School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Implementing a simple neural network in Accelerate

by

Ji Yong Jeong

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: May 30, 2017 Student ID: z2250189

Supervisor: A/Prof. Gabriele Keller Topic ID: 3733

Abstract

Parallel programming, including GPU programming, is commonly used to speed up neural network computations. However, the main languages for GPU programming are CUDA and OpenCL, which are very low-level languages that are cumbersome to use. This thesis includes the implementation of a feed-forward back-propagation neural network using Accelerate. Accelerate is an Embedded Domain-Specific Language in Haskell for high performance computing and has several accessibility advantages over CUDA, while still offering competitive performance on both GPUs and CPUs using LLVM. I assess its performance, benefits and disadvantages against the traditional approaches with CUDA and other languages. Future work includes the implementation of more sophisticated networks such as convolutional neural networks on top of this foundation.

Acknowledgements

First, I would like to express my deepest thanks to my supervisor, Gabriele Keller for her gentle encouragements, patience and guidance. I would like to thank Trevor McDonnell and Liam O'Connor for their encyclopedic knowledge and superbness.

Also much love and thanks to my parents and sister for their long-suffering patience and support.

Abbreviations

BGD Batch gradient descent learning algorithm

CUDA Computer Unified Device Architecture

DNN Deep neural network

EDSL Embedded domain-specific language

FFBP Feed-forward back-propagation algorithm

GPU Graphics Processing Unit

OpenCL Open Computing Language

PLS UNSW Programming Languages and Systems group

ReLU Rectified Linear Unit activation function

SGD Stochastic gradient descent learning algorithm

Contents

1 Introduction **1**

2 Background and Related Works **3**

2.1 Neural network architecture 3

2.2 Feed-forward back-propagation learning algorithm 6

2.3 GPU-accelerated programming and neural networks 9

2.4 Accelerate 10

2.5 Previous Implementations in Accelerate 13

3 Implementation **16**

3.1 Using a MATLAB-based neural network as reference 16

3.2 Accelerate Implementation 17

3.2.1 Neural network structure 17

3.2.2 Program structure 18

3.3 Known bugs 24

3.4 Other works 26

4 Testing and Results **27**

4.1 Test Data 27

4.2 Testing environment 28

4.3 Performance results 28

5 Evaluation **31**

5.1 Performance evaluation 31

5.2 Usability and Accelerate 33

5.3 Advantages of an Accelerate implementation 35

5.4 Incomplete works 36

5.5 Future works 38

6 Conclusion **39**

Bibliography **40**

List of Figures

2.1 Structure of a modern unit by [Kar16]. 4
2.2 Graphs of some common activation functions [Kar16]. 5
2.3 An example of a 2-layer neural network [Kar16]. 6
2.4 Structure overview of `Data.Array.Accelerate`. [CKL+11] 10
2.5 Types of array shapes and indices [CKL+11]. 11
2.6 Some Accelerate functions [Mar13]. 12
2.7 Key Accelerate functions used in [CEMCK17]. 14
2.8 Support functions for forward- and back-propagation used in [CEMCK17]. 14
2.9 Accelerate and Haskell code are mixed in [CEMCK17]. 15

3.1 Changing the number of hidden layers in `nnCostFunction` 18
3.2 Reshaping vectors into matrices in Accelerate. 21
3.3 Function minimise conjugate gradient function. 23
3.4 Assisting Accelerate by specifying types of unused variables in `unlift`. . 24
3.5 MATLAB's `double` is actually a `float`? What it means for `realmin`. . . 25

4.1 Graphs of training set 1 (top) and 2 (bottom) benchmarks. 30

5.1 Comparing `predict` in MATLAB [Ng16], Accelerate and C++ [Wol17]. 34

Chapter 1

Introduction

Neural networks are widely used for computer vision and one of the best methods for most pattern recognition problems [NVI14]. For instance, Deep neural networks (DNN) can already perform at human level on tasks such as handwritten character recognition (including Chinese), various automotive problems and mitosis detection.

One issue with DNN is its compute-intensiveness. Training a neural network with massive numbers of features require a lot of computations. Unfortunately, the most efficient and economical approach to testing the validity of many hypotheses is repeated trial-and-error [Ng16].

For the above reason, neural networks are commonly engineered to make use of multi-core parallelism, on both CPUs and GPUs. GPUs offer thousands of cores, and as such GPU-accelerated neural networks are generally faster than neural networks on a CPU cluster. The main languages for GPU programming are the Compute Unified Device Architecture (CUDA) from NVIDIA and the Open Computing Language (OpenCL). Both are very low-level languages based on C/C++.

On the other hand, Accelerate is a embedded domain-specific language (EDSL) created for high-performance GPU and CPU programming inside Haskell, with higher level semantics and cleaner syntax, while still offering competitive performance.

Thus, the motivations for this thesis is to explore the feasibility of implementing a neural network in a more on-the-fly, user-friendly approach using Accelerate. If successful, it may enable us test neural network hypotheses in a more convenient manner.

As an initial prototype, a feed-forward back-propagation (FFBP) neural network implementation has recently been made [CEMCK17], but it suffers from very poor accuracy. My project is to construct a network with competitive accuracy and performance to existing mature network implementations.

The following section, Chapter 2 outlines the background relating to this topic, from a general overview of neural networks and the mathematics behind FFBP algorithm to an overview of the Accelerate language and the previous implementation using Accelerate.

Chapter 3 introduces my implementation, discusses some issues in encoding the algorithm in Accelerate's combinator language, and identifies areas that need further development. Chapter 4 discusses the testing and analysis of my network implementation, and Chapter 5 compares and contrasts my implementation with other mature neural network implementations in terms of accuracy, speed, and scalability.

Finally, Chapter 6 summarises the contents of this report.

Chapter 2

Background and Related Works

2.1 Neural network architecture

Broadly speaking, a neural network can be described as a certain layering of nodes, or *units*, connected to each other by directed *links*, where each link has a certain numeric weight that signifies the strength of connection between the connected nodes.

Historically, the concept of 'net of neurons' whose interrelationship could be expressed in propositional logic was first proposed by [MP43] in 1943, inspired by a "all-or-none" behaviour of the biological nervous system. The first basic unit, also called the *perceptron*, was invented by [Ros62]. A perceptron will be activated if the sum of all the input values from its input links, say $x_1, ..., x_m$, multiplied by the links' corresponding weights, say $\theta_1, ..., \theta_m$, is above that unit's certain threshold value, or *bias* b, such that

$$\text{output} = \begin{cases} 0 & \text{if } \sum_{i=1}^{m} x_i \theta_i \leq b \\ 1 & \text{otherwise} \end{cases}$$

The above is equivalent to vectorizing the inputs to $\boldsymbol{x} = [x_1, ..., x_m]$, weights to $\boldsymbol{\theta} =$

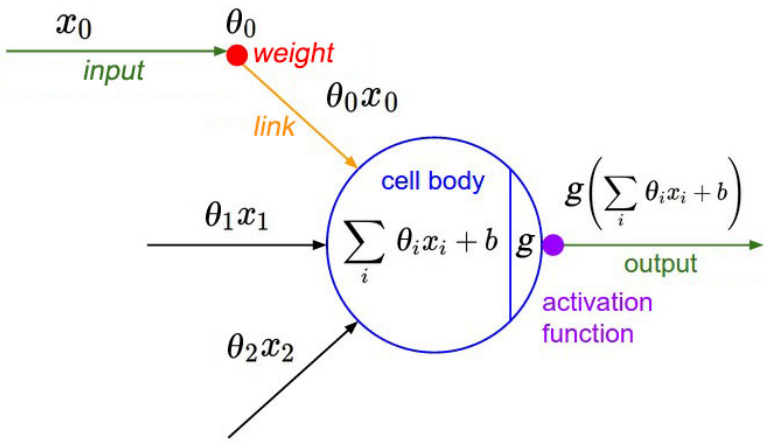

Figure 2.1: Structure of a modern unit by [Kar16].

$[\theta_1, ..., \theta_m]$ and inverting the sign on b, so that,

$$\text{output} = \begin{cases} 0 & \text{if } \boldsymbol{x} \cdot \boldsymbol{\theta} + b \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

The perceptron eventually evolved into the modern unit, which computes the output value as a *range* of values, obtained by applying an *activation function* to the sum of its inputs and bias. With this modification, a small change in the inputs only resulted a small change in the output, allowing a more convenient way to gradually modify the weights and consequently, improve the learning algorithm [Nie15].

There are various activation functions; historically, the most commonly used is the *sigmoid* function, $\sigma(x) = 1/(1+e^{-x})$. Its advantages and disadvantages are outlined in 2.5. [Kar16] recommends using less expensive functions with better performance, such as,

1. Tanh function, $tanh(x) = 2\sigma(2x) - 1$.
2. Rectified Linear Unit (ReLU) function, $f(x) = max(0, x)$.
3. Leaky ReLU function, $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$
4. Maxout function, $max(w_1^T x + b_1, w_2^T x + b_2)$.

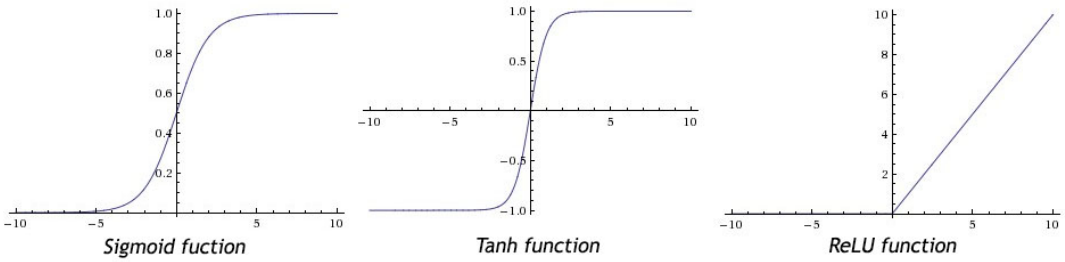

Figure 2.2: Graphs of some common activation functions [Kar16].

A unit's output can be expressed as $g(\boldsymbol{x} \cdot \boldsymbol{\theta} + b)$, where $g(x)$ is the chosen activation function.

As previously mentioned, the general architecture of a neural network can be described as distinct layers of these units, which are connected to units in its adjacent layers. The most common layer type is *fully-connected*, which means that each unit in a layer is connected to every unit in the adjacent layer [Nor14].

The *input layer* receives input values corresponding to the number of features[1] in the neural network. The last, or *output layer* usually corresponds to different classes in a multi-classification problem, or some real-valued target in a regression problem [Kar16]. The layers in between input and output layers are called *hidden layers*; a neural network is classified as DNN if it contains more than one hidden layer. Increasing the size and numbers of hidden layers also increases the *capacity* of the neural network [Kar16]; that is, the space of its representable functions. However, this may undesirably result in *overfitting*[2]. In the implementation (see Chapter 3), there is a λ or `lambda` parameter that adjusts the degree of overfitting to the training data set, which is set at the user's discretion.

There are numerous neural network classifications depending on their architecture; the

[1]For instance, in a 200×200 pixel image recognition problem, there may be $40,000$ features corresponding to each individual pixel's RGB values.

[2]Overfitting refers to modeling the learning algorithm to excessively fit to the training samples. It thereby increases the risk of including unnecessary noise in the data, resulting in more inaccurate model.

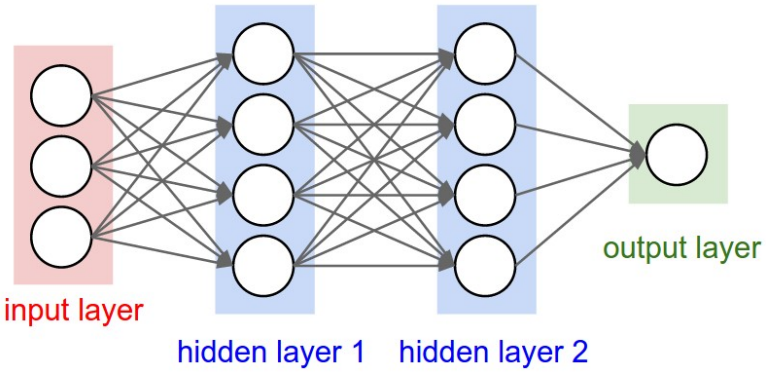

Figure 2.3: An example of a 2-layer neural network [Kar16].

design relevant to this thesis is *supervised*[3] FFBP neural network. Its training process and mathematical representation are briefly outlined in 2.2.

Finally, [Nor14] summarises the attractiveness of neural networks as follows: (1) its capacity to support parallel computation; (2) its fault tolerant nature against novel inputs; (3) *Graceful degradation*, which means a gradual performance drop-offs in worsening conditions; and, (4) The usage of inductive learning algorithms to train the networks.

2.2 Feed-forward back-propagation learning algorithm

This section explains the mechanics and the mathematical representation involved in training a fully-connected, supervised FFBP neural network, based on the works in [Ng16]. Given a set of features and training samples, this learning algorithm aims to find the correct weight distribution in the neural network in two stages: *feed-forward propagation* and *back-propagation*.

First, the weights are randomly initialised within the permitted range of the chosen activation function. For example, this range is $[0, 1]$ for a sigmoid function; for a tanh function, it is $[-1, 1]$.

[3]As in "supervised learning", a concept in machine learning where a set of training examples is paired up with a set of corresponding desired output values. A supervised neural network thus adjusts its links' weights to get the correct output values during its training phase.

Now, in a k-layer neural network, let the size or the number of units in layer j be denoted as $|j|$. Let a particular training sample, s, be denoted as $(\boldsymbol{x}^{(s)}, \boldsymbol{y}^{(s)})$, such that $\boldsymbol{x}^{(s)} = [x_1^{(s)}, ..., x_m^{(s)}]$ is the sample input and $\boldsymbol{y}^{(s)} = [y_1^{(s)}, ..., y_n^{(s)}]$ is the matching desired output. Let $a_i^{(j)}$ be the activation value of unit i in layer j, where $1 \leq i \leq |j|$, $1 \leq j \leq k$. Let $g(x)$ be the activation function; $\Theta_{qp}^{(j)}$ be the weight of a link from unit p in layer j to unit q in layer $j+1$; and, let $\Theta^{(j)} = [\Theta qp^{(j)}$ for $1 \leq q \leq |j+1|, 1 \leq p \leq |j|]$ be the matrix of weights controlling function mapping from layer j to $j+1$.

Then we can express $a_i^{(j)}$ as,

$$a_i^{(j)} = g(\Theta_{i1}^{(j-1)} a_1^{(j-1)} + \Theta_{i2}^{(j-1)} a_2^{(j-1)} + ... + \Theta_{i|j-1|}^{(j-1)} a_{|j-1|}^{(j-1)}) \tag{2.1}$$

For instance, the activation of unit i in the first hidden layer can be expressed as,

$$a_i^{(2)} = g(\Theta_{i1}^{(1)} x_1^{(s)} + ... + \Theta_{im}^{(1)} x_m^{(s)})$$

and, in the output layer as [Ng16],

$$a_i^{(k)} = g(\Theta_{i1}^{(k-1)} a_1^{(k-1)} + ... + \Theta_{i|k-1|}^{(k-1)} a_{|k-1|}^{(k-1)})$$

Also, unlike the approach taken above by [Ng16], [Kar16] states that the activation function is not commonly applied to output layer, because often the result as a real-value number received by the outer layer is the information sought by the user.

2.1 can be simplified using vectorised implementation. Let the activated units in layer j be denoted as $a^{(j)} = [a_1^{(j)}, ..., a_{|j|}^{(j)}]$. Then inputs to this layer can be expressed as $z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$ and so $a^{(j)}$ becomes,

$$a^{(j)} = g(z^{(j)}) \tag{2.2}$$

Forward-propagation process ends when the input values are thus propagated to the output layer.

Next, back-propagation involves re-distributing the error value between the expected output, $\boldsymbol{y}^{(s)}$, and actual output, $a^{(k)}$, back from the output layer through the hidden layers [RHW86]. This concept is based on the idea that the previous layer is responsible for some fraction of the error in next layer, proportional to the links' weights.

Let $\delta_i^{(j)}$ denote the error value in unit i in layer j and $\delta^{(j)} = [\delta_1^{(j)}, ..., \delta_{|j|}^{(j)}]$ be the vectorised error values. Then, for $1 < j < k$,

$$\delta^{(j)} = (\Theta^{(j)})^T \delta^{(j+1)} .* g'(z^{(j)}) \tag{2.3}$$

where $*$ is an element-wise multiplication. The error in the output layer is $\delta^{(k)} = a^{(k)} - \boldsymbol{y}^{(s)}$ and that there is no error in the first layer, because as input values, they cannot contain error.

Finally, the error in link weight $\Theta_{qp}^{(j)}$ is denoted as $\Delta_{qp}^{(j)}$, such that,

$$\Delta_{qp}^{(j)} = a_p^{(j)} \delta_q^{(j+1)}$$

This, too, can be simplified using vectorised implementation as,

$$\Delta^{(j)} = \delta^{(j+1)} (a^{(j)})^T \tag{2.4}$$

Back-propagation ends for s when the errors from the output layer is propagated to the first hidden layer.

The FFBP learning algorithm is then repeated for all the training samples. $\Delta^{(j)}$ accumulates all the errors in the training set during this process. Once the process is finished, the final values are averaged out by the size of the training set, a *regularisation term* is added, and finally the weights are updated. A regularisation term is a value that is added in gradient descent algorithm in machine learning, to prohibit features that are vastly different in its range of input values from one another from distorting the results [Ng16]. This entire process is also known as a form of *batch gradient descent* (BGD) learning algorithm in machine learning [Ng16] and is the method chosen to implement my Accelerate neural network (see Chapter 3).

There are other advanced optimization methods that can improve the performance of neural networks, but these have yet to be explored at the time of this report. As with other machine learning algorithms, the performance may also be improved with either altering various parameters, such as regularisation parameter λ and the learning rate parameter α, altering the number of features, gaining more training examples, adding polynomial features, or any combination of these [Ng16]. However, *diagnostic*

analysis is often the fastest and most economically sound method to determine which set of features is effective [Ng16]. Diagnostic analysis is a form of trial-and-error testing of features sets, or *hypotheses*, with a small set of training examples to assess its performance and thus its suitability.

2.3 GPU-accelerated programming and neural networks

Neural networks implemented with GPU-accelerated programming have shown a significant performance improvement. For instance, [OJ04] had a 20-fold performance enhancement of a multi-layered perceptron (MLP) network to detect text using an ATI RADEON 9700 PRO in 2004. More recently, [GdAF13] showed that performance gains from parallel implementation of neural networks in GPUs are scalable with the processing power of the GPU used. Their results show that performance enhancement over the pure CPU implementation increased with data set size before reaching a plateau, a limit which they contribute to the saturation of the GPU's processing cores.

[NVI14] explains that GPU computing is particularly well-suited to neural networks due to its massively parallel architecture, consisting of tens of thousands of cores. Thus, as single GPUs can hold entire neural network, they state that neural networks benefit from an overall bandwidth increase, reduction in communication latency, and a decrease in size and power consumption compared to a CPU cluster. Furthermore, as neural network units essentially repeat the same computation only with differing input values, the algorithm complements the GPU architecture as there is minimal need for conditional instructions that could trigger thread divergences, which can dramatically reduce the GPU throughput.

The main GPU programming language is CUDA, an API model created by NVIDIA in 2007. It has a C/C++ language style and the added benefits that it bypasses the need to learn graphics shading languages or learn about computer graphics in order to program the GPU. The alternative to CUDA is Open Computing Language (OpenCL) released in 2009; also based on C/C++, it is considered to be the more complicated

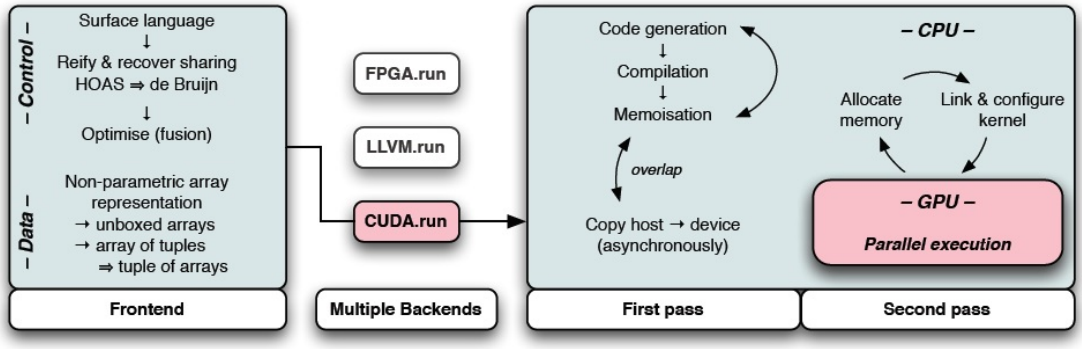

Figure 2.4: Structure overview of `Data.Array.Accelerate`. [CKL+11]

language of the two, but with enhanced portability. As both languages are very low-level [Mar13], there is a need for a way to create, manipulate and test neural networks in less complicated, more user-friendly, safer, higher-level language, such as a functional language.

2.4 Accelerate

Accelerate is an Embedded Domain-Specific Language (EDSL) for high-performance computing on both CUDA-enabled GPUs and CPUs via LLVM, released by UNSW PLS in 2011. EDSLs are restricted languages that are embedded in a more powerful language, so as to reuse the host language's infrastructure and enable powerful metaprogramming. In the case of Accelerate, Haskell is the host language and it compiles into CUDA C or LLVM PTX code that is then compiled for the GPU directly by the CUDA compiler.

Accelerate provides a framework for programming with arrays [Mar13] – Accelerate programs take arrays as input and output one or more arrays. The type of Accelerate arrays is,

```
data Array sh e
```

where `sh` is the *shape*, or dimensionality of the array, and `e` is the *element type*, for instance `Double`, `Int` or tuples. However, `e` cannot be an array type; that is, Accelerate

```
data Z            = Z
data tail :. head = tail :. head

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int

type Array DIM0 e = Scalar e
type Array DIM1 e = Vector e
```

Figure 2.5: Types of array shapes and indices [CKL+11].

does not support nested arrays. This is because GPUs only have flat arrays and do not support such structures [Mar13].

Types of array shapes and indices are shown in Fig. 2.5 [CKL+11]. It shows that the simplest shape is `Z`, which is the shape of an array with no dimensions and a single element, or a scalar. A vector is represented as `Z:.Int`, which is the shape of an array with a single dimension indexed by `Int`. Likewise, the shape of a two-dimensional array, or matrix, is `Z:.Int:.Int` where the left and right `Int`s denotes the row and column indexes or numbers, respectively.

Common shapes have type synonyms, such as scalars as `DIM0`, vectors as `DIM1` and matrices as `DIM2`. Similarly, common array dimensions of zero and one also have type synonyms as `Scalar e`, `Vector e`, respectively.

Arrays can be built using `fromList`; for instance, a 2×5 matrix with elements numbered from 1 to 10 can be created with,

```
fromList (Z:.2:.5) [1..]::Array DIM2 Int
```

To do an actual Accelerate computation on the GPU, there are two options [Mar13]:

1. Create arrays within the Haskell world. Then, using `use`, inject the array `Array a` into the Accelerate world as `Acc (Array a)`. Then, use `run`.

```
-- build an array in Haskell world
fromList :: (Shape sh, Elt e) => sh -> [e] -> Array sh e

-- to execute an Accelerate computation on the GPU
run :: Arrays a => Acc a -> a

-- inject Haskell world array into Accelerate world
use :: Arrays arrays => arrays -> Acc arrays

-- create an array in Accelerate world (array filled with user-specified
   function)
generate :: (Shape ix, Elt a)
         => Exp ix -> (Exp ix -> Exp a) -> Acc (Array ix a)

-- create an array in Accelerate world (array filled with same values)
fill :: (shape sh, Elt e) => Exp sh -> Exp e -> Acc (Array sh e)
```

Figure 2.6: Some Accelerate functions [Mar13].

2. Create arrays within the Accelerate world. There are various methods, such as using `generate` and `fill`. Then, use `run`.

`run` executes the Accelerate computation and copies the final values back into Haskell after execution. In `Acc a`, `Acc` is an Accelerate data structure representing a computation in the Accelerate world, that yields a value of type `a` (more specifically, an array) *once it is executed by a backend* [McD13, Mar13].

Now, as the first method may require the array data to be copied from computer's main memory into the GPU's memory, the second method is generally more efficient [Mar13]. All the aforementioned functions are outlined in Fig. 2.6.

Since Accelerate has its own backend compiler, it must be set before any code can be `run`. There can be a number of backends to execute Accelerate computations: (1) `Data.Array.Accelerate.Intepreter`, the slowest performing, simple Haskell interpreter; (2) `accelerate-llvm-native`, which supports parallel execution on multicore CPUs; and, (3) `accelerate-llvm-ptx`, which supports parallel execution on CUDA-capable NVIDIA GPUs. `accelerate-cuda` is deprecated in favour of (3).

2.5 Previous Implementations in Accelerate

A neural network in Accelerate is a fairly new concept and as such, there is only one known work in [CEMCK17]. In it, Everest implements a FFBP neural network with a *stochastic gradient descent* (SGD) learning algorithm, based on a Python implementation [Nie15]. This section briefly covers the details of this implementation.

SGD algorithm is a modification of BGD algorithm, shown in 2.2. Rather than adjusting the neural network parameters after going through the entire training set, SGD updates the parameters immediately after doing the FFBP algorithm for a single training example, which is picked at random [Ng16]. SGD can also be modified to select certain small batches of training sample at a time [LBBH98].

As such, SGD is computationally less expensive than the BGD, and allows the training algorithms to scale better to much larger training sets [Ng16]. [LBD+89] also claims that SGD trains the neural network faster. The disadvantages of SGD are that it is said to be "less accurate" than BGD as it may take longer to reach the local/global minima, however, it is still the more popular choice as it places less strain on system resources. Also, SGD is suitable to the work in [CEMCK17], because it deals with processing data in a streaming fashion to/from the GPU.

There are six key Accelerate functions that were used and their names and operations are listed in Fig. 2.7. These functions were also used in my own implementation (see Chapter 3). In particular, the functions `lift` and `unlift` are used to inject and extract a data constructor into and out of `Exp`[4], respectively, and are thus essential in interpreting the indices of arrays in Accelerate [Mar13].

In [CEMCK17], two support functions were also created to assist forward- and back-propagation processes – `mvm` and `cross` (see Fig. 2.8). In `mvm`, `h` first stores the row number of the matrix input using `unlift`. The vector input, `vec` is then replicated `h` times across the first dimension using `replicate`. The resulting matrix is then element-

[4] `Exp` is another Accelerate data structure, that is similar to `Acc` (refer to 2.4), but instead of an array, it represents an embedded *scalar* computation [CKL+11].

```
-- apply supplied function element-wise to corresponding elements of two
    input arrays to produce a third array
zipWith :: (Shape ix, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array ix a) -> Acc (Array ix b) -> Acc (Array ix c)

-- map function within the Accelerate world
map :: (Shape ix, Elt a, Elt b)
    => (Exp a -> Exp b) -> Acc (Array ix a) -> Acc (Array ix b)

-- replicate an array across one or more dimensions according to the
    first argument (a generalised array index)
replicate :: (Slice slix, Elt e)
          => Exp slix -> Acc (Array (SliceShape slix) e)
          -> Acc (Array (FullShape slix) e)

-- reduce the innermost dimension of an array
fold :: (Shape ix, Elt a)
     => (Exp a -> Exp a -> Exp a) -> Exp a
     -> Acc (Array (ix:.Int) a) -> Acc (Array ix a)

-- wraps a value in Exp
lift :: Z:.Exp Int:.Exp Int -> Exp (Z:.Int:.Int)

-- deconstruct Exp, get the structured value from within
unlift :: Exp (Z:.Int:.Int) -> Z:.Exp Int:.Exp Int
```

Figure 2.7: Key Accelerate functions used in [CEMCK17].

```
-- matrix vector multiplication function
mvm :: (Elt a, IsNum a)
    => Acc (Matrix a) -> Acc (Vector a) -> Acc (Vector a)
mvm mat vec
  = let Z:.h:._ = unlift (shape mat) :: Z:.Exp Int:.Exp Int
  in A.fold (+) 0 $ A.zipWith (*) mat (A.replicate (A.lift (Z:.h:.All)) vec)

-- vector multiplication function
cross :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Matrix Float)
cross v h = A.zipWith (*) (A.replicate (lift (Z:.All:.size h)) v)
                          (A.replicate (lift (Z:.size v:.All)) h)
```

Figure 2.8: Support functions for forward- and back-propagation used in [CEMCK17].

```
-- back-propagation part of backprop function
nabla_b_final = A.zipWith (*) ( costDerivative (P.last activations) y)
                                              (A.map sigmoid' (P.last zs) )
nabla_w_final = cross nabla_b_final ( P.last (P.init activations) )

...
```

Figure 2.9: Accelerate and Haskell code are mixed in [CEMCK17].

wise multiplied with `mat` using `zipWith`. Lastly, the values of the matrix are summed and flattened back into a vector using `fold`.

The process is similar in `cross`. `cross` is a vector multiplication function that returns a matrix. First, vector `v` is replicated $|h|$ number of times in the second dimension, whereas vector `h` is replicated $|v|$ number of times in the first dimension. The resulting two matrices are then element-wise multiplied using `zipWith`.

The actual FFBP algorithm is in the function `backprop`. In `backprop`, Accelerate code (imported as `A`) seamlessly interweaves with the Haskell part (imported as `P`), as seen in Fig. 2.9.

At the time of this report, [CEMCK17] has reported some performance issues with this implementation with the success rate of digit prediction at appropriximately 50%. The cause is yet unknown, but may possibly be due to the nature of data streaming. This implementation also purportedly running at a slower speed than an equivalent Theano implementation. Theano is a Python library that is optimised for multi-dimensional array computations, and can be optimised to use GPU for computations.

Chapter 3

Implementation

3.1 Using a MATLAB-based neural network as reference

MATLAB, or Matrix Laboratory, is a highly doman-specific programming language, specialised to conduct numerical calculations and analysis. It was created in the 1970s, and is an imperative procedural language with a wide array of built in linear algebra operations. It is widely used among the industry and academia, particularly in science, engineering and economics.

I have used the neural network MATLAB program in Andrew Ng's Machine Learning course in Coursera as the basis of my Accelerate program [Ng16]. There are several motivations for choosing this particular implementation.

Firstly, the MATLAB implementation clearly shows the mathematical structure of a neural network, succinctly expressing the matrix multiplication and vectorisation operations out of which a neural network is built. It was projected that, seeing as high level matrix operations can also be easily expressed in Accelerate, this implementation would be reasonably simple to adapt, and perhaps the close-in-spirit to an Accelerate implementation.

Secondly, the MATLAB implementation also included a data set with expected answers

to which I can compare my own implementations. Thus, I could be assured that my implementation in Accelerate was accurate enough to be reliable, particularly in the light of the difficulty of debugging Accelerate programs. Passing this accuracy test will indicate that my small network is reliable enough to be built upon in future machine learning developments.

Thirdly, the codebase on which I am basing my implementation is from Coursera, an online course service, specifically the Machine Learning course made by Andrew Ng of Stanford University. This course has drawn wide praise for its material and presumed to be reviewed by many in the same field. The structure of the course is reliable in terms of correctness and efficiency, and implemented already in a manner that is idiomatic for the Machine Learning community. Hence, I projected that the MATLAB implementation would be optimised for the best performance.

Lastly, in terms of familiarity, I was more conversed with MATLAB than with other neural network implementations, such as Python (see 2.5), C++ [Wol17], or Haskell's `hnn` and as such I could avoid needlessly increasing the time to start the project by choosing a language I was already comfortable using and an implementation I had already studied. While researching other implementations, I found that it took more time to understand these as I had to additionally learn the language in enough detail to understand the code being written, and the implementations differed in many details from the versions with which I was familiar.

3.2 Accelerate Implementation

3.2.1 Neural network structure

I have implemented a two-layered, fully connected simple neural network, with one hidden layer.

My implementation can easily be modified to take more than 1 hidden layer by changing the `nnCostFunction`. For instance, each extra hidden layer should only require

```
-- current feed-forward with 1 hidden layer
a3 :: Acc (Matrix Float)
a1 = xs
z2 = theta1 <> transpose a1
a2 = (fill (lift (Z :. h :. constant 1)) 1 :: Acc (Matrix Float))
     A.++ (A.transpose $ A.map sigmoid z2)
z3 = a2 <> A.transpose theta2
a3 = A.map sigmoid z3

-- an example feed-forward with 2 hidden layers
a4 :: Acc (Matrix Float)
a3' = (fill (lift (Z :. h' :. constant 1)) 1 :: Acc (Matrix Float))
      A.++ (A.transpose $ A.map sigmoid z3)
z4 = a3' <> A.transpose theta3
a4 = A.map sigmoid z4

-- an example backpropagate with 2 hidden layers
d4 = A.zipWith (-) a4 ys
d3 = A.zipWith (*)
     (d4 <> theta3)
     ((fill (lift (Z :. h' :. constant 1)) 1 :: Acc (Matrix Float))
       A.++ (A.transpose $ A.map sigmoidGradient z3))
d2 = A.zipWith (*)
     (d3 <> theta2)
     ((fill (lift (Z :. h :. constant 1)) 1 :: Acc (Matrix Float))
       A.++ (A.transpose $ A.map sigmoidGradient z2))
```

Figure 3.1: Changing the number of hidden layers in `nnCostFunction`

the addition of two matrix multiplications in the feed-forward and backpropagation components, as well as the necessary steps to update the theta layer (see 3.1).

3.2.2 Program structure

In terms of program structure, I closely followed the control flow of the MATLAB version. It can be partitioned into three main subroutines: (1) initialisation; (2) the neural network cost function; and, (3) function minimisation via the conjugate gradient method. The implementation code is available at [MJ17].

Initialisation

Initialisation is a straightforward process. We initialise (1) the regularisation parameter `lambda`; (2) initial weight vectors, in this instance `theta1` and `theta2`; (3) size of the input, hidden, output layers; and, (4) the input training data set `xs` and the output vector `ys` to conduct the supervised learning.

The number of input units correspond to the number of attributes in each sample of `xs`, whereas the output units correspond to the number of classifications that these samples could be divided into, or, the range of the values of `ys`. Setting the number of neurons in the hidden layer is mostly done by a mixture of guesswork and trial-and-error. In 4.1, 25 and 300 neurons comprise the hidden layer for two different data sets of handwritten numbers. These sizes were taken from other implementations with the same data sets, who had found these the most effective numbers by testing.

The matrix `xs` represents m training set data. After it has been loaded, each row of `xs` should represent a single training example, $x_1, x_2, ..., x_m$ where the length of vector x_i is the size of the input layer. A column vector of 1s is added to the leftmost of the array to represent the bias neuron, as shown below.

$$\texttt{xs} = \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{bmatrix}$$

This is implemented in Accelerate using a combination of `fill` and concatenation `A.++`. Accelerate's operations are not the same as Haskell Prelude's, even if some operation names coincide. Thus they are distinguished by a qualified import: `A`, `P` for Accelerate and Prelude correspondingly, for instance `A.++` and `P.++`. Other key array manipulation functions used throughout the program include `A.take, A.drop, A.map` and `A.zipWith`.

In contrast to the the clarity of the underlying structure offered by Accelerate's syntax, MATLAB functions automatically apply multiplications of a scalar term to a matrix,

by applying the operation of the scalar to each element of the matrix. Thus, if `a` is a scalar and `xs` is a matrix, it is possible to simply do `a` × `xs`. In Accelerate, this would be `A.map (*a) xs`. As a result, MATLAB code is deceptively compact — it may not reveal the actual structure of the underlying operation just from a surface examination of the program syntax.

Other notable MATLAB difference is the starting index of matrices and arrays being 1, not 0.

The effectiveness of the random initialisation of weight vectors will influence the accuracy of the neural network. Initial values should be in the range $[-\epsilon_{init}, \epsilon_{init}]$ [Ng16]. One popular choice of ϵ_{init} is,

$$\epsilon_{init} = \sqrt{\frac{6}{L_{in} + L_{out}}}$$

where L_{in} is number of units in the preceding layer, and L_{out} is the number of units in the subsequent layer of the weight vector-in-question.

The optimal regularisation parameter `lambda` is also discovered by trial-and-error. It is set by the user to determine the level of fitting of the neural network to the training set is considered 'overfitting'. Whether higher or lower overfitting to the training set is better or worse for making predictions on new samples depends entirely on the nature of the data.

Neural network cost function, `nnCostFunction`

This function applies both the feed-forward and back-progapation to sample data inputs and updates the weight vectors and error costs accordingly.

As described in 2.2 we apply the BGD method by grouping all the sample data into one large batch, passing them in all at once, for higher accuracy.

As the `nnCostFunction` is used in conjunction with the `fmincg` function (3.2.2), it was necessary to flatten all the weight vectors into a single dimension before passing them

```
-- type of reshape
reshape :: (Shape sh, Shape sh', Elt e) =>
           Exp sh -> Acc (Array sh' e) -> Acc (Array sh e)

-- unroll theta1, theta2
theta1 = reshape (index2 hiddenl (inputl +1)) $ A.take
    (hiddenl*(inputl+1)) thetas
theta2 = reshape (index2 outputl (hiddenl+1)) $ A.drop
    (hiddenl*(inputl+1)) thetas
```

Figure 3.2: Reshaping vectors into matrices in Accelerate.

into, and returning them from the `nnCostFunction`. The flattened vector is sliced and reshaped to restore the vectors to their original shapes inside `nnCostFunction`, implemented via the Accelerate `reshape` function as shown in 3.2.

Using `reshape` can be initially cumbersome, because it must be supplied with an `Exp` of the output shape. This can be supplied in various ways, and may require its type to be explicitly specified. For example, in the case where one `lifts` a `(Z:.m:.n)` expression to supply `fill` with an `Exp (Z:.Int:.Int)`, a type signature is required:

```
fill (lift (Z:.m:.n)) 1 ::  Acc (Matrix Float)
```

Thus, I found operations involving shape parameters quite tricky at times due to the necessity of explicit typing.

Matrix multiplications was initially implemented using `mmult`, but to improve performance on the CPU LLVM backend, the foreign function interface of Accelerate [CEMCK17] was used to call the matrix multiplication function from the `hmatrix` package, which has a better performance.

Function minimisation via conjugate gradient method, `fmincg`

Generally, function minimisation by conjugate gradients is a quadratically convergent gradient method used to locate a local minimum when the function has several variables, or, is *multivariate* [FR64]. This function repeats `nnCostFunction`, a multivariate

function, for certain number of iterations, and progressively tries to minimise the error in the weight matrices.

In MATLAB, function minisation by conjugate gradient is a function called `fminunc`, which states that given a starting point `x0` (can be a scalar, vector or a matrix), `x` is the the local minimum of unconstrained function `func` [Mat17]:

```
x = fminunc(func, x0)
```

`fmincg` function is a modified version of `fminunc` and has the following MATLAB equation:

```
function [X, fX, i] = fmincg(f, X, options, P1, P2, P3, P4, P5)
```

According to the author, it is more efficient than `fminunc` in that it uses 'Polack-Ribiere' conjugate gradients, quadratic and cubic polynomial approximations and the 'Wolfe-Powell stopping criteria' to more efficiently calculate slope and stepping sizes [Reb13]. The mathematics behind this method is beyond the scope of my understanding.

The `fmincg` function terminates when it either finds the local minimum, or if the progress is so insignificant that it is not worth any further exploration. It must be given a cost function `f`, initial weight vector `X` and maximum iteration number in `options`. Other parameters are not supplied.

The function returns the solution vector as `X` and a cost vector as `fX`. `fX` starts off as an empty vector, and `fmincg` pushes the cost/error of the newly recalibrated weights in each iteration to the back of `fX`. The end result is that the caller can see the progress made throughout the iterations by checking `fX`. Other variables are discarded.

The Haskell/Accelerate type signature of `fmincg` can be seen in 3.3. The vector argument is a concatenation of flattened weight matrices and is fed into the function argument. This is to enable `fmincg` to execute the function independently from the structure of the underlying neural network, i.e. the number of hidden layers. It is a artifact of MATLAB methodology that needs better implementation in Accelerate (see Chapter 5).

```
fmincg :: (Acc (Vector Float) -> Acc (Scalar Float, Vector Float))
       -> Acc (Vector Float)
       -> (Acc (Vector Float), Acc (Vector Float))
```

Figure 3.3: Function minimise conjugate gradient function.

I faced a couple of challenging factors in implementing this function. First, the MATLAB version had approximately 17 parameters, most of which are overwritten and interact in complex, intricate ways. Without a knowledge of the underlying mathematics, I could not fully comprehend the overall structure of this 175-line procedure. The ambiguity and similarity of parameter names, such as `d1,d2,...,v3`, did nothing to ease the problem.

Secondly, it is difficult to follow the flow of control in `fmincg`. In order to reduce the chance of errors arising, I chose to initially follow the procedure very closely, hoping to optimise incrementally later. Certain expressions were substituted where operations had not yet been implemented in Accelerate (see 3.3). Additionally, I could not model the extremely complex flow control perfectly, and the known issues and bugs are mentioned in 3.3.

Thirdly, MATLAB `fmincg` was composed of three `while` loops within each other and three `if-else` statements, one of the latter which resulted in a flow divergence. The last factor in particular may have greatly reduced this implementation's suitability for GPU execution, as such parallelism with flow divergence may can cause drastic problems in GPU performance, which favours non-divergent *flat-data parallelism*, where sequential operations can be applied in parallel over bulk data.

On the positive side, the flow divergence is only triggered upon a failure to find a local minimum; which, according to [Reb13] occurs when the function supplied to `fmincg` is inaccurate or unsuitable. For this particular implementation, I have taken the assumption that the function supplied, `nnCostFunction` is accurate and that there is always a local minimum that `fmincg` will find. In my limited tests (see Chapter 4), all results support this assumption. More broadly, the correctness of the function can be checked for prior to being supplied to `fmincg`.

```
d1, f1, limit :: Acc (Scalar Float)
m :: Acc (Scalar Int)
(theta, df2, d1, d2, d3, f1, f2, f3, z1, z2, z3, m, limit) = unlift $
    awhile cond body initial
```

Figure 3.4: Assisting Accelerate by specifying types of unused variables in `unlift`.

To make my implementation more manageable, I divided the MATLAB code into their `while` loops. These were implemented using Accelerate's `awhile` flow control combinator.

This combinator `awhile` requires variables that change during the loop and/or are returned when function ends, to be wrapped into a single `Acc` tuple to be used for the loop process. The functions `lift` and `unlift` are necessary to pack and unpack the variables respectively in each loop, for condition checks and for loop execution. An interesting fact I learnt was that Haskell's type inference quickly meets its match with Accelerate — it cannot infer the types of some variables unless they are unambiguously used after an `unlift` call. It is thus sometimes necessary to specify types of certain terms to assist Haskell with type inference as seen in 3.4.

3.3 Known bugs

There are several issues with my Accelerate implementation. One, it does not have the robust failure handling, allowing it withstand incorrect inputs analogously to the MATLAB programs. For instance, I have already mentioned in 3.2.2 that the function supplied to `fmincg` must have a clear minimum.

Also, there are cases in `fmincg` where some variables need to checked for `isinf`, `isreal` or `isnan`, that is, be checked for an infinity, a real number, or 'not-a-number' floating-point properities, respectively. As these functions are not yet available in Accelerate, I substituted them for expressions which I believe will cover those particular situations. For instance, `isreal` check may arise when a determinate may be less than zero; `isnan` may arise when a divisor is zero. I was not able to devise a check for `isinf`, however,

```
157       z1 = z1 * min(RATIO, d1/(d2-realmin));          % slope ratio but max RATIO
158       d1 = d2;
159       ls_failed = 0;                                  % this line search did not fail
```

Figure 3.5: MATLAB's `double` is actually a `float`? What it means for `realmin`.

so the implementation may not cover all the range of inputs and remain fault-free.

Secondly, MATLAB's `double` is by default a double-precision data type, and requires 64 bits [Mat17]. Yet, after testing it on the sample data, importing the values in Accelerate as `Double`s produces a result further away from the MATLAB result than when I pass them to Accelerate as `Float`s. Not only that, the accuracy of the Accelerate neural network predictions after training decreased with the same training set when the data was passed as a `Double`.

The source of the inaccuracy may be due to `fmincg`. There is one line that seems to account for overflow, by subtracting `realmin` from a dividend as seen in 3.5. The value `realmin` for a `float` is defined as $1.1755e^{38}$ and for a `double` is $2.2251e^{308}$ in MATLAB. These different by orders of magnitude ($1xe^{270}$) and could vastly affect the result. In my implementation, I have currently taken out `realmin` from the equation, which increases the risk of division by zero but does not seem to be problematic in practice.

Lastly, some of the optimisations in `fmincg` has been ignored, namely: (1) the innermost and middle loop iterations, thus the loop counter `M` is set to 1; and, (2) the handling of a failure case. The reason for (2) has already been discussed in 3.2.2. For (1), testing has shown that the iterations other than exactly once for the sections in question produces erroneous results, and the reasons for this is unfortunately still unknown at the time of writing. In addition, although setting `M=1` seems to closely align my program's results to MATLAB's results in the first training set, my neural network was unable to yield as high accuracy rate as in [LBBH98] for the second training set, deviating by almost 10 per cent (see Chapter 4, 5).

3.4 Other works

I have also implemented a logistic regression cost function, which is akin to a neural network without any hidden layers. This can be accessed at [MJ17]. This function produces analogous results to the equivalent MATLAB implementation.

Chapter 4

Testing and Results

4.1 Test Data

There are two data sets that I have tested the implementation with: (1) handwritten digits data set from [Ng16]; and, (2) handwritten digits data set from [LCBnd].

Set (1) consists of 5000 training examples of 20×20 pixel greyscale images of digits, each pixel represented by a floating point number that indicates the greyscale intensity at that location. The input layer will thus have 400 neurons. In this example, the hidden layer has 25 neurons and the output layer has 10 neurons. The size of weight vector between input and hidden layers is $401 \times 25 = 10025$ and the size of the weight vector between hidden and output layers is $26 \times 10 = 260$.

The output neuron with the highest activation value will be the digit that the sample will be classifed under. For instance, if the neuron '4' of the output vector, `ys[3]` had the highest value, then the sample will be classified as digit '4'.

This data set was mostly used during the development to assess the accuracy of the implementation, as I could easily compare the values produced by my program with the values from the MATLAB program. In 4.3 I evaluated the comparative performance of Accelerate and MATLAB, up to 2 cores.

Set (2) is Le Cunn's MNIST handwritten digits database. This consists of a training set of 60000 examples and a test set of 10000 examples. Each example is a 28×28 pixel greyscale image. This input layer has 784 neurons and a hidden layer of 300 neurons. Thus the size of weight vector between them is $785 \times 300 = 235500$. The size of weight vector between hidden and output layer is $301 \times 10 = 3010$. This data set was used to test the performance of completed Accelerate implementation from 1 to 8 cores.

Each test was repeated 10 times and the average value plotted. The initial weight vectors for training set (2) benchmarking was kept consistent to strictly measure the effect of cores on performance. In training set (1), the initial weight are randomly initialised, and thus up to 1% in fluctuations could result from this factor [Ng16].

4.2 Testing environment

There are two testing environments. Both environments ran on `llvm` backends as the testing data was too small for the GPU backend.

The first configuration is for testing during development. It is a 2-core Intel i5-5200U CPU (64-bit, 2.2GHz, 12GB RAM, hyperthreading enabled) running GNU/Linux (Ubuntu 16.04 LTS).

The second configuration was for dedicated multi-core testing. It is a 4-core "Ivy Bridge" Intel i7-3720QM (64-bit, 2.6GHz, 8GB SDRAM, hyperthreading enabled) running MacOS.

4.3 Performance results

Table 4.1 shows that in the development environment with training sample set (1), the accuracy of the Accelerate implementation is similar to MATLAB's; however, the time taken to calculate the weights grows much faster depending on the sample size. This may be due to some bugs mentioned in Chapter 5.

Input size	Accelerate (LLVM-CPU)(s)	Accelerate accuracy (%)	MATLAB (s)	MATLAB accuracy (%)
100	0.4912	75.22	0.2037	75.00
250	0.9312	83.42	0.3592	83.84
500	1.545	88.86	0.5028	88.66
750	2.106	89.94	0.7283	89.94
1000	2.821	91.40	0.8276	90.90
1500	4.052	92.64	1.2782	92.40
2000	5.915	93.44	1.3827	93.62
3000	8.273	94.10	1.9658	94.50
4000	13.1	94.74	2.4011	94.96
5000	17.95	95.54	2.6811	95.08

Table 4.1: Benchmarking training set 1 (Training set from [Ng16]).

Number of cores	Accelerate (LLVM-CPU)(s)	Accelerate accuracy (%)
1	239.695	83.97
2	184.37	84.46
3	151.967	84.46
4	135.719	84.46
5	133.421	84.46
6	130.372	84.46
7	124.314	84.46
8	122.310	84.46

Table 4.2: Benchmarking training set 2 (MNIST training set).

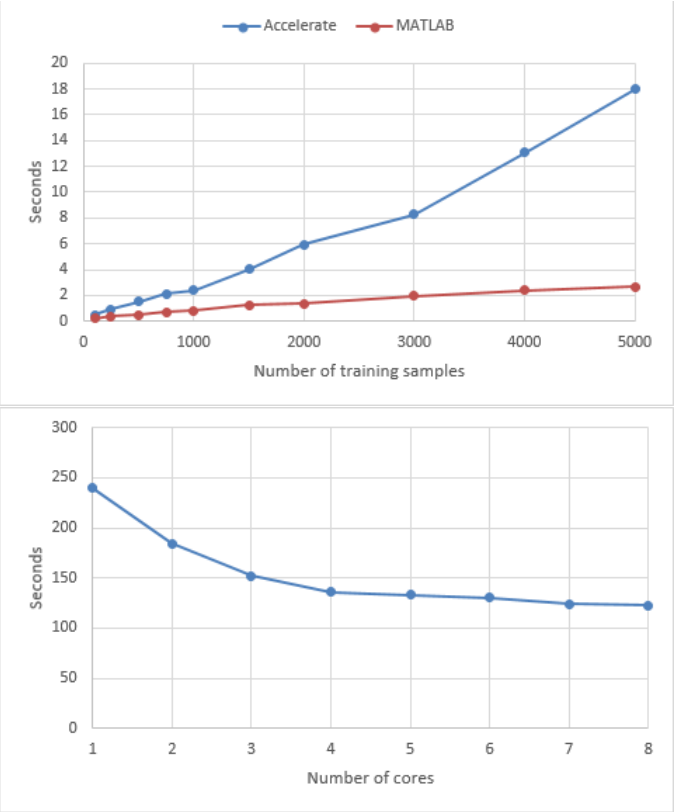

Figure 4.1: Graphs of training set 1 (top) and 2 (bottom) benchmarks.

Table 4.2 indicates a performance improvement with the addition of more cores. It levels off rapidly, however, perhaps indicating that there may be too much overhead to keep the cores synchronized to gain more benefit with the current implementation (see Chapter 5).

Another alarming result is the lower than expected accuracy at 84.46%. At `lambda=3.0`, this remains largely unchanged at 84.39%. I checked the MATLAB accuracy with training set (2) and got an average of 91.68% accuracy and 190.52 seconds time taken (10 tests on first testing environment), indicating that the issues outlined in 3.3 is having a significant impact on accuracy of my implementation.

Chapter 5

Evaluation

The intent of this thesis was to create a neural network in Accelerate that is (1) sufficiently well-performing; (2) with good useability; and, (3) analyse the benefits and disadvantages of such implementation. I will go through each one in the sections below.

5.1 Performance evaluation

In terms of performance, namely correctness, scaleability and relative speed, my implementation did not perform well as expected (see Chapter 4).

Although accuracy was on-par with MATLAB on training set (1), it had lower accuracy than other implementations, including MATLAB, on the second training set. According to [LBBH98], a two-layer fully connected neural network 300 hidden units for MNIST dataset should produce an error of approximately 4.7%. Albeit the fact that LeCun (1998) uses a carefully tuned, SGD algorithm method named *Stochastic Diagonal Levenberg-Marquardt*[1] to train his neural network, one should expect my Accelerate

[1]This is a complex method, but amongst other techniques, LeCun (1998) calculates an individual learning rate for each weight before parsing the training set — which is applied to the network around 20 times — in a randomised order of *patterns*, or the sorting of training samples in a properly randomised sequence.

implementation to mirror the accuracy of MATLAB's performance as it did for training set (1). However,4.2 indicates that it is approximately 7.2% less accurate than that of MATLAB. This seems to signify that the unresolved bugs and issues as previously mentioned in 3.3 is a significant dampener on the accuracy of my implementation.

Also worth noting is that the MATLAB implementation has a higher error rate of 8.3% compared to LeCun's SGD neural network at 4.7%. This seems to imply that BGD with conjugate gradient method may not be as effective as SGD (or perhaps, for this particular data set).

Interstingly, my program with an error rate of 15.5% is closer the error rate reported by a native Haskell neural network implementation called `neural` [Bru16]. Upon testing with MNIST training set, **neural** produced an error rate of approximately 17% after 50 iterations, taking 983.9 seconds with 8 cores. This program reports it can ultimately reach an accuracy of 90.05% after 450 iterations.

In terms of scaleability, my implementation linearly increases the time taken compared to size of input with constant number of cores similar to MATLAB, and reaches a plateau rapidly upon adding more (see 4.1). We believe this may be due to the rolling and unrolling of weight vectors during `fmincg` operation as mentioned in 3.2.2. The amount of work involved in copying the large matrices probably adds a significant overhead, reducing the benefit of having multicores.

Such redundant work is also likely a factor in negatively affecting the speed, particularly for smaller data sets. For instance, the MATLAB program seems to perform much faster with the smaller training set (1) (see 4.1). With MNIST training set, however, my Accelerate program does perform more competitively — but, further testing is required.

I was unable to find *speed* performance results for MNIST training set on neural networks with same architecture in other languages in order to do a relative speed performance apart from MATLAB[2].

[2]The closest one I could find was a C++ neural network[Wol17], but with only 30 hidden layers. This reduces the first weight vector from 235500 to 23550 and the second weight vector from 3010 to 310. This implementation also uses the faster SGD method

Yet, on a positive note, my Accelerate neural network finishes training at 122.3 seconds with 8 cores. This is despite the fact that this implementation is more or less a direct translation of the MATLAB code with minimal Accelerate optimisation. With further Accelerate naturalisation (and bug fixes), it may be possible to achieve a very reasonable speed performance!

5.2 Usability and Accelerate

It is fairly reasonable to say that the ease and convenience of a programming language can affect a programmer's work. One of the main reasons in starting this thesis was to gauge the the ease of creating a neural network using Accelerate.

Firstly, Accelerate has a convenient syntax that Haskell-users will find easy to use. Although MATLAB may seem more convenient than Accelerate at times, especially in operations that requires array or matrix manipulations, MATLAB is also ambiguous and loose in its language, and meanings can get lost without more effort being invested by the reader (see **??**). In contrast, Accelerate allows programmers to know exactly what is occuring with its syntax, accompanied by many high performing libraries for array computations that are easy-to-use.

Secondly, Accelerate is compact and succinct, but perhaps not as abbreviated as in MATLAB to become obscure to its users. Neither is Accelerate as verbose as C++, which can become cumbersome to read and write as seen in 5.1.

There are, however, several elements that can intimidate new users to Accelerate. For example, I found it was quite difficult to debug my program, as Accelerate computations are not observable until they are returned to Haskell world.

Secondly, as previously mentioned in 3.2.2, Haskell's type inference could not automat-

in the manner of [LBBH98] and also did not disclaim his testing environment. Training time taken is said to be 82 seconds, but there were too many unknown factors in this data to draw comparisons.

```
-- in MATLAB
function p = predict(Theta1, Theta2, X)
m = size(X, 1);
h1 = sigmoid([ones(m,1) X] * Theta1');
h2 = sigmoid([ones(m,1) h1] * Theta2');
[~,p] = max(h2, [], 2);
end

-- in Accelerate
predict :: Acc (Matrix Float) -> Acc (Matrix Float) -> Acc (Matrix Float)
    -> Acc (Vector Int)
 predict theta1 theta2 xs =
 let
     Z :. m :. n = unlift (shape h1) :: Z :. Exp Int :. Exp Int
     h1 = A.map sigmoid
        $ xs <> A.transpose theta1
     h2 = A.map sigmoid
        $ ((fill (lift (Z:.m:.(constant 1))) 1 :: Acc (Matrix Float)) A.++
            h1)
          <>
          (A.transpose theta2)

     getYs :: Acc (Vector Int)
     getYs
       = A.map ((+1) . A.indexHead . A.fst)
       $ A.fold1 (\x y -> A.snd x A.> A.snd y ? (x , y))
       $ A.indexed h2
 in
 getYs

-- in C++
uint8 ForwardPass (const float* pixels, uint8 correctLabel) {
     for (size_t neuronIndex = 0; neuronIndex < HIDDEN_NEURONS;
         ++neuronIndex) {
         float Z = m_hiddenLayerBiases[neuronIndex];
         for (size_t inputIndex = 0; inputIndex < INPUTS; ++inputIndex)
             Z += pixels[inputIndex] *
                 m_hiddenLayerWeights[HiddenLayerWeightIndex(inputIndex,
                 neuronIndex)];
         m_hiddenLayerOutputs[neuronIndex] = 1.0f / (1.0f + std::exp(-Z));
     }

     for (size_t neuronIndex = 0; neuronIndex < OUTPUT_NEURONS;
         ++neuronIndex) {
         float Z = m_outputLayerBiases[neuronIndex];
         for (size_t inputIndex = 0; inputIndex < HIDDEN_NEURONS;
             ++inputIndex)
             Z += m_hiddenLayerOutputs[inputIndex] *
                 m_outputLayerWeights[OutputLayerWeightIndex(inputIndex,
                 neuronIndex)];
         m_outputLayerOutputs[neuronIndex] = 1.0f / (1.0f + std::exp(-Z));
     }

     ...
     return maxLabel;
 }
```


Figure 5.1: Comparing `predict` in MATLAB [Ng16], Accelerate and C++ [Wol17].

ically infer the types of some Accelerate variables, particularly in `unlift` operations.

Other minor inconveniences include determining which situation called for `Exp` and `Acc` and why or when one should switch between those two data structures, the fact that it was not possible to extract values from `Exp`, and lastly, the initial set up was quite intimidating.

5.3 Advantages of an Accelerate implementation

There are several advantages to using Accelerate. First, it results in much simpler source programs as programs are written in Haskell syntax; Accelerate code is very similar to an ordinary Haskell code and there are minimal syntactic difference [Mar13].

Second, as Accelerate is embedded in Haskell, it can benefit from inheriting Haskell's functional language characteristics. For instance, Haskell as a pure language is advantageous for parallel computations as it will prohibit side effects that can disrupt other threads.

Another Haskell characteristic is having a more powerful type system, which could enforce a stronger checking for certain properties — thereby catching more errors — at compile time. In comparison, languages like MATLAB are dynamically typed, meaning that the types may change during runtime. Futhermore, as a weakly typed programming language, its types can be implicitly converted whenever a mismatch occurs, making its program unpredictable and unreliable.

Thirdly, Accelerate uses a number of optimisation techniques to mitigate the overheads, such as array fusion and sharing recovery[CKL$^+$11]. For instance, my implementation was very inefficient in the sense that it would have generated multiple copies of same variables due to many `while` loops in a non-optimising compiler. Accelerate will optimise the code to reduce the number of parameters to the bare minimum required during the production of the ASTs in compile time. It will also 'fuse' sequences of *producer/producer* operations and fuse *producer/consumer* operations to eliminate in-

termediate arrays — overall, resulting in a much cleaner, efficient program [MCGN15].

Finally, Accelerate is a dynamic code generator and as such it can, (1) optimise a program at the time of code generation simultaneously by obtaining information about the hardware capabilities; (2) customise the generated program to the input data at the time of code generation; and (3) allow host programs to "generate embedded programs on-the-fly" [CKL+11].

On the other hand, the disadvantage of using Accelerate is the extra overheads, which may originate at runtime (such as runtime code generation) and/or at execution time (such as kernel loading and data transfer) [CKL+11]. Some of the overheads however, such as dynamic kernel compilation overheads, may not be so problematic in heavily data- and compute-intensive programs, because the proportion to the total time taken by the program may become insignificant [CKL+11]; and, neural networks certainly fit in such a category of programs.

Indeed, even with the small data sets in this implementation, the results in Chapter 4 seem to support this: for training set (1) the overhead was probably disproportionately big, making Accelerate implementaion perform much worse compared to MATLAB's; however, Accelerate performed better than MATLAB for training set (2).

5.4 Incomplete works

As stated in 3.3, there are a number of issues that requires further work in my current implementation. First is resolving the unexpected behaviour in the inner and middle loops when `M` variable is not set at 1. Secondly, the rolling and unrolling of weight matrices should be optimised or eliminated, in order to reduce overhead.

I would also test the accuracy of the predictions by trying a different activation functions. I have used the sigmoid function in this implementation, initially in order to confirm that my development is in alignment with MATLAB's and also because it is

mathematically convenient, because its gradient, $\sigma'(x)$, is easy to calculate:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

However, sigmoid function is not widely used these days, due to these two disadvantages [Kar16]:

1. When the unit's activation *saturates* at either tail of 0 or 1, the gradient of the sigmoid function nears zero (refer to the sigmoid function graph in Fig. 2.2). Zero gradients effectively 'kill' any signal flows through the neuron in forward- and back-propagation. Thus saturated neurons results in a "network [that] will barely learn" [Kar16]. Accordingly, extra care must be taken not to initialise the weights with a large value.

2. The range of the sigmoid function is not centered around zero. Hence if the inputs are always positive, it could introduce an undesirable *zig-zagging dynamics* in the gradient updates for weights, as the gradient on the weights will be either all positive or all negative during back-propagation. However, this is a minor inconvenience as its impact is automatically mitigated in a BGD by the time the weights are updated.

It would be also interesting to implement a SGD neural network to test for its relative performance.

Finally, after optimising the current implementation, there are many complicated neural network architectures that could be built by using this simple neural network as preliminary building blocks. For instance, convolutional neural networks, or *convNets*, are one of the most reliable and efficient performers in image recognition problems. ConvNets work by processing only portions of the input image, which are tiled in subsequent layers in such a way that the input regions overlap and is deemed to obtain a better representation of the original image at a fraction of the computational cost [**?**]. Each of the tiles are a small simple neural network!

Chapter 6

Conclusion

An implementation of a neural network in Accelerate may offer a convenient alternative to current CUDA or Python implementations in testing for the validity of hypotheses.

The aim of the project is to implement a ConvNet in Accelerate. ConvNet is similar to FFBP neural networks, and thus should be possible to build upon the implementation in [CEMCK17].

However, further research and knowledge is necessary in order to achieve this goal.

Bibliography

[Bru16] Lars Bruenjes. neural: Neural networks in native haskell. https://hackage.haskell.org/package/neural, accessed 29/05/2017, 2016. hackage information page and git repository.

[CEMCK17] Robert Clifton-Everest, Trevor L. McDonell, Manuel M.T. Chakravarty, and Gabriele Keller. Streaming irregular arrays. Actual paper is in review, linked is the git repository for neural network work written for the paper, which was used for this thesis, 2017.

[CKL+11] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. *Declarative Aspects of Multicore Programming*, 2011.

[FR64] R. Fletcher and C.M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7:149–154, 1964.

[GdAF13] Sskya T. A. Gurgel and Andrei de A. Formiga. Parallel implementation of feedforward neural networks on gpus. pages 143–149, 2013.

[Kar16] Andrej Karpathy. Neural networks part 1: Setting up the architecture. http://cs231n.github.io/neural-networks-1/, accessed 01/01/2016 to 30/05/2016, 2016. Course notes from CS231n: Convolutional Neural Networks for Visual Recognition at School of Comp. Sci. Stanford University.

[LBBH98] Y. LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffiner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 1998*, 86, 1998.

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1 989.

[LCBnd] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. https://yann.lecun.com/exdb/mnist/, accessed 20/05/2017, n.d. Sample database.

[Mar13] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Sebastopol, CA, 1st edition, 2013.

[Mat17] MathWorks. fminunc. https://au.mathworks.com/help, accessed from 01/03/2017 to 30/05/2017, 2017. MATLAB.

[McD13] Trevor L. McDonell. Gpgpu programming in haskell with accelerate. https://speakerdeck.com/tmcdonell/gpgpu-programming-in-haskell-with-accelerate, accessed 01/04/2016, 2013. YOW! Lambda Jam 2013 Workshop.

[MCGN15] Trevor McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Type-safe runtime code generation: Accelerate to llvm. *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 2015.

[MJ17] Trevor McDonnell and Ji Yong Jeong. seng4911. https://github.com/suzumo/seng4911/blob/master/haskell/src/MHNN.hs, accessed 01/03/2017 to 31/05/2017, 2017. Thesis project repository.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133, 1943.

[Ng16] Andrew Ng. Machine learning. http://www.coursera.org/learnmachine-learning, accessed 01/01/2016 to 30/05/2016, 2016. Coursera Inc.

[Nie15] Michael Nielson. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Nor14] Russell Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education Limited, New York City, New York, 3rd edition, 2014.

[NVI14] NVIDIA. Cuda spotlight: Gpu-accelerated deep neural networks. https://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-deep-neural-networks, accessed 14/04/2016, 2014. NVIDIA Accelerated Computing.

[OJ04] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37:1311–1314, 2004.

[Reb13] Jason Rebello. Logistic regression with regularization used to classify hand written digits. https://au.mathworks.com/fileexchange/42770-logistic-regression-with-regularization-used-to-classify-hand-written-digits, accessed 14/05/2017, 2013. MATLAB.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Ros62] Frank Rosenblatt. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Spartan Books, Washington, D.C., 1962.

[Wol17] Alan Wolfe. Neural network recipe: Recognize handwritten digits with 95accuracy. https://blog.demofox.org/2017/03/15/neural-network-recipe-recognize-handwritten-digits-with-95-accuracy, accessed 25/05/2017, 2017. Blog post.