LHask: A Machine Learning DSL in Haskell

Daniel Collin

Bachelor's Thesis in Computer Science
at Stockholm University, Sweden, 2020

Stockholm University

LHask: A Machine Learning DSL in Haskell

Daniel Collin

Bachelor’s Thesis in Computer Science (15 ECTS credits)
Single Subject Course
Stockholm University year 2020
Supervisor at the Department of Mathematics was Lars Arvestad
Examiner was Erik Palmgren

Department of Mathematics
Stockholm University
SE-106 91 Stockholm, Sweden

Abstract

In this degree project report we describe and implement LHask, a domain-specific language (DSL) for machine learning. The design of LHask is inspired by the idea of compositional learning algorithms from the category Learn. Using the Accelerate language LHask is capable of training models on both CPU and GPU. Utilizing dependent types LHask has compile-time checking and deduction of array bounds. Furthermore, LHask is compared to Google's machine learning DSL TensorFlow by evaluating identical models on the problem of classifying handwritten digits from the MNIST database. LHask proves to be capable of expressing feed-forward neural networks and achieve accuracy on the MNIST problem similar to that of TensorFlow. Benchmarking shows that training LHask and TensorFlow on the MNIST data set take similar amounts of time for the CPU, but also shows that LHask lags behind when it comes to the GPU.

LHask: Ett domänspecifikt språk för maskininlärning i Haskell

Sammanfattning

I denna exjobbsrapport beskriver och implementerar vi LHask, ett domänspecifikt språk för maskininlärning i Haskell. LHasks design är inspirerad av den centrala tanken om sammansättningsbara inlärningsalgoritmer från kategorin Learn. Genom att använda Accelerate-språket kan LHask träna modeller både på CPU och GPU. Med hjälp av beroende typer kan LHask vid kompilering härleda och kontrollera uppställningars längd. För att verifiera implementationen så jämförs LHask med Googles TensorFlow genom att träna identiska modeller i båda ramverken i att klassificera handskrivna siffror från databasen MNIST. Det visar sig att LHask kan uttrycka feed-forward neurala nätverk och uppnå liknande träffsäkerhet som TensorFlow. Jämförelser visar att LHask och TensorFlow behöver ungefär lika mycket tid för att träna MNIST på CPU, men visar också att LHask är långsammare än TensorFlow på GPU.

Contents

Chapter 1

Introduction

The last few years have seen machine learning being employed in many different tasks over the world. As the models that are being trained grow more complex, there is demand for libraries and programming languages both to include domain-specific constructs like activation functions and differentiable programs and general programming capabilities like data preprocessing, I/O and control flow. Moreover, the resulting program needs to be able run on both CPUs and GPUs in order for the machine learning computations to scale. This need has led to DSLs being developed specifically for machine learning, one example being Google's TensorFlow [19].

The compositional nature of machine learning as observed by Fong et al. [7] provides a framework in which machine learning models can be understood as tuples of functions. Similar notions are explored by Olah [18] in how different machine learning constructs can be formulated in functional programming.

While functional programming seldom is used for numeric computing, the Accelerate language [1] equips Haskell with the ability to generate correct and performant numeric code that can run on many different targets.

We describe and implement the machine learning DSL *LHask*, written in Haskell and Accelerate, and use it to classify handwritten digits from the *Modified National Institute of Standards and Technology* (MNIST) database [14].

As we only cover the minimal amount of building blocks required to train on the MNIST database using standard methods, LHask is in no way as fully developed in terms of extent as an established DSL like TensorFlow.

1.1 Limitations

Comparing performance in terms of speed between DSLs with runtimes separated from the host language is no easy feat. We only provide rudimentary benchmarks by measuring the mean of wall clock time it takes to run the respective applications using the *perf* program from the Linux kernel. This does of course not only measure the time it takes to train the models, but also the time it takes to initialize the runtimes.

When comparing the different DSLs in terms of accuracy in training it is a bit simpler. By ensuring that loss functions and accuracy are computed in the same way *mathematically* we compare results after a certain amount of passes through the training data. We expect that a correct implementation of LHask should yield similar results to TensorFlow, modulo floating point differences and quality of random numbers generated for initial weights.

1.2 Code

The implementation described in Chapter 3 can be found at https://github.com/vonpost/LHask.git with instructions for reproducing the training done in Chapter 4.

Chapter 2

Theory

2.1 Neural networks

Formally speaking a *neural network* is a weighted digraph describing a function divided into sub graphs known as *layers*. The first layer is the *input layer*, the last layer the *output layer* and all layers between are known as *hidden layers*. The *width* or *size* of a layer is the number of vertices in that layer.

As seen in Figure 2.1 each layer is represented by a vertical row of vertices, of which none are connected to each other. If v' is a vertex with incoming edges $e_1 ... e_m$ originating at vertices $v_1 \ldots v_n$ the *value* at v' is given by output of a so called *activation function* which has as its input the sum of all incoming edges' weights multiplied by the value of the vertex from which the edge originates. Often a *bias* term, a constant, is added to each vertex's value.

An activation function is normally a non-linear, differentiable function. This non-linearity together with a single hidden layer allows for the neural network to approximate any function fulfilling certain constraints [10]. Some common activation functions include:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

$$^1\,\text{relu}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

[1] Rectified linear unit (ReLU) is not differentiable at $x = 0$, but this is not a problem in practice [9].

We say that a neural network has a *fully-connected* layer if all vertices in one layer are connected to all vertices in the subsequent layer. One can easily formulate a fully-connected neural net in matrix form: $f(x) = \sigma(Wx + b)$ where $f(x)_j = \sum_i w_{ji}x_i + b_j$ and σ is the activation function.

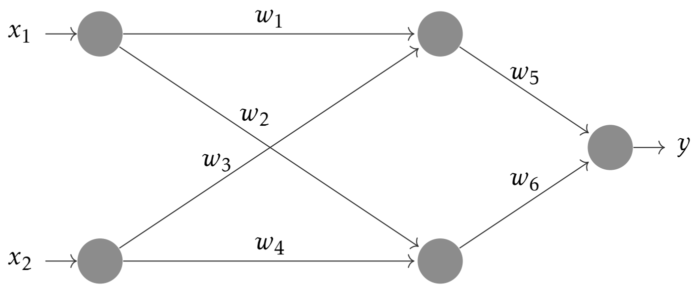

Figure 2.1: Fully-connected neural network representing the function $f(\bar{x}) = \sigma_2(W_2\sigma_1(W_1\bar{x} + b_1) + b_2)$

Neural networks are mainly used for approximating functions, the act of which we will from here on call *learning*.

Suppose that we have a function $g : \mathbb{R}^n \to \mathbb{R}^m$ that we wish to approximate. The output at the end of a chain of neural network layers $f_3 \circ f_2 \circ f_1 : \mathbb{R}^n \to \mathbb{R}^m$ is in the codomain $g(x)$ by letting the final layer be a function $f_3 : \mathbb{R}^k \to \mathbb{R}^m$ that transforms the output of the previous layers to be in that very codomain. Furthermore, the activation function of f_3 is often chosen such that it models the actual function we wish to approximate.

For example, one activation function that is often used when trying to classify what class the input belongs to among k different classes is the *softmax* function [9]:

$$\text{softmax} : \mathbb{R}^k \to \mathbb{R}^k$$
$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_i^k e^{x_i}}$$

After applying softmax to some vector x all of its components will be in the range $[0, 1]$. Furthermore, the sum of all the components will be 1 which is why the output of the softmax function can be interpreted as each component $f(x)_j$ being a probability that the input belonged to the jth class.

2.2 Gradient descent

Gradient descent [9] is an optimization algorithm finding a minimum of a differentiable function by iteratively taking steps in the opposite direction of its gradient. From calculus we know that by walking in the opposite direction of a function's gradient we should approach a local minimum.

If f is a neural network, consisting of some layers composed together, we extend our definition in Figure 2.1 to let the function be parametrized by its weights W such that $f : \mathbb{R}^k \times \mathbb{R}^n \to \mathbb{R}^m$. We then apply an *optimizer* which, given some training example of input and output, gives us new weights $W' \in \mathbb{R}^k$ such that $f(W', x)$ better approximates g than $f(W, x)$.

When training neural networks gradient descent is known as *stochastic gradient descent* (SGD) [9]. This is because rather than approximating the gradient of the neural network itself, we approximate the gradient of an error function between the output of a neural network and the true output using samples from the true distribution. One pass over the data set is usually referred to as an *epoch*.

If we define a function $h(W, x, y)$ that in some adequate way, depending on the function we wish to approximate, measures how well $f(W, x)$ approximates $g(x)$ then SGD is applied to the function f by taking an ϵ-step, usually referred to as the *learning rate*, in the opposite direction of the gradient of h with respect to W:

$$W' = W - \epsilon * \nabla_W h(W, x, y)$$

When $h(W, x, y) = e(f(W, x), y)$ the function $e : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is often referred to as the *error*, *cost* or *loss* function. Common examples of such functions are:

- *mean squared error*:

$$\text{mse}(\hat{y}, y)_i = \frac{1}{n} \sum_i^n (\hat{y}_i - y_i)^2$$

- *cross-entropy*:

$$\text{ce}(\hat{y}, y) = - \sum_i^n y_i * \log(\hat{y}_i)$$

2.2.1 Batch training

When using SGD as the optimizer for a neural network one has to compute the gradient sample by sample. This can be inefficient for GPUs and other hardware with a high level of parallelism.

One way to utilize parallelism when computing the gradient for SGD is by training on *batches* [9] which consist of the entire data set. Instead of taking a vector as an input the neural network now takes a matrix of vectors where either the columns or rows are all considered as an individual sample.

The matrix notation used in 2.1 for one layer of a fully-connected neural network when batch training becomes the equally succinct

$$XW + B$$

where X and B now are matrices and W is the weight matrix as before.

When computing the loss of a batch one either calculates the sum or the average across the batch. This can either be done inside the loss function, or when calculating the gradient, as sum and average are unchanged under derivation. Taking the average has the advantage that the loss function is comparable between batch sizes.

However, a high number of samples in a batch can lead to slower convergence [15]. Additionally, GPUs have fairly limited memory compared to the amount of RAM available to a CPU which means that computing over the entire training set rarely is feasible. In practice a compromise is often chosen where one trains over *mini-batches*, splitting the data set into n into matrices consisting of m training examples. Often m is chosen to be a power of 2 such as 32 or 64 which tends to improve performance on GPUs. [9]

2.2.2 Backpropagation

Backpropagation is an algorithm for computing the gradient of an arbitrary composed function given knowledge of the gradients of its composed parts. When training a neural network model with SGD, or another similar optimizer which requires gradients, backpropagation is often used [9].

The backpropagation algorithm is best understood by noticing a fundamental observation regarding the chain rule: suppose we have some function $f(x) = y$ and some function $g(y) = z$ and we wish to compute dz/dx, the derivative of a composed function, the chain rule then tells us that the partial derivative with respect to x is given by $dz/dx = dz/dy \cdot dy/dx$. Hence, if we know an expression for dz/dy and dy/dx then we know dz/dx.

If we construct our functions with building blocks of smaller functions that have known derivatives, we can always compute the derivative of any composed function we can come up with. Furthermore, since the composed functions share building blocks we can reuse many of the expressions computed in the different composed parts. This allows us to avoid approximation of gradients as well as purely symbolic derivation, both of which tend to be computationally expensive [9].

2.3 Learn

Suppose we have a learning algorithm that learns how to produce outputs of type B given inputs of type A. Suppose further that we have another learning algorithm that produces type C given inputs of type B, would it not be reasonable then to ask what the *composed* learning algorithm that produces type C given inputs of type A looks like? Compositionality tends to be the domain of category theory which in turn has a tight-knit relationship to Haskell.

Learn is a monoidal category where objects are sets and morphisms are tuples[2] consisting of the following components: a set called the *parameter space* P, an *implementation function* $I : P \times A \to B$, an *update function* $U : P \times A \times B \to P$ and a *request function* $R : P \times A \times B \to A$. [7]

As Learn is a category we must be able to compose morphisms. Composition of two morphisms is defined by a series of compositional rules for each individual component in the tuples.

The implementation function I is simply a function parametrized by some parameter space P. Hence, its compositional rule is simply to apply the composed functions with their respective parameters:

$$J \circ I(q, p, a) := J(q, I(p, a))$$

The update function, given some some training examples of input/output $(a, b) \in A \times B$ and some parameter $p \in P$ produces a new parameter $p' \in P$, ideally such that $I(p', -)$ is now closer than $I(p, -)$ to the function it wishes to approximate. As such, the composed update function has to produce parameters for both parts of the composed morphism. The request function is used to produce an output example for the update function of the innermost morphism:

$$U_J \circ U_I(p, q, a, c) := (U_I(p, a, R_J(q, I(p, a), c)), U_J(q, I(p, a), c))$$

For the request function itself, its compositional rule will propagate the request function from outer-most composed morphism to the request function of the innermost function:

$$R_J \circ R_I(p, q, a, c) := R_I(p, a, R_J(q, I(p, a), c))$$

Since Learn is a monoidal category it is also equipped with a sort of vertical composition, the tensor product. Like composition, tensoring is defined for each component of the tensored tuples separately.

1. $(P \otimes Q) := (P \times Q)$
2. $(I \otimes J)(p, q, a, b) := (I(p, a), J(q, b))$
3. $(U_I \otimes U_J)(p, q, a, b, c, d) := (U_I(p, a, b), U_J(q, c, d))$
4. $(R_I \otimes R_J)(p, q, a, b, c, d) := (R_I(p, a, b), R_J(q, c, d))$

The two component morphisms of a tensored morphism in Learn are independent from each other, as can be seen in the implementation, update and request function rules for tensoring: to compute one component in their output tuple no knowledge of the other morphism is needed.

[2]In order to satisfy certain laws of monoidal categories they are actually equivalence classes of tuples. However, for the sake of brevity, we will refer to them as tuples.

We illustrate the parallel nature of tensoring, and the sequential nature of composition, with *string diagrams* in Figure 2.2 and Figure 2.3, which describe the flow of input and output in morphisms of monoidal categories.

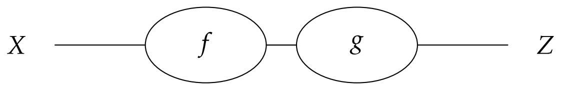

Figure 2.2: String diagram illustrating the morphism $g \circ f : X \to Z$.

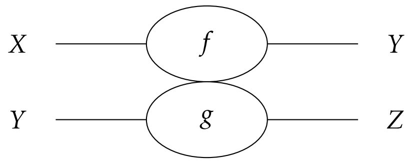

Figure 2.3: String diagram illustrating the morphism $f \otimes g : X \otimes Y \to Y \otimes Z$.

Since g in Figure 2.2 depends on the output of f, the composed morphism $g \circ f$ can be thought of as "first f, then g". In contrast, g in Figure 2.3 is independent of f, hence the morphism $f \otimes g$ can be thought of "both f and g".

Learn is general enough to encompass any learning algorithm from some set A to some set B. We can define specific morphisms in Learn that have SGD as their update function. There is a functor that, given some learning rate ϵ and error function $e : \mathbb{R} \to \mathbb{R}$, takes a parametrized and differentiable function $I : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^K$ to a morphism in Learn:

$$\begin{aligned}
U_I(p,a,b) &:= p - \epsilon \nabla_p E_I(p,a,b) \\
E_I(p,a,b) &:= \sum_k e(I_k(p,a), b_k) \\
R_I(p,a,b) &:= f_a(\nabla_a E_I(p,a,b)) \\
f_a(x)_i &:= \Big(\frac{\partial e}{\partial x}(a_i, -)\Big)^{-1}(x)_i
\end{aligned}$$

Note that the update function is indeed SGD as described in 2.2.

2.4 Accelerate Language

The Accelerate language [1, 4, 16, 17] is a DSL in Haskell for computing parallel array operations. Evaluating a Haskell program written in Accelerate defines an expression which in turn gets evaluated by the Accelerate runtime with the assigned back-end. By generating intermediate code for the Low Level Virtual Machine (LLVM) compiler framework Accelerate can retain Haskell's functional nature while targeting specialized hardware such as GPUs and multi-core CPUs.

Operating over Accelerate arrays is done with familiar functions such as `map`, `fold` and `zipWith`. Furthermore, the *accelerate-blas* package exports Basic Linear Algebra Subprograms (BLAS) [3, 5] bindings for efficient matrix multiplications and other linear algebra operations.

An Accelerate array `Array sh e` is an array parametrized by a shape `sh` and `e`, the type of its elements. A computation `Acc a` is a computation with a given Accelerate back-end which results in a value of type `a`.

Notably missing from the set of allowable types in an Accelerate array is an array itself, this means that Accelerate does not allow nested arrays: each parallel computation itself is sequential and cannot further dispatch other parallel computations.

Composing different functions into a pipeline of operations is a common pattern in functional programming. However, naively implemented this can create intermediate data structures. Accelerate will instead attempt to combine operations that work over entire arrays. Suppose we have the following expression:

```
map (+ 2) (map (+ 3) (arr :: Acc (Array sh e)))
```

The `map` function is a collective operation in Accelerate that applies its argument function to each element in the array in parallel. When Accelerate during runtime constructs an Accelerate expression it will notice that we are computing two expressions for each element, and optimize into something like:

```
map (\x -> x+2+3) arr
```

In this way, Accelerate can compute `\x -> x+2+3` for each element in parallel, with only one traversal of the array. This is what is known as *array-fusion* [16]. However, Accelerate will only attempt to fuse operations which dispatch parallel work on the elements. Consider instead the following expression:

```
map (+ 2) (fold (+) 0 (arr :: Acc (Array sh e)))
```

Since the input of `map` is the resulting array computed by `fold`, and `fold` requires all the elements in the array to compute its result, Accelerate will not fuse these operations into a single pass over the array.

One peculiarity of utilizing the Accelerate language is the fact that code is being written in a meta-programming fashion. As such, we have to be careful to not blow up the expression that Accelerate generates. One example of this is how one naively might circumvent the fact that Accelerate does not allow nested arrays:

```
foldl' (++)
       (acc1 :: Acc (Vector Float))
       (xs :: [Acc (Vector Float)])
```

This expression would evaluate to an Accelerate expression which concatenates all arrays in the list. Accelerate will attempt to optimize the expression with array fusion, however since the expression grows in size with a magnitude relative to the length of the list `xs` so does the optimization time. Hence, it is not desirable to try to avoid the limitation in this way.

The more Accelerate-friendly way to apply an Accelerate function multiple times is by using the higher order function `run1 :: (Acc a -> Acc b) -> (a -> b)`. This function tells Accelerate that the function can be compiled for the target code immediately, since the only thing that will change is the arguments and not the actual function body.

In order to use `run1` the function it is applied to needs to be from, and to, a single Accelerate computation. This can easily be remedied by putting multiple arguments inside a tuple, and then `curry` the function after applying `run1`. In the example above we could use `curry $ run1 $ \(a,b) -> a ++ b` as the folding function.

2.5 Backprop

Backprop [13] is a library written in Haskell in order to facilitate backpropagation of arbitrary differentiable Haskell functions. Any function that operates over types which are instances of `Backprop` can be rewritten as a backpropagatable function. For a type to be an instance of `Backprop` it needs to define the following methods:

```
class Backprop a where
zero :: a -> a
one :: a -> a
add :: a -> a -> a
```

The method `add` is to be thought of as adding together contributions of gradients. For scalar values, they are added with ordinary addition. When the values being added are containers, such as arrays or lists, they are added element-wise.

The `zero` method sends a value to its “zero” in the sense that when added to a gradient it does not change that gradient. For instance, an array gets sent to an array of the same shape consisting of only zeroes.

Similarly, `one` sends a value to its “one”, such that the gradient of the function `id a = a` is `one`. For an array this translates to sending the array to an array of the same shape consisting of only ones.

Suppose then that we have `instance Backprop A` and `instance Backprop B`, and furthermore that we wish to rewrite a function `f :: A -> B` as a backpropagatable function. Then the resulting type signature of the backpropagatable variant of `f` will be `f' :: (Reifies z W) => BVar z A -> BVar z B`. The constraint `Reifies z W` can be seen as saying that any `BVar z a` is part of the same backpropagation algorithm and thus any common expressions between these can be shared.

We can evaluate the original function with the higher order function:

```
evalBP :: (BVar z a -> BVar z b) -> a -> b
```

This function returns the original function `f`. More importantly, we can ask a function of its gradient with:

```
gradBP :: (BVar z a -> BVar z b) -> a -> a
```

Given `f'` this function will return the gradient of `f'`.

Writing a function `f' :: (Reifies z W) => BVar z A -> BVar z B` requires us to only write our function in terms of `BVar`, essentially restricting our building blocks to functions which are already differentiable.

We can also give an explicit definition for a specific function by lifting it into the `BVar` context with the `liftOp` function. This requires us to formulate a *scaled gradient*, something reminiscent of the request function from the category Learn. Recall that backpropagation works by computing $dz/dx = dz/dy \cdot dy/dx$, the scaled gradient is then the function that given dz/dy computes dz/dx. For scalar $\theta = dy/dx$ this is quite trivial, of course, since then dz/dx equals $dz/dy \cdot \theta$.

For more complicated functions defining the scaled gradient often includes transforming dz/dy to be of the same type as dy/dx, and then combining them with a suitable operation. If f is an isomorphism it can also be made backpropagatable by specifying its inverse using:

```
isoVar :: (a -> b) -> (b -> a) -> (BVar z a) -> (BVar z b)
```

2.6 Dependent types in Haskell

Simply speaking *dependent types* are types that depend on values. Unlike some functional languages, like Idris [11] and Agda [2], Haskell does not have first-class dependent types. It is however possible to achieve dependent types in Haskell by means of *type families* [6].

Type families are type-level functions, which allow for a sort of dependent types. One such type family for which the *Glasgow Haskell Compiler* (GHC) has built-in support is the `KnownNat` type family which is used as a constraint for a function to say that some type-level natural number, a type of kind `Nat`, is known at compile-time.

For example, suppose that we have a dependently typed list which is indexed by its length:

```
newtype DList (n :: Nat) a = DList [a]
```

It is dependent, because the type of the list depends on the length of the list. As such, `DList`s of different lengths are different types.

We can create a function for constructing a three element `DList`. Since we know that the list cannot have any other length than 3, we can at compile-time specify the length to GHC:

```
lengthThree :: a -> a -> a -> DList 3 a
lengthThree a1 a2 a3 = DList [a1,a2,a3]
 lengthThree 1 2 3 :: DList 3 Integer
```

GHC supports basic arithmetic expressions for `Nat` with type-families that put constraints in the type signature of functions involving natural number types. For example, we can define concatenation of two `DList`s:

```
concatD :: DList n [a] -> DList m [a] -> DList (n+m) [a]
concatD (DList xs) (DList xs') = DList (xs ++ xs')
 concatD (lengthThree 1 2 3) (lengthThree 2 3 4) :: DList 6 Integer
```

Natural number types can also be reflected down to values with the function

```
natVal :: forall n proxy. KnownNat n => proxy n -> Integer
```

which takes some proxy carrying the type `n` and reflects the type of the proxy down to an `Integer` with value n. This way we can, for instance, compute the length as a *value* at of two `DList`s:

```
addLengths :: (KnownNat n,
               KnownNat m)
           => DList n [a]
           -> DList m [b]
           -> Integer
addLengths DList _ Dlist _ = (natVal (undefined :: n))
                             + (natVal (undefined :: m))
```

The fact that we use `undefined`, which causes the GHC runtime to throw an exception when evaluated, is of no importance. The only thing that matters for `natVal` is the type of `undefined`, which we have set to be `n`. GHC will then convert `n` into its runtime representation as a value, which is n.

Note that, in contrast to `concatD`, we have to specify the constraint `KnownNat` for the function `addLengths`. This is because in order to compute the combined length of the two lists as *values* from their type-level indexing, we must know at compile-time what their respective lengths are.

Chapter 3

Implementation of LHask

Many of our design choices in LHask draw heavily from *A Purely Functional Typed Approach to Trainable Models* [12], in which the author describes how to compose machine learning models using the dependently-typed linear algebra package *hmatrix-static* as its linear algebra back-end and *backprop* for backpropagation. However, one of the big differences from LHask is the fact that *hmatrix-static* computes values using the GHC runtime, whereas LHask uses the Accelerate runtime allowing computations to be run on GPUs.

In LHask we try to capture the compositional nature described in the category Learn, but for design reasons we use the *backprop* library for backpropagation which then replaces the request and update functions.

In *A Purely Functional Typed Approach to Trainable Models* we have the type:

```haskell
type Model p a b = forall z. (Reifies z W)
                 => BVar z p
                 -> BVar z a
                 -> BVar z b
```

This type actually mirrors that of the morphisms in the category Learn. In LHask we will call them *learners* and define them as:

```haskell
type Learn p a b = forall z. (Reifies z W)
                 => BVar z p
                 -> BVar z a
                 -> BVar z b
```

There is an advantage in decoupling the update and request functions from the actual morphism, namely that the morphisms from Learn are now function types rather than tuples, allowing us to use the well-developed language support in Haskell for ordinary functions.

Since any function `f :: BVar z p -> BVar z a -> BVar z a` implicitly implements backpropagation by simply having arguments of the type `BVar`, the only thing that is not immediately available in comparison to a morphism in Learn is the update function. We can recover the update function with `gradBP` and `gradBP2`, the ability to ask any model of its gradient, and device our own update function:

```
update :: LearningRate
       -> ErrorFunction
       -> Learn p a b
       -> p
       -> a
       -> b
       -> p
update rate e i p a b = p - rate*gradient
        where gradient = (gradBP $ \p -> e (i p $ auto a) (auto b))
```

This is essentially the update from the category Learn achieved by asking the gradient of some error function `e` applied to output of some implementation function `i`, given some input and output examples `(a,b)`. Naturally, we could extend this example to implement a variety of different optimizers other than SGD by simply describing their functions in a similar manner.

3.1 Statically known bounds

We build dependently typed Accelerate arrays by packing them in the following newtypes:

```
newtype Dim (n :: Nat) t = Dim t
newtype R n = Dim n (Acc (Vector Float))
newtype L m n = Dim m (Dim n (Acc (Matrix Float)))
newtype SR n = Dim n (Vector Float))
newtype SL m n = Dim m (Dim n (Matrix Float)))
```

A newtype is mathematically isomorphic to the type it wraps. As such GHC erases any distinction between a newtype and that which it wraps during compilation, and hence newtypes carry no performance overhead during runtime.

We define a layer of dimension for each dimension of the array and then finally pack it inside a newtype to give concrete type information. So a vector will be packed in a single `Dim` whereas a matrix will packed inside a twice nested `Dim`.

Each dimension is parametrized by `n :: Nat`, a type-level natural number. A vector of length 1 and one of length 2 are in LHask of types `SR 1` and `SR 2`, and hence they are different types. This way we can make sure during compile-time that no operations are being done with mismatching shapes.

Furthermore, we define `Num`, `Fractional` and similar arithmetic instances for our dependently typed Accelerate computations allowing us to compute expressions such as:

```
exp (1 :: R 1) * (2 :: R 1) :: R 1
log (1 + 2 * 3) :: R 1
```

All the arithmetic functions have to be defined *component*-wise since, for example, the signature of multiplication is `Num a => a -> a -> a`. Because of this, vector and matrix multiplication does not align with the arithmetic type classes.

We define some type classes to handle conversion between runtime bound-checked Accelerate computations and arrays and our dependently typed versions to quickly be able to go between them:

```
class Packable a where
  type Unpack a
  pack   :: Unpack a -> a
  unpack :: a -> Unpack a

class (Packable a,
       Packable (Run a),
       Arrays (Unpack (Run a)),
       Unpack a ~ Acc (Unpack (Run a)))
       => Runable a where
  type Run a = result | result -> a
  run :: a -> Run a
  use :: Run a -> a

  instance Packable (R n) where
  type Unpack (R n) = Acc (Vector Float)

  instance Runable (R n) where
  type Run (R n) = SR n
```

Note that the `Unpack` type family is not injective since, for instance, `(R n)` will be taken to an `Acc (Vector a)` but since the compile-time information regarding the length of the vector is lost it is not clear what the other direction would be. This is the intended behavior, since non-dependently typed Accelerate arrays are considered unsafe in comparison to the dependently typed ones and lack the sufficient type information required to determine what the resulting type should be. It is up to the user to guarantee that the unsafe operation of packing an Accelerate computation is done correctly.

The `Run` type family on the other hand is injective since if we run it we get a dependently typed vector. In practice, this means that the compiler can always deduce the type of `Run a`, but if `Pack a` is not clear from the context then we have to annotate manually what the type should be:

```
      run (R 1) :: SR 1
      use (SR 1) :: R 1
```

```
unpack (R 1) :: Acc (Vector a)
unpack (R 2) :: Acc (Vector a)
pack (Acc (Vector a)) :: ?
```

We also define the category Learn's tensor product of objects for our dependently typed Accelerate computations:

```
newtype a ⊗ b = Tpl (Acc (Unpack (Run a), Unpack (Run b)))
```

The tuple type might have a curious definition, and indeed if we were to quite literally implement the tensor product of Learn, which is Cartesian product for objects, the natural choice would be ordinary tuples. However, `(Acc a, Acc b)` is not a single Accelerate computation and as such we cannot use `run1`. If we instead have tensor to mean the unpacked types being tupled in the output of an Accelerate computation then we retain the ability to use `run1` on nested tensors.

Using the GHC extension

```
{-# LANGUAGE PatternSynonyms #-}
```

we can define patterns for projecting and pattern matching on the respective components of a tensor. We do this both for an ordinary tensor and a tensor inside a `BVar`:

```
pattern (:::) ::  (Runable a,
                   Runable b)
              => a
              -> b
              -> a ⊗ b

pattern (:*:) ::  BVar z a
              ->  BVar z b
              ->  BVar z (a ⊗ b)
```

3.2 Learning in LHask

To do training in LHask we will need a few more tools. First of all, we need some way of gluing together different learners. We can implement composition of learners as it is described in Section 2.3 with a higher-order function:

```
(~>) :: Learn p a b
     -> Learn q b c
     -> Learn (p ⊗ q) a c
(~>) i j (p:*:q) = j q . i p
```

Since the composed learner is again a learner it is backpropagatable. We can also tensor morphisms, as in the category Learn, with the combinator:

```
(⊗) :: (Learn p a c)
    -> (Learn q b d)
    -> (Learn (p ⊗ q) (a ⊗ b) (c ⊗ d))
(⊗) i j (p:*:q) (a:*:b) = (i p a):*:(j q b)
```

However, tensoring in LHask is not quite as useful as composition. Since functions in Accelerate, like `map` and `fold`, already operate over entire arrays it is often more convenient to define a learner for a single array of some length, than to tensor individual operations up to that length.

Recall the fully-connected neural network on batches described in Section 2.2.1. To recreate such a network in LHask we will first need backpropagatable matrix multiplication. The Accelerate library *accelerate-blas* exports BLAS matrix multiplication as the function:

```
(<>) :: (Num a)
     => Acc (Matrix a)
     -> Acc (Matrix a)
     -> Acc (Matrix a)
```

This function is implemented either using some implementation of BLAS when the target is a CPU, or using NVIDIA's cuBLAS library when the target is a NVIDIA GPU.

We can define a backpropagatable, compile-time bound-checked variant of the matrix multiplication in LHask:

```
import Data.Array.Accelerate.Numeric.LinearAlgebra as N
(<>) :: (KnownNat m, KnownNat n, KnownNat k, Reifies z W)
     => BVar z (L m n)
     -> BVar z (L n k)
     -> BVar z (L m k)
(<>) = liftOp2 . op2 $ \(unpack -> m1)
                        (unpack -> m2) -> (pack $ m1 N.<> m2,
                                           \(unpack -> d)
                                           -> (pack $ d N.<> (transpose m2),
                                               pack $ (transpose m1) N.<> d)
                                          )
```

The actual function definition is clear: suppose that we know the dimensions `n`,`k` and `m` then we know that the output will be a matrix of type `L m k`. When it comes to scaled gradient we are guided by Haskell's type system and our dependent arrays: we *know* from the definition of `BVar` that the type of the scaled gradient function has to be `(L m k) -> (L m n, L m k)`. We can achieve this function type by multiplying the incoming gradient D with the transpose of the matrix from the opposite tuple component such that $scaled(X,Y) = (DY^T, X^T D)$. This is exactly the backpropagated value of matrix multiplication [8].

We can then define one layer of a fully-connected neural network as a function, which given an activation function returns a learner:

```
fullyConnectedL :: (KnownNat n, KnownNat m, KnownNat k, Reifies z W)
                => (BVar z (L m k) -> BVar z (L m k))
                -> Learn ((L n k) ⊗ (L m k)) (L m n) (L m k)
                fullyConnectedL act (w :*: b) x =
                                 act $ (x <> w)+b
```

A fully-connected neural network, considered as a learner in LHask, is simply a parametrized differentiable function of type:

```
Learn ((L n k) ⊗ (L m k)) (L m n) (L m k)
```

Its parameter type `(L n k) ⊗ (L m k)` is a tensor consisting of two components: a weight matrix of type `L n k` and a bias matrix `L m k`.

Formulating the neural network layer as a function, and combining multiple layers with the composition combinator ~>, allows us to easily define a three-layer fully-connected neural network:

```
nnet = fullyConnectedL act1 ~>
       fullyConnectedL act2 ~>
       fullyConnectedL act3
```

In order to train this neural network we can use `run1` to create a function `trainOnce` which, given a set of parameters to the neural network and a training example, returns a new parameter from the Accelerate expression generated by the SGD update-function:

```
trainOnce = curry $ run1 $ \(p:::(a:::b)) ->  update rate errorF nnet p a b
```

We can then train the network by folding over an entire data set, using `trainOnce` as the folding function:

```
trainedParameters = foldl nnet
                          initialParameter
                          (trainingData :: [(Input, Output)])
```

However, we do not have to restrict ourselves to using `foldl` to train the neural network. Instead, we could use the state monad and encode the weight as a state and use any traversable structure for the data set:

```
trainedParameters = do
 put initialParameter
 parameters <- forM trainingData $\sample -> do
                p <- get
                let p' = trainOnce p sample
                put p'
                return p'
 return $ last parameters
```

As such, training the neural network does not require a detailed training procedure specific to LHask. We can utilize the state monad, fold or use any other way of accumulating a state in Haskell. This highlights that LHask is well-integrated with the Haskell language: other than actually manipulating the underlying Accelerate arrays, which requires specialized operations, LHask is able to take advantage of common Haskell idioms and abstractions.

Chapter 4

Comparing LHask with TensorFlow

The neural network trained in this section was a fully-connected layers with hidden layer size 512 and with output activation function softmax. Learning rate was set to a constant 0.01.

The MNIST database contains 60,000 training examples, where one training example consists of a 28×28 picture of a handwritten digit and a label classifying the picture as a certain digit.

The loss function that was used was batched cross-entropy, with loss averaged across each batch:

$$-\frac{1}{m}\sum_{j}^{m}\sum_{i}^{n}\text{correct} * \log(\text{predicted})$$

Cross-entropy is implemented in LHask as:

```haskell
crossEntropyL :: forall m n z. (KnownNat m, KnownNat n, Reifies z W)
              => BVar z (L m n) -> BVar z (L m n) -> BVar z (R 1)
crossEntropyL x y = (/ m) . negate . sumAllL $ y * (log $ mapB (clipBy 1e-7) x)
  where m = fromInteger $ natVal (Proxy @m)
```

Equivalently for TensorFlow it is implemented as [1]:

```python
...
epsilon_ = _to_tensor(epsilon(), output.dtype.base_dtype)
output = clip_ops.clip_by_value(output, epsilon_, 1. - epsilon_)
return -math_ops.reduce_sum(target * math_ops.log(output), axis)
```

To avoid numerical instability the predicted array has its components clipped to make sure that the resulting logarithm is neither too small nor too big.

[1]https://github.com/tensorflow/tensorflow/blob/r1.13/tensorflow/python/keras/backend.py

The neural network itself is defined in LHask as:

```
nnet = fullyConnectedL (mapB relu) ~> fullyConnected @512 softMaxL
```

The TensorFlow code used can be found in Appendix A.

Initial weights for both TensorFlow and LHask were sampled from a uniform distribution. The training data was not shuffled between epochs in order to keep comparisons fair as LHask does not yet have the capability to shuffle data sets.

For measuring the accuracy the following formula was used:

$$\frac{\text{number of correct predictions}}{\text{number of possible predictions}}$$

All benchmarks were run using an Intel i5 6500 CPU and a NVIDIA GTX 1080 GPU with single-precision floating-points for calculations.

4.1 Accuracy

The graphs displayed in Figures 4.1 to 4.6 were trained for 10 epochs and all show similar results for LHask and TensorFlow across all batch sizes.

The largest difference can be seen in the loss function on batch size 32 in Figure 4.1 and Figure 4.4 where the loss is slightly higher for TensorFlow compared to LHask. For batch sizes 64 and 128 the graphs of TensorFlow and LHask are practically identical.

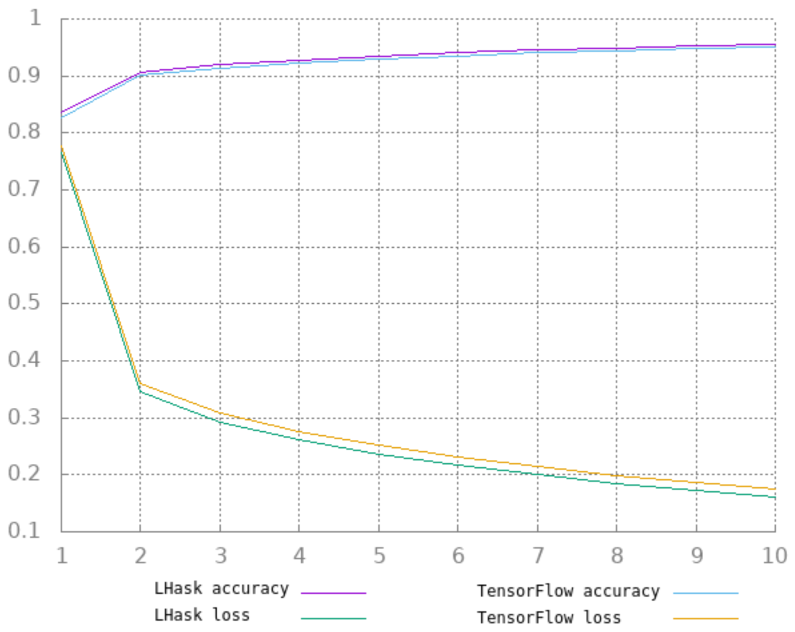

Figure 4.1: CPU training on MNIST with LHask and TensorFlow with batch size 32 trained.

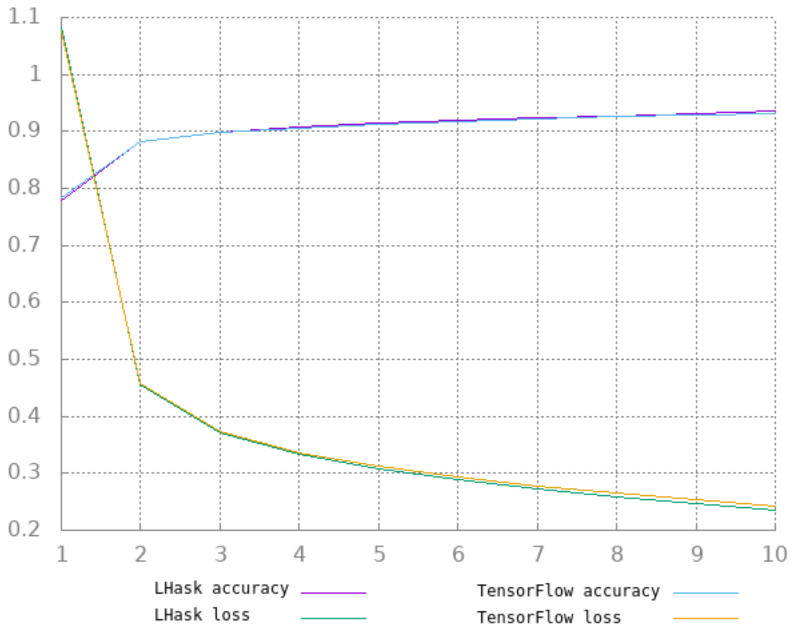

Figure 4.2: CPU training on MNIST with LHask and TensorFlow with batch size 64 trained.

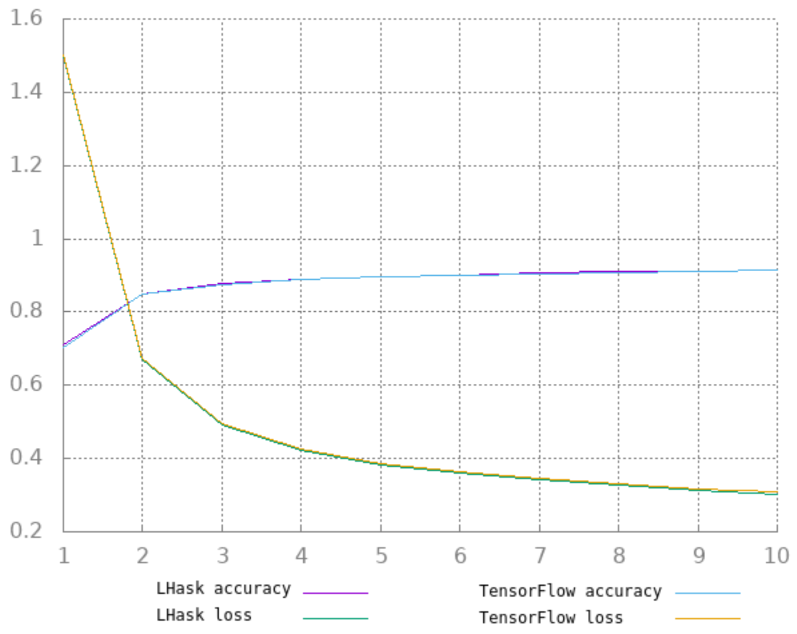

Figure 4.3: CPU training on MNIST with LHask and TensorFlow with batch size 128 trained.

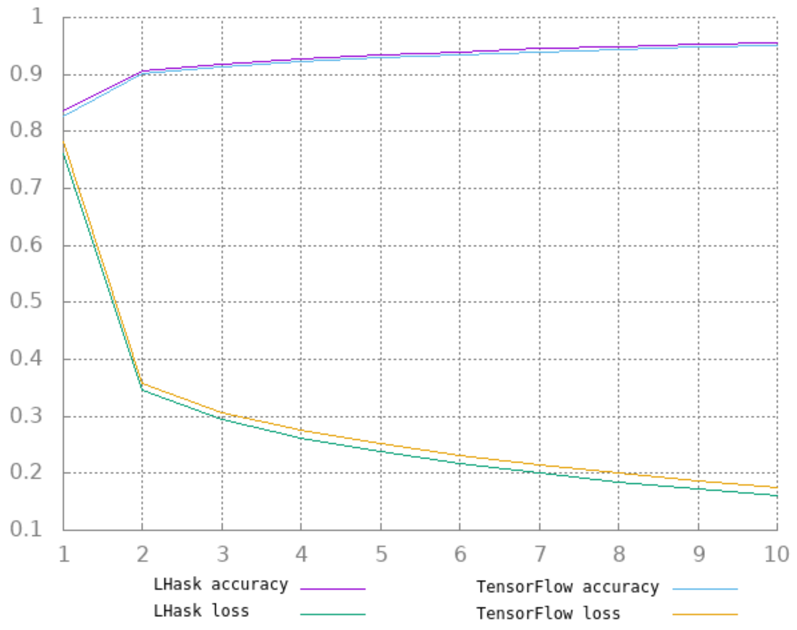

Figure 4.4: GPU training on MNIST with LHask and TensorFlow with batch size 32.

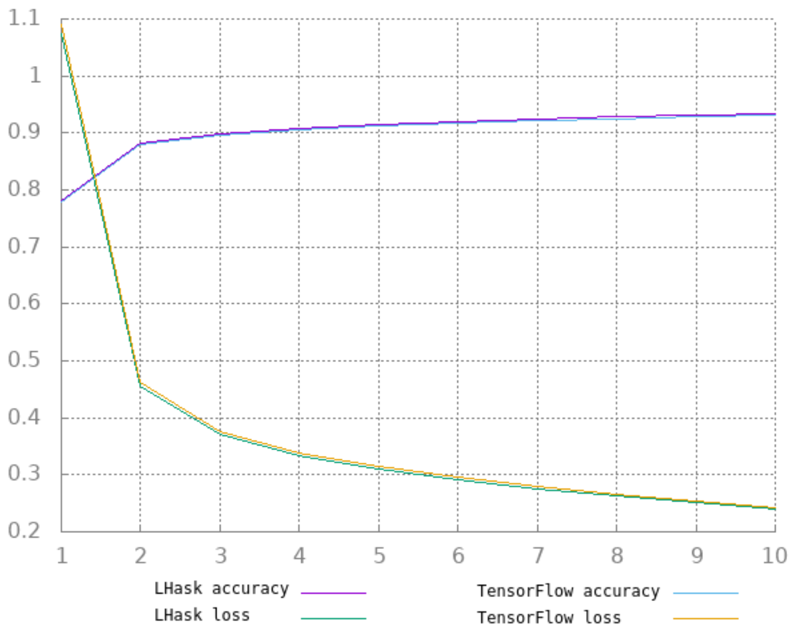

Figure 4.5: GPU training on MNIST with LHask and TensorFlow with batch size 64.

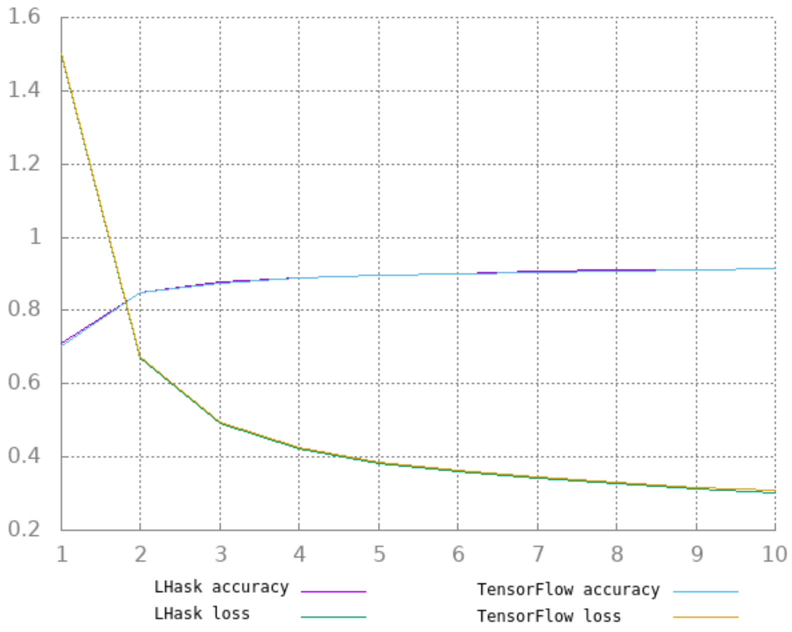

Figure 4.6: GPU training on MNIST with LHask and TensorFlow with batch size 128.

4.2 Performance

Figure 4.7, 4.8 and 4.9 display the mean time spent by TensorFlow and LHask after training the neural network from scratch on the MNIST database. The mean time was averaged over 10 runs.

Additional profiling data for the GPU can be found in Appendix B provided by NVIDIA's profiling tool *nvprof*. The raw time data used for Figures 4.7 to 4.9 put into tables can be found in Appendix C.

Figures 4.8 and 4.9 show that TensorFlow with GPU was by far the fastest, with second place belonging to TensorFlow with CPU in all but batch size 128.

Interestingly in Figure 4.7, which differs from the other graphs quite much, TensorFlow with GPU was the slowest for batch size 128 and the fastest for batch size 32. Instead in Figure 4.7 LHask with CPU was the fastest, and LHask with GPU the second fastest, for batch sizes 64 and 128.

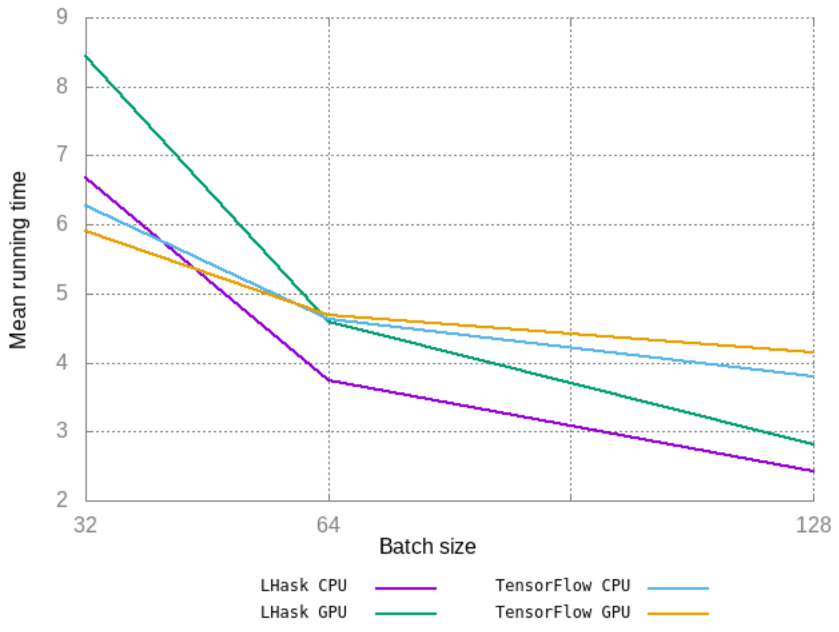

Figure 4.7: TensorFlow and LHask running time (seconds) when training MNIST for 1 epoch.

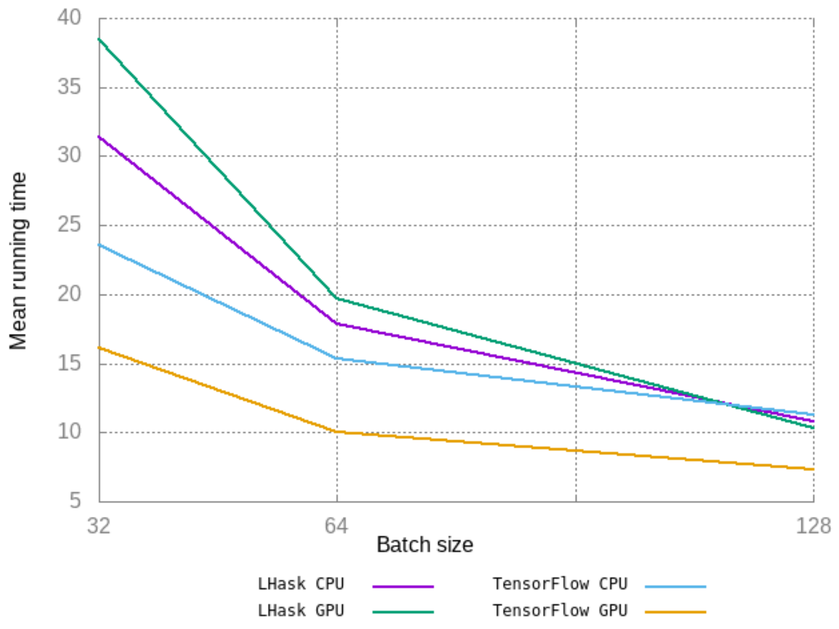

Figure 4.8: TensorFlow and LHask running time (seconds) when training MNIST for 5 epochs.

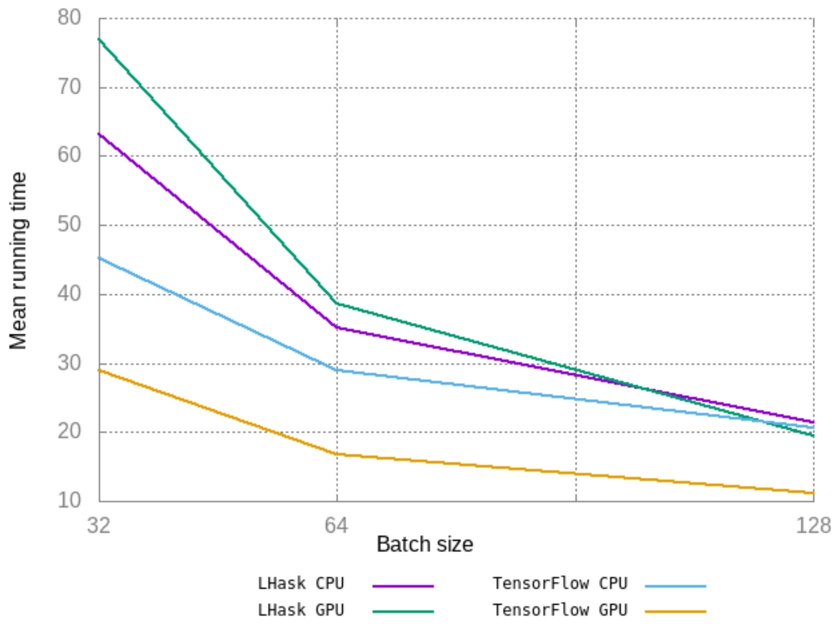

Figure 4.9: TensorFlow and LHask running time (seconds) when training MNIST for 10 epochs.

Chapter 5

Discussion

The fact that LHask achieved almost identical loss and accuracy as that of TensorFlow for all three batch sizes indicates that LHask is capable of learning MNIST with the given model to the same degree as TensorFlow can. Furthermore, since nothing in the implementation is specific to MNIST as a data set, it is reasonable to expect this result to extend to other data sets than MNIST.

Since training was only done with a single model, we cannot generalize and say that LHask is capable of learning to the same level as TensorFlow for *any* machine learning model. LHask as described in Section 3 cannot express more complicated machine learning models than fully-connected neural networks. If LHask in the future implements more complicated models it would warrant further testing. That being said, the results do indicate that backpropagation and SGD work as intended.

Looking at the benchmarks, TensorFlow is quicker than LHask in almost every benchmark other than a few batch sizes when running a single training pass. This could imply that while TensorFlow is faster, LHask is quicker at initializing its runtime system. Indeed, the entries in Table C.1 reinforce this notion as the LHask entries are the fastest in relation to TensorFlow when the number of epochs is low.

As for why the GPU implementation of LHask is *much* slower than that of TensorFlow, and often slower than the CPU implementation of LHask, we can look to profiling. In the detailed profiling output in Section B.1, we can see on line 2 that LHask spends a majority of the GPU time copying arrays from the GPU to the CPU. The profiling output of TensorFlow in Section B.2 looks more sound in this regard: as indicated by line 11, only a small amount of time is spent copying arrays from the GPU to the CPU. Since the only array that *has* to be copied between batches is the two-element array containing the loss and accuracy, optimizing the amount of memory transfers could could prove to be low-hanging fruit for improving LHask's performance on the GPU.

5.1 Conclusion

We have seen in Section 3 that LHask is fully capable of expressing basic models such as fully-connected neural networks. Furthermore, we have in Section 4 verified that such models are capable of learning at a similar rate to TensorFlow.

LHask still needs some optimization before it can compete with TensorFlow in terms of performance, but it has no problem arriving at the same loss and accuracy as TensorFlow after processing the same amount of samples from the data set. Profiling output suggests that redundant data transfers might be a major area to target in optimizing LHask.

Bibliography

[1] *Accelerate Language*. URL: https://www.acceleratehs.org/ (visited on 05/22/2019).

[2] *Agda*. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (visited on 06/10/2019).

[3] *BLAS*. URL: http://www.netlib.org/blas/ (visited on 05/22/2019).

[4] Manuel M T Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, Jan. 2011.

[5] *cuBLAS*. URL: https://developer.nvidia.com/cublas (visited on 05/22/2019).

[6] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. University of Pennsylvania. 2016. eprint: arXiv:1610.07978.

[7] Brendan Fong, David I. Spivak, and Rémy Tuyéras. *Backprop as Functor: A compositional perspective on supervised learning*. 2017. eprint: arXiv:1711.10455.

[8] M.B. Giles. "Collected matrix derivative results for forward and reverse mode algorithmic differentiation". In: *Advances in Automatic Differentiation* (2008), pp. 35–44.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[10] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. DOI: 10.1016/0893-6080(91)90009-t. URL: https://doi.org/10.1016/0893-6080(91)90009-t.

[11] *Idris*. URL: https://www.idris-lang.org/ (visited on 06/10/2019).

[12] Justin Le. *A Purely Functional Typed Approach to Trainable Models*. 2018. URL: https://blog.jle.im/entry/purely-functional-typed-models-1.html (visited on 05/22/2019).

[13] Justin Le. *Backprop Library*. URL: https://backprop.jle.im/index.html (visited on 05/22/2019).

[14] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *MNIST Database*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 06/10/2019).

[15] Dominic Masters and Carlo Luschi. *Revisiting Small Batch Training for Deep Neural Networks*. 2018. eprint: arXiv:1804.07612.

[16] Trevor L. McDonell et al. "Optimising Purely Functional GPU Programs". In: *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, Sept. 2013.

[17] Trevor L. McDonell et al. “Type-safe Runtime Code Generation: Accelerate to LLVM”. In: *Haskell ’15: The 8th ACM SIGPLAN Symposium on Haskell.* ACM, Sept. 2015, pp. 201–212.

[18] Cristopher Olah. *Neural Networks, Types, and Functional Programming.* 2015. URL: https://colah.github.io/posts/2015-09-NN-Types-FP/ (visited on 05/22/2019).

[19] *TensorFlow.* URL: https://www.tensorflow.org/ (visited on 06/10/2019).

Appendices

Appendix A

TensorFlow code

Adapted from the TensorFlow beginner example for training MNIST [1].

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test =
  tf.keras.utils.to_categorical(y_train, 10), tf.keras.utils.to_categorical(y_test, 10)
y_train.shape, y_test.shape

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu,
kernel_initializer='RandomUniform', bias_initializer='RandomUniform'),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax,
kernel_initializer='RandomUniform', bias_initializer='RandomUniform')
])
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=[tf.keras.metrics.categorical_accuracy])

hist = model.fit(x_train, y_train, epochs=5, batch_size=64, shuffle=False)
```

[1]https://www.tensorflow.org/tutorials

Appendix B

nvprof output

The profiling data in Sections B.1 and B.2 is output from *nvprof*[1] applied to GPU training of MNIST with batch size 64 for 5 epochs. DtoH means device-to-host. Some lines are shortened and marked with [...].

B.1 LHask

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   67.88%  2.73350s     57004  47.952us     960ns  697.04us  [CUDA memcpy DtoH]
                   14.14%  569.38ms    108535  5.2460us     800ns  24.097us  generate
                    4.04%  162.73ms      9682  16.807us  12.737us  20.321us  sgemm_32x32x32_NN_vec
                    3.00%  120.92ms     35610  3.3950us     832ns  557.96us  [CUDA memcpy HtoD]
                    2.78%  111.94ms     19208  5.8270us  2.5920us  12.769us  fold
                    2.61%  105.23ms      9682  10.868us  7.3930us  12.513us  sgemm_32x32x32_NN
                    2.49%  100.38ms      4685  21.424us  20.417us  22.401us  maxwell_sgemm_128x64_nt
                    0.95%  38.205ms      4685  8.1540us  7.6480us  12.736us  sgemm_32x32x32_TN
                    0.93%  37.294ms      9682  3.8510us  2.3040us  11.616us  foldAllS
                    0.92%  37.198ms      4685  7.9390us  7.6800us  10.592us  sgemm_32x32x32_NT
                    0.25%  10.239ms      4689  2.1830us  1.2160us  10.849us   map
      API calls:   35.21%  4.15423s     57004  72.876us  10.396us  38.329ms  cuMemcpyDtoHAsync
                   13.19%  1.55661s   2100029     741ns     393ns  964.46us  cuStreamQuery
                   12.56%  1.48182s   1056856  1.4020us     178ns  5.3122ms  cuEventDestroy
                   11.12%  1.31227s    142114  9.2330us  6.7960us  382.41us  cuLaunchKernel
                    8.60%  1.01471s   1318573     769ns     379ns  400.07us  cuEventQuery
                    4.56%  538.08ms   1058362     508ns     182ns  393.53us  cuEventCreate
                    3.93%  463.23ms   1058362     437ns     245ns  1.4308ms  cuEventRecord
                    3.55%  418.94ms     33419  12.535us  9.6980us  356.90us  cudaLaunchKernel
                    2.51%  295.68ms     35609  8.3030us  4.7320us  339.90us  cuMemcpyHtoDAsync
                    2.37%  279.72ms         1  279.72ms  279.72ms  279.72ms  cudaFree
                    1.17%  137.82ms         1  137.82ms  137.82ms  137.82ms  cuCtxCreate
```

[1]https://developer.nvidia.com/cuda-toolkit

```
                        0.64%  75.602ms      3300  22.909us  1.7120us  2.1020ms  cuMemAlloc
                        0.22%  26.541ms     45131     588ns     283ns  352.68us  cuStreamWaitEvent
                        0.12%  14.339ms     28430     504ns     167ns  351.84us  cuCtxPushCurrent
                        0.10%  11.997ms     28431     421ns     219ns  17.612us  cuCtxPopCurrent
                        0.07%  7.8654ms     33419     235ns     161ns  14.482us  cudaGetLastError
                        0.03%  3.6751ms       185  19.865us     928ns  1.0583ms  cuStreamCreate
                        0.02%  2.6122ms        21  124.39us  54.424us  336.64us  cuModuleLoadDataEx
                        0.01%  860.32us       146  5.8920us     106ns  244.03us  cuDeviceGetAttribute
                        0.01%  770.22us         2  385.11us  297.85us  472.37us  cuDeviceTotalMem
                        0.00%  158.51us       148  1.0710us     847ns  3.2540us  cuEventSynchronize
                        0.00%  87.709us         2  43.854us  40.887us  46.822us  cuDeviceGetName
                        0.00%  55.543us         1  55.543us  55.543us  55.543us  cuModuleUnload
                        0.00%  24.059us       155     155ns     124ns     364ns  cuFuncGetAttribute
                        0.00%  23.255us         3  7.7510us  3.7600us  12.142us  cudaMalloc
                        0.00%  10.447us        31     337ns     139ns     702ns  cuModuleGetFunction
                        0.00%  9.6060us         1  9.6060us  9.6060us  9.6060us  cudaMemcpy
                        0.00%  7.0910us        16     443ns     355ns  1.2900us  cudaEventCreateWithFlags
                        0.00%  4.7640us         1  4.7640us  4.7640us  4.7640us  cuDeviceGetPCIBusId
                        0.00%  3.4970us        11     317ns     214ns     812ns  cudaDeviceGetAttribute
                        0.00%  2.0380us         4     509ns     202ns  1.1620us  cuDeviceGetCount
                        0.00%  1.2160us         3     405ns     240ns     735ns  cuDeviceGet
                        0.00%  1.1680us         1  1.1680us  1.1680us  1.1680us  cuInit
                        0.00%  1.1150us         1  1.1150us  1.1150us  1.1150us  cudaGetDevice
                        0.00%     652ns         1     652ns     652ns     652ns  cuCtxSetCacheConfig
                        0.00%     363ns         1     363ns     363ns     363ns  cuDriverGetVersion
```

B.2 TensorFlow

```
               Type  Time(%)      Time     Calls       Avg       Min       Max  Name
    GPU activities:   23.38%  374.68ms     28178  13.296us     832ns  590.81us  [CUDA memcpy HtoD]
                       8.44%  135.22ms     42210  3.2030us     992ns  26.210us  _ZN5Eigen8internal15Eigen [...]
                       6.51%  104.36ms     18760  5.5620us  1.1520us  18.561us  _ZN5Eigen8internal15Eigen [...]
                       5.96%  95.525ms     18760  5.0910us  1.1520us  16.640us  _ZN5Eigen8internal15Eigen [...]
                       5.77%  92.533ms      4690  19.729us  15.521us  21.281us  maxwell_sgemm_128x64_nt
                       5.41%  86.695ms      4690  18.485us  14.401us  19.937us  sgemm_32x32x32_NN_vec
                       3.68%  58.970ms     18760  3.1430us  2.0800us  13.153us  _ZN5Eigen8internal15Eigen [...]
                       3.48%  55.834ms      4690  11.904us  11.425us  13.633us  sgemm_32x32x32_NN
                       2.66%  42.588ms     37520  1.1350us     960ns  13.057us  _ZN5Eigen8internal15Eigen [...]
                       2.59%  41.497ms     28150  1.4740us     928ns  39.874us  [CUDA memcpy DtoH]
                       2.34%  37.424ms      4690  7.9790us  7.6800us  11.905us  sgemm_32x32x32_NT
                       2.18%  34.958ms      9380  3.7260us  2.0480us  24.481us  _ZN5Eigen8internal15Eigen [...]
                       2.18%  34.904ms      4690  7.4420us  7.2320us  8.0000us  sgemm_32x32x32_TN
                       2.04%  32.684ms      9380  3.4840us  3.1680us  13.537us  _ZN10tensorflow95 [...]
                       1.79%  28.637ms      4690  6.1050us  1.3440us  38.242us  _ZN5Eigen8internal15Eigen [...]
                       1.53%  24.562ms     18760  1.3090us     992ns  13.569us  _ZN10tensorflow7functor15 [...]
                       1.41%  22.598ms      9380  2.4090us  1.3760us  11.393us  _ZN5Eigen8internal15Eigen [...]
                       1.32%  21.132ms     14070  1.5010us  1.1520us  12.896us  _ZN10tensorflow7functor17 [...]
                       1.25%  20.042ms     14070  1.4240us  1.1840us  12.641us  _ZN5Eigen8internal15Eigen [...]
                       1.10%  17.682ms      4690  3.7700us  3.5520us  8.6400us  _ZN10tensorflow7functor30 [...]
```

```
                  1.09%  17.391ms     14070  1.2360us  1.0880us  12.737us  _ZN5Eigen8internal15Eigen [...]
                  1.05%  16.767ms     14070  1.1910us     992ns  12.832us  _ZN5Eigen8internal15Eigen [...]
                  1.03%  16.445ms      9380  1.7530us  1.1210us  11.328us  _ZN10tensorflow14BiasNHWC [...]
                  0.86%  13.836ms      4690  2.9500us  2.8480us  10.912us  _ZN5Eigen8internal15Eigen [...]
                  0.81%  13.023ms      9380  1.3880us  1.0880us  13.473us  _ZN5Eigen8internal15Eigen [...]
                  0.80%  12.813ms      9380  1.3650us  1.1520us  11.297us  _ZN5Eigen8internal15Eigen [...]
                  0.75%  11.942ms      4686  2.5480us  2.4000us  12.768us  _ZN10tensorflow7functor18 [...]
                  0.66%  10.567ms      4690  2.2530us  2.1440us  12.192us  _ZN5Eigen8internal15Eigen [...]
                  0.57%  9.1643ms      4690  1.9540us  1.8880us  12.481us  _ZN10tensorflow88_GLOBAL [...]
                  0.57%  9.0707ms      4690  1.9340us  1.8560us  11.232us  _ZN5Eigen8internal15Eigen [...]
                  0.56%  8.9482ms      4690  1.9070us  1.8560us  12.576us  _ZN10tensorflow7functor15 [...]
                  0.55%  8.7854ms      9381     936ns     864ns  9.5360us  [CUDA memset]
                  0.54%  8.5776ms      4690  1.8280us  1.7280us  13.601us  _ZN10tensorflow7functor15 [...]
                  0.53%  8.5278ms      4690  1.8180us  1.5680us  12.928us  _ZN5Eigen8internal15Eigen [...]
                  0.52%  8.3466ms      4690  1.7790us  1.6640us  12.737us  _ZN5Eigen8internal15Eigen [...]
                  0.47%  7.5324ms      4690  1.6060us  1.3760us  8.1920us  _ZN5Eigen8internal15Eigen [...]
                  0.45%  7.1840ms      4694  1.5300us  1.1520us  11.264us  _ZN5Eigen8internal15Eigen [...]
                  0.45%  7.1443ms      4690  1.5230us  1.4080us  12.929us  _ZN5Eigen8internal15Eigen [...]
                  0.45%  7.1358ms      4690  1.5210us  1.3760us  13.025us  _ZN5Eigen8internal15Eigen [...]
                  0.42%  6.7617ms      4690  1.4410us  1.3760us  10.912us  _ZN5Eigen8internal15Eigen [...]
                  0.41%  6.6046ms      4690  1.4080us  1.3120us  11.680us  _ZN5Eigen8internal15Eigen [...]
                  0.37%  5.9988ms      4690  1.2790us  1.1840us  13.377us  _ZN5Eigen8internal15Eigen [...]
                  0.37%  5.9562ms      4690  1.2690us  1.2160us  13.217us  _ZN5Eigen8internal15Eigen [...]
                  0.36%  5.7932ms      4690  1.2350us  1.1840us  12.705us  _ZN5Eigen8internal15Eigen [...]
                  0.35%  5.6259ms      4690  1.1990us  1.1520us  12.513us  _ZN5Eigen8internal15Eigen [...]
                  0.00%  26.083us         8  3.2600us  2.8170us  4.6720us  _ZN10tensorflow26BiasGrad [...]
                  0.00%  21.953us         4  5.4880us  3.2320us  9.2480us  _ZN10tensorflow7functor28 [...]
                  0.00%  9.3760us         4  2.3440us  1.1520us  5.1840us  _ZN5Eigen8internal15Eigen [...]
                  0.00%  1.5360us         1  1.5360us  1.5360us  1.5360us  [CUDA memcpy DtoD]
      API calls:  57.02%  5.18110s    389286  13.309us  5.1400us  732.70ms  cudaLaunchKernel
                 19.60%  1.78053s    122052  14.588us     272ns  41.742ms  cuEventRecord
                  8.85%  804.43ms     28177  28.549us  3.4030us  1.6580ms  cuMemcpyHtoDAsync
                  6.07%  551.14ms     28150  19.578us  3.7850us  42.723ms  cuMemcpyDtoHAsync
                  3.31%  300.76ms         2  150.38ms  4.0250us  300.76ms  cudaFree
                  1.62%  146.85ms     90921  1.6150us     400ns  336.43us  cuEventQuery
                  1.51%  137.62ms         1  137.62ms  137.62ms  137.62ms  cuDevicePrimaryCtxRetain
                  0.90%  82.223ms     61018  1.3470us     342ns  412.26us  cuStreamWaitEvent
                  0.50%  45.496ms      9380  4.8500us  2.8860us  294.64us  cuMemsetD32Async
                  0.41%  37.514ms      4702  7.9780us  2.8380us  24.333us  cuCtxSynchronize
                  0.07%  5.9449ms         1  5.9449ms  5.9449ms  5.9449ms  cuMemAlloc
                  0.05%  4.1529ms     23450     177ns     104ns  13.742us  cudaGetLastError
                  0.03%  2.8689ms         6  478.16us  421.57us  529.34us  cudaGetDeviceProperties
                  0.02%  1.5267ms       295  5.1750us     106ns  207.13us  cuDeviceGetAttribute
                  0.01%  1.2938ms         2  646.92us  630.78us  663.05us  cuMemHostAlloc
                  0.01%  1.1951ms         4  298.78us  190.48us  357.12us  cuDeviceTotalMem
                  0.00%  300.86us         1  300.86us  300.86us  300.86us  cuDeviceGetProperties
                  0.00%  297.73us        11  27.065us  1.0020us  208.94us  cuStreamCreate
                  0.00%  278.61us        24  11.608us     228ns  105.16us  cuEventCreate
                  0.00%  227.52us         4  56.880us  38.995us  75.003us  cuDeviceGetName
                  0.00%  186.99us         1  186.99us  186.99us  186.99us  cudaMemcpyAsync
```

```
                    0.00%  175.99us         3  58.664us  3.7230us  162.24us  cudaMalloc
                    0.00%  163.09us         2  81.543us  65.705us  97.381us  cuMemGetInfo
                    0.00%  44.079us         8  5.5090us  4.8010us  6.5930us  cuEventSynchronize
                    0.00%  33.690us         1  33.690us  33.690us  33.690us  cuMemsetD32
                    0.00%  27.609us         7  3.9440us     932ns  15.583us  cuCtxSetCurrent
                    0.00%  13.580us         1  13.580us  13.580us  13.580us  cudaMemcpy
                    0.00%  7.3510us        16     459ns     352ns  1.5740us  cudaEventCreateWithFlags
                    0.00%  6.7710us         8     846ns     747ns  1.0190us  cuEventElapsedTime
                    0.00%  4.9860us         2  2.4930us  1.1190us  3.8670us  cudaGetDevice
                    0.00%  4.8740us         3  1.6240us  1.1890us  2.2670us  cuInit
                    0.00%  3.9220us        11     356ns     223ns     970ns  cudaDeviceGetAttribute
                    0.00%  3.5740us        11     324ns     129ns  1.2910us  cuDeviceGetCount
                    0.00%  3.5360us         5     707ns     248ns  2.1910us  cuDeviceGet
                    0.00%  3.3050us         8     413ns     242ns     749ns  cuEventDestroy
                    0.00%  2.4680us         4     617ns     166ns  1.3410us  cuDriverGetVersion
                    0.00%  1.5540us         2     777ns     310ns  1.2440us  cudaSetDevice
                    0.00%  1.3280us         1  1.3280us  1.3280us  1.3280us  cuDeviceGetPCIBusId
                    0.00%  1.1010us         1  1.1010us  1.1010us  1.1010us  cudaGetDeviceCount
                    0.00%     806ns         1     806ns     806ns     806ns  cuDeviceComputeCapability
                    0.00%     695ns         3     231ns     205ns     284ns
                    0.00%     593ns         1     593ns     593ns     593ns  cuDevicePrimaryCtxGetState
                    0.00%     250ns         1     250ns     250ns     250ns  cuCtxGetCurrent
```

Appendix C

Profiling data

Table C.1: Mean time (in seconds) of 10 runs with 1 pass over the MNIST dataset

DSL	Target	Batch size	Mean time	Std. Dev.
LHask	CPU	32	6.68	0.05
TensorFlow	CPU	32	6.28	0.00
LHask	GPU	32	8.43	0.26
TensorFlow	GPU	32	5.90	0.02
LHask	CPU	64	3.73	0.01
TensorFlow	CPU	64	4.63	0.00
LHask	GPU	64	4.59	0.01
TensorFlow	GPU	64	4.69	0.00
LHask	CPU	128	2.43	0.05
TensorFlow	CPU	128	3.79	0.01
LHask	GPU	128	2.81	0.01
TensorFlow	GPU	128	4.14	0.01

Table C.2: Mean time (in seconds) of 10 runs with 5 passes over the MNIST dataset

DSL	Target	Batch size	Mean time	Std. Dev.
LHask	CPU	32	31.39	0.27
TensorFlow	CPU	32	23.59	0.04
LHask	GPU	32	38.46	0.07
TensorFlow	GPU	32	16.07	0.03
LHask	CPU	64	17.85	0.11
TensorFlow	CPU	64	15.38	0.01
LHask	GPU	64	19.69	0.03
TensorFlow	GPU	64	10.05	0.01
LHask	CPU	128	10.84	0.05
TensorFlow	CPU	128	11.27	0.01
LHask	GPU	128	10.31	0.05
TensorFlow	GPU	128	7.31	0.04

Table C.3: Mean time (in seconds) of 10 runs with 10 passes over the MNIST dataset

DSL	Target	Batch size	Mean time	Std. Dev.
LHask	CPU	32	63.13	0.40
TensorFlow	CPU	32	45.29	0.04
LHask	GPU	32	76.88	0.09
TensorFlow	GPU	32	28.88	0.09
LHask	CPU	64	35.23	0.16
TensorFlow	CPU	64	28.87	0.03
LHask	GPU	64	38.64	0.05
TensorFlow	GPU	64	16.82	0.04
LHask	CPU	128	21.40	0.11
TensorFlow	CPU	128	20.62	0.02
LHask	GPU	128	19.49	0.02
TensorFlow	GPU	128	11.18	0.03