School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Implementing neural networks in Accelerate for GPU execution

by

Ji Yong Jeong

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: 24 May 2016 Student ID: z2250189

Supervisor: A/Prof. Gabriele Keller 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 also like to thank all the members of UNSW Programming Languages and Systems (PLS) group for additional support and assistance, and for being an invaluable wealth of knowledge. In particular, I would like to thank Liam O'Connor for his generous aid.

I am very grateful to have learned so much during Thesis A, and look forward increasing my knowledge further in Thesis B!

Abbreviations

BGD Batch gradient descent learning algorithm

ConvNet Convolutional neural network

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 4 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 $\boldsymbol{\theta} =$

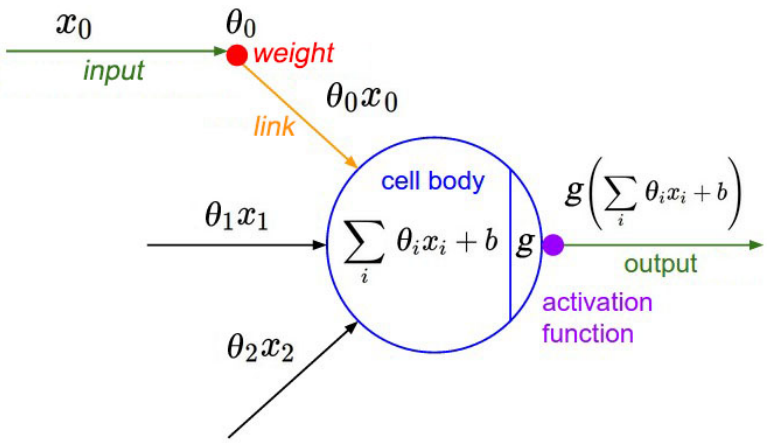

Figure 2.1: Structure of a modern unit by [Kar16].

$[\theta_1, ..., \theta_m]$ 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 2.6. [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, $max(w_1^T x + b_1, w_2^T x + b_2)$.

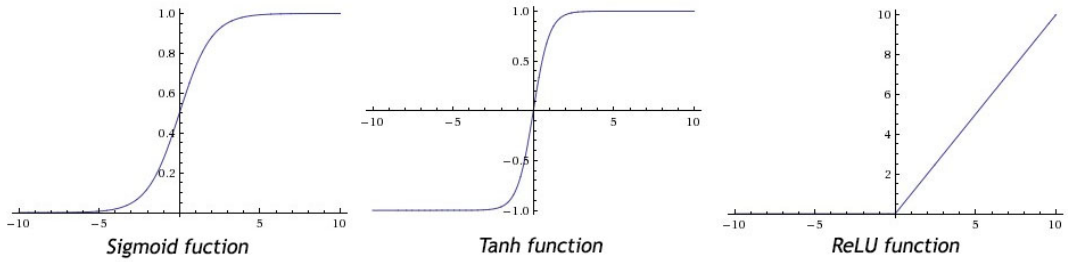

Figure 2.2: Graphs of some common activation functions [Kar16].

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

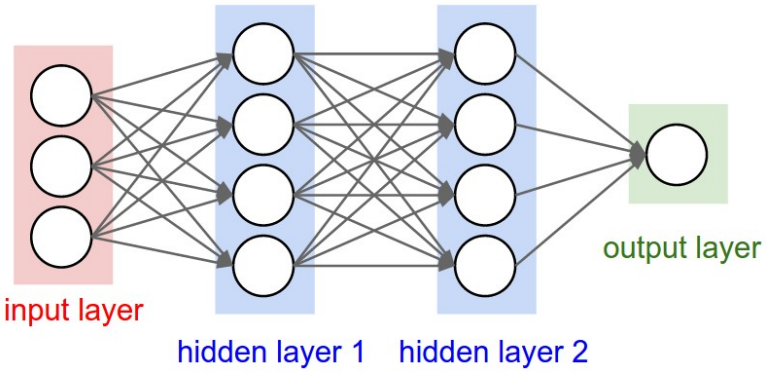

Figure 2.3: An example of a 2-layer neural network [Kar16].

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

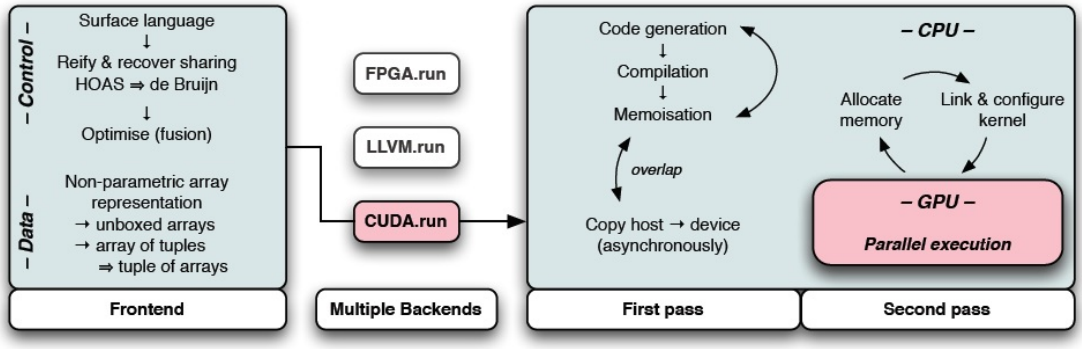

Figure 2.4: Structure overview of `Data.Array.Accelerate`. [CKL+11]

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 2.6.

2.5 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].

2.6 Previous Implementations

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]. 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

Thesis Proposal

3.1 Choice of neural network

For this project, I plan to implement a ConvNet in Accelerate. There are two motivations for choosing this particular neural network. First, ConvNets are currently one of the most reliable and efficient performer in image recognition problems. Second, ConvNets are architecturally very similar to FFBP neural networks [Kar16]; thus, I can build upon the works in [Eve16].

The fundamental concept behind the ConvNet model is based on the works of neurophysiologists David Hubel and Torsten Wiesel in the 1970s that successfully analysed a cat's visual system [BC16]. The first successful implementation was by Yann LeCun (1989) in [LeC89]. Its name comes from the fact that it replaces general matrix multiplication for some of its neural network layers with a mathematical operation called *convolution* [BC16].

The architecture of ConvNet is complicated and at the time of this report, I am still in the process of understanding its mechanism. The principal idea, however, is that it processes 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 [BC16]. The efficiency of ConvNets is partly due to the fact that it assumes that the inputs are of a fixed type, for instance, inputs that are exclusively images [Kar16].

There are also a diverse range of ConvNet architectures; the most common are LeNet, AlexNet, ZF Net, GoogLeNet, VGGNet and ResNet [Kar16]. This also, is an area that needs further research.

3.2 Measuring performance

One of the issues that has been raised during Thesis A is the need to find a suitable test subject, that provides an appropriate complexity, size and number to fully evaluate the performance of the implementation.

Currently, I have tentatively decided to test the system with the basic National Institute of Standards and Technology (MNIST) data set of handwritten digits. Should this prove unsuitable as a test subject, I may try an online repository for image data sets at the University of California, Irvine Center for Machine Learning and Intelligent Systems.

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].

3.3 Completed tasks

These are the objectives that have been completed for this project at this point:

- Complete Coursera online course on Machine Learning.
- Learn about neural network mechanisms.
- Read literature on Accelerate.
- Revise Haskell.

3.4 Areas requiring development

There are a number of problem areas that may require further attention and development in order.

During the course of this project, I have found that I lacked much of the background knowledge to comprehend the reading material, particularly in regards to the machine learning concepts and the programming languages' various semantics. Hence, I plan to build a more comprehensive understanding about the theory of programming languages in semester 2, 2016, while continuing to research about ConvNets.

More concretely, the problem areas and my strategies to combat them are as follows:

1. Need more familiarity with Haskell and Accelerate in general.
    - Learn how to utilise Accelerate through `accelerate-examples` package.
    - Investigate further into the details of Accelerate's internal workings to truly optimise an implementation.
    - Increase functional language knowledge by taking COMP3161 in semester 2, 2016.
2. Gain information, technical details about the practical implementation of ConvNets.
    - Learn from [Kar16, BC16], which seem cover ConvNets in detail.
    - Look for ConvNet implementations in other languages.
    - Research for and pick a ConvNet architecture.
    - Try to make a simple, working implementation as early as possible, and find ways to improve its performance from there.
3. Search for ways to improve the performance of the program once it has been implemented.
    - Learn if program can be improved using other parallel algorithms with better performance.
4. Identify an appropriate set of large sample materials to test the performance of the implementation (refer to 3.2).
5. Obtain a ConvNet implementation in CUDA for the comparison analysis. I believe that this is one of the latter tasks to be completed and plan to discuss with my supervisor at a later point.

3.5 Expected schedule

I intend to undertake Thesis B in Semester 1, 2017. The expected completion of this project is in May, 2017. Predicted timeline is depicted in Fig. 3.1.

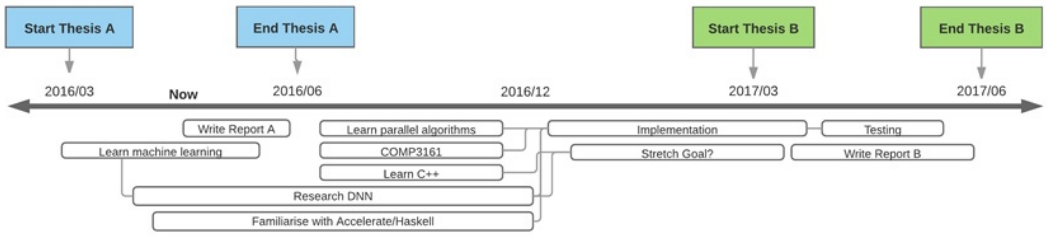

Figure 3.1: Estimated timeline for this project.

Chapter 4

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

[BC16] Ian Goodfellow Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[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.

[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.

[LeC89] Y. LeCun. Generalization and network design strategies. *University of Toronto Connectionist Research Group Technical Report*, 89, 1989.

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

[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.

[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.

[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.