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

Supervisor: A/Prof. Gabriele Keller

Student ID: z2250189

Topic ID: 3733

Abstract

GPU-accelerated programming is used to speed up neural network computations. However, the main languages for GPU programming are CUDA and OpenCL, which are very low-level languages. This thesis proposes implementing a simple feed-forward back-propagation neural network, namely a convolutional neural network using Accelerate. Accelerate is an Embedded Domain-Specific Language in Haskell for GPU programming and has several accessibility advantages over CUDA, while still offering competitive performance. Once the implementation is successful, I plan to assess its performance, benefits and disadvantages against the traditional approaches with CUDA.

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

List of Figures

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 implemented with GPU-accelerated computing are common, as such arrangements are generally faster than with a CPU cluster. The main languages for GPU programming are Compute Unified Device Architecture (CUDA) or 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 GPU 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 [Eve16].

This project aims to build upon that work and create a convolutional neural network (ConvNet), which is often specialised for image recognition problems.

It is crucial that the performance of the implementation is fairly competitive to existing high-level language implementations of neural networks, such as Python. Thus ways to enhancing the performance will also be explored once the basic implementation is finished.

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

Chapter 3 introduces my thesis proposal and predicts some issues as well areas that need further development.

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 $\boldsymbol{\theta} =$

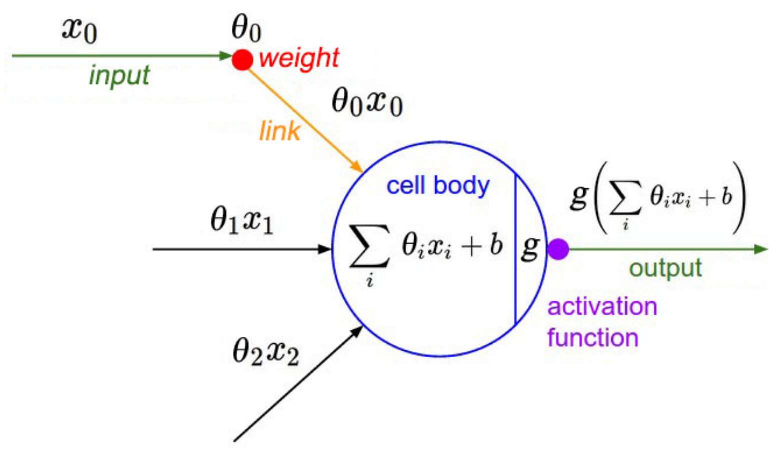


Figure 2.1: Structure of a modern unit by [Kar16].

$[\theta_1, ..., \theta_m]$ 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, $max(w_1^T x + b_1, w_2^T x + b_2)$.

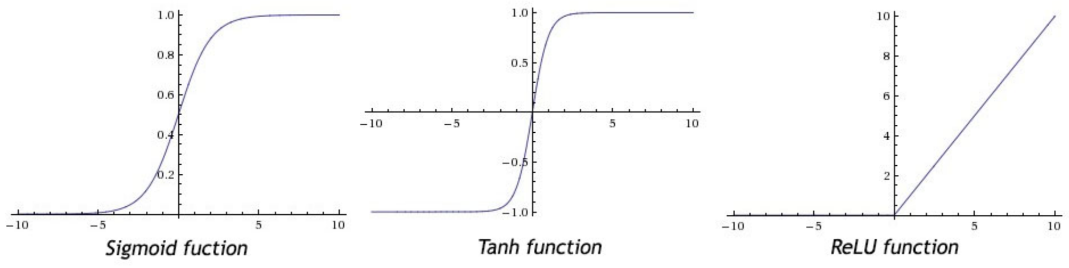


Figure 2.2: Graphs of some common activation functions [Kar16].

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].

There are numerous neural network classifications depending on their architecture; the design relevant to this thesis is *supervised*[3] FFBP neural network. Its training process

[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.

[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

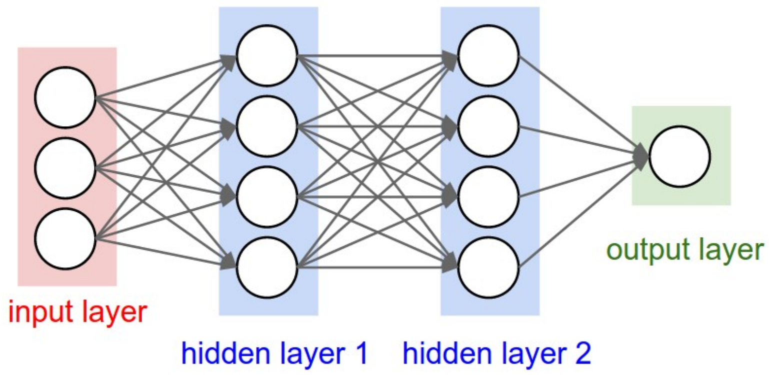


Figure 2.3: An example of a 2-layer neural network [Kar16].

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.
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*.

---

neural network thus adjusts its links' weights to get the correct output values during its training phase.

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 $[-1, 1]$.

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 Ng (2016), [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].

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

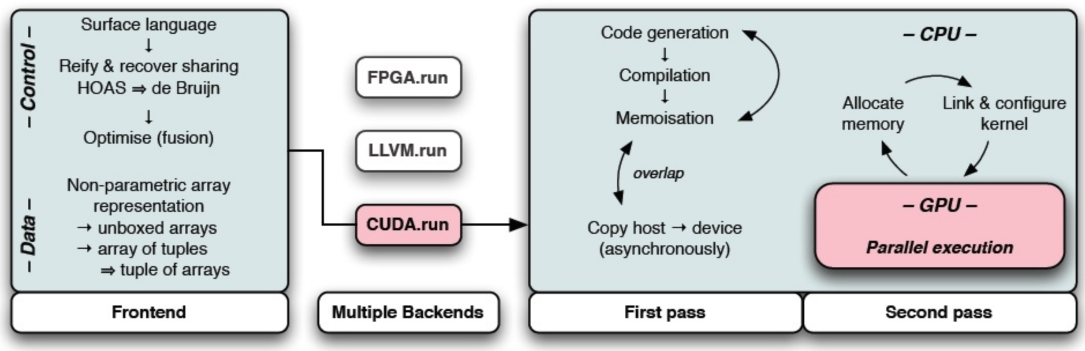


Figure 2.4: Structure overview of `Data.Array.Accelerate`. [CKL+11]

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 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.

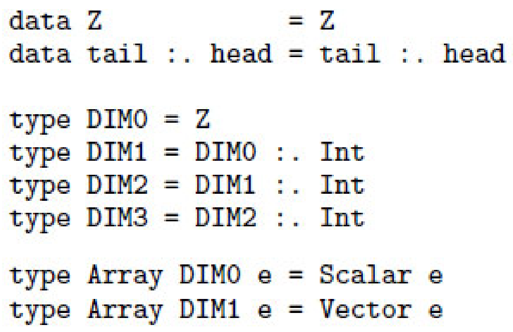
2.4 Accelerate

Accelerate is an Embedded Domain-Specific Language (EDSL) for GPU programming, 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 code that runs directly on the GPU.

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



```
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].

instance `Double`, `Int` or tuples. However, `e` cannot be an array type; that is, Accelerate 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 executes* [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.

Further details about the Accelerate language is continued in 2.5.

2.4.1 Advantages of Accelerate

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 [Mar13]. For instance, Fig. 2.7 shows a dot product function

```
-- dot product in Haskell
dotp :: [Float] -> [Float] -> Float
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)

-- dot product in Accelerate
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Figure 2.7: Haskell and Accelerate versions for dot product [McD13].

comparison in Haskell and in Accelerate by [McD13]. There are minimal syntactic difference; for example, in the input and output types (Haskell's version takes and gives Haskell arrays while Accelerate's `dotp` deals with only Accelerate arrays), and in that Haskell's `foldl` is a left-to-right traversal, whereas Accelerate's `fold` is neither left nor right as it occurs in parallel [McD13].

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; and, GPUs particularly require an extremely tight control flow due to massive numbers of threads that are generated. 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. An example of this is the use of types `Acc` and `Exp`, which separates array from scalar computations and prevents many cases of nested data parallelism, which is unsupported in Accelerate.

Finally, Accelerate is a dynamic code generator [CKL$^+$11] and as such it can,

- Optimise a program at the time of code generation simultaneously by obtaining information about the hardware capabilities.
- Customise the generated program to the input data at the time of code generation.
- Allow host programs to "generate embedded programs on-the-fly".

On the other hand, the disadvantages of using Accelerate over CUDA are 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$^+$11]; and, neural networks certainly fit in such a category of programs. Accelerate also uses a number of other techniques to mitigate the overheads, such as fusion, sharing recovery, caching and memoisation of compiled GPU kernels [CKL$^+$11].

In terms of performance, Accelerate can be competitive with CUDA depending on the nature of the data input and the program [MCKL13].

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 [Eve16]. In it, Everest (2016) implements a FFBP neural network with a *stochastic gradient descent* (SGD) learning algorithm. 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]. 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 less accurate than BGD, and that it may take longer to reach the local/global minima.

The activation function is the sigmoid function. This function is mathematically convenient, because its gradient, $\sigma'(x)$, is easy to calculate:

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

The sigmoid function suffers from the following two disadvantages and is now mostly unused [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.

The third interesting observation is how Accelerate was utilised in this implementation. First, there are six key Accelerate functions that were used and their names and operations are listed in Fig. 2.8.

`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]. Functions `lift` and `unlift` are used to inject and extract a value into and out of `Exp`, respectively, and are thus essential in interpreting the indices of arrays in Accelerate [Mar13].

Two support functions were also created to assist forward- and back-propagation processes – `mvm` and `cross` (see Fig. 2.9). 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-wise multiplied with `mat` using `zipWith`. Lastly, the values of the matrix are summed and flattened back into a vector using `fold`.

```
  -- 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 2.8: Key Accelerate functions used in [Eve16].

```
-- 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 2.9: Support functions for forward- and back-propagation used in [Eve16].

```
-- 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 2.10: Accelerate and Haskell code are mixed in [Eve16].

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. 2.10.

At the time of this report, Everest (2016) has reported some performance issues with this implementation, but has yet to determine the cause. He noted that the implementation was running at a slower speed than an equivalent neural network that was implemented with Theano. Theano is a Python library that is optimised for multi-dimensional array computations.

Chapter 3

Implementation

3.1 Using a MATLAB-based neural network as reference

There are several motivations for choosing this particular implementation of neural network as reference. First, MATLAB implementation is heavily reliant upon the mathematical interpretation of neural network and is clearly understandable where matrix multiplication and vectorisation is concerned. It was projected that it would be fairly simple and in the closest-in-spirit implementation to how it may work in Accelerate due to this.

Secondly, the MATLAB implementation had a certain data set with expected 'answers' to compare my own implementations to. Thus, I could be assured that my implementation in Accelerate was accurate enough to be reliable, particularly in the light of difficulty of debugging Accelerate programs. Passing this 'accuracy' test will prove the small network reliable enough to be built upon for further development in the future.

Thirdly, the codebase that I am basing my program is from Coursera, an online self-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 with much knowledge in the field. 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, C++, or Haskell, (PUT REFERENCES TO THESE IMPLEMENTATIONS) which may needlessly increase the time to start the project. I had researched about other implementations, but found that it took more time to understand these in order to integrate or implement an Accelerate neural network.

TO CHECK::::: Can C++ or Python versions use multiple threads/GPU?

3.1.1 Key differences from Accelerate

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, object-orientated, procedural language. It is widely used among the industry and academia, particularly in science, engineering and economics.

MATLAB is a weakly typed programming language, as types are implicitly converted. This means that variables can be declared without an explicit declaration of their type [Unk17]. Furthermore, MATLAB is also dynamically typed, meaning that the types may change during runtime.

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 and does not reveal the actual structure of the underlying operation just by looking at the code.

Other points include starting index of matrices and arrays being 1, not 0. Also, MATLAB method dispatching does not strictly adhere to the method signature, but selects the appropriate method by first matching the arguments in a class precedence and then by left-most priority [Mat17].

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 implementing two extra matrix multiplications in feed-forward ad backpropagate parts, plus all the steps required to update the new theta layer calculations (see 3.1).

3.2.2 Program structure

In terms of program structure, I closely followed the flow in the MATLAB version. It can be portioned into 3 parts: (1) initialisation; (2) neural network cost function; and, (3) function minimise conjugate gradient function. 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 there are `theta1`, `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`. The number of 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

```
-- 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`

of neurons in the hidden layer is a bit of a mystery and seems to be done by trial-and-error. In 4.1, the hidden units are 25 neurons and 300 neurons for the two different handwritten numbers data sets and these are taken from other implementations with the same data sets, who had found these the most effective numbers by testing.

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 prefix `A`, `P` for Accelerate and Prelude correspondingly, for instance `A.++` and `P.++`. Other key matrix manipulating functions used throughout the program include `A.take, A.drop, A.map` and `A.zipWith`.

An 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.

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

```
  -- 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.

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 massive batch to parse them in all at once for higher accuracy.

As `nnCostFunction` is used in conjunction with `fmincg` function 3.2.2, it was necessary to flatten all the weight vectors into one before passing into, and also out of the `nnCostFunction`. The fused vector is sliced and reshaped this its original shapes inside `nnCostFunction`, which is implemented using `reshape` as shown in 3.2.

`reshape` is tricky to maneuver at first, because it must be supplied with a `Exp sh`. `Exp sh` can be made in various ways and sometimes requires its type to be defined. For example, in the case where one `lifts` a `(Z:.m:.n)` form to supply `fill` with an `Exp sh`,

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

Thus, I found operations involving `Exp sh` quite tricky at times due to typing.

Matrix multiplications was initially implemented using `mmult`, but was eventually replaced with `hmatrix` package, which has a better performance.

Function minimise nonlinear conjugate gradient function, `fmincg`

This function repeats `nnCostFunction` for certain number of iterations, and progressively tries to minimise the error in the weight matrices. 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].

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 '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.

`fmincg` 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.

`fmincg` 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 to see progress made throughout the iterations upon checking `fX`. Other variables are discarded.

My Accelerate implementation 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.

```
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.

This is to enable `fmincg` to execute the function indepedent of 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).

I faced a couple of challenging factors in implementing this function. First, the MATLAB version had approximately 17 parameters, most of which overwrote itself and interacted with each other. Without a knowledge of the underlying mathematics, I could not get fully comprehend the purpoes behind this 175-line procedure. The ambiguity and similarity of parameter names, such as `d1, d2, d3, f1,..., v1,...`, did not assist the issue.

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, and optimise incrementally later (for which, I ran out of time). Certain expressions were substituted where operations had not yet been implemented in Accelerate (see 3.3). I could not get the flow control perfectly, however, 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 flat-data parallelism with flow divergence may slow down the GPU execution. Flat-data parallelism is where a sequential operations can be applied in parallel over bulk data.

On the positive side, the flow divergence was 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 assump-

```
    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 with predefining types of unused variables in `unlift`.

tion 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`.

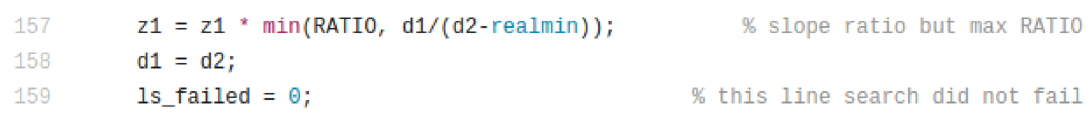
To explain the implementation more concretely, I divided the MATLAB code into their `while` loops. These were implemented using Accelerate's `awhile` flow control.

`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. `lift` and `unlift` are necessary to pack and unpack the variables each loop for condition checks and for loop execution. An interesting fact I learnt was that Accelerate cannot infer the types of some variables unless if they are used after an `unlift` process. It is thus sometimes necessary to predefine types of certain terms to assist Accelerate 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 enough to withstand wrong inputs as well as the MATLAB programs. For instance, I have already mentioned in 3.2.2 that the function supplied to `fmincg` must be correct.

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' properities, correspondingly.



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

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

As these functions were not yet available during the time of development in Accelerate, I substituted them for expressions which I believe will cover those particular situations. For instance, `isreal` check may arise when an determinate may be less than zero; `isnan` check may arise when divisor is 0. I was not able to devise a check for `isinf`. Thus 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 `Doubles` produces a result further away from the MATLAB result than had I parsed them in Accelerate as `Floats`. Not only that, the accuracy of the Accelerate neural network predictions after training decreased with the same training set when the data was parsed as a `Double`.

The source of the inaccuracy may be due to `fmincg`. There is one line that seems to account for an overflow, by minusing a `realmin` from a dividend as seen in 3.5. `realmin` for a `float` is defined as $1.1755e^{38}$ and for a `double` is $2.2251e^{308}$ in MATLAB. These are orders of magnitude of $1xe^{270}$ times different and could vastly affect the result. In my implementation, I have currently taken out `realmin` from the equation.

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, almost off by 10 per cent (see Chapter 4).

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].

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].

(1) is a 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. 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 as. 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 check the values that I get from my program with the values in the MATLAB program. In 4.3 I evaluated the comparative performance of Accelerate and MATLAB, up to 2 cores.

(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 images. 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.

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

Second is a multi-core testing configuration. 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 (see 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.

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.

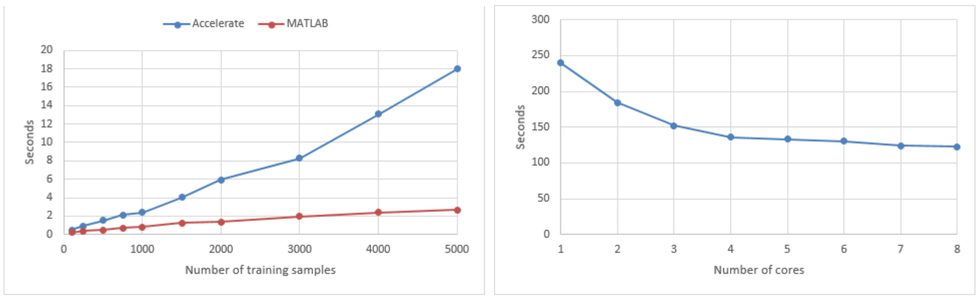


Figure 4.1: Graphs of training set 1 (left) and 2 (right) benchmarks.

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).

Chapter 5

Evaluation

The intent of this thesis was to evaluate Accelerate's suitability in creating a neural network 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

I believe that Chapter 4 training set 2 results indicate that a decently well-performing neural network should be possible.

Current implementation has an issue with accuracy: according to [LBBH98], a two-layer fully connected neural network 300 hidden units for MNIST dataset should produce an error of approximately 4.7%. According to the paper, LeCun uses a carefully tuned, SGD algorithm to train the neural network 2.5. My implementation, on the other hand, uses BGD with conjugate gradient method. Perhaps due to the bugs in `fmincg` or otherwise, my implementation has a high error rate of approximately 15.5%, which is closer to the error rate of LeCun's classic linear classifier at 12%. A linear classifier architecture is similar to a neural network without a hidden layer.

In terms of speed, I have found a C++ neural network that tested on the MNIST dataset[Wol17], but with only 30 hidden layers. This reduces the number of first weight vector from 235500 to 23550 and second weight vector from 3010 to 310. This implementation also uses the faster SGD method following [LBBH98]. Althought the testing environment of [Wol17] is unclear, the training period is said to be 82 seconds.

With 8 cores, my Accelerate implementation runs at 122.3 seconds. This is despite the fact that this implementation is more or less a direct translation of the MATLAB code with minimal Accelerate naturalisation, and the list of unresolved issues (see 3.3). With further optimisation and bug fixes, it may be possible to achieve a very reasonable performance and accuracy.

5.2 Benefits of Accelerate implementation

Acclerate has inherent benefits.

For instance, even if the code is 'inefficient' in the sense that it has repeating parameters, by having 'sharing recovery', Accelerate will reduce the number of parameters to the bare minimum during the production of the ASTs.

5.3 Incomplete works

This was the expected behaviour –¿ First, the comparative training set prediction results from Accelerate and MATLAB using sample data (1) indicates that my Accelerate implementation does align with the MATLAB program and should be thus, fairly accurate (see 4.3).

But what I got was –¿ even if prediction is accurate, it does not always apply to everything else

Second: talk about difficulties with developing with Accelerate – debugging. Maths.

Type checking.

I did not conduct testing the neural network with different activation functions as I was aiming to test the speed and load performance of my implementation and such activation functions should not have a drastic impact on the performance of the neural network.

– a full comparative analysis of other languages' implementation difficulty, performance, GPU harnessibility

5.4 Future works

– from thesis A about creating testing data

Another problem that I may encounter using such pre-made repositories is that there may be insufficient training samples in the provided data set to train my neural network. In machine learning, a concept called *artificial data synthesis* may resolve this dilemma [Ng16].

Artificial data synthesis comprises of the following two methods to create *synthetic data*:

1. Create new *labeled* samples by superimposing original sample with a suitable replacement. For example, an original image of an alphabet can be edited to a new sample by overlaying the existing letter with a new font type of the same letter.
2. Amplify the data set by introducing *smart* distortion to original sample. For instance, edit an original image of an alphabet into multiple versions by introducing various wave-like distortions.

Artificial data synthesis is generally more efficient than obtaining raw data and labeling them manually. In the second method, the distortion must be non-trivial and the resulting sample must lie within a naturally occurring variance – it must be reasonably found in the real world [Ng16].

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 [Eve16].

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

Bibliography

[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.

[Eve16] Rob Everest. Untitled. 2016.

[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.

[MCKL13] Trevor McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *ACM SIGPLAN International Conference on Functional Programming*, 2013.

[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.

[Unk17] Unknown. Matlab. https://en.wikipedia.org/wiki/MATLAB, accessed 14/05/2016, 2017. MATLAB, Wikipedia article.

[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.