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 back-propagation) 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:

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 combinator (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 $28 \times 28$ 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.
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.

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...
With multiclass problems, it is common to transform each
where
which is defined by
will then repeat the parallel computation of network changes
until the network stabilises. The process is parameterised
over a learning rate (often named \( \alpha \)), a batch size, and, of
course, the size of the training data set.

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 \( x_1, x_2, \ldots, x_D \) be the input into the
network, we can write the first layer calculation as:

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

where \( j \in \{1 \ldots M\} \). 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

\[
z_j = \sigma(a_j)
\]

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

\[
a_k = \sum_{j=1}^{M} w_{kj} z_j + b_k^{(2)} \quad \text{for } k = 1, \ldots, K
\]

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

\[
\text{softmax}(a_k) = y_k = \frac{e^{a_k}}{\sum_{i=1}^{K} e^{a_i}} \quad \text{for } k = 1 \ldots K
\]

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

<table>
<thead>
<tr>
<th>Loss function</th>
<th>( E(w) )</th>
<th>( \frac{\partial E(w)}{\partial w_{kj}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-entropy (CE)</td>
<td>( - \sum_{k=1}^{K} (t_k \ln y_k) )</td>
<td>( -\frac{t_k}{y_k} )</td>
</tr>
<tr>
<td>CE with softmax</td>
<td>( - \sum_{k=1}^{K} (t_k \ln \frac{y_k}{\sum_{k=1}^{K} y_k}) )</td>
<td>( y_k - t_k )</td>
</tr>
<tr>
<td>Sum of squares</td>
<td>( \frac{1}{2} \sum_{k=1}^{K} (y_k - t_k)^2 )</td>
<td>( y_k - t_k )</td>
</tr>
</tbody>
</table>

Figure 3. Common loss functions and their derivatives.

\( y_k(x, W) = \text{softmax} \left( \sum_{i=1}^{M} w_{ki}^{(2)} \sigma \left( \sum_{j=1}^{D} w_{ji}^{(1)} x_i + b_j^{(1)} \right) + b_{k}^{(2)} \right) \) \((5)\)

Here we have grouped the weights and bias parameters into
\( W \). Thus a dense neural network is a nonlinear function
from a set of input variables \( x \) to a set of output variables \( y \),
controlled by a set of adjustable parameters, \( W \). 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

\[
y(x, W) = \text{softmax} \left( W^{(2)} \sigma \left( W^{(1)} x + B^{(1)} \right) + B^{(2)} \right)
\]

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 \( \text{tanh}, \text{ReLU} \) (i.e., rectified linear unit), \( \text{sigmoid} \), and
\( \text{softmax} \). As an example, the \( \text{sigmoid} \) function is defined as
\( \text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \) with the derivative
\( \text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \).

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 \( y_k(x, W) \) 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 \( E(W) \). 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
with the weights \( W \). Letting \( \nabla E(W) \) 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
\( \nabla E(w) = 0 \), as our loss function is a smooth continuous function of \( w \). To achieve this effect, we want to determine \( \nabla E(w) \) 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
\[
 w^{r+1} = w^r - \alpha \nabla E(w^r)
\] (7)
where \( r \) is the iteration step and \( \alpha \) denotes the learning rate.

Stochastic gradient descent (SGD) is the method where the gradients of a single data point (e.g., the vector \( x \)) 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:
\[
 w^{r+1} = w^r - \frac{1}{N} \sum_{n=1}^{N} \nabla E_n(w^r)
\] (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 \( \nabla E(w) \), 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 backpropagation 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 \( \delta^{(l)} \), \( W^{(l)} \), \( B^{(l)} \), and \( z^{(l)} \) denoting the errors, weights, biases, and activations of the \( l \)th layer, respectively. Moreover, let the input \( x \) into the network be \( z^{(0)} \). The backpropagation algorithm can then be expressed as follows:
\[
z^{(l)} = \sigma^{(l)}(W^{(l)}z^{(l-1)} + B^{(l)})
\] (9)
\[
\delta^{(d)} = \frac{\partial E}{\partial z^{(d)}} \cdot \sigma''^{(d)}(W^{(d)}z^{(d-1)} + B^{(d)})
\] (10)
\[
\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \cdot \sigma''^{(l)}(W^{(l)}z^{(l-1)} + B^{(l)})
\] (11)
\[
\frac{\partial E}{\partial W^{(l)}} = \delta^{(l)}(z^{(l-1)})^T
\] (12)
\[
\frac{\partial E}{\partial B^{(l)}} = \delta^{(l)}
\] (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 \( f_n(f_{n-1}(\cdots(f_1(x)))) \) for the forward pass and \( b_1(b_2(\cdots(b_n(\cdots))) \) 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:

```futhark

**type** forwards 'input 'w 'output 'cache =
bool \rightarrow w \rightarrow input \rightarrow (cache, output)

**type** backwards 'w 'cache 'err_in 'err_out 'u =
bool \rightarrow u \rightarrow w \rightarrow cache \rightarrow err_in \rightarrow (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:

```futhark

**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:

```futhark

**type** std_weights 't = ([[][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][][[1]}
The implementation makes use of Futhark’s higher-order abstract type as follows: the activation function is represented as a pair, containing the backward pass to calculate the gradient. Therefore, an activation function should also be differentiable, so that we can use its derivative 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 5, 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 lr inp lba bsz lfun` returns the network `nn` modified based on the training data, available in the input and label arrays (`inp` and `lba`), and based on the supplied learning rate `lr`, 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.
At each step, the input is forward propagated and the error term can be backpropagated correctly. The remaining part, $\sigma^{(l)}(W^{(l)}z^{(l-1)} + B^{(l)})$ is calculated at layer $l$, where the term $W^{(l)}z^{(l-1)}$ 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 parallelism is optimised.

Figure 5. Abstract optimiser module.

Figure 6. Train function in the gradient_descent module.

Figure 7. The generic layer module type.
We will first consider the case of depth equal to one, thus 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
\[
\left( \frac{(n - k)}{s} + 1 \right) \times \left( \frac{(n - k)}{s} + 1 \right)
\]
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
\[
\left( \frac{(n - k)}{s} + 1 \right) \times \left( \frac{(n - k)}{s} + 1 \right) \times N_f
\]
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
\[
(I \otimes F_f)_{ij} = a_{ij} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I[i+m, j+n, c']F_f[m, n, c'] + b_f
\]
for \( i, j = 1, \ldots, \frac{(n-k)}{s} + 1 \). 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.
\[
z_{ij} = \sigma(a_{ij})
\]
Removing subscripts, the output of a convolutional layer is:
\[
Z = \sigma(I \otimes F + B)
\]
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:
\[
Z^{(l)} = \sigma \left( Z^{(l-1)} \otimes F^{(l)} + B^{(l)} \right)
\]
\[
\delta^{(l)} = \delta^{(l+1)} \otimes \left( F^{(l+1)} \right)^{rot(180)} \bullet \sigma' \left( Z^{(l-1)} \otimes F^{(l)} + B^{(l)} \right)
\]
\[
\frac{\partial E}{\partial F^{(l)}} = Z^{(l-1)} \otimes \delta^{(l)}
\]
\[
\frac{\partial E}{\partial B^{(l)}} = \sum_m \sum_n \delta^{(l)}_{mn}
\]
\(^{4}\)The 2D refers to the dimensions the filter is slid in and not the dimensions of the filter nor the input image.
with a stride of 1, Figure 10 shows how the transformation 4
5
linearly with the image size.

duplicates the data by a factor of 2.25. This factor increases
downside is that it uses a lot of auxiliary memory and also
operation as a matrix multiplication. Matrix multiplication
filters as matrices as well, we can perform the convolutional
function called
images, by transforming image data into matrices using a
performance. Another approach is to lower the convolution
order follows the same logic, but is uncommon, because
errors to substitute
δ
errors back to the input with the corresponding weights, e.g.
see that the result has the same dimensions as the example
example of the full convolutional operation, where one can
perform a normal convolutional operation. Figure 9 shows an
δ
out-of-bounds. This is in practice solved by adding k – 1 zero
padding around δ, 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 δ(l+1) ∗ (F(l+1)) rot(180°) in equation (22) with
(W(l+1))T δ(l+1) from equation (11) in order to calculate the
errors δ(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 con-
volution 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 op-
eration. Because these matrices need to be pre-computed, each
filter size requires a special case and therefore the WMF al-
gorithm 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
determined, 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 rea-
sonably 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 sub-
matrices 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 ap-
proach does not hide the memory latency and seems like a
half solution to the problem, as different systems have differ-
ent 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

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

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

Here δ(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 δ, 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.

\(^5\)This rotation is a consequence of the derivation and is necessary regardless
of whether a cross-correlation or a convolutional operation is used.
The forward pass is done by using the im2col function, which transforms the input image 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 \( \delta^{(l)} \), 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 \( \delta^{(l)} \) 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.
The network is designed to be trained on the MNIST data-set. The implementation defines a neural_network module.

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.

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:

```bash
$ 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.

<table>
<thead>
<tr>
<th>Library</th>
<th>Batch size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>16</td>
</tr>
<tr>
<td>Futhark</td>
<td>852ms</td>
</tr>
<tr>
<td>Relative speedup</td>
<td>3.29 ×</td>
</tr>
</tbody>
</table>

### Table 1. Benchmark results for the dense neural network.

<table>
<thead>
<tr>
<th>Library</th>
<th>Batch size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>7158ms</td>
</tr>
<tr>
<td>Futhark</td>
<td>4535ms</td>
</tr>
<tr>
<td>Relative speedup</td>
<td>1.57 ×</td>
</tr>
</tbody>
</table>

### Table 2. Benchmark results for the convolutional network.
TensorFlow examples, 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 (“epoch7”) 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 1m2c01 algorithm. The Futhark compiler does a decent job of optimising the resulting code, but TensorFlow makes use of NVIDIA’s 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, 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-compiled 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 multilayer 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


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

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