss.setUniform<float>(1, (float)viewportHeight / (float)aux.regs
      [ri.x].texSize.y);
  SetTextures<1> st(aux.regs[ri.x]);
  SetViewport sv(vec2i(1, viewportHeight));
  drawScreenQuad();
 }
 swap(ri.x, ri.y);
```

```
 }
}
```

5.7.10 matVecRowReduce16to1Shader

The `matVecRowReduce16to1Shader`, `matVecRowReduce8to1Shader`, and `matVecRowReduce4to1Shader` shaders are all derived from the same GLSL fragment program by merely alternating the `NUM_COMBINES` preprocessor macro value. Each of them reads in texture values from a viewport assumed to be $\frac{n}{4}$ in size, for n the number of values that the shader reduces per-pass. The four is due to the four channels read per texture lookup. It then writes them out into an output of size $max(\frac{n}{16}, 1)$. The sixteen is due to the fact that four lookups are used and that each lookup reads four channels at a time.

```
varying vec2 pos;
uniform sampler2D readtex;

#if NUM_COMBINES < 4
#define writewidth 1.
#else
uniform float writewidth;
#endif

uniform float readwidth;
```

```
uniform float tyscale;

void main() {
 vec2 tc = vec2(pos.x * writewidth - .5, pos.y * tyscale);
 vec2 readtc;
 vec4 readcolor;
 const vec4 ones = vec4(1.);
 readtc.y = tc.y;

 readtc.x = (4. * tc.x + .5) / readwidth;
 readcolor = texture2D(readtex, readtc);
 gl_FragColor.r = dot(readcolor, ones);

#if NUM_COMBINES >= 2
 readtc.x = (4. * tc.x + 1.5) / readwidth;
 readcolor = texture2D(readtex, readtc);
 gl_FragColor.r += dot(readcolor, ones);

#if NUM_COMBINES >= 3
 readtc.x = (4. * tc.x + 2.5) / readwidth;
 readcolor = texture2D(readtex, readtc);
 gl_FragColor.r += dot(readcolor, ones);

#if NUM_COMBINES >= 4
 readtc.x = (4. * tc.x + 3.5) / readwidth;
 readcolor = texture2D(readtex, readtc);
```

```
 gl_FragColor.r += dot(readcolor, ones);
#endif
#endif
#endif

 gl_FragColor.gba = vec3(0.);
}
```

5.7.11 GPURNN::updateHistory

The GPURNN::updateHistory method is the heart and soul of the GPU RNN algorithm. All the prior computing was matrix-vector multiplications, adds, and scalar functions. The GPURNN::updateHistory is where the tensor operation between the $\frac{\partial a}{\partial w}$ components is stored. In terms of our implementation, from this point on $\frac{\partial a}{\partial w}$ will refer to $\xi_{s,t,j}^{m,p_{m,n},i}$ for feed-forward connections and $\zeta_{s,t,j}^{m,p_{m,n},i}$ for recurrent.

```
void GPURNN::updateHistory() {
 frames[CURRENT_FRAME]->updateHistory(aux, frames[PREVIOUS_FRAME],
     dEdYTex);
}

void GPURNNTimeframe::updateHistory(GPURNNAuxInfo &aux,
   GPURNNTimeframe *lastframe, const FloatTex &dEdYTex) {
 SetFBO fbo(aux.fboID);
```

```
for (int recurDst = 0; recurDst < 2; recurDst++) {
 _mat<0,0,RNNWeight*> &weightSrc = recurDst ? recurWeights :
     weights;
 _tensor3<0,0,0,RNNHistory*> &historyDst = recurDst ?
     recurHistory : history;
 _tensor3<0,0,0,RNNHistory*> &lastHistoryDst = recurDst ?
     lastframe->recurHistory : lastframe->history;
 for (int m = 1; m < layers.dim(); m++) {
  int maxn = recurDst ? layers.dim() : m;
  for (int n = 0; n < maxn; n++) {
   if (!weightSrc(m,n)) continue;
   if ( ((GPURNNWeight*)weightSrc(m,n))->constant) continue;
   GPURNNLayer *xn = (recurDst ? lastframe : this)->layers[n];
   for (int i = 1; i < layers.dim(); i++) {
    GPURNNHistory *hmni = (GPURNNHistory *)historyDst(m,n,i);
    GPURNNLayer *xi = layers[i];
    vec3i ri(0,1,2);
    SetViewport sv(hmni->texs[0].texSize);
    for (int j = 0; j < hmni->texs.dim(); j++) {
     if (fbo.target(aux.regs[ri.x])) {
      glClearColor(0,0,0,0);
      glClear(GL_COLOR_BUFFER_BIT);
     }
     for (int recurSrc = 0; recurSrc < 2; recurSrc++) {
      int maxk = recurSrc ? layers.dim() : i;
```

```
const _tensor3<0,0,0,RNNHistory*> &historySrc = recurSrc ?
    lastHistoryDst : historyDst;
const _mat<0,0,RNNWeight*> &wikSrc = recurSrc ?
    recurWeights : weights;
for (int k = 1; k < maxk; k++) {
 const GPURNNWeight *wik = (GPURNNWeight*)wikSrc(i,k);
 if (!wik) continue;
 const GPURNNHistory *hmnk = (const GPURNNHistory *)
     historySrc(m,n,k);
 int l = 0;
 for (; l < (hmnk->texs.dim() >> 2); l++) {

  if (fbo.target(aux.regs[ri.y])) {
   SetShader ss(aux.backpropAccumShader[3]);
   ss.setUniform<float>(0, ((float)l + .5f) / (float)wik->
       tex.texSize.x);
   ss.setUniform<float>(1, ((float)j + .5f) / (float)wik->
       tex.texSize.y);
   SetTextures<6> st(aux.regs[ri.x], wik->tex,
    hmnk->texs[4*l], hmnk->texs[4*l+1], hmnk->texs[4*l+2],
        hmnk->texs[4*l+3]);

   st.draw(sv);
  }

  swap(ri.x, ri.y);
```

```
}
int lastchannels = hmnk->texs.dim() & 3;
if (lastchannels) {
 if (fbo.target(aux.regs[ri.y])) {
  SetShader ss(aux.backpropAccumShader[lastchannels-1]);
  ss.setUniform<float>(0, ((float)l + .5f) / (float)wik->
      tex.texSize.x);
  ss.setUniform<float>(1, ((float)j + .5f) / (float)wik->
      tex.texSize.y);

  switch (lastchannels) {
  case 1:
   {
    SetTextures<3> st(aux.regs[ri.x], wik->tex, hmnk->texs
        [4*l]);
    st.draw(sv);
   }
   break;
  case 2:
   {
    SetTextures<4> st(aux.regs[ri.x], wik->tex, hmnk->texs
        [4*l], hmnk->texs[4*l+1]);
    st.draw(sv);
   }
   break;
  case 3:
```

```
      {
       SetTextures<5> st(aux.regs[ri.x], wik->tex, hmnk->texs
           [4*l], hmnk->texs[4*l+1], hmnk->texs[4*l+2]);
       st.draw(sv);
      }
      break;
     }
    }
    swap(ri.x, ri.y);
   }
  }
}

if (fbo.target(hmni->texs[j])) {
 {
  SetTextures<1> st(aux.regs[ri.x]);
  st.draw(sv);
 }
 if (i == m) {
  SetShader ss(aux.backpropAccumDeltaShader);
  SetTextures<2> st(aux.regs[ri.x], xn->x);
  float y1 = (float)j / (float)hmni->texs[j].texSize.y;
  float y2 = (float)(j+1) / (float)hmni->texs[j].texSize.y;

  st.draw(sv, box2f(vec2f(0,y1), vec2f(1,y2)));
 }
```

```
      }
     }
     for (int j = 0; j < hmni->texs.dim(); j++) {
      if (fbo.target(aux.regs[ri.x])) {
       SetTextures<1> st(hmni->texs[j]);
       st.draw(sv);
      }
      if (fbo.target(hmni->texs[j])) {
       SetShader ss(aux.backpropScaleByDerivShader[j & 3]);
       ss.setUniform<float>(0, (float)hmni->texs[j].texSize.x / (
           float)aux.regs[ri.x].texSize.x);
       ss.setUniform<float>(1, (float)hmni->texs[j].texSize.y / (
           float)aux.regs[ri.x].texSize.y);
       ss.setUniform<float>(2, ((float)(j >> 2) + .5f) / (float)xi
           ->x.texSize.x);

       SetTextures<2> st(aux.regs[ri.x], xi->dXdNet);
       drawScreenQuad();
      }
     }
    }
   }
  }
 }
}
```

5.7.12 backpropAccumShader

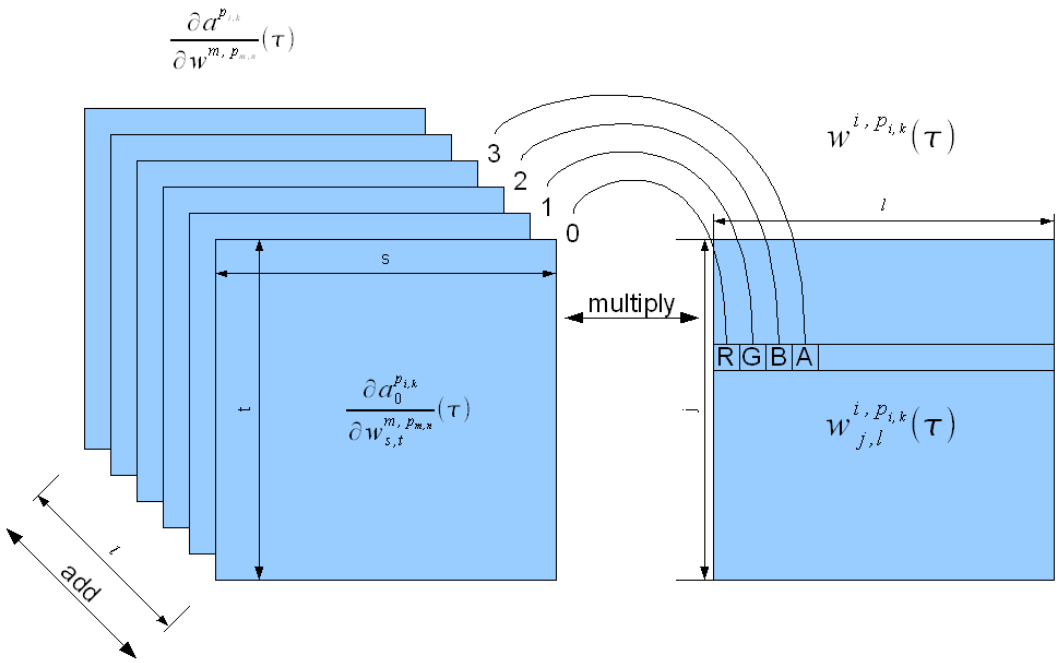

Figure 5.5: Visual example of the history tensor computation. The above shows the computation of the $\sum_{k,l} w_{j,l}^{i,p_{i,k}}(\tau)\xi_{s,t,l}^{m,p_{m,n},p_{i,k}}(\tau)$ component.

The `backpropAccumShader` is responsible for computing the sums within the history tensors. For $\xi_{s,t,j}^{m,p_{m,n},i}$ this includes the $\sum_{k,l} w_{j,l}^{i,p_{i,k}} \mid_{\tau} \xi_{s,t,l}^{m,p_{m,n},p_{i,k}} \mid_{\tau}$ (Eqn. 3.6) and $\sum_{k,l} \mu_{j,l}^{i,q_{i,k}} \mid_{\tau} \xi_{s,t,l}^{m,p_{m,n},q_{i,k}} \mid_{\tau-1}$ terms. For $\zeta_{s,t,j}^{m,p_{m,n},i}$ this includes the

$\sum_{k,l} w_{j,l}^{i,p_{i,k}} \mid_\tau \zeta_{s,t,l}^{m,q_{m,n},p_{i,k}} \mid_\tau$ and $\sum_{k,l} \mu_{j,l}^{i,q_{i,k}} \mid_\tau \zeta_{s,t,l}^{m,q_{m,n},q_{i,k}} \mid_{\tau-1}$ terms (Eqn. 3.8). The computation of the $\sum_{k,l} w_{j,l}^{i,p_{i,k}} \mid_\tau \xi_{s,t,l}^{m,p_{m,n},p_{i,k}} \mid_\tau$ sum is visually displayed in Figure 5.5.

The `backpropAccumShader[n]` is mapped such that the n'th entry corresponds to `CHANNELS_AT_ONCE = n+1`.

```
uniform sampler2D accumtex;
uniform sampler2D weighttex;

#if CHANNELS_AT_ONCE >= 1
uniform sampler2D hist0tex;
#endif
#if CHANNELS_AT_ONCE >= 2
uniform sampler2D hist1tex;
#endif
#if CHANNELS_AT_ONCE >= 3
uniform sampler2D hist2tex;
#endif
#if CHANNELS_AT_ONCE >= 4
uniform sampler2D hist3tex;
#endif

uniform float weighttcx;
uniform float weighttcy;
```

```
void main() {
 gl_FragColor = texture2D(accumtex, gl_TexCoord[0].st);
 vec2 weighttc = vec2(weighttcx, weighttcy);
 vec4 weightcolor = texture2D(weighttex, weighttc);
#if CHANNELS_AT_ONCE >= 1
 vec4 histcolor = texture2D(hist0tex, gl_TexCoord[2].st);
 gl_FragColor += histcolor * weightcolor.r;
#endif
#if CHANNELS_AT_ONCE >= 2
 histcolor = texture2D(hist1tex, gl_TexCoord[2].st);
 gl_FragColor += histcolor * weightcolor.g;
#endif
#if CHANNELS_AT_ONCE >= 3
 histcolor = texture2D(hist2tex, gl_TexCoord[2].st);
 gl_FragColor += histcolor * weightcolor.b;
#endif
#if CHANNELS_AT_ONCE >= 4
 histcolor = texture2D(hist3tex, gl_TexCoord[2].st);
 gl_FragColor += histcolor * weightcolor.a;
#endif
}
```

5.7.13 backpropAccumDeltaShader

The backpropAccumDeltaShader is then responsible for adding the last Kronecher delta terms. For $\xi_{s,t,j}^{m,p_{m,n},i}$ this is $\delta_{i,m}\delta_{j,s}a_t^{p_{m,n}} \mid_\tau$ (Eqn. 3.6) and for $\zeta_{s,t,j}^{m,p_{m,n},i}$ this is $\delta_{i,m}\delta_{j,s}a_t^{q_{m,n}} \mid_{\tau-1}$ (Eqn. 3.8).

```
uniform sampler2D accumtex;
uniform sampler2D rowtex;
void main() {
 vec4 accumcolor = texture2D(accumtex, gl_TexCoord[0].st);
 vec4 rowcolor = texture2D(rowtex, gl_TexCoord[1].st);
 gl_FragColor = accumcolor + rowcolor;
}
```

5.7.14 backpropScaleByDerivShader

The backpropScaleByDerivShader is responsible for applying the final scaling of $\xi_{s,t,j}^{m,p_{m,n},i} \mid_\tau = \frac{\partial net_j^i}{\partial w_{s,t}^{m,p_{m,n}}} \mid_\tau \cdot\sigma' net_j^i \mid_\tau$ (Eqn. 3.7) and $\zeta_{s,t,j}^{m,p_{m,n},i} \mid_\tau = \frac{\partial net_j^i}{\partial \mu_{s,t}^{m,q_{m,n}}} \mid_\tau \cdot\sigma' net_j^i \mid_\tau$ (Eqn. 3.9).

In prior versions I had only a single backpropScaleByDerivShader with a uniform int for accessing which channel to extract. NVIDIA's GLSL implementations only support the operator[] for vector element access when the value in the brackets is a constant int. Variables are not supported under NVIDIA. For

this reason I recreated the shader four times with each shader option accessing a different component of the result via the CHANNEL macro.

```
varying vec2 pos;
uniform sampler2D histtex;
uniform sampler2D dydnettex;
uniform float histscaletcx;
uniform float histscaletcy;
uniform float dydnettcx;

void main() {
 vec2 histtc = vec2(pos.x * histscaletcx, pos.y * histscaletcy);
 vec4 histcolor = texture2D(histtex, histtc);
 vec2 dydnettc = vec2(dydnettcx, .5);
 vec4 dydnetcolor = texture2D(dydnettex, dydnettc);
 float dydnet = dydnetcolor[CHANNEL];
 gl_FragColor = histcolor * dydnet;
}
```

5.7.15 GPURNN::updateWeights

The GPURNN::updateWeights function is a simple per-element scale between the last layer's $\frac{\partial y_j}{\partial w^{i,k}}$ and the provided $\frac{\partial E}{\partial y_j}$. Unlike feedforward weight updates the weight input vector needs not be considered. It is already incorporated into the history tensor. Instead a simple tensor product multiplication expressing

the derivative chain rule. Figure 5.6 shows this operation.

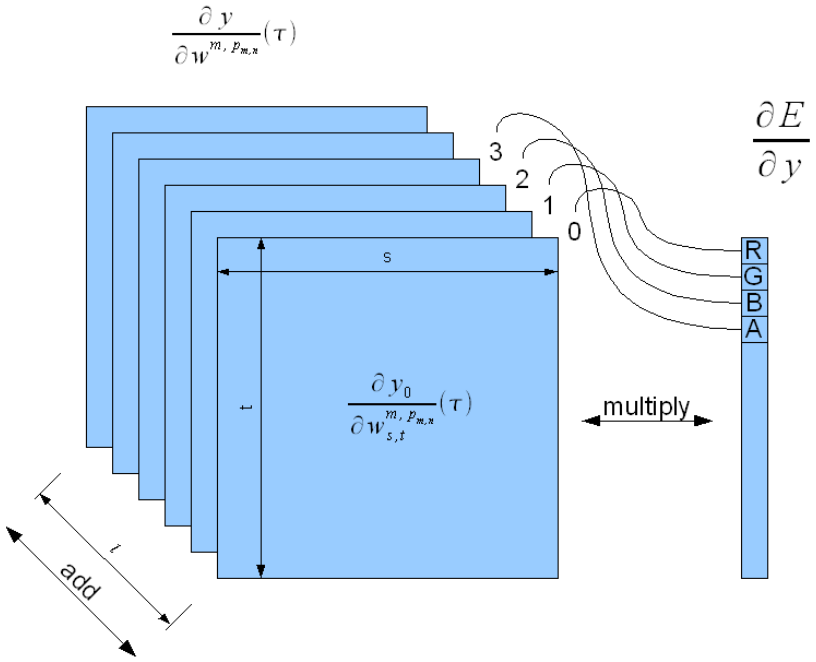

Figure 5.6: Example of the $\frac{\partial E}{\partial w}$ computation from the $\frac{\partial E}{\partial y}$ vector and the $\frac{\partial y}{\partial w^{m,p_{m,n}}}$ tensor. Each tensor slice is scaled by an associated component of the error vector and summed to produce the change in the $\partial w^{m,p_{m,n}}$ weight.

```
void GPURNN::updateWeights() {
 const int nextframeIndex = (CURRENT_FRAME + 1) % frames.dim();
```

```
 frames[CURRENT_FRAME]->updateWeights(aux, frames[nextframeIndex],
     dEdYTex, timestep);
}


void GPURNNTimeframe::updateWeights(GPURNNAuxInfo &aux,
   GPURNNTimeframe *nextframe, const FloatTex &dEdYTex, float
   stepSize) {
 SetFBO fbo(aux.fboID);
 vec3i ri(0,1,2);
 for (int m = 1; m < layers.dim(); m++) {
  for (int n = 0; n < layerInfos[m].inputs.size(); n++) {
   GPURNNHistory *hmny = (GPURNNHistory *)layerInfos[m].inputs[n].
       dydw;
   GPURNNWeight *wmnRead = layerInfos[m].inputs[n].w;
   GPURNNWeight *wmnWrite = nextframe->layerInfos[m].inputs[n].w;
   const GPURNNLayer *y = layers[layers.dim()-1];
   int ysize = y->size();
   if (!ysize) continue;
   int basemaxi = ysize >> 2;
   int maxi = basemaxi + !!(ysize & 3);
   SetViewport sv(wmnWrite->tex.texSize);
   FloatTex *accumReadTex = &wmnRead->tex;
   int i = 0;
   for (; i < maxi; i++) {
    int channels = i < basemaxi ? 4 : ysize & 3;
    bool lastpass = i == maxi - 1;
```

```
FloatTex &desttex = lastpass ? wmnWrite->tex : aux.regs[ri.x];
if (fbo.target(desttex)) {
 SetShader ss(aux.updateWeightsAccumShader[channels-1]);
 ss.setUniform<float>(0, stepSize);
 ss.setUniform<float>(1, (float)sv.size.x / (float)
    accumReadTex->texSize.x);
 ss.setUniform<float>(2, (float)sv.size.y / (float)
    accumReadTex->texSize.y);
 ss.setUniform<float>(3, ((float)i + .5f) / (float)y->x.
    texSize.x);
 glActiveTextureARB(GL_TEXTURE0_ARB);
 glBindTexture(GL_TEXTURE_2D, accumReadTex->texID);
 glActiveTextureARB(GL_TEXTURE1_ARB);
 glBindTexture(GL_TEXTURE_2D, dEdYTex.texID);
 glActiveTextureARB(GL_TEXTURE2_ARB);
 glBindTexture(GL_TEXTURE_2D, hmny->texs[4*i].texID);
 if (channels >= 2) {
  glActiveTextureARB(GL_TEXTURE3_ARB);
  glBindTexture(GL_TEXTURE_2D, hmny->texs[4*i+1].texID);
  if (channels >= 3) {
   glActiveTextureARB(GL_TEXTURE4_ARB);
   glBindTexture(GL_TEXTURE_2D, hmny->texs[4*i+2].texID);
   if (channels == 4) {
    glActiveTextureARB(GL_TEXTURE5_ARB);
    glBindTexture(GL_TEXTURE_2D, hmny->texs[4*i+3].texID);
   }
```

```
      }
     }

     drawScreenQuad();
    }
    if (!lastpass) {
     swap(ri.x, ri.y);
     accumReadTex = &aux.regs[ri.y];
    }
   }
  }
 }
 for (int i = 7; i >= 0; i--) {
  glActiveTextureARB(GL_TEXTURE0_ARB + i);
  glBindTexture(GL_TEXTURE_2D, 0);
 }
}
```

5.7.16 updateWeightsAccumShader

The `updateWeightsAccumShader` operates by scaling each component of the $\frac{\partial E}{\partial y_l}$ vector with its matching $\frac{\partial y_l}{\partial w^{m,p_{m,n}}}$ slice of the history computed previously in `GPURNN::updateHistory`.

```
varying vec2 pos;
```

```
uniform sampler2D accumtex;
uniform sampler2D dedytex;
#if CHANNELS_AT_ONCE >= 1
uniform sampler2D dydw0tex;
#endif
#if CHANNELS_AT_ONCE >= 2
uniform sampler2D dydw1tex;
#endif
#if CHANNELS_AT_ONCE >= 3
uniform sampler2D dydw2tex;
#endif
#if CHANNELS_AT_ONCE >= 4
uniform sampler2D dydw3tex;
#endif

uniform float stepSize;
uniform float accumscaletcx;
uniform float accumscaletcy;
uniform float ytcx;

void main() {
 vec2 accumtc = vec2(pos.x * accumscaletcx, pos.y * accumscaletcy)
    ;
 gl_FragColor = texture2D(accumtex, accumtc);
 vec2 ytc = vec2(ytcx, .5);
 vec4 dedydtcolor = texture2D(dedytex, ytc) * stepSize;
```

```
#if CHANNELS_AT_ONCE >= 1
 vec4 dydwcolor = texture2D(dydw0tex, pos);
 gl_FragColor -= dedydtcolor.r * dydwcolor;
#endif
#if CHANNELS_AT_ONCE >= 2
 dydwcolor = texture2D(dydw1tex, pos);
 gl_FragColor -= dedydtcolor.g * dydwcolor;
#endif
#if CHANNELS_AT_ONCE >= 3
 dydwcolor = texture2D(dydw2tex, pos);
 gl_FragColor -= dedydtcolor.b * dydwcolor;
#endif
#if CHANNELS_AT_ONCE >= 4
 dydwcolor = texture2D(dydw3tex, pos);
 gl_FragColor -= dedydtcolor.a * dydwcolor;
#endif
}
```

5.8 GPU With Unravelled Loops

Early hardware running fragment programs was slowed heavily by branching. Even in later hardware such as the NVIDIA Tesla, when two threads in a warp branch in different ways, there is a major slowdown. Because of this, for-loops, which require a compare every iteration, offer significant potential damage to shader performances. One solution for this is to unravel the loops

in your shader code. This is only effective if the size of the loop is fixed at compile time. This restriction was accommodated for by dynamically compiling the shader kernels during the RNN class initialization. From this point the networks were subsequently fixed in size. Any such modifications to any layer sizes would require the shader code to be recompiled.

A benefit of writing out unravelled loops of iteration across the shaders is that they remove our dependency upon the ping-pong effect. This reduces the number of passes required to complete our operation and increases performance. A downside to this is that a separate unique shader needs to be written for each layer. Another restriction to loop unravelling in the shader is that, for each new unravelled line of code, a new set of instructions must be generated for the GPU shader. This poses a threat to earlier graphics cards whose memory size was restricted.

In our RNN implementation, the implementation of unravelled loops only occurs to replace the $log(n)$-pass `rowReduce` method. The two methods that previously relied on this were `GPURNNTimeframe::feedForward` and `GPURNN::calcError`. In place of each `rowReduce` call is a reference to a dynamically-generated shader that is created during the `GPURNN::prepare` method.

5.8.1 GPURNNTimeframe::feedForward

The `GPURNNTimeframe::feedForward` is changed in our implementation of unravelled loops. Whereas before four different shaders had to be made use of to complete the operation, now only one is: `layerShader->feedForwardShader`. This saves in the number of passes for the kernel to run over the data. The tradeoff is, however, that more texture units need to be utilized for this single pass. If more units are required than the graphics card allows, the network topology can be redesigned to work equivalently while still working on hardware with such restrictions.

Unique shaders are generated per-layer. The number of input texture units is proportional to the number of input layers to that layer. The shader passes along the texture of the output layer, performing lookups across the width of the matrix.

```
void GPURNNTimeframe::feedForward(GPURNNTimeframe *lastframe,
    GPURNNAuxInfo &aux) {
 glColor3f(1,1,1);
 setUnitViewport();
 SetFBO fbo(aux.fboID);
 for (int i = 1; i < layers.dim(); i++) {
  GPURNNTimeframeLayerInfo &layerInfo = layerInfos[i];
  GPURNNLayer *y = layers(i);
```

```
GPURNNAuxLayerShaderInfo *layerShader = aux.layerShaders[i];
if (fbo.target(y->net, y->x, y->dXdNet)) {
 {
  SetViewport sv(layerShader->viewportSize);
  SetShader ss(layerShader->feedForwardShader);
  for (int j = 0; j < layerInfo.inputs.size(); j++) {
   glActiveTextureARB(GL_TEXTURE0_ARB + 2 * j);
   glBindTexture(GL_TEXTURE_2D, layerInfo.inputs[j].x->x.texID);

   glActiveTextureARB(GL_TEXTURE0_ARB + 2 * j + 1);
   glBindTexture(GL_TEXTURE_2D, layerInfo.inputs[j].w->tex.texID
       );
  }
  drawScreenQuad();
  for (int j = 0; j < 2 * layerInfo.inputs.size(); j++) {
   glActiveTextureARB(GL_TEXTURE0_ARB + j);
   glBindTexture(GL_TEXTURE_2D, 0);
  }
 }
 {
  SetViewport sv(y->x.texSize);
  glColorMask(
   (y->size() & 3) == 0,
   (y->size() & 3) == 1,
   (y->size() & 3) == 2,
   (y->size() & 3) == 3);
```

```
    glBegin(GL_QUADS);
    int qsize = y->size() >> 2;
    float x1 = (float)qsize / (float)y->x.texSize.x;
    float x2 = (float)(qsize + 1) / (float)y->x.texSize.x;
    glVertex2f(x1, 0);
    glVertex2f(x1, 1);
    glVertex2f(x2, 1);
    glVertex2f(x2, 0);
    glEnd();
    glColorMask(1,1,1,1);
   }
  }
 }
}
```

5.8.2 GPURNN::calcError

True to the prior demonstration of loop optimizations in the shader, the `GPURNN::calcError` can likewise be optimized into a single pass operation. The ping-pong based method required one pass for the difference to be calculated and another $log(n)$ passes to reduce the sum via `rowReduce`. The unravelled shader runs completely inside of `calcErrorShader`, which is generated per-network during the `GPURNN::prepare` method.

```
float GPURNN::calcError() {
 GPURNNTimeframe *frame = (GPURNNTimeframe*)getCurrentFrame();
 GPURNNLayer *outputLayer = (GPURNNLayer*)frame->getOutputLayer();
 SetFBO fbo(aux.fboID);
 if (fbo.target(aux.regs[0])) {
  SetShader ss(aux.calcErrorShader);
  SetTextures<2> st(outputLayer->x, desTex);
  glViewport(0,0,1,1);
  drawScreenQuad();
 }
 float e = -fInf;
 glReadPixels(0,0,1,1, GL_RED, GL_FLOAT, &e);
 return e;
}
```

5.9 GPU With For Loops

The addition of for-loops is a harmless step up from unravelled loops generated in shader programs. For loop upper bounds are generated uniquely to each shader in each layer. This can, in theory, allow the shader compiler the opportunity to perform the same unravelling that I otherwise would've. The benefit to using this over explicitly unravelled loops is that the number of shader instructions blooms at a rate of $O(1)$ rather than $O(n)$ with respect to the number of inputs used.

5.10 $\frac{\partial E}{\partial net}$ Optimization

To circumvent the weighty use of tensors, it is possible to optimize in the user of the $\frac{\partial E}{\partial net}$ wherever able. This can be found by checking all paths from a weight's output and seeing if any of them are the destination of a recurrent connection's edge. If they are, then a cycle in derivative computation dependencies exists.

The acyclic paths can be optimized in the old fashioned manner. Back-propagation, an $O(n^2)$ algorithm, can be applied from the output node back to each of these nodes before the general recurrent $O(n^3)$ algorithm is applied to all weights which subsequently connect into this layer. This completely reduces the algorithm from $O(n^3)$ to $O(n^2)$ for all networks which do not make use of recurrent connections.

In this project the $\frac{\partial E}{\partial net}$ optimization was implemented for both CPU, GPU with unravelled loops, and GPU with explicit loops.

5.11 Future Optimization

There are still several avenues of optimization that I have not yet pursued. The above $\frac{\partial E}{\partial net}$ optimization does reduce the complexity of certain edges with acyclic paths to the y layer from $O(n^3)$ to $O(n^2)$. It still leaves all the remaining

edges in need of the original full forward pass to compute $\frac{\partial a}{\partial w}$. There still may still be means of optimizing out certain redundant calculations in the tensor product chain rule.

The implementation can be improved upon in several different areas in regards to hardware as well. One such means would be to design a proxy network class to abstract any hardware limitations that the network might face, such as a limit on texture units or on texture sizes. Another feature still to be implemented is the ability to let the user define what transfer function they would like to use between layers. Further advances could include automatic gradient generation – even through arbitrary rank tensors and functions.

Chapter 6 – Results

For this project several examples were implemented to benchmark the GPU RNN in different circumstances. For each example I will list out comparisons between the performance with CPU and GPU versions. For certain examples I will also list what preprocessor macros were applied as well.

The following configurations will be benchmarked: CPU, GPU with ping-pong reductions, GPU with unravelled loops, GPU with for-loops. The same tests will be composed again with the $\frac{\partial E}{\partial net}$ optimizations applied.

6.1 Accuracy

GPU and CPU floating point implementations can vary slightly in their rounding and accuracy. For this reason verifying their accuracy was pretty important. Shadowing is a condition in mathematics when a simulation varies slightly from its true algebraic value due to rounding errors. Dynamic system theory reminds us that these errors can grow exponentially within our system. RNNs, thanks to their feedback nature, especially fall into this category.

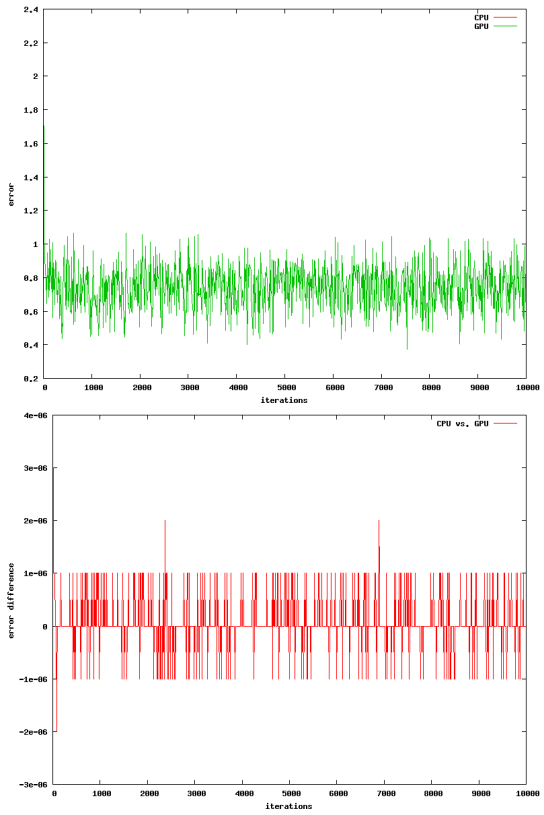

Figure 6.1: An example of a sequence for which the CPU and GPU values align closely

6.2 Performance

For our performance example two networks were used: one to test feedforward performance and one to test recurrence performance. The feedforward network consisted of two weight connections between an input layer of size 2n, a hidden layer of size 2n, and an output layer of size n. The recurrent network consisted of the same problem in the form of an Elman network, that is to say

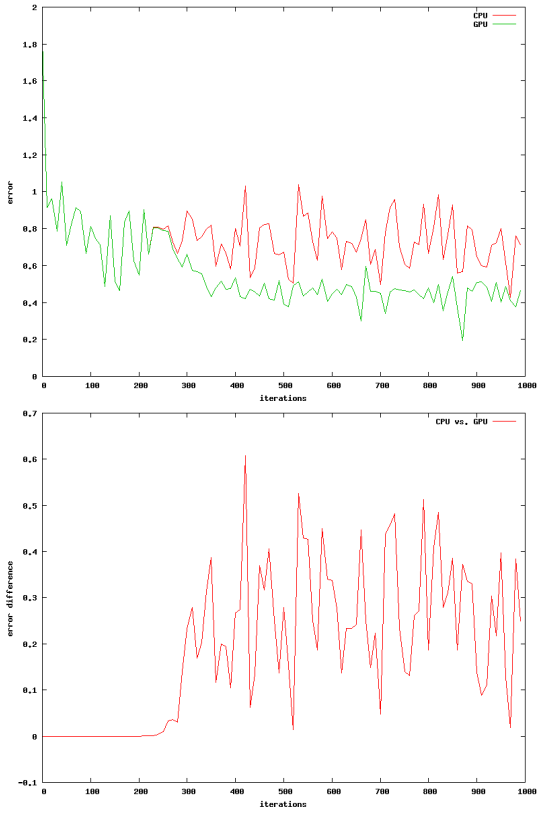

Figure 6.2: An example of a sequence for which the CPU and GPU values eventually diverge

it had an addition recurrent connection from the hidden layer back into itself. The networks were trained to an addition problem, feeding in two numbers of size of n-bits, in binary, and training the network to its n-bit sum, in binary.

We start with an unoptimized CPU implementation of the $O(n^3)$ $\frac{\partial a}{\partial w}$ implementation. Converting this algorithm to the GPU with ping-pong reduction improves performance times dramatically, but not for small vector sizes due to

the overhead of using the GPU. Next we replace the ping-pong reduction with unravelled loop reduction. Finally the unravelled reduction loop is replaced with a for-loop in the shader.

Figure 6.3 shows the comparison of algorithms on the ATI X1600. This is running without subsequent $\frac{\partial E}{\partial net}$ optimizations. Note how the GPU quickly outperforms the CPU as the layer sizes are progressively increased. Also note the stair-stepping in time taken that occurs exponentially further apart as bit size is increased. This is due to the fact that all texture objects are created with dimensions rounded up to the next power of two in size.

Figure 6.4 shows the comparison of algorithms on the NVIDIA GeForce 8600 GTS, demonstrating the same tradeoff of performance gain between the CPU and GPU as the layer sizes increase.

The CPU performs better for initial network sizes but is soon undercut by the parallelism of the GPU. Soon after the GPU maintains better performance for larger networks within which more data is computed in parallel. It doesn't appear that there is much performance difference between different GPU implementations, however some implementations have different restrictions. The unravelled loops implementation can be seen to succumb to the listed restrictions as the shader program instruction count reaches a critical capacity: on the ATI the GPU fails to execute its fragment shader for problems of size 67

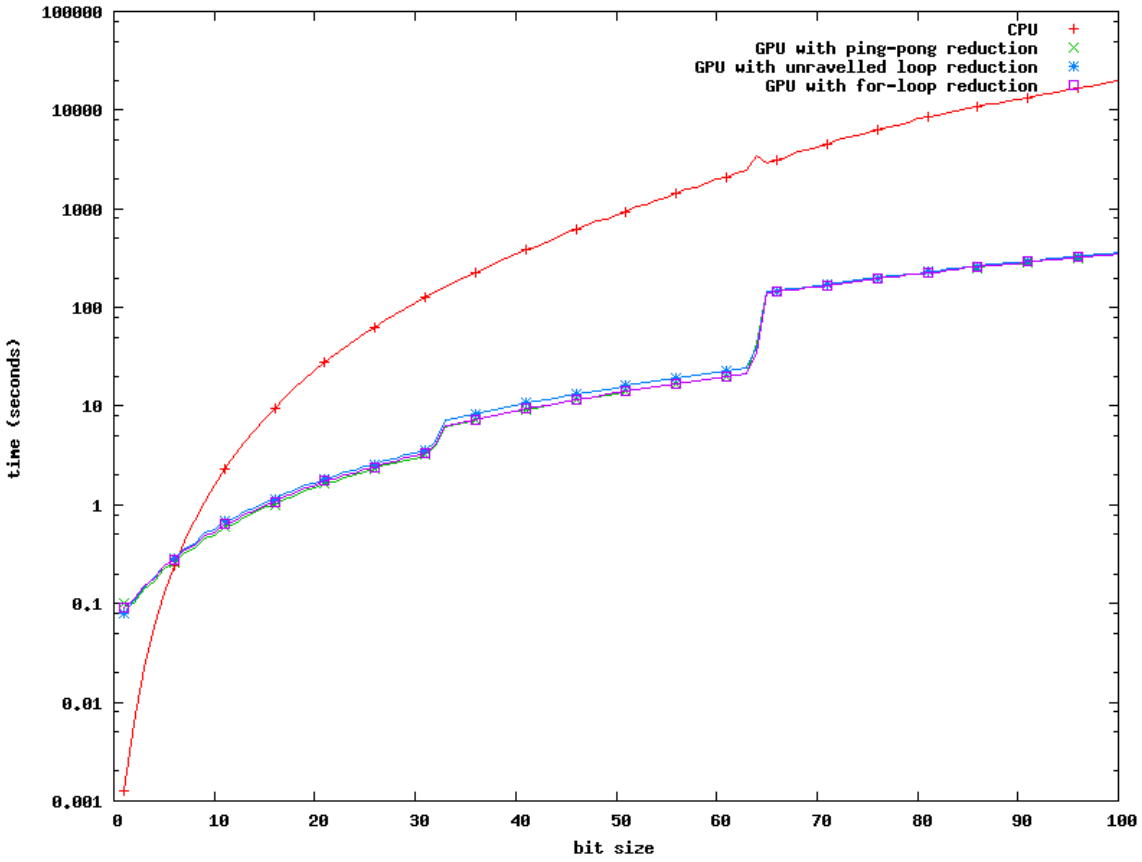

Figure 6.3: Logarithmic plot of time versus bit size. Performance comparison for an Elman network on an 1.83 GHz Intel dual core with 2GB RAM and an ATI X1600 graphics card. Stair-stepping of performance as bit size increases exponentially is due to the textures being rounded up to the next power of two in size.

bits and greater. Rendering of the fragments still occurs but the results stop converging.

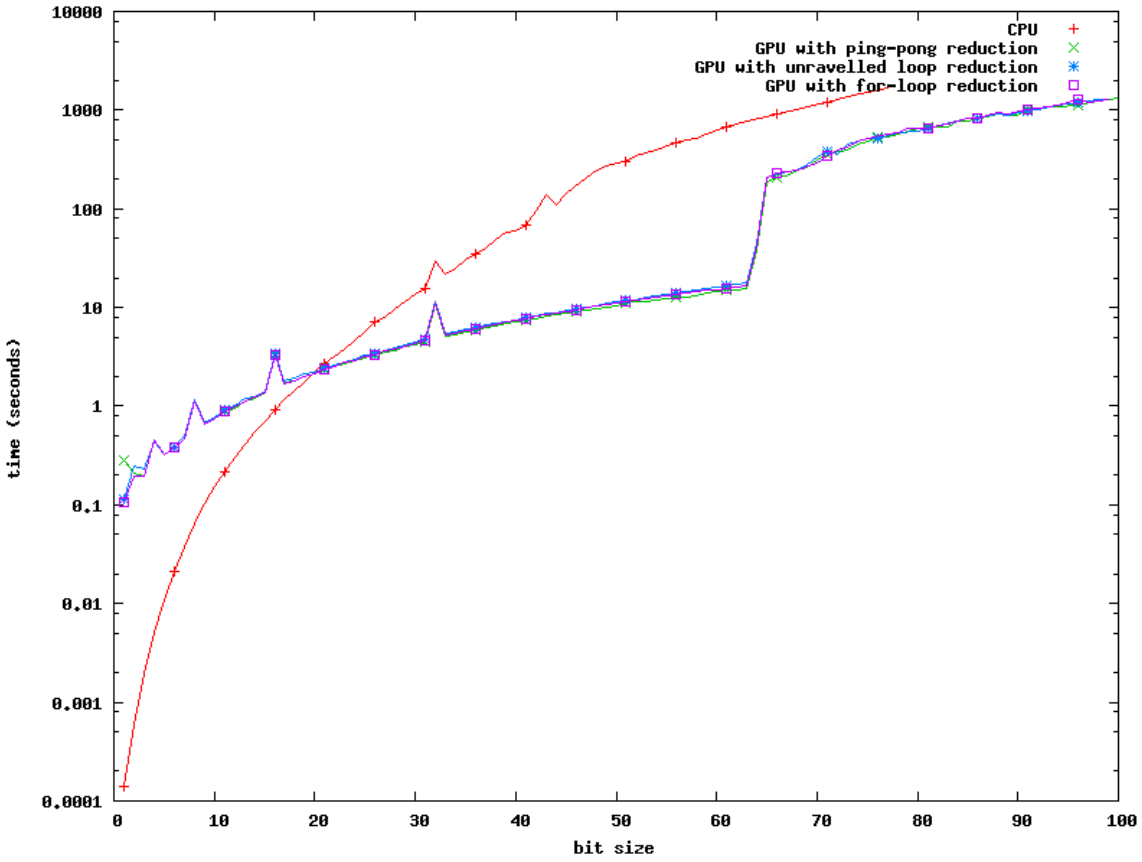

Figure 6.4: Logarithmic plot of time versus bit size. Performance comparison for an Elman network on an 3.20 GHz Intel dual core with 2GB RAM and a NVIDIA GeForce 8600 GTS graphics card.

Introducing $\frac{\partial E}{\partial net}$ optimizations gives a noticeable performance increase. Figures 6.5 and 6.6 shows this. The optimized CPU implementation performs 10x to 100x faster than the fastest unoptimized GPU implementation. It also performs better than the optimized GPU implementation for the first few bit sizes. This was true as well for the unoptimized version: due to constant-time

GPU overhead the CPU is more efficient for smaller network sizes, though for larger networks the GPU performs better.

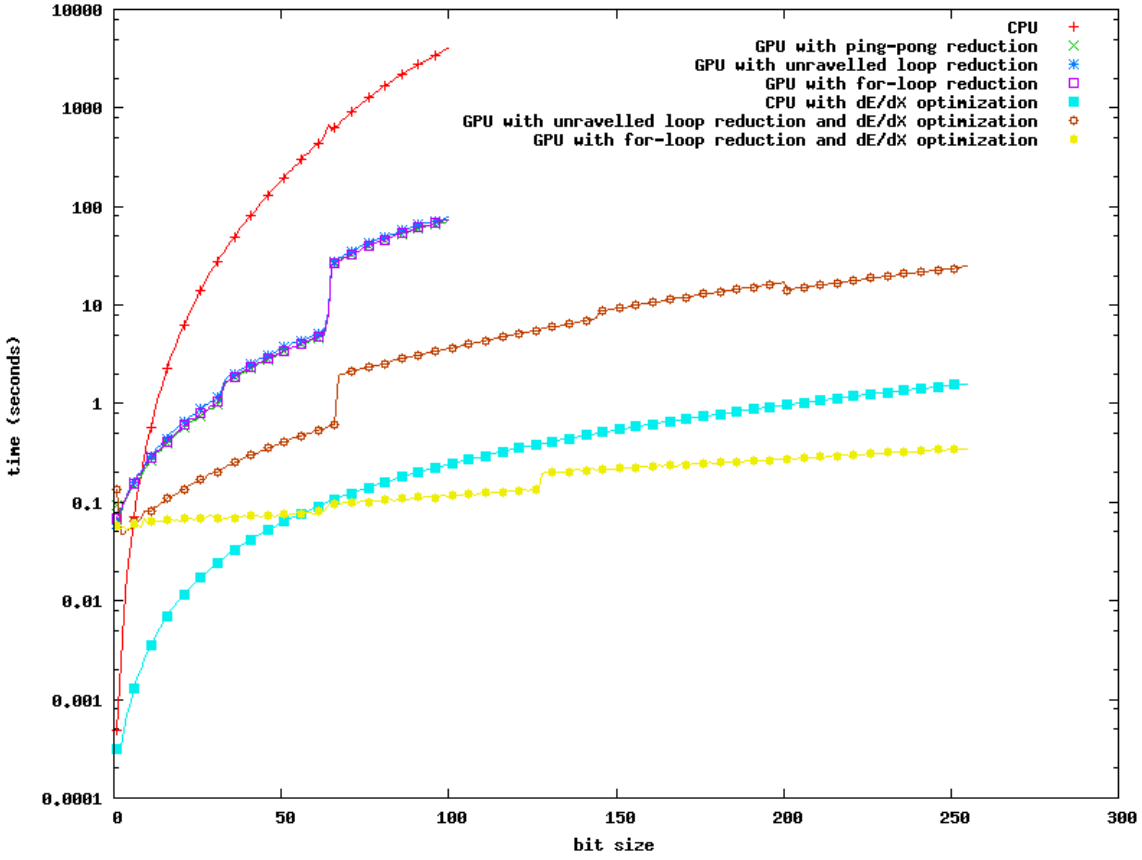

Figure 6.5: Logarithmic plot of time versus bit size. Performance comparison for a feedforward network between unoptimized computation and $\frac{\partial E}{\partial net}$ optimized computation on an 1.83 GHz Intel dual core with 2GB RAM and an ATI x1600 graphics card.

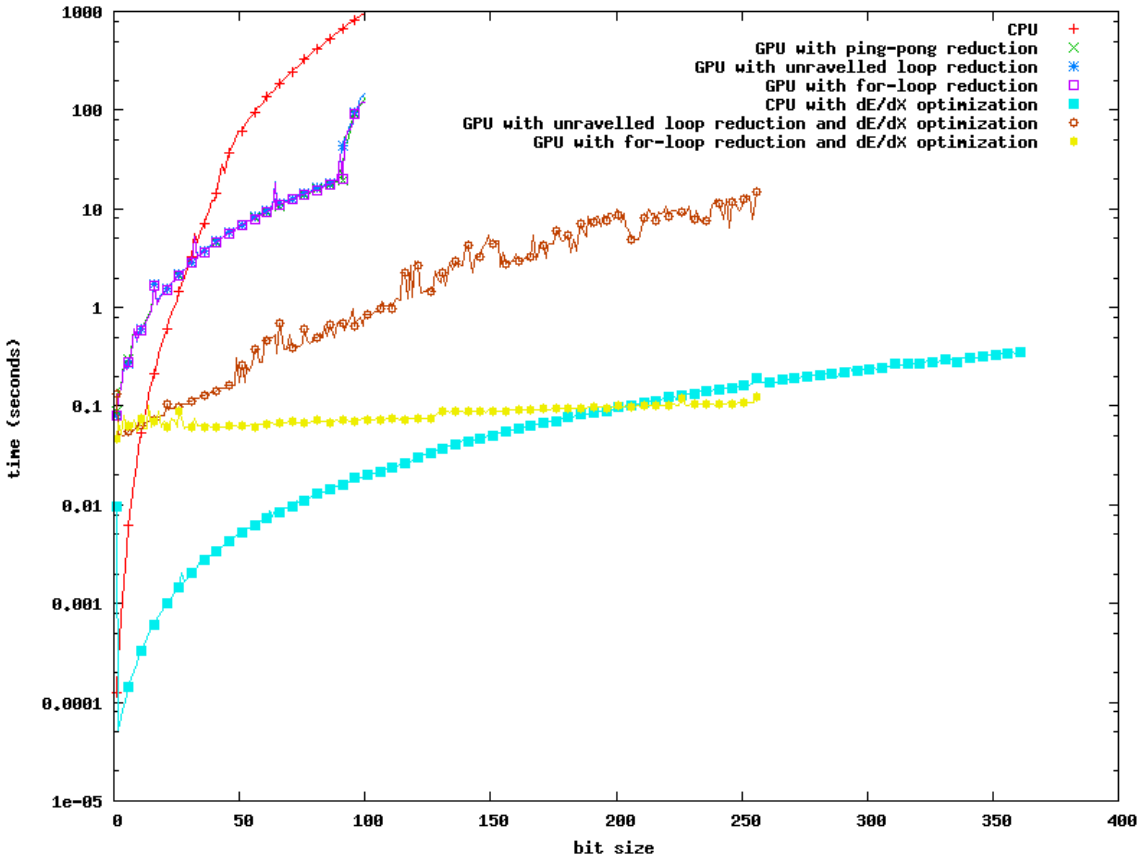

Figure 6.6: Logarithmic plot of time versus bit size. Performance comparison for a feedforward network between unoptimized computation and $\frac{\partial E}{\partial net}$ optimized computation on an 3.20 GHz Intel dual core with 2GB RAM and a NVIDIA GeForce 8600 GTS graphics card.

6.2.1 Hardware Trends

As GPU power increases this implementation's effectiveness will also better perform. Figure 6.7 shows the increase in performance over a span of different platforms. The graph shows the ATI X1600 holds a performance advantage

over the later generation NVIDIA GeForce chipsets for small vector sizes. However as the system size increases, the performance favors the newer chips.

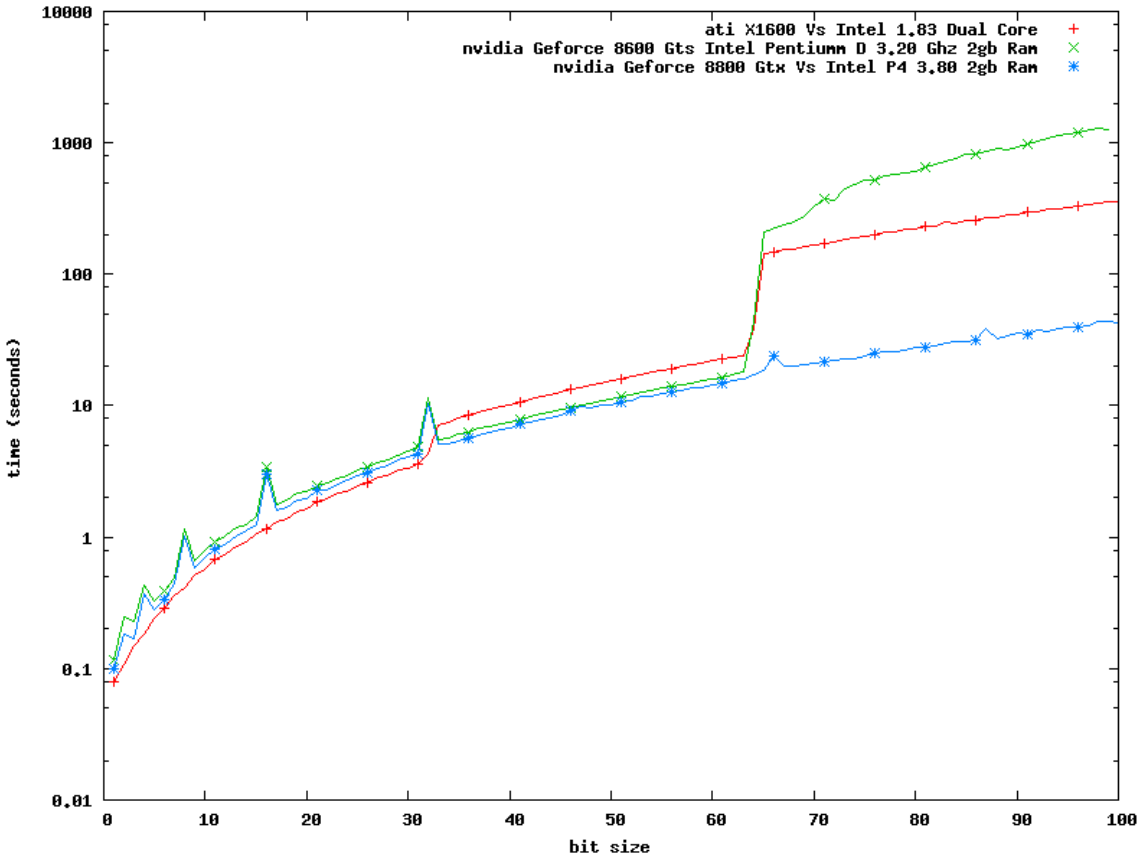

Figure 6.7: Logarithmic plot of time versus bit size. Comparison between different machines of the GPU ping-pong implementation.

Chapter 7 – Conclusion

For recurrent systems there is an eventual tradeoff in performance between the CPU and GPU implementations. For the ATI x1600 card, for networks of dimension less than 20, the CPU outperforms the GPU. Beyond that the GPU maintains a better speed, even up to as much as 100 times for significantly large vectors (of 200 dimensions or more).

For feed-forward systems the CPU maintains better performance for small dimensions as well. On the NVIDIA GeForce 8600 GTS the point of trade-off occurred at 35 dimensions. The performance benefit of the GPU implementation over the CPU does not provide as much of a benefit as it does for the recurrent network. The GPU best outperforms the CPU at a dimensionality just below 130.

Overall it appears the GPU based RNN implementation outperforms CPU based implementations for networks with dimensions above 30 or so. For smaller networks the CPU implementation is optimal to use. For larger, however, the GPU can outperform the CPU by rates of up to 100 times faster.

Bibliography

[1] Nvidia cuda compute unified device architecture - programming guide. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[3] Maria Charalambous, Pedro Trancoso, and Alexandros Stamatakis. Initial experiences porting a bioinformatics application to a graphics processor. *Panhellenic Conference on Informatics*, 3746:415–425, 2005.

[4] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[5] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, March 1995.

[6] Govindaraju, Naga K., Lloyd, Brandon, Wang, Wei, Lin, Ming, Manocha, and Dinesh. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM.

[7] Harris and Mark. Gpu physics. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 15, New York, NY, USA, 2007. ACM.

[8] Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, New York, June 1949.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[10] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. pages 112–127, 1990.

[11] Jian-Ming Li, Xiao-Jing Wang, Rong-Sheng He, and Zhong-Xian Chi. An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *NPC '07: Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, pages 855–862, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Marco Maggini. Recursive neural networks and automata. In *Adaptive Processing of Sequences and Data Structures, International Summer School on Neural Networks, "E.R. Caianiello"-Tutorial Lectures*, pages 248–295, London, UK, 1998. Springer-Verlag.

[13] Gordon E. Moore. Progress in digital integrated electronics. *IEEE, IEDM Tech Digest*, pages 11–13, 1975.

[14] Aaftab Munshi. Opencl: Parallel computing on the gpu and cpu. http://s08.idav.ucdavis.edu/munshi-opencl.pdf, 2008.

[15] K.S. Oh and K.C. Jung. Gpu implementation of neural networks. *PR*, 37(6):1311–1314, June 2004.

[16] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[17] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks, 1989.

[18] Luo Zhongwen, Liu Hongzhi, Yang Zhengping, and Wu Xincai. Self-organizing maps computing on graphic process unit.