FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis in Computer Science

Duc Minh Tran <cwz688>

Implementation of a deep learning library in Futhark

Supervisor: Troels Henriksen

01. August 2018

Abstract

Neural networks are a powerful class of machine learning models, which are able to solve complex problems, such as image classification. The increased power comes with a cost though as they are computationally very expensive models and thus are trained on GPGPUs. This thesis explored implementation options of a deep learning library, which can support feed-forward types of neural networks, in the data-parallel language Futhark. Furthermore, the thesis investigates if the language is expressive enough to handle the complex nature of a deep learning library. The final design of the library in this thesis uses a representation of a neural network as a composition of functions, which avoids the limitation of non-regular arrays and shows that the module and type system is expressive enough to provide the necessary abstraction. The library is then benchmarked against the popular machine-learning library, Tensorflow, which showed that the performance of the library is faster for larger batch sizes on a multilayer perceptron, but falls short on smaller batch sizes. The library is up to 2.7 times slower on a convolutional network for large batch sizes, but only 2.0 times slower for small batch sizes. The performance gap for the latter result can be lessened significantly by implementing better convolutional algorithms, along with the Futhark compiler continously being optimized in the future.

Keywords: Deep learning, Neural networks, General-Purpose Graphics Processing Units, Futhark.

Contents

Chapter 1

Introduction

In the recent decade the amount of data has exploded with the increased usage of electronic devices resulting in new problems arising due to this technological development. These problems occur within, but not limited to, image classification, natural language processing and many more. Given that the value of data is limited to how well we can find patterns in it and the sheer complexity and size of this data, classical statistical methods are no longer sufficient, calling for the need of other methods. For such problems and data, *machine learning* approaches have been used as a substitute to the aforementioned classical statistical methods. Such approaches use a combination of mathematics and computational power to search for patterns in the data. The main advantage of machine learning approaches is their ability to "learn" iteratively through data examples without explicit rules. A popular class of machine learning models is artificial neural networks (also known as deep learning models) which attempt to replicate the biological neural system where information flows through layers of neurons. This approach has seen great success in multiple applications such as self-driving vehicles and board games, for example when the AI application AlphaGo beat the world champion Go player, Lee Sedol, using a deep learning approach. The term artificial neural network dates all the way back to W. McCulloh & W.Pitts in 1943, where they modeled a one layer neural network using electrical circuits. F. Rosenblatt (1958) expanded this thought and invented the perceptron algorithm, which allowed such models to learn through data. The model created a lot of excitement in the artificial intelligence community that only lasted a decade when M. Minsky and S. Papert (1969) showed that the perceptron model only applied to problems that were linearly separable; in particular, the XOR problem couldn't be solved using the perceptron model. The enthusiasm faded and it wasn't until 1986 when D. Rumelhart et al. presented the *backpropagation* algorithm that neural networks again became popular. The backpropagation algorithm was very similar to the perceptron algorithm, but applied to a neural network of *arbitrary depth*, meaning they could now solve non-linear problems. The backpropagation algorithm is based on the concept of letting errors flow backwards into the network, which are then used to adjust learning parameters. This process is repeated continuously, also called training, until a sufficient model is obtained. As such, neural networks can be called slow learners, requiring potentially thousands or millions of iterations. Thus the need for faster hardware that can perform training is essential in the success of neural network applications and training is therefore usually performed on General-Purpose Graphics Processing Units (GPGPU), which can leverage the highly parallel nature of neural networks.

This thesis explores how an implementation of a deep learning library can be achieved in the data-parallel GPGPU language, Futhark[1], which was developed at the Department of Computer Science at the University of Copenhagen.

1.1 Thesis objective

The main objective of this thesis is to investigate to what extend a general-purpose data-parallel language such as Futhark, can implement a deep learning library. As Futhark imposes limitations on the language semantics in order to produce high-performance data-parallel GPU code, porting existing libraries one-to-one is not possible. Instead, alternative approaches are required. The complex nature of a deep learning library will also require an implementation of many separate components which will show if Futhark is capable of providing the necessary level of abstraction. Such a library should ideally be flexible enough to be maintained over an extended period of time. Lastly, will this thesis explore how well such a library in Futhark can compete, in terms of performance, with dedicated DSL solutions, such as Tensorflow[2].

1.1.1 Limitations

The implementation is limited to the most essential blocks required for building and training a *feed-forward* type of neural network. Achieving the same flexibility as state of the art libraries, like Tensorflow is not a goal in this thesis work. Furthermore, Tensorflow, and other popular deep learning libraries, are merely a front-end for the cudNN API[18] - an API provided by NVIDIA specifically designed for deep learning applications. To achieve the best performance, the API provides multiple algorithm options and hardware specific optimizations depending on the system and neural network. This will be difficult for the thesis' implementation and Futhark to match. However, Tensorflow will be used as a benchmark to compare the performance of this thesis' library to one of the fastest libraries available.

1.2 Thesis structure

Chapter 2 presents the mathematical foundation behind neural networks based on chapter 5 in C. Bishop's book "Pattern Recognition and Machine Learning"(2006) and present the components needed to build and train a neural network. The chapter will also show

[1]`https://futhark-lang.org`

[2]`https://tensorflow.org`

the derivation of the backpropagation algorithm for a multilayer perceptron, while only the final results will be shown for a convolutional network. The reader is not expected to understand this chapter in detail, but should at least read through the main concepts and results. Chapter 3 will first discuss implementation alternatives of a deep learning library in Futhark, and discuss why some approaches, which would be applicable in other languages, doesn't apply to Futhark. The main concept behind the implementation will then be presented, followed by the implementation details. Chapter 4 will compare the performance of training simple neural networks with the library against equivalent networks in Tensorflow. Chapter 5 presents the future work of the library and provides a conclusion.

1.3 Introduction to Futhark

Futhark is a small high-level, purely functional array language from the ML-family, which is designed to generate efficient data-parallel code [10]. Futhark currently generates GPU code via OpenCL, although the language is hardware independent. The Futhark compiler can compile directly to an executable for the GPU or it can generate a reusable library for either Python or C. The latter is how the language is meant to be used, to accelerate computer-intensive parts of an application, and as such not meant as a general-purpose language.

Futhark supports regular nested data-parallelism, as well as imperative-style in-place updates of arrays, but maintains its purely functional style through a uniqueness type system, which prohibit the use of an array after it has been updated in-place. As most languages from the ML-family, Futhark also has parametric polymorphic and uses type parameters, which allows functions and types to be polymorphic. Type parameters are written as a name preceded by an apostrophe. For example Listing 1.1 shows a type abbreviation *number* with a type parameter t, which is instantiated with concrete types in line 2, where $f32$ denotes 32-bit floating point.

```
1 type number 't = t
2 type float = number f32
```

Listing 1.1: Example of type abbreviation with type parameter in Futhark

Type abbreviations are mostly for syntactic convenience and for abstract types that hide their definition we need to use the higher-order module system [5]. Futhark allows for abstract modules, called module types, which provide a powerful abstraction mechanism. For example Listing 1.2 shows how we can write a module type, *number* to have an abstract type t and an abstract `add` function.

```
module type number = {
  type t
  val add: t → t → t
}
```

Listing 1.2: Example of a module type, *number*, in Futhark

Module types are used to classify the contents of modules, meaning that an implementation of *number* must provide a concrete type t and an `add` function with the signature $t \to t \to t$. We can then define a *float* module as follows:

```
module float : number {
  type t = f32
  let add (x:t) (y:t) : t =  x + y
}
```

One can specify that an abstract type is allowed to be functional by specifying it in the module type using ^, (e.g. `type ^my_func`). Lastly is it also possible to specify a parametric module, meaning that the module can take another module as an argument, (i.e. module-level functions), which allows for abstraction over modules. The module system is an important factor in Futhark for providing abstraction and code reuse into larger applications, which are structured using the module system.

Futhark achieves much of its data-parallelism through its Second-Order Array Combinators (SOACs), `map` `reduce` `scan` and `scatter`. The semantics of the first three are similar to the ones found in other functional languages, such as SML, F# and Haskell, but there are, however, some aspects to note about these functions in Futhark. The operator given to `reduce` and `scan` must be associative, in order to produce a result, that is equivalent to applying the operator in sequential order. Along with, it must be the neutral element of the operator (e.g., 1 for multiplication and 0 for addition). The `scatter` function takes three array arguments, x, *idx* and *vals*, where *idx* and *vals* must be of same length. The function performs in-place updates in x on indices *idx* with values *vals* and returns the updated array. The input array x is *consumed* and is not allowed to be used afterwards nor through aliasing. These SOACs permits the Futhark compiler to generate parallel code, which means that Futhark programs are commonly written as bulk operations on arrays. Through the SOACs is the Futhark compiler able to provide aggressive optimizations. For example is a composition of nested `map-reduce` computation efficiently supported based on fusion and code generation [9, 13] and the compiler also provide support for 1-d and 2-d tiling [10].

As Futhark focuses less on expressiveness and elaborate type systems, but more on compiling to high-performance parallel code, it puts some constraints on the language semantics and in turn on the design of the deep learning library For example, the language does not support irregular arrays, meaning that all inner arrays must have the same shape. For example, is this two dimensional array $[[\mathbf{1}], [\mathbf{2}, \mathbf{3}]]$ not allowed. Another key limitation is that arrays of functions are not permitted. How these limitations affect the design of the library will be discussed in Chapter 3.

1.4 Code and data

The library produced from this thesis can be found at `https://github.com/HnimNart/deep_learning`, which includes benchmark programs and tests. The data used throughout the example programs is the MNIST dataset[3], containing images of handwritten digits, which is often used as the "Hello-World" example in deep learning. As these files are too big for GitHub, the data used for the Futhark examples can be found at `http://napoleon.hiperfit.dk/~HnimNart/mnist_data/`.

[3]`http://yann.lecun.com/exdb/mnist/`

Chapter 2

Neural Network

A neural network is a type of machine learning model, which is represented through a hierarchical structure of layers. Each layer consists of a fixed set of units or neurons with adaptable parameters, which can be changed during training. When some input data is passed into the network, each successive layer uses the output from the previous layer as input. The process is then repeated until an output layer has been reached. The interpretation of the output from a network depends on the modeling problem. A common one is the *multiclass* problem, where there are N prediction classes. The goal of the neural network is then to predict which class the input data belongs to. In such a case the output is interpreted as probabilities.

How each layer process the input depends on the type of layer, where this thesis will show the process for a fully-connected- and convolutional layer. Before doing so, introducing some terminology is needed. When a neural network consists solely of fully-connected layers, it is called a *multilayer perceptron* and if a neural network has at least one convolutional layer, but maybe one or more fully-connected, it is called a *convolutional network*. The architecture of a convolutional network usually consists of convolutional layers at the beginning of the network and ends with a number of fully-connected layers at the end. The next section will show how information is processed in a multilayer perceptron and a derivation showing the backpropagation algorithm applied to a multilayer perceptron of an arbitrary depth, which in the subsequent section will be extended to a convolutional network.

2.1 Multilayer perceptron

Multilayer perceptrons (MLP) are the simplest form of neural networks, where every output from a given layer is connected to every single neuron of the next layer. Figure 2.1 shows a MLP with two layers[1].

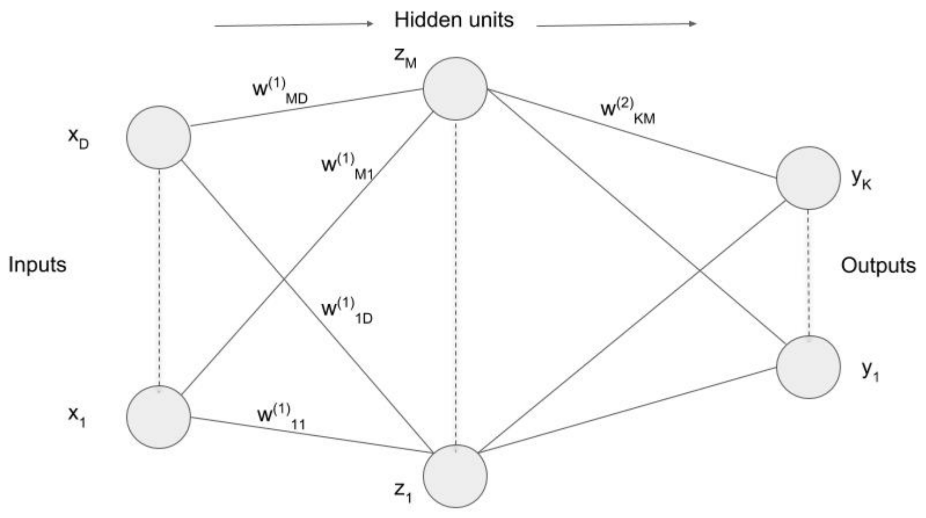


Figure 2.1: A 2-layer MLP network showing only the outer most neurons. (source:[1, ch. 5])

Using Figure 2.1 as an example we can express the first layer with M neurons as having M linear combinations with D inputs. Using this formulation and letting $x_1, x_2, \cdots, x_D$ be the input into the network in Figure 2.1, we can write the first layer calculation as:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)} \tag{2.1}$$

where $j \in \{1...M\}$, that is each neuron in the layer and the superscript (1) refers to the first layer. Here the w_{ji} are called the weight, the b_j are called the bias and each quantity a_j is called the activation of a neuron and is transformed using a *differentiable, non-linear* activation function, $\sigma(\cdot)$ giving

$$z_j = \sigma(a_j) \tag{2.2}$$

[1]There is some confusion about counting layers in a network, where some would call this network a three layer network. Using same terminology as C. Bishop I will also call this a two-layer network, since there are two layers with weights.

The z_j are the output of the first layer, which are then passed onto the next layer, where the same process is continued, until reaching the final layer. Following the same procedure for the next layer we can write

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + b_k^{(2)} \quad \text{for } k = 1, .., K \tag{2.3}$$

Once the end of a network is reached, the output activations are transformed using an appropriate activation function into y_k, depending on the problem the network tries to solve. With multiclass problems, it is common to transform each output unit into probabilities, by using the *softmax* function, which is defined by

$$softmax(a_k) = y_k = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}} \text{ for } k = 1...K \tag{2.4}$$

where $0 \leq y_k \leq 1$ with $\sum_{k=1}^{K} y_k = 1$, which can be interpreted as a probability. Combining (2.1), (2.2), (2.3) and (2.4) we can express the network as a composition of operations and the network function therefore takes the form

$$y_k(\mathbf{x}, \mathbf{w}) = softmax\left(\sum_{j=1}^{M} w_{kj}^{(2)} \sigma\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)}\right) + b_k^{(2)}\right) \tag{2.5}$$

where we have grouped the weights and bias parameters into $\mathbf{w}$. Thus a MLP is simply a nonlinear function from a set of input variables $\{x_i\}$ that maps to a set of output variables $\{y_k\}$ controlled by a set of adjustable parameters, $\mathbf{w}$. For implementation purposes, we can rewrite this into matrix form and use matrix-vector multiplication instead of summations. For a neuron j in the first layer we can see that $\sum_{i=1}^{D} w_{ji}^{(1)} x_i$ is just the dot-product. As we have M neurons each with a set of weights, we can therefore represent the weights in the first layer as $W^{(1)} : \mathbb{R}^{M \times D}$ with the biases $B^{(1)} : \mathbb{R}^{M}$. Likewise for the next layer we define $W^{(2)} : \mathbb{R}^{K \times M}$ with the biases $B^{(2)} : \mathbb{R}^{K}$. Doing this we can write

$$\mathbf{y}(\mathbf{x}, \mathbf{W}) = softmax\left(W^{(2)} \sigma\left(W^{(1)} \mathbf{x} + B^{(1)}\right) + B^{(2)}\right) \tag{2.6}$$

The above process of evaluating (2.6) is called the *forward propagation* of information through the network.

2.1.1 Activation functions

Activation functions are required to be differentiable; this is necessary when training networks, since we need to use the derivative of the input when we backpropagate through the network. Common activation functions are *tanh, rectified linear unit (ReLU), sigmoid* and *softmax*. Table 2.1 shows these activation functions and their derivatives for an input x. The use of activation functions is an important factor for introducing non-linearity into the network; otherwise we could simply express a network as a linear combination, which in general is less powerful.

Activation function	$\sigma(x)$	$\sigma'(x)$
tanh	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - tanh(x)^2$
ReLU	$max(0, x)$	if $x \leq 0$ then 0 else 1
sigmoid	$\frac{1}{1+e^{-x}}$	$sigmoid(x)(1 - sigmoid(x))$
softmax	$\frac{e^x}{\sum_{k=1}^{K} e^{z_k}}$	$softmax(x_i)(1(i = j) - softmax(x_j))$

Table 2.1: Common activation functions and their derivatives

2.2 Network training

As the goal with neural network is to be able to provide some prediction, given some input, we need to train the network first. The idea is that given our $y_k(\mathbf{x}, \mathbf{w})$, we want it to predict our target values t_k for all k. For each set of y_k and t_k we can calculate a *loss*, defined by some function $E(\mathbf{w})$. Table 2.2 shows some common loss functions.

Loss function	$E(\mathbf{w})$	$\frac{\partial E(\mathbf{w})}{\partial y_k}$
Cross-entropy	$-\sum_{k=1}^{K} (t_k \; ln \; y_k)$	$-\frac{t_k}{y_k}$
Cross-entropy w. softmax	$-\sum_{k=1}^{K} \left(t_k \; ln \; \frac{e^{y_k}}{\sum_{i=1}^{K} e^{y_i}}\right)$	$y_k - t_k$
Sum of squares	$\frac{1}{2}\sum_{k=1}^{K} (y_k - t_k)^2$	$y_k - t_k$

Table 2.2: Common loss functions and their derivatives w.r.t. to activation unit, y_k, used in the backpropagation algorithm

Note that the definitions of cross-entropy functions in Table 2.2 assumes that target values are encoded as *one-hot*, meaning that for a n target values there exists only one $t_k = 1$ and $t_j = 0, \forall j \neq k$. This is a common coding for multiclass problems, as we are trying to predict a single class. As the goal of training the network to give better predictions, we want to *minimize* the loss function, w.r.t. the weights $\mathbf{w}$. Letting $\nabla E(\mathbf{w})$ denote the gradient, that is the direction of the greatest rate of increase of the error function, we can see that the smallest value of E will occur when $\nabla E(\mathbf{w}) = 0$, as our loss function is a smooth continuous function of $\mathbf{w}$. To achieve this we want to determine $\nabla E(\mathbf{w})$ and *subtract* it from our weights such that we approach a minimum, which ideally is a global minimum. By doing this iteratively, we improve the neural networks prediction power a little step at the time. This is also called the *gradient descent* optimization algorithm, which can be written as

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \alpha \nabla E(\mathbf{w}^{\tau}) \tag{2.7}$$

where τ is the iteration step and α denotes the *learning rate*. The value of the learning rate is often chosen to be of the order of 10^{-1}, when starting training on a new network and then decrease the learning rate as the model becomes more and more trained. This avoids the problem of ”valley-hopping”, where the weight updates jumps between a local minimum and never properly converges.

2.2.1 Stochastic- and Batch gradient descent

Stochatic gradient descent (SGD) is the method where the gradients of a *single* data point, (e.g. the vector $\boldsymbol{x}$ from Section 2.1), is applied to the weights at the time. In contrast, there is batch gradient descent, where the gradients of the multiple data points, i.e. $\{\boldsymbol{x}_1, .., \boldsymbol{x}_n\}$, are applied to the weights at the same time. Because of the large amount of data that needs to be processed, batches of data points are often used in practice, which decreases the computational time, but comes with a cost. Le Cun et al. (1989) showed that SGD provides the ability to escape stationary points, e.g. local minima and likewise N. Keskar et. al.(2016) show that too large of a batch size decreases the model accuracy, since they tend to get ”stuck” in local minima. Thus there is a trade-off between computational time and model accuracy, which model developers need to take into account. The sizes of batches are usually chosen to be of power of two, with common sizes being 32, 64 and 128, but the exact choice depends on modeling problem. When using batches of size N during training we obtain N-sets of gradients. The common approach is to take the mean of the N gradients before applying them, which enables comparisons across different batch sizes. Therefore equation (2.7) becomes:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \alpha \frac{1}{N} \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{\tau}) \tag{2.8}$$

C. Bishop applies the sum of the gradients in his book[1], but one can see that the learning rate, can be adjusted to achieve the same result, i.e. if α is applied for the mean approach, then $\frac{\alpha}{N}$ can be applied for the sum approach. The downside is that you'll need to adjust the learning rate accordingly for each batch size in order to compare, thus using the mean is more common.

2.3 Backpropagation algorithm

This section will show the derivation of the backpropagation algorithm for an arbitrary feed-forward topology with arbitrary differentiable, nonlinear activation functions. The intuition of the backpropagation algorithm is that based on our output error, we want the let these errors flow backwards into the network, which are then used to adjust the weights. The backbone of the backpropagation algorithm is the chain rule which states that if f and g are two differentiable functions then the derivative of $\frac{\partial f(g(x))}{\partial x}$ is

$$\frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x} = f'(g(x))g'(x) \tag{2.9}$$

As we want to determine $\nabla E(\mathbf{w})$, we need to determine $\frac{\partial E}{\partial W^l}$, by applying the chain rule recursively back through the network for each layer l.

2.3.1 Derivation

Recall that for the last layer we calculate:

$$a_k = \sum_{j=1}^{M} w_{kj} z_j + b_k \tag{2.10}$$

$$y_k = \sigma(a_k) \tag{2.11}$$

where σ is some activation function. We want to derive $\frac{\partial E}{\partial w_{kj}}$ first, i.e. the derivative of a single weight, w_{kj}. We see that E depends on w_{kj} through both y_k and a_k, so by applying the chain rule twice we get that

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} \tag{2.12}$$

Using the loss function, *sum of squares*, defined in table 2.2 as an example, we can evaluate the partial derivatives of each part separately which gives us

$$\frac{\partial E}{\partial y_k} = y_k - t_k, \quad \frac{\partial y_k}{\partial a_k} = \sigma'(a_k), \quad \frac{\partial a_k}{\partial w_{kj}} = z_j \tag{2.13}$$

And combining them back together we get that

$$\frac{\partial E}{\partial w_{kj}} = (y_k - t_k)\sigma'(a_k) z_j \tag{2.14}$$

We have now evaluated the derivative for a single weight w_{kj}, which also applies to the other weights in the last layer, giving us our gradient. A similar process is repeated for the remaining layers of the network. We will introduce a general notation,

$$\delta_j = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j} \tag{2.15}$$

which is called the error of neuron j, which semantically means if the activation of neuron j changes by 1, then the loss function changes by δ_j. Letting the previous layer be defined by:

$$a_j = \sum_{i} w_{ji} z_i + b_j \tag{2.16}$$

$$z_j = \sigma(a_j) \tag{2.17}$$

where we want to determine $\frac{\partial E}{\partial w_{ji}}$. By applying the chain rule, we can write the derivative as

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{2.18}$$

From equation (2.16) it's easy to see that $\frac{\partial a_j}{\partial w_{ji}} = z_i$ and by using our previously defined δ_j notation we find that

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i \tag{2.19}$$

We lastly need to evaluate δ_j. Using our definition and applying the chain rule gives us:

$$\delta_j = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j} \tag{2.20}$$

$$= \sum_k \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} \tag{2.21}$$

The first two terms of (2.21) can be written as δ_k and we note that the derivative of $\frac{\partial a_k}{\partial z_j}$ is the weight from neuron j to k (i.e. w_{kj}). The sum comes from the fact that the activation of neuron j is spread through of all its connections to the neurons in the next layer, which in this case is all of the output nodes. The last term $\frac{\partial z_j}{\partial a_j} = \sigma'(a_j)$ does not depend on k, and we can move it outside the summation. Substituting into (2.21) gives us:

$$\delta_j = \sigma'(a_j)\sum_k w_{kj}\delta_k \tag{2.22}$$

By substituting (2.22) into (2.19) we can now determine $\frac{\partial E}{\partial w_{ji}}$. Lastly we must also remember the derivative of the biases. As the bias term is simply a constant, we can write the derivative $\frac{\partial a_j}{\partial b_j} = 1$ and we easily see that $\frac{\partial E}{\partial b_j} = \delta_j$. We have finally derived the backpropagation algorithm, which when applying (2.22) and (2.19) recursively back through the network gives us the gradients for each of the layers. The backpropagation algorithm is summarized in Algorithm 1.

Algorithm 1 Backpropagation algorithm

```
1: Apply an input vector $\mathbf{x}$ to the network and forward propagate through the network
   finding the activations of each layer
2: Evaluate $\delta_k$ for all of the output units for some loss function $E$ using equation (2.15)
3: Backpropagate $\delta$'s using equation (2.22) for each neuron in the network.
4: Use equation (2.19) to calculate the gradients w.r.t. $w_{ji}$ for each weight in each layer.
```

For implementation purpose, we want to rewrite the backpropagation algorithm into matrix form [19]. Assuming a network with depth d and let $\boldsymbol{\delta}^{(l)}, W^{(l)}, B^{(l)}, \boldsymbol{z}^{(l)}$ denote the errors, weights, biases and activations of the l'th layer respectively. We also let the input $\mathbf{x}$ into the network be $\boldsymbol{z}^{(0)}$. We can express the backpropagation algorithm as follows

$$\boldsymbol{z}^{(l)} = \sigma^{(l)}(W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}) \tag{2.23}$$

$$\boldsymbol{\delta}^{(d)} = \frac{\partial E}{\partial \boldsymbol{z^{(d)}}} \circ \sigma'^{(d)}(W^{(d)}\boldsymbol{z}^{(d-1)} + B^{(l)}) \tag{2.24}$$

$$\boldsymbol{\delta^{(l)}} = (W^{(l+1)})^T\boldsymbol{\delta}^{(l+1)} \circ \sigma'^{(l)}(W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}) \tag{2.25}$$

$$\frac{\partial E}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{z}^{(l-1)})^T \tag{2.26}$$

$$\frac{\partial E}{\partial B^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{2.27}$$

Where $\circ$ denotes the Hadamard product or element-wise product.

2.4 Convolutional neural network

A convolutional neural network has the same hierarchical structure as a MLP, but convolutional layers treats the input data in a different manner. The main idea behind convolutional layers is that data, like images, contain "hidden" information in the spatial structure, which can be utilized when searching for patterns. The input into a convolutional layer is therefore three dimensions, described by height, width and depth, $H \times W \times D$. I'll consider the case of depth equal to one first, thus reducing the dimensions to two, and since it's common to have square images, the input dimensions becomes $N \times N$. When data is fed into a MLP, the input is stretched out into a single dimension, resulting in the spatial information being lost, but with a convolutional network we want to make use of this information. In convolutional layers weights are now called filters[2], which a layer can have multiples of. These filters are (usually) small square matrices denoted by $k \times k$. Each filter is slided across the input image in both dimensions with a predetermined stride constant and computes the dot-product for each stride, which is called the convolutional operation. Figure 2.2 shows an example. For compactness the filter f is written as a vector $[f_{00}\ f_{10}\ f_{01}\ f_{11}]^T$.

$$\begin{bmatrix} x_{00} & x_{10} & x_{20} \\ x_{01} & x_{11} & x_{21} \\ x_{02} & x_{12} & x_{22} \end{bmatrix} \otimes \begin{bmatrix} f_{00} & f_{10} \\ f_{01} & f_{11} \end{bmatrix} = \begin{bmatrix} [x_{00}\ x_{10}\ x_{01}\ x_{11}\ ] \cdot f & [x_{10}\ x_{20}\ x_{11}\ x_{21}\ ] \cdot f \\ [x_{01}\ x_{11}\ x_{02}\ x_{12}\ ] \cdot f & [x_{11}\ x_{21}\ x_{12}\ x_{22}\ ] \cdot f \end{bmatrix}$$

Figure 2.2: Example of convolutional operation, with input image of size 3x3 and filter size 2x2 with a stride 1, where $\cdot$ denotes the dot-product

An important property of a convolutional layer is weight sharing. The sharing of weights causes *equivariance*, which means that if the input changes, then the output changes in the same way [7, ch. 9], though are convolutional layers only naturally equivariance to

[2]Filters are also called kernels, but to not confuse with GPU kernels, the term filters is used instead

shifts, but not rotation nor scaling. The goal is to have each filter adapt to certain characteristics of the input images, for example one filter detects to edges, another detects to depth and so on. We can define a convolutional operation[3] for a single output value, a_{ij} given an image, I and a single filter, F_f of size $k \times k$ as:

$$(I \otimes F_f)_{ij} = a_{ij} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I[i+m, j+n] F_f[m,n] \tag{2.28}$$

The outputs from a convolutional layer is now called an *activation map*, and we can calculate the dimensions of the activation map, given an input image of dimensions $n \times n$ and filter size of $k \times k$ with a stride of s as

$$\left(\frac{(n-k)}{s} + 1 \right) \times \left(\frac{(n-k)}{s} + 1 \right) \tag{2.29}$$

In the case of the depth dimension, also called channels, is larger than one, the image channels *must* match the filter channels, because we are doing 2D convolutions[4], (i.e. if the input is of dimensions $n \times n \times c$, then the filter must have dimensions $k \times k \times c$). The output value, a_{ij} is then the sum of the dot-products from each channel. Therefore is depth dimension of the output from a convolutional layer only determined by the *number of filters*, N_f, in a convolutional layer and we can write the output dimension as

$$\left(\frac{(n-k)}{s} + 1 \right) \times \left(\frac{(n-k)}{s} + 1 \right) \times N_f \tag{2.30}$$

The activations, a_{ij} from a convolutional layer, given an image, I of size $n \times n \times c$ and a filter F_f of size $k \times k \times c$ is then

$$(I \otimes F_f)_{ij} = a_{ij} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c'=0}^{c-1} I[i+m, j+n, c'] F_f[m,n,c'] + b_f \quad \text{for } i,j = 1,.., \frac{(n-k)}{s} + 1 \tag{2.31}$$

The operation is then repeated for each filter in F. Note that there is only one bias value for each filter, i.e. there are N_f bias values in a convolutional layer. The convolutional layer also applies an activation function $\sigma()$ resulting in the output from the convolutional layer.

$$z_{ij} = \sigma(a_{ij}) \tag{2.32}$$

Removing subscripts we can write the output of a convolutional layer as:

$$Z = \sigma\left(I \otimes F + B\right) \tag{2.33}$$

Like in the MLP case this is also called the *forward propagation* of information, but in this case there is no natural way to transform it into matrix form.

[3]This is technically a cross-correlation operation, as a convolution operation requires flipping the filter, but when training a network it doesn't matter which is used, as long as one is consistent during forward and backward passes. This is also how Tensorflow performs its convolutional operation, `https://tensorflow.org/api_guides/python/nn#Convolution`

[4]The 2D refers to the dimensions the filter is slided in and not the dimensions of the filter nor the input image

2.4.1 Backpropagation algorithm

The backpropagation algorithm for the convolutional network is similar to the one of MLP, but with the matrix multiplications replaced by convolutional operations. The full derivation is omitted here; instead we refer to [11] for a full derivation. The equations of the backpropagation algorithm for the convolutional network are:

$$Z^{(l)} = \sigma\left(Z^{(l-1)} \otimes F^{(l)} + B^{(l)}\right) \tag{2.34}$$

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \otimes \left(F^{(l+1)}\right)^{rot(180^\circ)} \circ \sigma'\left(Z^{(l-1)} \otimes F^{(l)} + B^{(l)}\right) \tag{2.35}$$

$$\frac{\partial E}{\partial F^{(l)}} = Z^{(l-1)} \otimes \boldsymbol{\delta}^{(l)} \tag{2.36}$$

$$\frac{\partial E}{\partial B^{(l)}} = \sum_m \sum_n \boldsymbol{\delta}^{(l)}_{mn} \tag{2.37}$$

where $\boldsymbol{\delta}^{(l)}$ have the same semantics as in the case of MLP. Note that in equation (2.35) each filter is rotated 180°, [5] (or flipped), since we need to map the errors back to the input with the corresponding weights, e.g. from the example in Figure 2.2 x_{00} is only affected by f_{00}. In order to do so, we need to flip the filter and perform a *full* convolutional operation, meaning that some of the filter goes out-of-bounds. This is in practice solved by adding $k-1$ zero padding around $\boldsymbol{\delta}$, where k is the filter size and then one can perform a normal convolutional operation. Figure 2.3 shows an example of the full convolutional operation, where one can see that the result has same dimensions as the example in figure 2.2, and verify that we have correctly mapped the errors back to it's input through the filter, f.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{00} & \delta_{10} & 0 \\ 0 & \delta_{01} & \delta_{11} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} f_{11} & f_{01} \\ f_{10} & f_{00} \end{bmatrix} = \begin{bmatrix} [0\ 0\ 0\ \delta_{00}\ ] \cdot f' & [0\ 0\ \delta_{00}\ \delta_{10}\ ] \cdot f' & [0\ 0\ \delta_{10}\ 0\ ] \cdot f' \\ [0\ \delta_{00}\ 0\ \delta_{01}\ ] \cdot f' & [\delta_{00}\ \delta_{10}\ \delta_{01}\ \delta_{11}\ ] \cdot f' & [\delta_{10}\ 0\ \delta_{11}\ 0\ ] \cdot f' \\ [0\ \delta_{01}\ 0\ 0\ ] \cdot f' & [\delta_{01}\ \delta_{1}\ 0\ 0\ ] \cdot f' & [\delta_{11}\ 0\ 0\ 0\ ] \cdot f' \end{bmatrix}$$

Figure 2.3: Example of a full convolution operation by padding the errors $\boldsymbol{\delta}$ with zeroes and applying the flipped filter, $f' = [f_{11}\ f_{01}\ f_{10}\ f_{00}]^T$ from Figure 2.2.

Having defined our backpropagation algorithms, we need to be able to combine a convolutional layer with a fully-connected layer, which swapping part of the algorithms. For the forward pass, we simply stretch out the output of the convolutional layer, before applying it to the fully-connected one. For the backward pass we need to substitute $\boldsymbol{\delta}^{(l+1)} \otimes \left(F^{(l+1)}\right)^{rot(180^\circ)}$ in (2.35) with $(W^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}$ from (2.25) in order to calculate

[5]This is just a consequence of the derivation and is done regardless even if one uses cross-correlation or convolutional operation

the errors $\boldsymbol{\delta}^{(l)}$ in equation (2.35). Combining the other way follows the same logic, but is uncommon, since you lose the spatial information in the fully-connected layer.

2.5 Max-pooling layer

A pooling layer is often used after a convolutional layer in order to down-sample a $n \times n \times c$ input to $n' \times n' \times c$ where $n' < n$, and as such a pooling layer doesn't have any learning parameters associated with it. Max-pooling works by sliding a $w_1 \times w_2$ window over the input image and select the maximum value within the window and discards the other values. Figure 2.4 shows an example of a max-pooling operation using a 4×4 input image and window of size 2×2.

$$\begin{bmatrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \underbrace{\Rightarrow}_{\text{down-sample w. max-pooling}} \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

Figure 2.4: Example of max-pooling with a 2×2 window and stride of 2

As the goal is to down-sample, the stride parameter is usually chosen to be the same size as the sliding window, such that there is no overlap for each stride.

Backwards pass

The backwards pass in a max-pooling layer work by doing the reverse operation, i.e. up-sample. Using the same example as above and say we have a 2×2 error matrix, we want to be able to remap the error values with maximum value during the forward pass. This is in practice done by remembering which index the down-sampled values come from. Figure 2.5 shows an up-sampling operation from the previous forward pass.

$$\begin{bmatrix} \delta_{00} & \delta_{01} \\ \delta_{10} & \delta_{11} \end{bmatrix} \underbrace{\Rightarrow}_{\text{up-sample}} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{00} & 0 & \delta_{01} \\ \delta_{10} & 0 & 0 & 0 \\ 0 & 0 & 0 & \delta_{11} \end{bmatrix}$$

Figure 2.5: Example of up-sampling continued from example in figure 2.4

As the other values don't contribute to the output they are simply set to zero.

2.6 Weight initialization

The initialization of weights plays an important part of having a network converge towards an acceptable minima. Usually a normal or uniform distribution is chosen, but sampling from either distribution also requires choosing a mean and variance. A zero mean is normally chosen, but the choice of variance is not as simple. If the variance is too small, then for neural networks with many layers, the activations will tend towards zero as the data reaches the output layers, which results in close to zero gradients and thus will the network not be updated. Choosing a too large variance will result in saturated neurons, for example when using the *tanh* activation function, the output values tends to have absolute large values and the gradient will be close to zero, thus resulting in the network not updating. The *Xavier uniform initialization* [6] provides a good estimate of the variance, which is also the default in Tensorflow[6], which samples numbers from the uniform distribution:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{(N_{in}+N_{out})}}, \frac{\sqrt{6}}{\sqrt{(N_{in}+N_{out})}}\right] \tag{2.38}$$

where N_{in} and N_{out} denotes the number of input and output values in a given layer. There exists other initialization methods (e.g. the He initialization [8]), which all have strength and weaknesses depending on network architecture, activation functions used, etc., but this initialization is fairly robust across most network types. It is therefore chosen to be the standard choice in the library. Bias terms are just initialized to zeros.

[6]`https://tensorflow.org/api_docs/python/tf/get_variable`

Chapter 3

Design and implementation

As Futhark focuses on producing efficient parallel code, it also imposes constraint on the language semantics, in order to do so. These constrains will in turn impose limitations on the design of the library, where the most significant limitations are that arrays cannot be irregular and cannot hold function types. Without these limitations we could define a network as an array of layers each containing a set of functions and weights. Forward and backward passes could then be performed by a `fold`-like operation, which would be the approach in any other functional language, but unfortunately not possible in Futhark. We therefore need to look for alternative solutions.

3.1 Design

The initial solution was to split layer functions and weights into two separate parts, which would avoid the two limitations mentioned above. As layers in a neural network have different sizes, the limitation of irregular arrays, makes a natural one-to-one array representation of weights impossible. To overcome this my initial design used a weight representation of a one-dimensional array, but this required the need for additional information of indexes, to keep track of which weight slices belongs to which layers. Furthermore, additional information was required, such as the layer type, weight shape, activation functions etc., for each layer in the network, where layer type and activation functions where defined by integer values. The idea was that depending on the layer type, the appropriate layer logic would be applied to handle, for instance the forward pass for a fully-connected layer. I could then define a network as `type nn = (weights, layer_information)`. The separation of weights and layer functions meant that we needed a function with some control structure to perform a forward and backward pass for a network. The pseudo-code in Algorithm 2 shows such a function for the forward pass. This would be the approach in most imperative languages and at first sight seems reasonable in Futhark as well. But one main optimization in implementing the backpropagation algorithm efficiently, is to store information produced during the forward pass for the backward pass to avoid recomputing results. These can only be stored as a one-dimensional array, because of the non-regular sizes of layers, and would

Algorithm 2 Forward propagation

```
1: procedure FORWARD PROPAGATION(input, nn)
2:     for l in nn.layer_information do
3:         (activation, dims, weight_indexs) ← l // Extract layer information
4:         weights_flat ← nn.weights[weights_indexs]
5:         weights ← reshape dims weights_flat
6:         output ← apply weights and activation to input
7:         input ← output // Write output to input for next layer
       return output
```

again require an additional set of indices. The general problem with this representation is that the restriction on non-regular arrays, means that most data needs to be stored as one-dimensional arrays, with a corresponding set of indices and shape parameters. Every time a layer needs to perform an operation, it first needs to extract the data and reshape it. While reshaping might not result in compute-intensive operations, it is an unfortunate consequence of this representation. The additional set of indices and shape dimensions creates a library that uses additional information and operations, which can be avoided in most other languages. Thus using an array representation of a network in Futhark is not an optimal solution. Additionally as weights and layer logic is separated, dependencies would be scattered all over the library and extending it would require changes in several places, which will inevitably hinder the maintainability. Splitting weights and layer logic is therefore not a viable solution either.

Recall from Section 2.1 that the network was combined by letting the output of the first layer be the input to next one and that we can express a neural network as a function, that takes some weights and input and returns some output. From the derivation of the backpropagation algorithm, we saw that the errors from the output layer was passed back through the network, where each layer passes errors to the previous one. Using these observations, we can view a neural network as two main functions(i.e., for a network of depth n we can write $f_n(f_{n-1}(\cdots(f_1(\cdot)))$ for the forward pass and likewise for the backward pass $b_1(b_2(\cdots(b_n(\cdot)))))$. This is the main idea behind this implementation, where neural networks are represented as a composition of functions. With this representation, a single layer is now essentially the same as a one-layer network, which defines two functions $f(\cdot)$ and $b(\cdot)$. Conceptually the idea is simple, but the functions need to carry some additional information for this idea to be implemented efficiently. Therefore we need to define abstract functions that can do so:

```
type forwards   'input 'w 'output 'cache =
                        bool -> w -> input -> (cache, output)
type backwards   'w  'cache 'err_in   'err_out '^u =
                        bool -> u -> w -> cache -> err_in -> (err_out , w)
```

Listing 3.1: Auxiliary abstract types for specifying neural networks

where '(apostrophe) means that *input, w, output, cache*, and so on, are abstract types and ^ means that u is a functional type (and also abstract because of the apostrophe). The abstract semantics of the functions are:

- `forwards:` Values of this type take, a boolean, some weights, and input and returns a cache and the output from the network. The cache stores intermediate results, such that when we backpropagate we do not have to recompue the values. The boolean argument is there to indicate if the function is called during training; if it is not, then we can just return an empty cache.

- `backwards:` Values of this type take, a function u for applying the gradients to the weights, some weights, the information stored in the cache from the forward pass, and the errors that are backpropagated from the above layer. The returned value is the updated weights and the errors that is backpropagated further down the network. The function u is provided by the type of optimizer that is used, e.g. gradient descent, which enables the handling of gradients in different ways. Other types of optimizers can then be implemented later on through the function u.

  The boolean is there to indicate if it is the first layer of the network, and if it is, then we do not need to calculate and backpropagate the error. This is a small optimization, but can give a performance increase on longer training passes.

As these are fully abstract function types, means that a layer implementation needs to define it's own concrete types, but note that, how one layer chooses it's concrete types will affect if the layer can be combined with other layer types, which in some cases will require an utility layer. In the implementation we require a layer to supply these two functions with the abstract semantics above, but the internal logic is defined by the layer it self. Additionally must the layer also define and possibly provide some weights. Combining it all we can write the complete network type as:

```
type NN 'input 'w 'output 'c 'e_in 'e_out '^u =
        { forward  : forwards input w output c,
          backward: backwards c w e_in e_out u,
          weights  : w }
```

Listing 3.2: Abstract type for representing a neural network

In the implementation I'm using a more concrete function in the network type instead of the abstract function u:

```
-- The weight definition used by optimizers
type std_weights 't = ([][]t, []t)
type apply_grad 't  = std_weights t → std_weights t → std_weights t
```

Listing 3.3: Function signature for applying gradient

As optimizers have to operate on the weights and gradients, they need to know its concrete type, and therefore the abstract function `apply_grad` with a concrete signature is used. Layers that do not use this weight representation needs to reshape their weights and gradients before applying the function. Most optimizers update gradients and weights in bulk operations (e.g. all gradients gets the same learning rate applied), and therefore will reshaping not affect the update, but if some optimizer does not treat gradients in bulk, then this design may be a non-optimal.

Networks can be combined by using the core function of the library `connect_layers`, which takes two networks and combines them into one through lambda expressions.

```
1  let connect_layers 'w1 'w2 'i 'o1 'o2 'c1 'c2 'e1 'e2 'e
2      ({forward=f1, backward=b1,
3      weights=ws1}: NN i w1 o1 c1 e e1 (apply_grad t))
4      ({forward=f2, backward=b2,
5      weights=ws2}: NN o1 w2 o2 c2 e2 e (apply_grad t))
6      : NN i1 (w1,w2) (o2) (c1,c2) (e2) (e1) (apply_grad t) =
7
8    {forward = \(is_training) (w1, w2) (input) ->
9      let (c1, res)  = f1 is_training w1 input
10     let (c2, res2) = f2 is_training w2 res
11     in ((c1, c2), res2),
12
13   backward = \(_) u (w1,w2) (c1,c2) (error) ->
14     let (err2, w2') = b2 false u w2 c2 error
15     let (err1, w1') = b1 true u w1 c1 err2
16     in (err1, (w1', w2')),
17
18   weights = (ws1, ws2)}
```

Listing 3.4: Function for combining two networks

The forward functions are straightforwardly combined and like so for the backward functions, just in reverse order. As mentioned above, there are some type restrictions when combining two networks, nn_1 and nn_2. The output type of nn_1 must match the input type of nn_2 and the error output type of nn_2 must match the error input type of nn_1, which are the only restrictions when connecting two networks. These restriction are also reflected in the two neural network types in Listing 3.4, (e.g. the output type from the first network, $o1$ on line 3 is the same as the input to the second network on line 5). Weights and caches are also combined into tuples, becoming more and more nested as the network depth increases. The tuples are then unfolded when functions are called deeper into a multilayer network. This representation completely avoids the use of 1-dimensional arrays and indexing, but instead inline functions and make use of tuples to represent a network. We can store information in their correct form and also avoid storing auxiliary information. Additionally, extending the library with a new layer, is only limited to the concrete layer implementation itself and does not affect other parts. Such a representation is clearly favored and is also more natural to the definition of a neural network.

3.2 Library structure

Having defined the core types and functions of the library, this section will provide an overview of the library structure. The implementation makes use of the ML-style module system in Futhark. As a neural network consists of many components, we naturally should separate each of these into their own modules. Weight initialization-, activation- and loss functions are implemented within their own modules respectively. Layer implementations are a bit different, where I define a module type (i.e. an abstract module) *layer_type*, which each concrete layer implementation uses. Each concrete layer implementation is then collected in the module *layers*, such that we can access all of the layers through a single module. Optimizer implementations follow the same structure as layers. As layers and optimizers in general have more complex implementations, using a level of abstraction is needed. Lastly is the *neural_network* module, where the `connect_layers` function is implemented. This module contains more generic utility functions, such as functions for calculating the loss or accuracy of a network. The next sections will provide details on the implementations, starting with activation functions.

3.3 Activation functions

Recall that an activation function has the characteristic of being *differentiable*, such that. we can use it's derivative during the backward pass in order to calculate the gradient. Therefore are activation functions represented as a pair, containing the function it self and it's derivative, i.e. we can define it's abstract type as

```
type activation_func 'o = {f:o -> o, fd:o -> o}
```

Listing 3.5: Abstract activation function type

The activation functions provided in this implementation all use a 1-dimensional array as their concrete type, which is required since the softmax function is applied on a sequence of activations. A key reason for this representation is that the user can define their *own* pair of functions and use them should the library not contain them. This ensures a flexible system, where the user is not limited to the library implementation. Supported activation functions are *sigmoid, tanh, ReLU, identity* and *softmax*, though softmax can not be used during training in a layer[1]. The implementations follows the definitions given in Table 2.1 and only softmax has been implemented differently by subtracting the maximum value on each element, before applying the exponential function such that overflowing is avoided. The activation functions can be accessed through the *neural_network* module through simple wrappers, following the same interface as Tensorflow.

[1]See Section 3.11 for more details

3.4 Loss function

The definition of loss functions follows the same idea as activation functions, but they just have a different signature.

```
type loss_func 'o 't = {f:o -> o -> t, fd:o -> o -> o}
```

Listing 3.6: Abstract loss function type

Again, this allows users to define their own loss functions and in this implementation a loss functions concrete type of *'o* is a 1-dimensional array. Supported loss functions are *cross entropy, cross entropy with softmax* and *sum of squares*. The implementation of these follow the definitions given in Tabel 2.2.

Both activation- and loss function types are defined globally as they are also used by concrete layer modules and the *neural_network* module.

3.5 Optimizers

In this implementation optimizers performs the backpropagation algorithm, by calling the forward and backward passes on a network and applies the gradients through it's own implementation of the abstract function `apply_grad`. In order to provide options on future implementations, optimizers are separated with their own module type. The module type of an optimizer is defined as follows:

```
module type optimizer_type = {
 type t
 type ^learning_rate
 -- | Train function with signature
 --    network -> learning_rate -> input data -> labels ->
 --    batch_size -> loss function ->
 --    Returns the same network with updated weights
 val train 'i 'w 'g 'e2 'o :
                NN ([]i) w ([]o) g ([]o) e2 (apply_grad t)
             -> learning_rate
             -> (input_data:[]i)
             -> (labels:[]o)
             -> i32
             -> loss_func o t
             -> NN ([]i) w ([]o) g ([]o) e2 (apply_grad t)
}
```

Listing 3.7: Abstract optimizer module

Where *learning_rate* is allowed to be a function type, which allows an optimizer implementation to adapt the learning rate for different training steps with a user defined function. The restrictions on the abstract function `train` is straightforward, given a neural network, the input and output should match the input data and the labels respectively. The input data and labels must be an array type, where each data point is an element, such that we can easily loop through the input data. The loss function given as argument should also match the output type. The only optimizer the implementation provides is gradient descent defined in the *gradient_descent* module. The process of training a network is performed by a loop:

```
1  let train [n] 'w 'g 'o 'e2 'i (
2            {forward=f,
3             backward=b,
4             weights=w} : NN ([]i) w ([]o) g ([]o) e2 (apply_grad t))
5           (alpha:learning_rate)
6           (input:[n]i)
7           (labels:[n]o)
8           (batch_sz: i32)
9           ({f=_, fd=loss'}:loss_func o t) =
10
11   let i = 0
12   let updater = apply_grad_gd alpha batch_sz
13
14   let (w',_) = loop (w, i) while i < length input do
15     let input'          = input[i:i+batch_sz]
16     let label'          = labels[i:i+batch_sz]
17     let (cache, output) = f true w (input')
18     let error           = map2 (\o l -> loss' o l) output label'
19     let (_, w')         = b false updater w cache error
20     in (w', i + batch_sz)
21   in {forward = f, backward = b,  weights = w'}
```

Listing 3.8: Train function in gradient descent module

We forward propagate the input and calculate the error(line 17 and 18), which are then backpropagated through the network to get the updated weights (line 18). The `updater` value is a function that takes the gradients and weights of a layer and performs the update using equation (2.8). The process is repeated, until all of the given input data has been processed, returning the same network with the updated weights.

3.6 Layers

Recall that the errors are backpropagated back through the network using this following equation for MLP:

$$\boldsymbol{\delta}^{(l)} = \underbrace{(W^{(l+1)})^T\boldsymbol{\delta}^{(l+1)}}_{error'} \circ \sigma'^{(l)}(W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}) \tag{3.1}$$

The *error'* part is backpropagated in this implementation as the error term, as it can be calculated at layer $l + 1$, but not at layer l. The same part of the error term is used for a convolutional network. Recall also that when connecting a convolutional layer with a fully-connected one, we'd need to swap this exact part. Therefore must the layer implementations follow this convention, such that we can backpropagate the errors correctly. The remaining part, $\sigma'_{(i)}(W^{(l)}\boldsymbol{z}^{(l-1)}+B^{(l)})$ is calculated at layer l, where $W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}$ is retrieved from the cache. The abstract layer module *layer_type* is defined as:

```
module type layer_type = {
  type t
  type input_params
  type ^activations

  type input
  type output
  type weights
  type err_in
  type err_out
  type cache
  -- Initialize layer given input params,
  -- activation func and seed
  val init: input_params -> activations -> i32 ->
      NN input weights output cache err_in err_out (apply_grad t)
}
```

Listing 3.9: Abstract layer module

Thus a concrete layer implementation must define it's own *input, output, weights, err_in, err_out*, and *cache* types and must provide a function, that initializes the layer given its own defined input parameters and activation function. The integer given is used as a seed parameter, which is used for layers with weight initialization. Note that layer functions are expecting a batch of data points at the time for forward and backward passes, such that the parallelism is optimized.

The implemented layers are *dense (fully-connected), 2D convolutional, max-pooling*, and *flatten*. The latter is a utility layer, which allows a convolutional layer to be combined with a dense one.

3.6.1 Dense

The dense layer is initialized with a tuple of two integers, (m, n), which represents the input and output dimensions respectively. The weights are then represented as a matrix of dimensions $n \times m$, where each row represents the weights associated with a neuron, along with a 1-dimensional array of length n for the biases, following the same representation as in Section 2.1. The forward pass is implemented one-to-one using equation (2.23), with appropriate transposing of the input data, as it is in row format. Matrix multiplication is performed using the function `matmul` from the `futlib/linalg` library. The cache in a dense layer consists of the original input and the result after applying the biases, which are used during backpropagation. The backward pass is implemented using equation (2.25), (2.26) and (2.27) directly.

3.6.2 Convolutional

The implementation of a convolutional layer is not as straightforward as the dense one. There are many ways convolutions can be implemented efficiently today, with each method having different strengths and weaknesses. The need for performing fast convolution is evident in that convolutional networks are used in real-time applications such as self-driving cars for detecting pedestrian, which requires low latency, so the success of a convolutional network is limited by how fast we can perform the convolution [14]. Convolutional operations are compute-expensive operations and H. Kim et. al. [12] show that more than 70% of the training time is spent in convolutional layers on the AlexNet network, which consists of 5 convolutional layers and 4 fully-connected layers. The search for faster convolutional algorithms is still an active area of research, where the common approach is to reduce the amount of multiplication operations at the expense of additions and/or use of auxiliary memory.

The simplest way is to implement the convolutional operation directly, following equation (2.31), but this also leads to very slow performance. Another approach is to lower the convolution into a matrix multiplication, which can be done by transforming image data into matrices using a function called `im2col`, which arranges the image slices into a column matrix. This is also called the GEneric Matrix-Matrix multiplication (GEMM) approach. By representing filters as matrices as well, we can perform the convolutional operation as a matrix multiplication. Matrix multiplication can be done very efficiently on GPUs, as it can utilize the local memory that has low latency and high bandwith. The downside is that it uses a lot of auxiliary memory and also additional resources to transform the data to matrix form. As an example given a 4×4 image and a filter of size 2×2 with a stride of 1, Figure 3.1 shows how the transformation duplicates the data by a factor of 2.25. This factor increases linearly with the image size, fixing everything else.

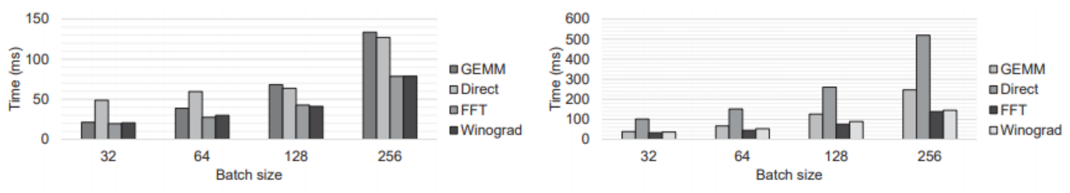
$$\begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \underbrace{\Rightarrow}_{\text{im2col}} \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{10} & x_{11} & x_{12} & x_{20} & x_{21} & x_{22} \\ x_{01} & x_{02} & x_{03} & x_{11} & x_{12} & x_{13} & x_{21} & x_{22} & x_{23} \\ x_{10} & x_{11} & x_{12} & x_{20} & x_{21} & x_{22} & x_{30} & x_{31} & x_{32} \\ x_{11} & x_{12} & x_{13} & x_{21} & x_{22} & x_{23} & x_{31} & x_{32} & x_{33} \end{bmatrix}$$

Figure 3.1: Example of *im2col* operation on a 4×4 image with a 2×2 filter and stride of 1

The Fast Fourier Transformation (FFT) along with the convolution theorem is another popular approach [23], but this approach only performs well for large filter sizes as the extra padding on small filters and unused calculations outweighs the benefit of performing element-wise multiplications. It also only works with a stride equal to one [12, p. 59]. For filters of smaller sizes, 5 and below, the Winograd minimal filtering (WMF) [14] algorithm is usually better than FFT. The WMF algorithm is also based on the GEMM approach and achieves its performance gain by transforming the data with pre-computed auxiliary matrices, for each filter size, which reduces the arithmetic complexity of the convolutional operation. Because of these matrices needs to be pre-computed means that each filter size requires a special case and therefore is the WMF algorithm only applicable to a small set of filter sizes. The two latter methods also uses excess amount of auxiliary memory to hold intermediate results.

Note that determining which algorithm is fastest cannot always be predetermined, as it also depends on batch size, stride, and on. For example, Tensorflow executes all available algorithms, to figure out which is best [12, p. 60]. cudNN [18] have all three approaches implemented into their library, but clearly implementing all three algorithms is a huge task and not feasible in this project. As both FFT and WMF algorithms doesn't apply for a general convolutional layer, these algorithms have been neglected for now.

This leaves us with one last option, which is to use the GEMM approach, which can be applied in the general case. H. Kim et. al [12] (see Figure 3.2) show that the GEMM approach performs reasonably well for small batch sizes, but scales poorly as the batch sizes increases, because the transformation to matrix form becomes too expensive.



(a) The execution time of the forward computation

(b) The execution time of the backward computation

Figure 3.2: Performance comparison on forward and backward passes between Direct convolution, GEMM, FFT and WMF algorithm on the AlexNet (source: [12])

The auxiliary memory usage can also become an issue and caused memory allocation failures on a NVIDIA GTX 970 with 4GB of memory. The cudNN library solves this by reading fixed sized sub matrices of the input data from the off-chip memory onto the on-chip memory successively and computes a subset of the output. This is done while fetching the *next* sub matrices onto the on-chip memory, essentially hiding the memory latency associated with the data transfer, thus is the computation only limited by the time it takes to perform the arithmetic, while limiting the auxiliary memory usage[2, p. 5]. The cudNN library provide both options in their API (i.e. to form the matrix explicitly or implicitly). Interestingly enough, [12] shows that the GEMM method has a less or equal peak memory usage than direct convolution, though it is not clear which GEMM method is used in the paper, but assuming that the explicit method is used, since they have $4 \times 12GB$ of GPU memory and the showed peak memory usage is below

6GB. The same result was also encountered in practice[2], where the direct convolution not only had worse performance, but got memory allocation failures at fewer data points. The exact reason for this is unknown, but it seems that the GEMM method is superior to direct convolution in terms of performance and memory usage.

The implicit GEMM approach is not possible in Futhark, and the closest approach is to perform a loop, which iterate through the input, computing each in chunks, but this approach does not hide the memory latency and seems like a half solution to the problem, as different systems have different memory capacities. As the memory allocation failures only occur, when calculating the accuracy for many data points at the same time, well above the normal batch sizes used during training, the implicit method has been skipped in this work, but consider this future work to find an alternative solution.
Therefore have the explicit GEMM approach been implemented, which transforms the input image into matrix form explicitly.

Implementation

The convolutional layer implementation takes four parameters, *number of filters*, *filter size*, *stride*, and *input depth.* The latter is needed to initialize the proper filter depth to match the input depth. Note that it is only possible to have square filters, but as it is the most common, it is not a big issue, but extending it to allow for non-square filter sizes is not difficult. The input layout for N images is assumed to be $N \times D \times H \times W$, where D, H and W is depth, height, and width respectively.

Forward pass

The forward pass is done by using the `im2col` function, which transforms the image given filter size and image offsets into a matrix. By representing filters as a matrix and with the image matrix in place the convolutional operation can be performed by a matrix multiplication. The biases and the activation function is then applied to the result.

$$\begin{bmatrix} f_{0,0,0} & f_{0,1,0} \\ f_{1,0,0} & f_{1,1,0} \end{bmatrix} \begin{bmatrix} f_{0,0,1} & f_{0,1,1} \\ f_{1,0,1} & f_{1,1,1} \end{bmatrix} \Rightarrow \begin{bmatrix} f_{0,0,0} & f_{0,1,0} & f_{1,0,0} & f_{1,1,0} & f_{0,0,1} & f_{0,1,1} & f_{1,0,1} & f_{1,1,1} \end{bmatrix}$$

Figure 3.3: A single filter of size $2 \times 2 \times 2$ representation in a convolutional layer

The cache consists of the image matrix, which avoids doing the transformation again during the backward pass, but we need to store the original image dimensions additionally. We also cache the result of the convolutional operation after applying the bias in the *correct* format, such that we do not have to reshape it when we perform the Hadamard product during backpropagation.

[2]There is an experimental implementation on branch 'directconv', which uses direct convolution, but the implementation is rather messy and no longer maintained

Backward pass

The backward pass is based on equations (2.35) and (2.36). Having calculated $\boldsymbol{\delta}^{(l)}$, we need to flatten each of the layers, to be able to perform a matrix multiplication for the convolutional operation in (2.36) with the image matrix from the cache. For backpropagation of the errors to the previous layer, we need to flip the filters first, which can be done by slicing into the filter vector and reverse each filter separately using the Futhark function `reverse`.

$$\begin{bmatrix} f_{1,1,0} & f_{1,0,0} & f_{0,1,0} & f_{0,0,0} & f_{1,1,1} & f_{1,0,1} & f_{0,1,1} & f_{0,0,1} \end{bmatrix}$$

Figure 3.4: Flipped filter from Figure 3.3.

Recall that equation (2.35) is a full convolution and we need to pad $\boldsymbol{\delta}^{(l)}$ before transforming it into matrix form. From that representation we can perform a matrix multiplication again to perform the convolutional operation.

3.6.3 Max pooling

A max pooling layer is initialized with a tuple of two integers, (wm, wn), which represents the dimensions of the sliding window, where the stride in the two dimensions respectively is implied from those parameters. The forward pass is done by sliding the pooling window over the input image, where each slice is given to the function `max_val`, which returns the index of the maximum value in the slice and the value itself as a tuple. The index is then transformed to an offset in the original image as if it was an 1-dimensional array and stored along with the maximum value. Lastly we unzip the offsets from the values and keep the offsets in the cache and forward propagate the down-sampled values.

The backward pass is done by using the offsets from the cache along with the `scatter` function. The original image size is first created as a 1-dimensional array and filled with zeros. Each of the 2-dimensional errors given is then flattened, and we can then perform a `scatter` operation on each of them with the corresponding set of offsets. Now every value is in the correct place and before returning, we reshape the errors into the correct shape.

3.7 Weight initialization

The Xavier initialization is implemented using the module *uniform_real_distribution*, which generates numbers from a uniform distribution, from the Futhark library `futlib/random`. The Xavier initialization function is defined in the *weight_initializer* module in the library. The implementation samples each number separately by generating a state from a seed s.

```
1 let gen_rand_uni (s:i32) (dist:(uni.num.t, uni.num.t)): t =
2   let rng = uni.engine.rng_from_seed [s]
3   let (_, x) = uni.rand dist rng in x
```

Listing 3.10: Function for sampling a uniform number from the seed s, by generating a state in line 2, which is used to sample the number in line 3

Normally you would only generate a single state for a sequence of random number, where the state is updated and passed along for each call to `uni.rand` in Listing 3.10, but with this implementation the function can be called with `map`, which allows one to generate the numbers in parallel, rather than in a sequential order. The seeds given into the function `gen_rand_uni` are generated through simple arithmetic, based on the users chosen seed and requested weight dimensions. The tests also show that the numbers are sampled within the ranges provided by the distribution parameters, which is sufficient. The initialization function are used by the *dense* and *convolutional* layers, as they are the only ones with weights.

3.8 Additional network functions

As we not only want to train a network, but also want to be able to evaluate a model a number of additional functions are provided:

- `predict`: Given a network, input data and an activation function, the `predict` function performs the forward pass of the network with the input data and returns the output activations.
- `accuracy`: The `accuracy` function takes a network, input data, labels and an activation function and performs the forward pass like `predict`, but additionally compares the output activations from the network with the labels. The choice of comparison is done using either an `argmax` or `argmin` function[3], which is given into the `accuracy` function as argument. The function returns an percentage on how many data points the model correctly predicts, $\frac{\#hits}{\#datapoints}$.
- `loss`: The function calculates the loss on a network given some input data, labels and a loss function. The accumulated loss of the input data and labels is returned.

These functions are defined in the *neural_network* module.

3.9 Putting it all together

The implementation defines a *deep_learning* module, which combines all of the modules, *layers*, *optimizers*, *loss* and *neural_network* such that we only have to instantiate a single

[3]Recall from Chapter 2 that the output are interpreted as probabilities, and therefore will the output activation with highest probability be the prediction of the input. The `argmin` comparison is usually used for cases, where the goal is predict, which is *not* correct.

module. Having defined all of these components, we can now see how one can built the convolutional network defined in Table 3.1

Layer type	**Filters/neurons**	**Filter/window size**	**Stride**	**Activation function**
conv2d	32	5×5	1	relu
max pooling	0	2×2	2	N/A
conv2d	64	3×3	1	relu
max pooling	0	2×2	2	N/A
dense	1024	N/A	N/A	identity
dense	10	N/A	N/A	identity

Table 3.1: Convolutional network with input dimension of $1 \times 28 \times 28$

The network in Table 3.1 is build to be trained on the MNIST data-set. Listing 3.11 show how we can put together such a network using the library:

```
import "../lib/deep_learning"
module dl = deep_learning f32
let seed = 1

let conv1     = dl.layers.conv2d (32, 5, 1, 1) dl.nn.relu seed
let max_pool1 = dl.layers.max_pooling2d (2,2)
let conv2     = dl.layers.conv2d (64, 3, 1, 32) dl.nn.relu seed
let max_pool2 = dl.layers.max_pooling2d (2,2)
let flat      = dl.layers.flatten
let fc        = dl.layers.dense (1600, 1024) dl.nn.identity seed
let output    = dl.layers.dense (1024, 10)   dl.nn.identity seed

let nn0   = dl.nn.connect_layers conv1 max_pool1
let nn1   = dl.nn.connect_layers nn0 conv2
let nn2   = dl.nn.connect_layers nn1 max_pool2
let nn3   = dl.nn.connect_layers nn2 flat
let nn4   = dl.nn.connect_layers nn3 fc
let nn    = dl.nn.connect_layers nn4 output
```

Listing 3.11: Example of building a convolutional network w. the library

where we first define each of our layers separately. We can then build our network using the `connect_layers` function. Having defined our network we can train it and calculate the accuracy as such:

```
1   let main [m] (input: [m][]dl.t) (labels: [m][]dl.t) =
2     let input' = map (\img -> [unflatten 28 28 img]) input
3     let train = 64000
4     let validation = 10000
5     let batch_size = 128
6     let alpha = 0.1
7     let nn' = dl.train.gradient_descent nn alpha
8       input'[:train] labels[:train]
9       batch_size dl.loss.softmax_cross_entropy_with_logits
10    in dl.nn.accuracy nn'
11      input'[train:train+validation]
12      labels[train:train+validation]
13      (dl.nn.softmax) (dl.nn.argmax)
```

Listing 3.12: Example of training a network w. the library

The program first trains the network on 64000 data points with a batch size of 128 and learning rate of 0.1 on line 7. The accuracy of the trained network is then calculated on 10000 separate data points on line 10. The program can be found at `https://github.com/HnimNart/deep_learning/blob/master/programs/mnist_conv.fut` and the data used to run this can be downloaded at `http://napoleon.hiperfit.dk/~HnimNart/mnist_data/mnist_100000_f32.bindata`. On a Unix-like system you can compile it with the `futhark-opencl` compiler and run it with these two commands:

```
$ futhark-opencl mnist\_conv.fut
$ ./mnist_conv < path/to/mnist_100000_f32.bindata
```

3.10 Testing

As we have seen, a neural network implementation has many components, and if just one of these is implemented wrong, then the network will not train properly. One way to test is to compare the same types of networks to existing libraries and check if one can achieve the same accuracy on the same data modulo the weight initialization. For sufficiently large training data, the accuracies should converge towards the same. Such testing have been done with a MLP and a convolutional network. The comparison is done with Aymeric Damien's Tensorflow examples[4], *neural_network.py* and *convolutional_network.py*, with some minor modifications. Particularly, the dropout layer in the convolutional network is removed and the optimizer is changed to gradient descent. The data used is the MNIST dataset and both networks uses the loss function, *cross entropy with softmax*. The modified Tensorflow programs along with the Futhark programs can be found in Appendix C.1 and the accuracies from running each programs ten times with different seeds is shown in Appendix B. The network structure and training parameters used for the MLP are in Table 3.2 with results in Table 3.3.

[4]`https://github.com/aymericdamien/TensorFlow-Examples/tree/master/examples/3_NeuralNetworks`

Layer type	Neurons	Activation function
dense	256	identity
dense	256	identity
dense	10	identity

Parameter	Value
Training steps	500
Batch size	128
Learning rate	0.1
Training data #	64.000
Validation data #	10.000

Table 3.2: MLP network with input dimension of 1×784 and training parameters.

Program	Accuracy	Std deviation	95% confidence interval
Tensorflow	90.78 %	0.003142	$[0.9058, 0.9097]$
Futhark	90.21 %	0.002777	$[0.9004, 0.9038]$

Table 3.3: Mean, standard deviation and confidence interval of accuracy test for MLP

The confidence interval do not overlap, which could be due to unlucky variable initialization, however the intervals are relatively close to each other, which indicates that the two libraries computes similar results. Repeating the same process for the convolutional network defined in previous section, where we use the same training parameters as for the MLP, we get the results shown in Table 3.4

Program	Accuracy	Std deviation	95% confidence interval
Tensorflow	97.26 %	0.00296	$[0.9708, 0.9745]$
Futhark	97.27%	0.00301	$[0.9708, 0.9746]$

Table 3.4: Mean, standard deviation and confidence interval of accuracy test for convolutional network

The results show very similar accuracy, and provide some confidence that the implementation computes the same as Tensorflow. Furthermore is unit tests provided, which are located at `https://github.com/HnimNart/deep_learning/tree/master/tests`. These tests are relatively simple with small numbers and input sizes, such that the results can be calculated by hand. The tests can be run by using `futhark-test ./` from the folder.

3.11 Shortcomings of the implementation

This section will discuss some of the short comings in the library. Most of these are minor issues, which does not effect the performance or accuracy, but should be noted nevertheless.

3.11.1 Specifying input dimension

In the library, when creating a convolutional or dense layer you must specify the input size, such that the correct filter or weight sizes can be generated. This is due to Futhark's lacking support for recursive types, and when creating a layer, it cannot know the input size from the previous layer. It's a minor issue and a neural network developer should know these sizes, but would provide a tiny improvement to the usability. To solve this with the current version of Futhark, you can wait to initialize the weights until the entire network is assembled, but would require another function to the neural network type.

3.11.2 Swap layers and weights

A common approach when one has sparse data available, say for image classification, is to use an existing network architecture, with a large amount of data available, which you perform initial training on. The trick is that the initial convolutional layers, will have learned to recognize general image patterns, which can be used in your own network. To adapt to your own data, you swap 2 or 3 layers (usually the last ones) out with "fresh" layers, and continue training on the modified network with your own data. Now, since weights for a given network is represented as a nested tuple, it is rather cumbersome to extract and swap weights relative to an array representation. This is an unfortunate consequence of the network representation.

3.11.3 Softmax during backpropagation

The softmax activation function is different from the others, because it is not applied element-wise, but rather element-wise for a sequence. This means that for a sequence, the derivative of the softmax function is a matrix rather than a vector, and therefore when the errors are calculated during backpropagation, it requires to do matrix multiplication rather than the Hadamard product. This is not supported currently, which means that you can not use the softmax activation function during training, but it is also rare that you would use the softmax function in a layer, so it is a minor issue. Rather if you want to calculate the loss using softmax probabilities, you should use `cross_entropy_w_softmax`, which when combined has a simple derivative.

3.11.4 Long compilation times

Though this issue is not directly related to the library, if you compile the convolutional network provided in programs, you would probably notice that compilation time is rather long. If you are in the process of testing different model architectures, this can become annoying. Rerunning networks in other libraries, like Tensorflow, takes a fraction of the time, as they are interpreted and simply execute pre-compiled binaries, which is a benefit when testing different architectures. Thus the benefit in using this implementation comes from the deployment of a model given that it is faster than other frameworks.

3.12 Expanding to recurrent type of networks

Another powerful type of model in deep learning is the recurrent types, which are capable of modeling problems with temporal behavior (i.e. where there is dependencies towards previous input) and are commonly used in natural language processing [22]. Using the unfinished sentence ”The quick brown fox jumps over the ..” as an example, where the goal of the network is to predict the next words, (i.e. ”lazy dog”), then it is clear that a correct prediction is based on the previous words. The recurrent layer would then process each word in succession, but has a ”memory” of previous processed words, which affects the output results. The implementation therefore requires that a layer additionally has a feedback loop, where the previous computation is fed back into the same layer again. Such implementation can be done by feeding the input (e.g. the sentence) into a layer, and let layer handle the processing it self, i.e. process each word[5] of the sentence in succession, (e.g. by using `scan`), thereby is the feedback loop self contained within the layer implementation and therefore is the network representation in this thesis also applicable for a recurrent type of network. Optimizers, activation- and loss function can also be reused and an extension is limited to new recurrent layers.

[5]Note that a fixed size input is normally used, (e.g. 4 letters), and just using a word as an example here for simplicity. We therefore don’t have the problem with the irregular array limitation as all input and output will be of fixed sizes.

Chapter 4

Benchmarks

The benchmarks in this section compares Tensorflow and Futhark performance on the two networks used in the accuracy test, but the benchmark only include the training part of the programs. Each test run is done with 64.000 data points, with four different batch sizes 128, 64, 32 and 16. Recall from section 2.2.1, that there are no universal standard batch size and that batch sizes affects the modeling power, which means that model developers, might need to run their models several times to fine-tune their model. The batch size is also limited by hardware memory combined with size of network architecture. What this means is that a benchmark should not be limited to only one batch size, but rather a range of batch sizes, to provide a comprehensive overview of the performance.
Note that the training steps doubles each time the batch sizes is halved. Because the run time is so small for the MLP, keeping the same training steps, would lead to too small time measurements, which when including noise creates too much fluctuation. Because of there is some noise when running each training run, each training run is done 10 times and the mean of time results are shown. Futhark programs have been benchmarked with `futhark-bench --compiler=futhark-opencl <prog>.fut` using version 0.7.0. Tensorflow version 1.8 is used and the *time* library have been used around the training instructions inside a for-loop with 11 iterations. The first call to the `train` function in Tensorflow will load data to the GPU, but subsequent calls will already have data available in the GPU memory. Tensorflow will also run different algorithms on different layers during the initial training step, to figure out which algorithms performs the best, and this information will be remembered for later calls to the `train` function[12, p.60]. The Futhark benchmark programs can be found at `https://github.com/HnimNart/deep_learning/tree/master/benchmark`[1], and the Tensorflow programs are the same as used in accuracy tests, just with a loop around the `train` function. The graphics card used is NVIDIA GTX 970 with 4GB of memory and all programs uses 32 bit floating points. The times from each run in Tensorflow is in appendix

[1]You need to place the data in the *data*-folder, before running the benchmark in Futhark. The data file is the same as used earlier, i.e. `http://napoleon.hiperfit.dk/~HnimNart/mnist_data/mnist_100000_f32.bindata`

A, along with the output from using `futhark-bench`

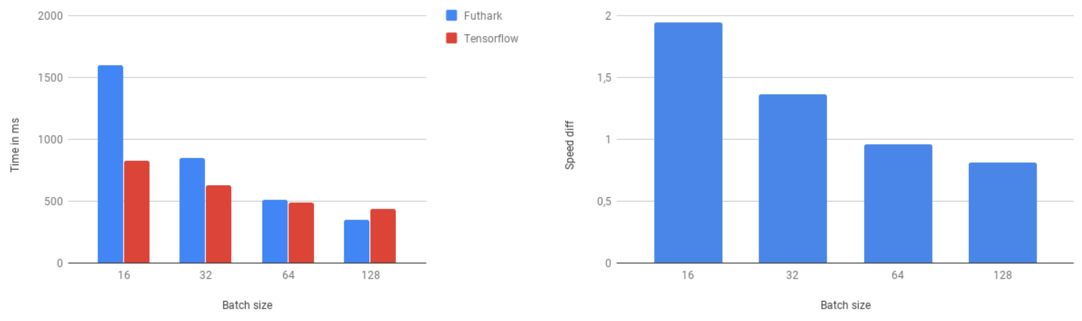
4.1 MLP network

The first benchmark is using the same MLP as used in the accuracy test. As most of the computer-intensive parts in this type of network is implemented using matrix multiplication, I would expect that the times are at least comparable. The mean of the times are shown Figure 4.1 in milliseconds along with a relative speed-difference calculated as $\frac{time_{futhark}}{time_{Tensorflow}}$.

Batch size / Library	16	32	64	128
Tensorflow	824 ms	624 ms	488 ms	434 ms
Futhark	1594 ms	846 ms	506 ms	350 ms
Relative speed diff.	1.94	1.36	1.04	0.81

Table 4.1: Benchmark results for MLP.

The results show that for batch size of 128, Futhark is faster than Tensorflow, and for batch size of 64 the performance is the same. As the batch size decreases to 32 Tensorflow, starts to perform better, and for batch size of 16 Tensorflow is significantly faster than Futhark. This is due to the degree of parallelism shrinks too much for Futhark, though is a batch size of 16 also relatively small and often only used on hardware with limited space.



(a) Benchmark times (ms)

(b) Relative speed diff.

Figure 4.1: Benchmark results for MLP

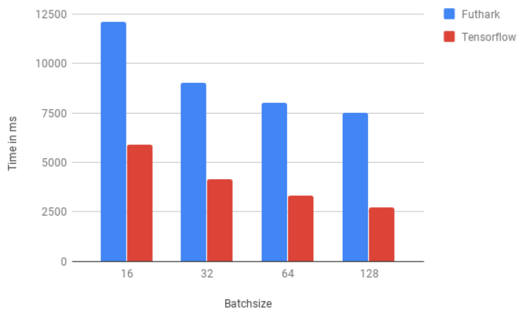
4.2 Convolutional Network

The second benchmark is on the convolutional network used in the accuracy test. The expectation are here lower than before, as we know that the implemented algorithm is not optimal for the convolutional layers. The best that we can hope for are that Futhark is not orders of magnitudes slower. The results are shown in Table 4.2

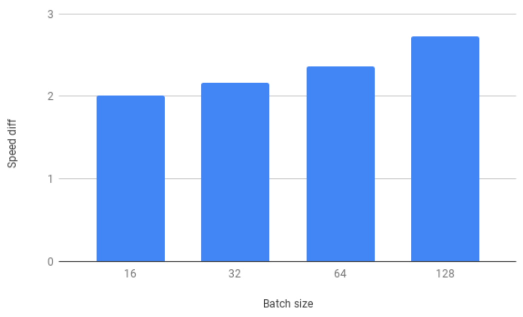
Library \ Batch size	16	32	64	128
Tensorflow	5881 ms	4168 ms	3343 ms	2748 ms
Futhark	12091 ms	9010 ms	7992 ms	7515 ms
Relative speed diff.	2.06	2.16	2.39	2.73

Table 4.2: Benchmark results for convolutional network

The results show that for a batch size of 128, Futhark is 2.73 times slower than Tensorflow, but as the batch decreases the relative difference also shrinks, showing a similar results as [12], i.e. the GEMM approach works best at smaller batch sizes, but the performance are still relatively far from each other. This is primarily due to that the GEMM approach doesn't perform better than the WMF or FFT algorithms on the filter sizes used in this network, even at smaller batch sizes.



(a) Benchmark times (ms)



(b) Relative speed diff.

Figure 4.2: Benchmark results for convolutional network

4.3 Reflection on results

To be able to reflect on the performance on the benchmark results, we should first try assess how fast Tensorflow is, in absolute terms. S. Chintala [3] have benchmarked different convolutional networks with other popular libraries and shows that Tensorflow are among the fastest libraries, but is beaten by Torch[2] and Neon[3], though is Neon significantly faster on larger networks. A comparison between Tensorflow and other cudNN based deep learning libraries in [12] shows that they have similar in performance, when run in 'benchmark' mode, meaning that they choose the best algorithms, either

[2]`http://torch.ch/`
[3]`https://ai.intel.com/neon/`

based on heuristics or simple brute-force. Only Torch is a bit faster in the paper. While these results show that Tensorflow isn't the fastest it's clear that Tensorflow is not considered a slow library either as it is based on cudNN, which have been developed over multiple years, aiming at providing fast code for deep learning applications. The fact that the benchmark shows similar or better performance for the MLP, where the underlying methods is the same, for a batch size of 64 and above is a great result. The small gap on batch size of 32 is acceptable, but the large performance gap for a batch size of 16 is not so great, but one can argue that this result is less important, as this batch size is more uncommon than the others. For the convolutional network, the benchmark results are clear; Tensorflow currently is the faster library, but we know that there is about a 2x performance gain in using better algorithms [12]. Assuming this, the library will be within a factor of 1.5 of Tensorflow, which would be a great accomplishment for a high-level language like Futhark. Once those algorithms have been implemented, a more comprehensive comparison between those two will tell if the performance gap is only due to the choice of algorithms and if Futhark is capable of competing, in terms of performance, with Tensorflow. Overall the benchmark results are promising for a deep learning library in Futhark, which eventually can perform at a similar level as Tensorflow. Additionally as Futhark is a hardware independent language and thus cannot provide as much hardware specific optimizations as cudNN does, makes the benchmark results impressive, especially in the case of the MLP with a batch size of 128. Considering that Futhark still is a relatively new, we can only expect that Futhark compiler will yield even better performance in the future as well.

Chapter 5

Future work and conclusion

5.1 Future work

There are some unsolved issues and shortcomings as described earlier in the implementation. A discussion on how these should be resolved in a proper manner should be prioritized. Having resolved those we can consider the future of the library which will consist of two main parts. First of, implementation of better performing algorithms for the convolutional layer will inevitably increase performance [12]. This might also apply to implementing a matrix multiplication algorithm, with a better asymptotic bound, but might not be easy to properly to do in Futhark efficiently. Nevertheless improving these two main compute-intensive parts would improve the performance of the library. Secondly is to extend the library with more activation functions, loss functions, optimizers and layers etc., giving a more complete library providing users with options.

5.2 Conclusion

This thesis explored implementation options of a deep learning library in Futhark. The final implementation is based on a composition of functions for representing a neural network, which avoids the non-regular array and array of functions limitation of Futhark. The representation also avoids the need for auxiliary information. The implementation also showed that the module system in Futhark is capable of providing the abstraction needed for the complex nature of a deep learning library. While there are still some minor issues in the library, the proposed framework showed that Futhark is fully capable of an implementation of a deep learning library, which shows a high degree of flexibility and maintainability for future work.

The benchmarks showed that for a MLP and larger batch sizes, Futhark is more than capable of competing with dedicated DSL solutions like Tensorflow, but Futhark falls behind when the batch size decreases. For the convolutional network there is still some work to do, if the library is to be able to match Tensorflow's performance. The main reason for the performance gap is due to Tensorflow's superior selection of algorithms.

When those algorithms are implemented into the library, a new benchmark will provide a more fair comparison of the libraries. As this thesis has only scraped the surface of a deep learning library in Futhark, future iterations will undoubtedly progress the library into being faster and more complete.

Appendix A

Benchmark results

A.1 Futhark benchmarks results

```
Results for mnist_128.fut:
dataset ../data/mnist_100000_f32.bindata:  349954.80us (avg. of 10 runs; RSD: 0.00)
Results for mnist_64.fut:
dataset ../data/mnist_100000_f32.bindata:  506426.60us (avg. of 10 runs; RSD: 0.00)
Results for mnist_32.fut:
dataset ../data/mnist_100000_f32.bindata:  845774.50us (avg. of 10 runs; RSD: 0.00)
Results for mnist_16.fut:
dataset ../data/mnist_100000_f32.bindata: 1593802.60us (avg. of 10 runs; RSD: 0.00)
```

Figure A.1: Benchmark results for MLP in Futhark

```
Results for mnist_conv_128.fut:
dataset ../data/mnist_100000_f32.bindata: 7514796.40us (avg. of 10 runs; RSD: 0.00)
Results for mnist_conv_64.fut:
dataset ../data/mnist_100000_f32.bindata: 7991749.90us (avg. of 10 runs; RSD: 0.00)
Results for mnist_conv_32.fut:
dataset ../data/mnist_100000_f32.bindata: 9010751.10us (avg. of 10 runs; RSD: 0.00)
Results for mnist_conv_16.fut:
dataset ../data/mnist_100000_f32.bindata: 12091464.30us (avg. of 10 runs; RSD: 0.00)

```

Figure A.2: Benchmark results for CNN in Futhark

A.2 Tensorflow benchmark results

Batch size / run #	16	32	64	128
1	806060	604798	546822	439358
2	880370	679317	476656	476263
3	809124	631627	472093	531199
4	810173	600780	538741	408783
5	877590	692555	479202	413419
6	806486	622508	478880	407778
7	812005	603003	475040	430840
8	824111	608722	469326	413238
9	811256	603799	473371	413990
10	805981	600677	475712	411898

Table A.1: Benchmark time in μs for each run w. MLP in Tensorflow.

Batch size / run #	16	32	64	128
1	6116550	4465761	3431302	2772002
2	5947696	4164861	3424191	2762104
3	5774723	4102609	3279976	2743420
4	5816716	4081254	3343420	2756505
5	5873528	4276965	3538718	2711210
6	5891137	4114870	3239401	2707600
7	5826807	4160194	3295297	2743304
8	5852604	4080076	3347933	2762100
9	5852070	4129724	3281702	2762154
10	5867789	4107558	3252794	2762101

Table A.2: Benchmark time in μs for each run w. CNN in Tensorflow

Appendix B

Accuracy results

run # \ Library	Tensorflow	Futhark
1	0,9127	0,906
2	0,9117	0,9009
3	0,9086	0,9065
4	0,9061	0,9028
5	0,9102	0,8972
6	0,9011	0,9033
7	0,9066	0,9021
8	0,9063	0,9014
9	0,9069	0,8992
10	0,9084	0,9001
Mean	0.9078	0.9021

Table B.1: Accuracy results for MLP

Library / run #	Tensorflow	Futhark
1	0,9676	0,9709
2	0,974	0,9731
3	0,9708	0,972
4	0,9707	0,9658
5	0,9695	0,9701
6	0,9761	0,9728
7	0,9713	0,9738
8	0,9762	0,9757
9	0,9726	0,9757
10	0,9766	0,977
Mean	0.9726	0.9727

Table B.2: Accuracy results for convolutional network

Appendix C

Programs

C.1 Accuracy programs

C.1.1 MLP Futhark program

```
import "../lib/deep_learning"
module dl = deep_learning f32

let seed = 1

let l1 = dl.layers.dense (784, 256) dl.nn.identity seed
let l2 = dl.layers.dense (256, 256) dl.nn.identity seed
let l3 = dl.layers.dense (256, 10) dl.nn.identity seed

let nn1 = dl.nn.connect_layers l1 l2
let nn  = dl.nn.connect_layers nn1 l3

let main [m] (input:[m][]dl.t) (labels:[m][]dl.t) =
  let train = 64000
  let validation = 10000
  let batch_size = 128
  let alpha = 0.1
  let nn1 = dl.train.gradient_descent nn alpha
    input[:train] labels[:train]
    batch_size dl.loss.softmax_cross_entropy_with_logits
  in dl.nn.accuracy nn1 input[train:train+validation]
    labels[train:train+validation] dl.nn.softmax dl.nn.argmax
```

C.1.2 MLP Tensorflow program

```
""" Neural Network.

A 2-Hidden Layers Fully Connected Neural Network (a.k.a Multilayer Perceptron)
```

```
implementation with TensorFlow. This example is using the MNIST database
of handwritten digits (http://yann.lecun.com/exdb/mnist/).

Links:
[MNIST Dataset](http://yann.lecun.com/exdb/mnist/).

Author: Aymeric Damien
Project: https://github.com/aymericdamien/TensorFlow-Examples/
"""
from __future__ import print_function

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf

# Parameters
learning_rate = 0.1
num_steps = 500
batch_size = 128

# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# Define the neural network
def neural_net(x_dict):
        # TF Estimator input is a dict, in case of multiple inputs
        x = x_dict['images']
        # Hidden fully connected layer with 256 neurons
        layer_1 = tf.layers.dense(x, n_hidden_1)
        # Hidden fully connected layer with 256 neurons
        layer_2 = tf.layers.dense(layer_1, n_hidden_2)
        # Output fully connected layer with a neuron for each class
        out_layer = tf.layers.dense(layer_2, num_classes)
        return out_layer

# Define the model function (following TF Estimator Template)
def model_fn(features, labels, mode):
        # Build the neural network
        logits = neural_net(features)

        # Predictions
        pred_classes = tf.argmax(logits, axis=1)
        pred_probas = tf.nn.softmax(logits)

        # If prediction mode, early return
        if mode == tf.estimator.ModeKeys.PREDICT:
                return tf.estimator.EstimatorSpec(mode, predictions=pred_classes)

        # Define loss and optimizer
        loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        logits=logits, labels=labels))
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
        train_op = optimizer.minimize(loss_op,
        global_step=tf.train.get_global_step())

        # Evaluate the accuracy of the model
        acc_op = tf.metrics.accuracy(labels=tf.argmax(labels,1), predictions=pred_classes

        estim_specs = tf.estimator.EstimatorSpec(
                mode=mode,
                predictions=pred_classes,
                loss=loss_op,
                train_op=train_op,
                eval_metric_ops={'accuracy': acc_op})

        return estim_specs

# Build the Estimator
model = tf.estimator.Estimator(model_fn)

# Define the input function for training
input_fn = tf.estimator.inputs.numpy_input_fn(
x={'images': mnist.train.images}, y=mnist.train.labels,
        batch_size=batch_size, num_epochs=None, shuffle=False)
        # Train the Model
model.train(input_fn, steps=num_steps)

test_input, test_labels = mnist.train.next_batch(10000)
# Evaluate the Model
# Define the input function for evaluating
input_fn = tf.estimator.inputs.numpy_input_fn(
        x={'images': test_input}, y=test_labels,
        shuffle=False)
# Use the Estimator 'evaluate' method
e = model.evaluate(input_fn)
print("Testing Accuracy:", e['accuracy'])
```

C.1.3 Futhark convolutional program

```
import "../lib/deep_learning"
module dl = deep_learning f32
let seed = 1

let conv1     = dl.layers.conv2d (32, 5, 1, 1) dl.nn.relu seed
let max_pool1 = dl.layers.max_pooling2d (2,2)
let conv2     = dl.layers.conv2d (64, 3, 1, 32) dl.nn.relu seed
let max_pool2 = dl.layers.max_pooling2d (2,2)
let flat      = dl.layers.flatten
let fc        = dl.layers.dense (1600, 1024) dl.nn.relu seed
let output    = dl.layers.dense (1024, 10)   dl.nn.identity seed

let nn0   = dl.nn.connect_layers conv1 max_pool1
let nn1   = dl.nn.connect_layers nn0 conv2
let nn2   = dl.nn.connect_layers nn1 max_pool2
let nn3   = dl.nn.connect_layers nn2 flat
let nn4   = dl.nn.connect_layers nn3 fc
let nn    = dl.nn.connect_layers nn4 output

let main [m] (input: [m][]dl.t) (labels: [m][]dl.t) =
  let input' = map (\img -> [unflatten 28 28 img]) input
  let train = 64000
  let validation = 10000
  let batch_size = 128
  let alpha = 0.1
  let nn' = dl.train.gradient_descent nn alpha
    input'[:train] labels[:train]
    batch_size dl.loss.softmax_cross_entropy_with_logits
  in dl.nn.accuracy nn'
    input'[train:train+validation]
    labels[train:train+validation]
    (dl.nn.softmax) (dl.nn.argmax)
```

Tensorflow convolutional program

```
""" Convolutional Neural Network.

Build and train a convolutional neural network with TensorFlow.
```

```
This example is using the MNIST database of handwritten digits
(http://yann.lecun.com/exdb/mnist/)

Author: Aymeric Damien
Project: https://github.com/aymericdamien/TensorFlow-Examples/
"""
from __future__ import division, print_function, absolute_import

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf

# Training Parameters
learning_rate = 0.1
num_steps = 500
batch_size = 128

# Network Parameters
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# Create the neural network
def conv_net(x_dict, n_classes,  reuse, is_training):
        # Define a scope for reusing the variables
        with tf.variable_scope('ConvNet', reuse=reuse):
        # TF Estimator input is a dict, in case of multiple inputs
                x = x_dict['images']
                # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
                # Reshape to match picture format [Height x Width x Channel]
                # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
                x = tf.reshape(x, shape=[-1, 28, 28, 1])
                # Convolution Layer with 32 filters and a kernel size of 5
                conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu)
                # Max Pooling (down-sampling) with strides of 2 and kernel size of 2
                conv1 = tf.layers.max_pooling2d(conv1, 2, 2)
                # Convolution Layer with 64 filters and a kernel size of 3
                conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.relu)
                # Max Pooling (down-sampling) with strides of 2 and kernel size of 2
                conv2 = tf.layers.max_pooling2d(conv2, 2, 2)
                # Flatten the data to a 1-D vector for the fully connected layer
                fc1 = tf.contrib.layers.flatten(conv2)
                # Fully connected layer (in tf contrib folder for now)
                fc1 = tf.layers.dense(fc1, 1024)
                # Output layer, class prediction
                out = tf.layers.dense(fc1, n_classes)
        return out

# Define the model function (following TF Estimator Template)
def model_fn(features, labels, mode):
        logits_train = conv_net(features, num_classes, reuse=False,is_training=True)
        # Predictions
        pred_classes = tf.argmax(logits_train, axis=1)
        pred_probas = tf.nn.softmax(logits_train)

        # If prediction mode, early return
        if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode, predictions=pred_classes)

        # Define loss and optimizer
        loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        logits=logits_train, labels=labels))
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
        train_op = optimizer.minimize(loss_op,
        global_step=tf.train.get_global_step())

        # Evaluate the accuracy of the model
        acc_op = tf.metrics.accuracy(labels=tf.argmax(labels,1) , predictions=pred_classe

        # TF Estimators requires to return a EstimatorSpec, that specify
        # the different ops for training, evaluating, ...
        estim_specs = tf.estimator.EstimatorSpec(
                mode=mode,
                predictions=pred_classes,
                loss=loss_op,
                train_op=train_op,
                eval_metric_ops={'accuracy': acc_op})

        return estim_specs

# Build the Estimator
model = tf.estimator.Estimator(model_fn)

# Define the input function for training
input_fn = tf.estimator.inputs.numpy_input_fn(
```

```python
        x={'images': mnist.train.images}, y=mnist.train.labels,
        batch_size=batch_size, num_epochs=None, shuffle=False)
# Train the Model
model.train(input_fn, steps=num_steps)

test_input, test_labels = mnist.train.next_batch(10000)
# Evaluate the Model
# Define the input function for evaluating
input_fn = tf.estimator.inputs.numpy_input_fn(
        x={'images': test_input}, y=test_labels,
        shuffle=False)
# Use the Estimator 'evaluate' method
e = model.evaluate(input_fn)
print("Testing Accuracy:", e['accuracy'])
```

Bibliography

[1] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag, Berlin, Heidelberg (2006).

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, cudnn: Efficient primitives for deep learning, *CoRR* **abs/1410.0759** (2014).

[3] S. Chintala, *convnet-benchmarks* (2018). `https://github.com/soumith/convnet-benchmarks`

[4] A. Damien, *TensorFlow-Examples* (2018). `https://github.com/aymericdamien/TensorFlow-Examples/tree/master/examples`

[5] M. Elsman, T. Henriksen, D. Annenkov, and . Cosmin E Oancea, Static Interpretation of Higher-Order Modules in Futhark, *Functional GPU Programming in the Large*, ACM Program (September 2018), 30 pages. `https://doi.org/10.1145/3236792`

[6] X. Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATSâĂŹ10). Society for Artificial Intelligence and Statistics* (2010).

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press (2016). `http://www.deeplearningbook.org`

[8] K. He, X. Zhang, S. Ren, and J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, *ICCV '15*, IEEE Computer Society, Washington, DC, USA (2015), 1026–1034.

[9] T. Henriksen, K. F. Larsen, and C. E. Oancea, Design and GPGPU Performance of Futhark's Redomap Construct, *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY 2016*, ACM, New York, NY, USA (2016), 17–24.

[10] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates, *SIGPLAN Not.* **52**, 6 (2017), 556–571.

[11] J. Kafunah, *Backpropagation in Convolutional Neural Networks*, Stanford (2016). `https://canvas.stanford.edu/files/1041875/download?download_frd=1&verifier=tFv4Jc7bCezxJg9rG2yhEKEERi70zJ3ScmFbNlbN`

[12] H. Kim, H. Nam, W. Jung, and J. Lee, Performance analysis of CNN frameworks for GPUs, *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2017), 55–64.

[13] R. W. Larsen and T. Henriksen, Strategies for regular segmented reductions on gpu, *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, FHPC 2017*, ACM, New York, NY, USA (2017), 42–52.

[14] A. Lavin, Fast algorithms for convolutional neural networks, *CoRR* **abs/1509.09308** (2015).

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, Backpropagation applied to handwritten zip code recognition, *Neural Comput.* **1**, 4 (1989), 541–551.

[16] W. Mcculloch and W. Pitts, A logical calculus of ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics* **5** (1943), 127–147.

[17] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA, USA (1969).

[18] NVIDIA, *Deep Learning SDK documentation* (2018). `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html`

[19] S. Raja, *A Derivation of Backpropagation in Matrix Form* (2017). `https://sudeepraja.github.io/Neural/`

[20] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, *Psychological Review* (1958), 65–386.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature* **323** (1986), 533–.

[22] H. Salehinejad, J. Baarbe, S. Sankar, J. Barfett, E. Colak, and S. Valaee, Recent Advances in Recurrent Neural Networks, *CoRR* **abs/1801.01078** (2018).

[23] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, Fast Convolutional Nets With fbfft: A GPU Performance Evaluation, *CoRR* **abs/1412.7580** (2014).