Compositional Deep Learning in Futhark

Duc Minh Tran
University of Copenhagen
Denmark
cwz688@alumni.ku.dk

Troels Henriksen
University of Copenhagen
Denmark
athas@sigkill.dk

Martin Elsman
University of Copenhagen
Denmark
mael@di.ku.dk

Abstract

We present a design pattern for composing deep learning networks in a typed, higher-order fashion. The exposed library functions are generically typed and the composition structure allows for networks to be trained (using backpropagation) and for trained networks to be used for predicting new results (using forward-propagation). Individual layers in a network can take different forms ranging over dense sigmoid layers to convolutional layers. The paper discusses different typing techniques aimed at enforcing proper use and composition of networks. The approach is implemented in Futhark, a data-parallel functional language and compiler targeting GPU architectures, and we demonstrate that Futhark's elimination of higher-order functions and modules leads to efficient generated code.

CCS Concepts • **Computing methodologies → Parallel programming languages**; **Neural networks**; • **Software and its engineering → Software performance**; *Software libraries and repositories.*

Keywords deep learning, data-parallelism, functional languages

ACM Reference Format:
Duc Minh Tran, Troels Henriksen, and Martin Elsman. 2019. Compositional Deep Learning in Futhark. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '19), August 18, 2019, Berlin, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3331553.3342617

1 Introduction

Deep learning artificial neural networks are becoming increasingly important for a large variety of application areas, including areas such as speech recognition, image classification, and search. With the rapid development of new application domains and increased performance demands, the architectures and configurations of deep neural networks have increased in complexity. Today's deep neural network toolkits, including Caffe [17], Torch [6], CNTK [30], and TensorFlow [1], allow end users to configure and combine different kinds of layers in a variety of ways using library APIs or embedded domain specific languages.

Implementations of particular instances of deep neural networks benefit, with respect to performance, from specialisation and the use of massively parallel architectures, such as GPGPUs. Such specialisations can be achieved, for instance, using runtime code generation, as supported by, for instance TensorFlow's XLA [21], or using computation graph analysis frameworks, such as R-Stream.TF [24].

In this paper, we present a prototype deep neural network library, written entirely in the data-parallel functional language Futhark [14], which is a statically typed language, featuring a set of data-parallel second-order array combinators (SOACs), such as `map`, `reduce`, `scan`, and `filter`. The language also features polymorphism, higher-order functions [16], and higher-order modules [7]. Common to these surface language abstraction features is that they come with zero overhead as the features are compiled away at compile time. The effect is that traditional compiler optimisations and optimisations that are essential for obtaining a high degree of data-parallel performance, such as fusion [11–13] and tiling [14], work for SOACs across modules and function boundaries. As a consequence of using Futhark for implementing a deep neural network library, the compilation of specified neural network computation graphs benefits from fusion and other GPU architecture-specific optimisations that Futhark implements.

Figure 1 shows a specification of a dense 3-layer neural network in Futhark. After being trained with images of handwritten digits and information about the digit each image represents (the MNIST data set), the network can be used to infer the denotation of new handwritten digits. The first layer is the *input layer*, consisting of 784 neurons representing 28 × 28 grey-scale pixels. The second layer is a so-called *hidden layer*, which is essential for representing features in the image. The third layer is the *output layer*, from which we can read 10 output values and infer which digit (0-9) the input image, most probably, represents. The first two lines import and instantiate the library module to work with 32-bit floats.

FHPNC '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6814-8/19/08…$15.00
https://doi.org/10.1145/3331553.3342617

```
module dl = deep_learning f32
let (>-) = dl.nn.connect_layers

let mk_dense (i,o) =
  let seed = 1
  in dl.layers.dense (i,o) dl.nn.identity seed

let nn = mk_dense (784,256) >-   -- input layer
         mk_dense (256,256) >-   -- hidden layer
         mk_dense (256,10)       -- output layer

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 nn2 = dl.train.gradient_descent nn alpha
            input[:train] labels[:train]
            batch_size
            odl.loss.softmax_cross_entropy
  in dl.nn.accuracy nn2
       input[train:train+validation]
       labels[train:train+validation]
       dl.nn.softmax dl.nn.argmax
```

Figure 1. Futhark code for specifying a dense neural network for training and inferring hand-written digits based on the MNIST data set.

The mk_dense function takes a number of input edges and a number of output edges and creates a layer of neurons. The infix function >- is used to connect layers. The main function first trains the network with 64,000 images using batch-based stochastic gradient descent. It then measures the accuracy of inferring digits for another 10,000 images.

As we shall see in Section 8, the Futhark program ends up running faster than when the same network is trained with TensorFlow. The Futhark DNN library also has support for convolutional layers. However, due to Futhark not generating sufficiently efficient code for certain matrix operations, Futhark is (still) slower than when the same experiment is performed with TensorFlow.

The contributions of this paper are the following:

- We present a functional and typed design for deep neural networks that is easily extensible by the user.
- We demonstrate by a nontrivial application that high-level language constructs such as higher-order functions do not preclude high performance.
- We show that a non-domain-specific parallel functional language can perform competitively with specialised deep learning frameworks like TensorFlow.

The remainder of this paper is organised as follows. In Section 2, we present some essential background on deep neural networks. In Section 3, we present some of the essential features of the Futhark language. In Section 4, we present how we can define a network essentially to be a pair of two functions, one that operates in forward mode through the layers of the network (for inference) and one that operates in backwards mode (for learning). We also show how two networks are composed by, essentially, composing the forward and backward functions, individually. In Section 5, we show how the different kinds of layers are implemented, including dense layers, convolutional layers, and other administrative kinds of layers. Sections 6 and 7 present additional network functionality and how an elaborate convolutional network is assembled using the library. In Section 8, we evaluate the performance of code generated using the approach. In Section 9, we present related work and in Section 10, we conclude and describe possible future work.

2 A Neural Network Primer

A deep neural network is defined as a composition of layers. Different kinds of layers exist, including dense sigmoid layers, convolutional layers, max-pooling layers, and others.

Each layer consists of a fixed set of units (or neurons) with adaptable parameters, which can be changed by a process called training. When some input data is passed to the network, each successive layer uses the output from the previous layer as input. The result of evaluating a network on some input will then be available as the output of the final layer. The interpretation of the output from a network depends on the modeling problem. A common one is the *multiclass* problem with 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.

The simplest form of a deep neural network consists of a number of dense layers, for which each output of a layer is connected to every neuron in the next layer. We have already seen an example of such a network in Section 1.

A key property of a deep neural network is that it can adjust its internal weight parameters by a training process called back-propagation [22, 26]. This feature comes from the representation of each neuron, which is a function that takes n input values and results in a single output value. Based on the partial derivatives of these n-variate functions that make up the network layer, we can deduce how a particular suggested change in the output should affect the weights defining the neuron functions and how we should suggest that the output of previous layers should change.

For utilising parallel architectures efficiently, the back-propagation phase is organised in a so-called *batched stochastic gradient descent* process, for which we will compute suggested changes for a batch of training data (e.g., stochastically chosen sets of training images) in parallel. The process will then repeat the parallel computation of network changes until the network stabilises. The process is parameterised over a learning rate (often named α), a batch size, and, of course, the size of the training data set.

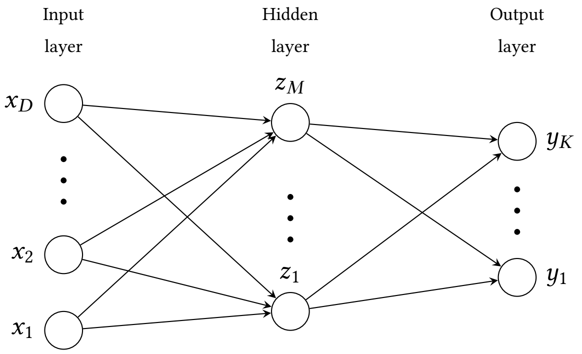

Figure 2. A 3-layer dense neural network with an input layer, a hidden layer, and an output layer.

An example 3-layer dense neural network is shown in Figure 2. We can describe 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, we can write the first layer calculation as:

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

where $j \in \{1...M\}$. The superscript (1) refers to the layer being the first layer. The w_{ji} is called the weight, the b_j is 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

$$z_j = \sigma(a_j) \tag{2}$$

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 the final layer is reached. Following the same procedure for the next layer, we can write

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + b_k^{(2)} \quad \text{for } k = 1, .., K \tag{3}$$

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

$$\mathit{softmax}(a_k) = y_k = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}} \text{ for } k = 1...K \tag{4}$$

where $0 \leq y_k \leq 1$ with $\sum_{k=1}^{K} y_k = 1$, which can be interpreted as a probability. Combining (1), (2), (3), and (4), we can express the network as a composition of operations and the network function therefore takes the form

$$y_k(\mathbf{x}, \mathbf{w}) = \mathit{softmax}\left(\sum_{j=1}^{M} w_{kj}^{(2)} \sigma\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)}\right) + b_k^{(2)}\right) \tag{5}$$

Loss function	$E(\mathbf{w})$	$\frac{\partial E(\mathbf{w})}{\partial y_k}$
Cross-entropy (CE)	$-\sum_{k=1}^{K} (t_k \; ln \; y_k)$	$-\frac{t_k}{y_k}$
CE with 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	$\frac{1}{2} \sum_{k=1}^{K} (y_k - t_k)^2$	$y_k - t_k$

Figure 3. Common loss functions and their derivatives.

Here we have grouped the weights and bias parameters into $\mathbf{w}$. Thus a dense neural network is a nonlinear function from a set of input variables $\mathbf{x}$ to a set of output variables $\mathbf{y}$, controlled by a set of adjustable parameters, $\mathbf{w}$. For implementation purposes, we can rewrite this formulation into matrix form and use matrix-vector multiplication instead of summations. For a neuron j in the first layer, we have 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 $B^{(2)} : \mathbb{R}^{K}$. All in all, we have

$$\mathbf{y}(\mathbf{x}, \mathbf{W}) = \mathit{softmax}\left(W^{(2)} \sigma\left(W^{(1)}\mathbf{x} + B^{(1)}\right) + B^{(2)}\right) \tag{6}$$

The above process of evaluating (6) is called the *forward propagation* of information through the network.

Activation functions ($\sigma(\cdot)$) are required to be differentiable, which is necessary when training networks, since we need to use the derivative of the input for backpropagation through the network. Common activation functions include *tanh*, ReLU (i.e., rectified linear unit), *sigmoid*, and *softmax*. As an example, the *sigmoid* function is defined as $\mathit{sigmoid}(x) = \frac{1}{1+e^{-x}}$ with the derivative $\mathit{sigmoid}'(x) = \mathit{sigmoid}(x)(1 - \mathit{sigmoid}(x))$.

2.1 Training

Before we can use a neural network to make a prediction about a given input, we first need to train the network. The idea is that we want $y_k(\mathbf{x}, \mathbf{w})$ 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 $E(\mathbf{w})$. Figure 3 shows some common loss functions.

Notice that the definitions of cross-entropy functions in Figure 3 assume that target values are encoded as $t_i = 1$ for exactly one i and is zero otherwise (one-hot). To learn from training, we want to minimise the loss function with respect to 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 effect, 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 iterating, we improve the neural network's prediction power a small step at a time. This process is called the *gradient descent* optimisation algorithm, which can be written

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \alpha \nabla E(\mathbf{w}^{\tau}) \tag{7}$$

where τ is the iteration step and α denotes the *learning rate.*

Stochastic gradient descent (SGD) is the method where the gradients of a *single* data point (e.g., the vector $\boldsymbol{x}$) is applied to the weights at the time. In contrast, for batch gradient descent, gradients of multiple data points are applied to the weights at the same time. In short, for batch gradient descent, Equation (7) becomes:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \alpha \frac{1}{N} \sum_{n=1}^{N} \nabla E_n(\mathbf{w}^{\tau}) \tag{8}$$

The intuition of the backpropagation algorithm is that starting from the output error, we want the errors to flow backwards into the network and thereby 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 $(f \circ g)'(x) = f'(g(x))g'(x)$, where $\circ$ means function composition. 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. We shall not present the mathematical foundation for the back-propagation algorithm in detail here, but rather refer to [3, Chapter 5]. We shall however, present the backpropagation algorithm in matrix form [25].

Assume a network of depth d with $\boldsymbol{\delta}^{(l)}$, $W^{(l)}$, $B^{(l)}$, and $\boldsymbol{z}^{(l)}$ denoting the errors, weights, biases, and activations of the l'th layer, respectively. Moreover, let the input $\mathbf{x}$ into the network be $\boldsymbol{z}^{(0)}$. The backpropagation algorithm can then be expressed as follows:

$$\boldsymbol{z}^{(l)} = \sigma^{(l)}(W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}) \tag{9}$$

$$\boldsymbol{\delta}^{(d)} = \frac{\partial E}{\partial \boldsymbol{z}^{(\boldsymbol{d})}} \bullet \sigma'^{(d)}(W^{(d)}\boldsymbol{z}^{(d-1)} + B^{(l)}) \tag{10}$$

$$\boldsymbol{\delta}^{(\boldsymbol{l})} = (W^{(l+1)})^T \boldsymbol{\delta}^{(l+1)} \bullet \sigma'^{(l)}(W^{(l)}\boldsymbol{z}^{(l-1)} + B^{(l)}) \tag{11}$$

$$\frac{\partial E}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{z}^{(l-1)})^T \tag{12}$$

$$\frac{\partial E}{\partial B^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{13}$$

Here $\bullet$ denotes the element-wise product, also called the Hadamard product and $(\cdot)^T$ denotes transposition.

3 Futhark

Futhark is a small high-level, purely functional array language from the ML-family, which is designed to generate efficient data-parallel code. The Futhark compiler can generate GPU code via OpenCL or CUDA, although the language itself is hardware-independent. A deep knowledge of Futhark is not required to understand this paper, but a few notational details will be clarified below. As most languages from the ML-family, Futhark supports parametric polymorphism via type parameters, which are written as a name preceded by an apostrophe. Here is a parametric type abbreviation and a polymorphic identity function:

```
type vector 't = []t        -- array of 't's
let id 't (x: t): t = x
```

Futhark also supports *size parameters*, which can be used to impose constraints on the sizes of the arrays accepted by a function, for example that vector addition requires arrays of identical sizes:

```
let vadd [n] (xs:[n]f32) (ys:[n]f32) :[n]f32 = ...
```

Size parameters are not passed explicitly when a function is applied, but inferred from normal parameters. Size constraints are checked dynamically. Futhark supports higher-order functions with some restrictions [16], and also higher-order modules. The latter is used in the deep learning library to abstract over the scalar type—in practice, whether we use single-precision `f32` or double-precision `f64` scalars.

Futhark supports regular nested parallelism (i.e., nested parallelism where all inner parallel constructs do the same amount of work) through a process called *moderate flattening*, which translates regular nested parallelism into flat parallelism suitable for executing on a GPU [14]. Moreover, Futhark sometimes generates multiple code versions of the same nested parallel constructs through a process called *incremental flattening*, which leads to code that, at runtime, selects a code version based on possibly auto-tuned parameters [15]. Futhark does not, like NESL [4], support irregular parallelism, but exposes a number of higher-order library functions for supporting irregular nested parallel patterns [8].

4 Representing Networks

Recall from Section 2 that one aspect of a neural network is that two layers can be combined by letting the output of the first layer be the input to the next one and that we can express this aspect of a neural network as a function that takes some weights and input and returns some output. From the derivation of the backpropagation algorithm [3, Chapter 5], we have that errors from the output layer are passed back through the network, where each layer passes errors to the previous one. Using these observations, we can view a neural network as a pair of two functions. That is, for a network of depth n, we can write $f_n(f_{n-1}(\cdots(f_1(\cdot)))$ for the forward pass and $b_1(b_2(\cdots(b_n(\cdot))))$ for the backward pass. This simple insight is the main idea behind the technique, for which neural networks are represented as the composition of pairs 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 additional information for the idea to be implemented efficiently. We first define a couple of parameterised type abbreviations for specifying forward functions and backwards functions, respectively:

```
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)
```

Intuitively, a function of a particular instance of the `forwards` type takes a boolean, some weights, and input and returns a pair of a cache and the output from the network. The cache[1] stores intermediate results, so that, during backpropagation, values need not be recomputed eagerly. The boolean argument is there to indicate if the function is called during training; if it is not, the function just returns an empty cache.

A function of a particular instance of the `backwards` type takes 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 following layer. The returned value is a pair of the updated weights and errors to be backpropagated further down the network. The function `u` is provided by the particular optimiser used (e.g., gradient descent), which enables the handling of gradients in different ways. Other types of optimisers can use different instantiations of the type `u`. The initial boolean argument is there to indicate if it is the first layer of the network. In this case, we do not need to calculate and backpropagate errors. This is a minor optimisation, but can give a performance increase on longer training passes.

In an abstract sense, a neural network (or an individual layer of a neural network) is a record holding a forwarding function, a backpropagation function, and a representation of the network weights. Here is how it can be specified in Futhark as a parameterised type abbreviation `NN`:

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

We shall later see how we can allow for networks, given as instances of the above type, to be composed. Notice, however, that the specification allows for each layer implementation to use its own concrete representations of caches and weights. However, how one layer chooses its concrete types will affect whether the layer can be combined with other layer types, which in some cases will require a utility layer.

For composing networks, we shall be more specific about the abstract type `u`. We first define the following two parameterised type abbreviations:

```
type std_weights 't = ([][]t, []t)
type apply_grad 't = std_weights t → std_weights t
                     → std_weights t
```

Because optimisers operate on weights and gradients, the concrete types of these concepts need to be known to the optimisers. As a consequence, `apply_grad` is defined as a transparent type. Layers that do not use this weight representation needs to reshape their weights and gradients before applying the function. Most optimisers update gradients and weights in bulk operations (e.g., all gradients gets the same learning rate applied), and therefore reshaping will not affect the update. However, if an optimiser does not treat gradients in bulk, then this design may be non-optimal.

Here is the type of the function `connect_layers`, which takes two networks and combines them into one:

```
type pair 'x 'y -- abstract
val connect_layers 'w1 'w2 'i 'o1 'o2
                   'c1 'c2 'e1 'e2 'e :
  NN i1 w1 o1 c1 e e1 (apply_grad t) →
  NN o1 w2 o2 c2 e2 e (apply_grad t) →
  NN i1 (pair w1 w2) o2 (pair c1 c2) e2 e1
     (apply_grad t)
```

Notice that the representation of how caches and weight structures are composed is kept abstract using type abstraction at the modules level.

The implementation of the function `connect_layers` is given in Figure 4. Forward functions are combined in a forward fashion whereas backpropagation functions are combined in a backwards fashion. As mentioned already, some type restrictions apply 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 Figure 4. That is, the output type from the first network, `o1` on line 5 is the same as the input to the second network on line 7. Weights and caches are also combined into tuples, which become more and more nested as the network depth increases. This representation completely avoids the use of 1D arrays and indexing, but instead inlines functions and makes use of tuples to represent a network. The representation allows layer information to be stored in their native form along with layer-specific auxiliary information. Additionally, extending the library with a new layer type, is only limited to the concrete layer implementation itself and does not affect other parts.

[1] The term "cache" is used with the meaning of "stored away for future use", and has no relation to memory caches as a computer-architectural concept.

```
type pair 'x 'y = (x,y)
let connect_layers 'w1 'w2 'i 'o1 'o2
                   'c1 'c2 'e1 'e2 'e
  ({forward=f1, backward=b1, weights=ws1}
   : NN i1 w1 o1 c1 e e1 (apply_grad t))
  ({forward=f2, backward=b2, weights=ws2}
   : NN o1 w2 o2 c2 e2 e (apply_grad t))
  : NN i1 (pair w1 w2) o2 (pair c1 c2) e2 e1
       (apply_grad t) =
     {forward = \is_training (w1,w2) input →
        let (c1,res)  = f1 is_training w1 input
        let (c2,res2) = f2 is_training w2 res
        in ((c1,c2), res2),
      backward = \_ u (w1,w2) (c1,c2) error →
        let (err2,w2') = b2 false u w2 c2 error
        let (err1,w1') = b1 true u w1 c1 err2
        in (err1, (w1',w2')),
      weights = (ws1, ws2)}
```

Figure 4. Function for combining two networks.

4.1 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 Futhark's higher-order modules [7]. As a neural network consists of many components, we naturally separate each of these components into separate modules. Weight initialisation functions, activation functions, and loss functions are implemented within their own modules, respectively. Layer implementations are kept abstract and each concrete layer implementation must match an abstract module type `layer_type`.

All concrete layer implementations are collected in the module `layers`, which allows us to access all layer functionality through a single module. Optimiser implementations follow the same structure as layers. As layers and optimisers in general have more complex implementations, an additional level of abstraction is used.

Finally, the `neural_network` module contains a number of generic utility functions, including the `connect_layers` function (or >-) and functions for calculating the loss or accuracy of a network. The next sections will provide details on the implementations, starting with activation functions.

4.2 Activation Functions

Recall that an activation function has the characteristic of being differentiable, so that we can use its derivative during the backward pass to calculate the gradient. Therefore, an activation function is represented as a pair, containing the function itself and its derivative. That is, we can define its abstract type as follows:

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

Activation functions provided in this implementation all use a 1D 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 users can define their own pair of functions and use them should the library not contain them. This choice ensures a flexible system, where the user is not limited to the library implementation. Supported activation functions include `sigmoid`, `tanh`, `ReLU`, `identity`, and `softmax`, although `softmax` cannot be used during training in a layer. The implementations are straightforward and follow the outline in Section 2. Activation functions can be accessed through the `neural_network` module through simple wrappers, following the same interface as TensorFlow.

4.3 Loss Functions

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

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

Again, this abstract type allows users to define their own loss functions and in this implementation, a loss function's concrete type of `'o` is a 1D array. Following the definitions in Figure 3, supported loss functions include `cross_entropy`, `cross_entropy_with_softmax`, and `sum_of_squares`.

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

4.4 Optimisers

Optimisers may implement the backpropagation algorithm by calling the forward and backward passes on a network and apply the gradients through its own implementation of the abstract function `apply_grad`. The library is open in the sense that end users may implement new optimisers with the constraint that they must match the optimiser module type, which is shown in Figure 5. Here `learning_rate` is allowed to be a function type, which allows an optimiser implementation to adapt the learning rate to different training steps with a user defined function.

A call `train nn lrate inp lbls bsz lfun` returns the network `nn` modified based on the training data, available in the input and label arrays (`inp` and `lbls`), and based on the supplied learning rate `lrate`, the batch size `bsz`, and the loss function `lfun`. 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 types of the input data and labels must be arrays, 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 optimiser the implementation provides is a stochastic gradient descent optimiser, which is defined in the `gradient_descent` module.

```
module type optimizer_type = {
  type t
  type learning_rate
  val train 'i 'w 'g 'e2 'o :
       NN ([]i) w ([]o) g ([]o) e2 (apply_grad t)
       → learning_rate → (input_data:[]i)
       → (labels:[]o) → (batch_size:i32)
       → loss_func o t
       → NN ([]i) w ([]o) g ([]o) e2 (apply_grad t)
}
```

Figure 5. Abstract optimiser module.

```
let train [n] 'w 'g 'o 'e2 'i
     ({forward=f, backward=b, weights=w}
      : NN ([]i) w ([]o) g ([]o) e2 (apply_grad t))
     (alpha:learning_rate) (input:[n]i)
     (labels:[n]o) (batch_sz: i32)
     ({f=_, fd=loss'}:loss_func o t) =
  let updater = apply_grad_gd alpha batch_sz
  let (w',_) =
    loop (w,i) = (w,0) while i < length input do
      let input'          = input[i:i+batch_sz]
      let label'          = labels[i:i+batch_sz]
      let (cache, output) = f true w input'
      let error   = map2 loss' output label'
      let (_, w') = b false updater w cache error
      in (w', i + batch_sz)
  in {forward=f, backward=b, weights=w'}
```

Figure 6. Train function in the `gradient_descent` module.

```
module type layer_type = {
  type t
  type input_params
  type activations
  type input
  type output
  type weights      -- Initialise a layer given
  type err_in       -- input parameters, an
  type err_out      -- activation function, and
  type cache        -- a seed.
  val init : input_params → activations → i32 →
             NN input weights output cache
                err_in err_out (apply_grad t)
}
```

Figure 7. The generic layer module type.

Training a network is done by a loop as shown in Figure 6. At each step, the input is forward propagated and the error is calculated (line 16 and 17), which is then backpropagated through the network to get the updated weights (line 18 and 19). The `updater` value is a function that takes the gradients and weights of a layer and performs the update using equation (8). The process is repeated, until all input data has been processed, returning a network with updated weights.

5 Layers

Errors are backpropagated back through the network as described by Equation 11, which we restate here:

$$\delta^{(l)} = \underbrace{(W^{(l+1)})^T \delta^{(l+1)}}_{error'} \bullet \sigma'^{(l)}(W^{(l)} z^{(l-1)} + B^{(l)}) \tag{14}$$

The *error'* part is backpropagated as the error term, as it can be calculated at layer $l+1$, but not at layer l. All layer implementations must follow this convention, so that errors can be backpropagated correctly. The remaining part, $\sigma'^{(l)}(W^{(l)} z^{(l-1)} + B^{(l)})$ is calculated at layer l, where the term $W^{(l)} z^{(l-1)} + B^{(l)}$ is retrieved from the cache. The abstract module type `layer_type` is defined as shown in Figure 7.

A concrete layer implementation must define its own `input`, `output`, `weights`, `err_in`, `err_out`, and `cache` types and it must provide a function that initialises 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 random weight initialisation (dense and convolutional layers). Notice that layer functions are expecting a batch of data points at the time for forward and backward passes, such that the parallelism is optimised.

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

5.1 Dense Layers

Dense layers are initialised with a tuple (m, n) of two integers, each of 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 1D array of length n for the biases, following the same representation as in Section 2. The forward pass is implemented directly using equation (9), 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 (11), (12), and (13) directly.

5.2 Convolutional Layers

Implementing a convolutional layer is not as straightforward as implementing a dense layer. There are many ways convolutions can be implemented efficiently, with each method having different strengths and weaknesses. The need for fast convolutions is evident in that convolutional networks are used in real-time applications, such as self-driving cars for detecting pedestrians, which requires low latency. Thus, the success of a convolutional network is limited by how fast the convolution can be performed [20]. Convolutional operations are compute-expensive operations and H. Kim et. al. [19] show that more than 70 percent 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 an active research area, where the common approach is to reduce the amount of multiplication operations at the expense of additions and/or use of auxiliary memory.

The main idea behind convolutional layers is that data, like images, contain "hidden" information in the spatial structure, which can be utilised when searching for patterns. The input into a convolutional layer is therefore three dimensions, described by height, width and depth, $H \times W \times D$. We will first consider the case of depth equal to one, 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 an ordinary dense neural network, the input is stretched out into a single dimension, resulting in the spatial information being lost. With a convolutional network, however, the spacial information is maintained. In convolutional layers, weights are often called filters,[2] which a layer can have multiples of. These filters are (usually) small square matrices. 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 8 shows an example.

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 [9, ch. 9]. Convolutional layers are only naturally equivariant to shifts, but not to rotation or scaling. The goal is to have each filter adapt to certain characteristics of the input images. For example, one filter should detect edges, another should detect 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:

$$(I \otimes F_f)_{ij} = a_{ij} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I[i+m, j+n] F_f[m,n] \tag{15}$$

The output 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

$$\left(\frac{(n-k)}{s} + 1 \right) \times \left(\frac{(n-k)}{s} + 1 \right) \tag{16}$$

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, the depth dimension of the output from a convolutional layer is only determined by the *number of filters*, N_f, in a convolutional layer and we can write the output dimension as

$$\left(\frac{(n-k)}{s} + 1 \right) \times \left(\frac{(n-k)}{s} + 1 \right) \times N_f \tag{17}$$

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

$$(I \otimes F_f)_{ij} = a_{ij} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c'=0}^{c-1} I[i+m, j+n, c'] F_f[m,n,c'] + b_f \tag{18}$$

for $i, j = 1, .., \frac{(n-k)}{s} + 1$. The operation is then repeated for each filter in F. Notice that there is only one bias value for each filter. That is, 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.

$$z_{ij} = \sigma(a_{ij}) \tag{19}$$

Removing subscripts, the output of a convolutional layer is:

$$Z = \sigma\left(I \otimes F + B\right) \tag{20}$$

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.

5.2.1 Backpropagation

The backpropagation algorithm for the convolutional network is similar to the one for a dense neural network, but with the matrix multiplications replaced by convolutional operations. The full derivation is omitted here; instead we refer to [18] for a full derivation. The equations of the backpropagation algorithm for the convolutional network are:

$$Z^{(l)} = \sigma\left(Z^{(l-1)} \otimes F^{(l)} + B^{(l)}\right) \tag{21}$$

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \otimes \left(F^{(l+1)}\right)^{rot(180^\circ)} \bullet \sigma'\left(Z^{(l-1)} \otimes F^{(l)} + B^{(l)}\right) \tag{22}$$

$$\frac{\partial E}{\partial F^{(l)}} = Z^{(l-1)} \otimes \boldsymbol{\delta}^{(l)} \tag{23}$$

$$\frac{\partial E}{\partial B^{(l)}} = \sum_m \sum_n \boldsymbol{\delta}_{mn}^{(l)} \tag{24}$$

[2] Filters are also called kernels, but in order not to confuse with GPU kernels, the term filter is used here.

[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.

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

Figure 8. A convolutional operation, with input image of size 3x3 and filter size 2x2 with a stride 1, where $\cdot$ denotes the dot-product and f is the vector $[f_{00}\ f_{10}\ f_{01}\ f_{11}]^T$.

Here $\boldsymbol{\delta}^{(l)}$ have the same semantics as in the case of a dense neural network. Notice that in equation (22) each filter is rotated 180 degrees, [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 8 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 9 shows an example of the full convolutional operation, where one can see that the result has the same dimensions as the example in Figure 8, and verify that we have correctly mapped the errors back to its input through the filter, f.

Having defined the backpropagation algorithms, we now show how we can combine a convolutional layer with a dense layer. 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 equation (22) with $(W^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}$ from equation (11) in order to calculate the errors $\boldsymbol{\delta}^{(l)}$ in equation (22). Combining layers in the opposite order follows the same logic, but is uncommon, because spatial information is lost in a fully-connected layer.

The simplest implementation of a convolutional operation follows equation (18) directly, which, however, leads to poor performance. Another approach is to lower the convolution into a matrix multiplication, which can be done, in case of images, by transforming image data into matrices using a function called `im2col`, which arranges the image slices into a column matrix. This approach is 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 utilise the local memory that has low latency and high bandwidth. 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 10 shows how the transformation duplicates the data by a factor of 2.25. This factor increases linearly with the image size.

The Fast Fourier Transformation (FFT) along with the convolution theorem is another popular approach [28], but this approach performs well only 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 [19, p. 59]. For filters of smaller sizes (5 and below), the Winograd minimal filtering (WMF) algorithm [20] 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 these matrices need to be pre-computed, each filter size requires a special case and therefore the WMF algorithm is applicable only to a small set of filter sizes. The two latter methods uses an excess amount of auxiliary memory to hold intermediate results.

Determining which algorithm is fastest cannot always be predetermined, as it depends on batch size, stride, and so on. For example, TensorFlow executes all available algorithms to identify which is best [19, p. 60]. Also the NVIDIA cudNN library [23] implements all three approaches. Here we shall be concerned only with implementing the GEMM approach, which can be applied in the general case and performs reasonably well for small batch sizes, but scales poorly as the batch sizes increases, because the transformation to matrix form becomes too expensive [19].

The cudNN library solves this issue by reading fixed sized submatrices 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 succeeding submatrices onto the on-chip memory, essentially hiding the memory latency associated with the data transfer. The effect is that the computation is limited only by the time it takes to perform the arithmetic, while limiting the auxiliary memory usage [5, p.5]. The cudNN library provides both options in their API (i.e., to form the matrix explicitly or implicitly).

The implicit GEMM approach is not possible in Futhark, and the closest approach is to perform a loop, which iterates 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

[5] This rotation is a consequence of the derivation and is necessary regardless of whether a cross-correlation or a convolutional operation is used.

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

Figure 9. Example of a full convolution operation by padding the errors $\boldsymbol{\delta}$ with zeroes and applying the flipped filter, $g = [f_{11}\ f_{01}\ f_{10}\ f_{00}]^T$ from Figure 8.

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

Figure 10. Example of *im2col* operation on a 4 × 4 image with a 2 × 2 filter and a stride of 1.

$$\begin{bmatrix} f_{000} & f_{010} \\ f_{100} & f_{110} \end{bmatrix} \begin{bmatrix} f_{001} & f_{011} \\ f_{101} & f_{111} \end{bmatrix} \Rightarrow \begin{bmatrix} f_{000} & f_{010} & f_{100} & f_{110} & f_{001} & f_{011} & f_{101} & f_{111} \end{bmatrix}$$

Figure 11. A single filter of size 2 × 2 × 2 representation in a convolutional layer.

used during training, we have implemented only the explicit GEMM approach, which transforms the input image into matrix form explicitly.

5.2.2 The Implementation

The convolutional layer implementation takes four parameters, `number_of_filters`, `filter_size`, `stride`, and `input_depth`. The latter is needed to initialise the proper filter depth to match the input depth. Currently only square filters are possible, although it will not be difficult to support also non-square filter sizes. 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.

5.2.3 The 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.

The cache consists of the image matrix, which helps avoiding performing the transformation again during the backward pass. We need, however, to store the original image dimensions. We also cache, in a suitable format, the result of the convolutional operation after applying the bias, which means that we do not have to reshape it when we perform the Hadamard product during backpropagation.

5.2.4 The Backward Pass

The backward pass is based on equations (22) and (23). Having calculated $\boldsymbol{\delta}^{(l)}$, we flatten each of the layers and perform a matrix multiplication for the convolutional operation in (23) with the image matrix from the cache. For backpropagation of the errors to the previous layer, we flip the filters first, which is done by slicing into the filter vector and reverse each filter separately using the Futhark function `reverse`. Recall that equation (22) is a full convolution and we need to pad $\boldsymbol{\delta}^{(l)}$ before transforming it into matrix form. From that representation, we perform a matrix multiplication again to perform the convolutional operation.

5.3 Max Pooling

A max pooling layer is initialised with a tuple (wm, wn) of two integers, which represent 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 a 1D array and stored along with the maximum value. Finally, 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 implemented using the offsets from the cache along with the `scatter` function. The original image size is first created as a 1D 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.

6 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.

Layer type	Filters/ neurons	Filter/ window sz	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

Figure 12. CNN with input dimension of 1 × 28 × 28.

```
module dl = deep_learning f32
open dl.layers dl.nn
let (>-)      = connect_layers
let seed      = 1
let conv1     = conv2d (32, 5, 1, 1) relu seed
let max_pool1 = max_pooling2d (2,2)
let conv2     = conv2d (64, 3, 1, 32) relu seed
let max_pool2 = max_pooling2d (2,2)
let fc        = dense (1600, 1024) identity seed
let output    = dense (1024, 10) identity seed
let nn  = conv1 >- max_pool1 >- conv2 >-
          max_pool2 >- flatten >- fc >- output
```

Figure 13. Example CNN built with the library.

- 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,[6] which is given to the accuracy function as argument. The function returns the degree of accuracy of the network.
- loss: The function calculates and returns the accumulated loss on a network given some input data, labels, and a loss function.

These functions are defined in the neural_network module.

7 Putting it All Together

The implementation defines a deep_learning module, which combines the modules layers, optimizers, loss, and neural_network so that we only have to instantiate a single module. Having defined all of these components, we can now see how one can build the convolutional network defined in Figure 12. The network is designed to be trained on the MNIST data-set.

Figure 13 shows how we can put together such a network using the library. Notice that we first define each of our layers separately. We then build our network using the connect_layers function. Having defined our network, we can train it and calculate the accuracy, as shown in Figure 14.

```
let main [m] (inp: [m][]dl.t)
             (labs: [m][]dl.t) =
  let input = map (\i → [unflatten 28 28 i]) inp
  let train = 64000
  let validation = 10000
  let batch_size = 128
  let alpha = 0.1         -- learning rate
  let nn' = dl.train.gradient_descent nn alpha
              input[:train] labs[:train]
              batch_size
              dl.loss.softmax_cross_entropy
  in dl.nn.accuracy nn'
       input[train:train+validation]
       labs[train:train+validation]
       dl.nn.softmax dl.nn.argmax
```

Figure 14. Example of training a network with the library.

Table 1. Benchmark results for the dense neural network.

	Batch size			
Library	16	32	64	128
TensorFlow	5072ms	2831ms	1614ms	970ms
Futhark	862ms	448ms	215ms	123ms
Relative speedup	5.88 ×	6.32 ×	7.51 ×	7.88 ×

Table 2. Benchmark results for the convolutional network.

	Batch size			
Library	16	32	64	128
TensorFlow	7158ms	3906ms	2118ms	1358ms
Futhark	4533ms	3091ms	2434ms	2135ms
Relative speedup	1.57×	1.26 ×	0.87 ×	0.63 ×

The program first trains the network on 64,000 data points with a batch size of 128 and a learning rate of 0.1. The accuracy of the trained network is then calculated on 10,000 separate data points.

On a Unix-like system, the program can be compiled with the futhark-opencl compiler and executed as follows:

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

8 Empirical Evaluation

While the contribution of this paper is not primarily performance, we wish to demonstrate that our presented design can perform well. Therefore, this section compares the performance of our implementation with Aymeric Damien’s

[6] Recall from Section 2 that the outputs are interpreted as probabilities, and therefore, the output activation with the highest probability will be the prediction of the input.

TensorFlow examples,[7] *neuralnetwork.py* (a *multilayer perceptron*—MLP) and *convolutionalnetwork.py* (a convolutional network), with some minor modifications. Specifically, we removed the dropout layer in the convolutional network and changed the optimiser to gradient descent. We then ported the two networks to Futhark. As training data, we use the classic MNIST dataset for digit recognition, and both networks use the loss function *cross entropy with softmax*. While we do not claim that TensorFlow is the fastest neural network implementation available (performance is lost to interpretive overhead by going through Python), it is widely used, and so serves as a good point of comparison to the level of performance used in practice. The implementations are available at https://github.com/diku-dk/futhark-fhpnc19, including instructions (and scripts) for downloading the MNIST data set and for executing and benchmarking the code.

We run our experiments on a single NVIDIA RTX 2080 Ti GPU with CUDA 10 and TensorFlow 1.13.1. Each training run is done with a single traversal ("epoch") of 54,000 data points, with four different batch sizes: 128, 64, 32 and 16. There is no universally best batch size. In practice, model developers run their models several times to find the best value. The batch size is also limited by hardware memory combined with the size of the network architecture. Thus, a benchmark should not be limited to only one batch size, but rather to a range of batch sizes, to provide a comprehensive performance overview. The training is done ten times (plus a warmup run), from which we report the average runtime. The runtime results are shown in Table 1 and 2.

We see that Futhark significantly outperforms TensorFlow on the MLP. We believe this is due to the individual layers being relatively simple, and thus easy for the Futhark compiler to optimise. In contrast, TensorFlow outperforms Futhark on the convolutional network. This is because of the convolutional layers, which in our implementation are implemented with the `im2col` algorithm. The Futhark compiler does a decent job of optimising the resulting code, but TensorFlow makes use of NVIDIAs heavily hand-optimised and proprietary cuDNN library [5], whose implementation—and perhaps choice of algorithm—for convolution far outperforms the ones generated by Futhark.

9 Related Work

The dataflow programming model used by neural network libraries, such as TensorFlow [1] and Theano [2], is similar to pure functional programming, but is usually exposed through object-oriented or procedural interfaces. Such library approaches suffers from the friction between the low-level language used for implementing layer primitives, and the high-level language (often Python) used to describe the network topology. These libraries can be made available in a functional language, as seen for example in Hasktorch,[8] but the fundamental friction is still present.

Also related to the present work is the work on implementing a convolutional neural network in APL [29], a dynamically typed functional array language. Whereas the APL implementation itself is not shown to run very efficiently, it is used as a specification for a hand-compilation into SaC [27], which shows good performance and is demonstrated to run faster than TensorFlow on CPU. Compared to our work, however, their solution implements the forward pass independent from the backward pass. An interesting experiment would be to investigate whether previous work on compiling APL to Futhark [10] could provide decent performance for an APL implementation of a convolutional network.

Polyhedral optimisations provide a common technique for optimising loops, and have also been applied to optimising TensorFlow graphs [24]. In principle, techniques that directly take into account the semantics of neural networks should be able to perform better optimisations than a compiler for an ordinary parallel language, for example, by exploiting nondeterminism, which the Futhark compiler will not do, as Futhark has deterministic semantics.

10 Conclusion

We have shown a functional design for a neural network library structured as the composition of functions. While the design is language-agnostic, our implementation is in Futhark. Our benchmarks suggest that for a multiplayer perceptron, our implementation is capable of competing with existing libraries like TensorFlow. For convolutional networks there is still some work to do to achieve performance parity, but we are within a factor of two in all cases. Considering that our library is algorithmically not very sophisticated, and that we are competing with expertly hand-written GPU code, our results are promising.

In the future, we would like to extend our approach to handle more complex network topologies, such as recurrent neural networks. Further, we would like to extend Futhark's support for size parameters to statically verify that the output of a layer is compatible with the input of the following layer (such failures are currently not detected until run-time).

Acknowledgments

This work has been supported by the Independent Research Fund Denmark grant under the research project *FUTHARK: Functional Technology for High-performance Architectures*.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser,

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

[8] https://github.com/hasktorch/hasktorch

Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. white paper.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU Math Compiler in Python. In *Procs. of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 3 – 10.

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

[4] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. ACM, New York, NY, USA, 213–225. https://doi.org/10.1145/232627.232650

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014).

[6] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[7] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin E. Oancea. 2018. Static Interpretation of Higher-order Modules in Futhark: Functional GPU Programming in the Large. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 97 (July 2018), 30 pages.

[8] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. 2019. Data-parallel Flattening by Expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/3315454.3329955

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

[10] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing (FHPC'16)*. ACM, New York, NY, USA, 38–43.

[11] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. 2014. Size Slicing: A Hybrid Approach to Size Inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '14)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/2636228.2636238

[12] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 17–24.

[13] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 Graph-reduction Approach to Fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '13)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2502323.2502328

[14] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[15] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

[16] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-performance defunctionalization in Futhark. In *Symposium on Trends in Functional Programming (TFP'18)*. Springer-Verlag.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014). arXiv:1408.5093 http://arxiv.org/abs/1408.5093

[18] Jefkine Kafunah. 2016. *Backpropagation in Convolutional Neural Networks*. Stanford. https://canvas.stanford.edu/files/1041875/download?download_frd=1&verifier=tFv4Jc7bCezxJg9rG2yhEKEERi70zJ3ScmFbNlbN.

[19] H. Kim, H. Nam, W. Jung, and J. Lee. 2017. Performance analysis of CNN frameworks for GPUs. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 55–64. https://doi.org/10.1109/ISPASS.2017.7975270

[20] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308

[21] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled! TensorFlow Development Summit 2017.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* 1, 4 (Dec. 1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[23] NVIDIA. 2018. Deep Learning SDK documentation. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[24] Benoît Pradelle, Benoît Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. 2019. Polyhedral Optimization of TensorFlow Computation Graphs. In *Programming and Performance Visualization Tools*, Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz (Eds.). Springer International Publishing, Cham, 74–89.

[25] Sudeep Raja. 2017. A Derivation of Backpropagation in Matrix Form. https://sudeepraja.github.io/Neural/.

[26] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (Oct. 1986), 533–. http://dx.doi.org/10.1038/323533a0

[27] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[28] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. *CoRR* abs/1412.7580 (2014).

[29] Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. ACM, New York, NY, USA, 69–79. https://doi.org/10.1145/3315454.3329960

[30] Dong Yu, Kaisheng Yao, and Yu Zhang. 2015. The Computational Network Toolkit. *IEEE Signal Processing Magazine* (November 2015), 123–126.