Design and Implementation of DeepDSL: A DSL for Deep Learning

Tian Zhao
University of Wisconsin – Milwaukee
tzhao@uwm.edu

Xiaobing Huang
University of Wisconsin – Milwaukee
xiaobing@uwm.edu

Abstract

Deep Learning (DL) has found great success in well-diversified areas such as machine vision, speech recognition, and multimedia understanding. However, the state-of-the-art tools (e.g. Caffe, TensorFlow, and CNTK), are programming libraries with many dependencies and implemented in languages such as C++ that need to be compiled to a specific runtime environment and require users to install the entire tool libraries for training or inference, which limits the portability of DL applications.

In this work, we introduce *DeepDSL*, a domain specific language (DSL) embedded in Scala, that compiles DL networks encoded with DeepDSL to efficient, compact, and portable Java source programs for DL training and inference. DeepDSL represents DL networks as abstract tensor functions, performs symbolic gradient derivations to generate Intermediate Representation (IR), optimizes the IR expressions, and translates the optimized IR expressions to Java code that runs on GPU without additional dependencies other than the necessary GPU libraries and the related invocation interfaces: a small set of *JNI* (Java Native Interface) wrappers. Our experiments show DeepDSL outperforms existing tools in several benchmark programs adopted from the current mainstream Deep Neural Networks (*DNN*s).

1 Introduction

Multimedia has become the most valuable resource for insights and information [9]. In recent years, a new set of machine learning algorithms denoted as *Deep Learning*(DL) [29], which aims at learning multiple levels of representation and abstraction that help infer knowledge from multimedia data such as text, audio, image, and video, is making astonishing gains in areas such as machine vision, speech recognition, and multimedia analysis.

Deep learning leverages neural network of many layers to perform learning tasks such as classification. The learning process is iterative where a typical iteration has a forward inference step to make a prediction using the training data and a backward update step to adjust each network parameter using the gradient of the *loss* with respect to the parameter, where the loss is a scalar that measures the difference between the prediction and actual classification of the training data. Users specify the structure of the DL network that forms the forward inference step while the gradient update step is derived from the DL network directly or through symbolic derivation from the gradient expressions.

Deep learning is very computationally intensive and most solutions leverage parallel computing platforms such as GPUs for better performance. For instance, convolutional neural network (CNN), a popular type of DNN, applies multiple convolution operations (among others) to training data such as images, where the image data are represented as 4-dimensional arrays (or tensors) and the dimensions represent the number of images in a training batch, the number of channels, and the height and width of each image. Among the main challenges of implementing DL networks are runtime and memory efficiency. The runtime efficiency is important since training a DL network requires many iterations and inefficient solution can take days to complete. Memory efficiency is important since operations like convolution can use a large portion of the GPU memory where memory inefficient solutions can exhaust the GPU memory and cause the training program to crash.

Optimization is critical to the efficiency of DL applications, which can be implemented at high level or low level. High-level optimization includes steps such as the simplification of computation (to eliminate redundancies and reuse intermediate results) and computation steps reordering to reduce peak memory usage. Low-level optimization improves the efficiency of individual operations such as convolution and matrix multiplication. For low-level optimization, there are GPU libraries such as Cuda that supports high-performance linear algebraic compu-

tation and Cudnn that supports DL-specific computation such as convolution of image tensors. However, these GPU libraries consist of low-level C functions with complex interfaces and explicit memory management, which are difficult to use directly and hard to debug.

The optimization of DL applications is domain specific since the meaning of the mathematical computation is not recognized by programming language compilers. The optimization is also complicated in that the gradient-update step of DL training loop is indirectly derived and it can reuse some of the intermediate results of the forward inference step. The degree of reuse depends on how fine-grained the computation abstraction is and how the gradients are derived. The suitable level of granularity differs at different stages of the computation. To encode the DL networks, it is more convenient to use coarse-grained abstractions such as DL layers that pass tensor data between them. However, fine-grained abstractions are more suitable for concrete definition of DL network layers, symbolic gradient derivation, and optimization. On the other hand, to utilize high-performance GPU libraries and to automate memory management, the DL computation should be encoded using abstractions such as tensors and matrices. In short, DL applications need to use varying levels of domain-specific abstractions during their computation process, which can be conveniently supported by a domain specific language.

In recent years, researchers have developed a number of popular tools such as Torch7 [11], Theano [8], Caffe [21], TensorFlow [1], and Computational Network Toolkit (CNTK) [2]. These tools are programming libraries with fixed bindings for key data structures such as tensors and opaque internal representation of control flow logic. Most of the these libraries represent the DL networks using some form of *directed acyclic graphs* (DAG) or computation graphs. The gradient derivation and optimization are based on graph transformation while runtime execution and memory management are based on the optimized computation graph.

Computation graphs are similar to dataflow graphs that depict the order of execution in DL programs. However, the graphs are not convenient for program optimization with multiple levels of abstractions and the heuristics-based optimization of these libraries through graph traversal and transformation are often less than optimal.

Computation graphs are not designed for user-level access, which make it difficult to define customized DL applications and to debug runtime errors. This also limits the runtime environments of the DL applications to what the libraries provide. Debugging errors or making low-level changes to the existing libraries are difficult without in-depth understanding of how the libraries are designed and implemented. In addition, these tools have a number of software dependencies and require platform dependent installation. Most of these tools directly or indirectly depend on languages such as C++ that need to be compiled to specific platforms, which limits the portability of DL applications using these tools.

To address these limitations, we developed DeepDSL[1] a domain specific language [2] embedded in Scala, for composing DL networks. DeepDSL differs from other DL tools in several aspects:

1. DeepDSL represents DL network as expressions where tensor functions define DL layers, function compositions define DL networks, and indexed scalars define tensor expressions. Gradient derivation and optimization are based on term-rewriting rules that transform DSL expressions from one form to another. The optimized expressions are then scheduled to reduce peak memory usage before target code is generated. Before code generation, the DSL expressions are fully abstract, with distinct stages of symbolic evaluation, optimization, and memory management. The scalar, tensor, and tensor function expressions are the abstractions of the mathematical computations of a DL network and they do not have concrete bindings to actual computations before target code is generated. Language-based representation is more flexible for optimization than the computation graphs, where parts of the graphs are tied to concrete data structures such as tensors and the graphs are executed by invocation of low-level code of a specific language such as C.

2. DeepDSL can statically detect errors such as incorrect network composition (as typing error) and report memory consumption at each computation step, Users can adjust the memory allocation strategy before execution.

3. DeepDSL program is compiled to Java source code. Unlike other DL tools, compiled source program does not need to repeat the phase of gradient derivation and optimization. The runtime of initialization is not significant compared to the training time but it can be important when adjusting the parameters of a DL network on small datasets.

4. The target code of DeepDSL is high-level source code that is human readable, customizable, and easy for debugging. The target code is more portable since it can run on any platforms with Java Virtual Machine (JVM) and GPU runtime library available.

[1] This paper extends our conference publication [52] by including the formalization of the DSL, implementation details of gradient derivation and optimization, enhanced performance evaluation, and detailed discussion on related works.

[2] `https://github.com/deepdsl/deepdsl`

Other DL tools often have far more dependencies and are specific to language versions and operating systems.

For the rest of the paper, Section 2 is an informal overview of DeepDSL using a simple DL network. In Section 3, we define the formal syntax, type system, and operational semantics of DeepDSL. We explain gradient derivation of DeepDSL in Section 4, optimization through symbolic reduction in Section 5, optimization through inlining and in-place computation in Section 6, and code scheduling in Section 7. Target code generation and runtime memory management are explained in Section 8. The performance of DeepDSL is evaluated in Section 9, which is followed by related work in Section 10. We conclude in Section 11.

2 Overview

A high-level overview of DeepDSL is shown in Figure 1, where a DSL program is transformed through the stages of symbolic gradient, optimization, and code generation into a Java source program. The generated Java program is human readable and runs on Nvidia GPU through a Java API that calls Cuda/Cudnn library through a JNI library JCuda.

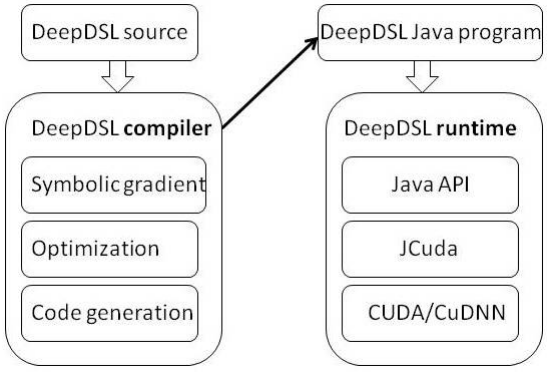


Figure 1: DeepDSL architecture, where the compiler and runtime are completely separate.

The core concepts of DeepDSL are abstract *tensor* and *scalar* expressions and *tensor functions* that transform tensors to tensors and tensors to scalars. DeepDSL directly encodes the mathematical representation of DL networks, where each layer is represented as a tensor function. The entire network is then represented as a composition of these functions. The training loss of a DL network is represented as a scalar expression. The gradients of the loss expression against network parameters are derived symbolically so that they are also DSL expressions. The gradient and the loss expressions go through several stages of simplification, optimization, transformation to become expressions of intermediate representation (IR), which remain abstract and human readable. A final stage of code generation transforms the IR expressions to a Java program for DL network training and inference.

Also, since DeepDSL programs consist of DSL expressions that represent abstract mathematical computations, they can be statically analyzed by the DSL compiler to infer the dimensions of tensors in each layer, to check whether the layers are properly connected, and to automatically insert tensor reshaping operations as necessary. Errors caused by incorrect parameter dimensions are caught before code generation.

DeepDSL analyzes the dependencies of the DSL expressions during optimization stage to determine when each DSL expression is ready to run. For example, the gradients of the weight and bias of convolution layer can start as soon as the backward gradient of the previous layer is known and before the backward gradients of other layers can be computed. Such information is obtained by analyzing the variable dependency of the IR expressions. There is no dedicated data structure such as a graph for representing the relations between layers. DeepDSL also reorders the execution of IR expressions so that tensor objects are allocated as late as possible but deallocated as early as possible to reduce the peak memory consumption.

DeepDSL is embedded in Scala and its syntax is defined using Scala classes and methods as syntactic sugar. After evaluation, DeepDSL programs are de-sugared to a form of *abstract syntax tree* (AST). In this section, we use examples to illustrate how a DL network such as Lenet is defined using DeepDSL.

2.1 Lenet

Figure 2 shows the network structure of a variant of the classic *LeNet-5*[30]. This network has 2 convolution and subsampling/pooling layer alternatively arranged for 2 times, followed by a fully-connected layer, a ReLU activation layer, another fully-connected layer. A softmax layer is attached to the very end to produce a normalized K-dimensional vector of real values in the range $[0,1]$ that add up to 1.

LeNet DNN was originally designed to recognize visual patterns directly from pixel images with extreme variability, such as handwritten characters, as well as robustness to distortions and simple geometric transformations[3]. Due to its simplicity and latent feature capturing power, Its variations have been applied to domains such

[3]http://yann.lecun.com/exdb/lenet/

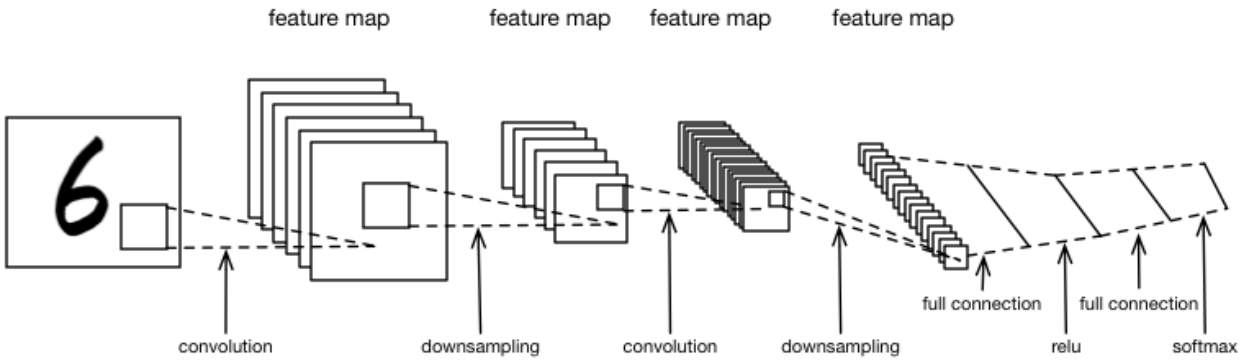


Figure 2: A version of the LeNet-5 network structure [30]

as facial recognition, scene labeling, and image classification, etc.

The parameters of convolution layer consist of learnable kernels or filters. Each unit of this layer receives inputs from a set of units located in small neighboring area in the previous layer, the neighboring area is called the receptive field. During the forward pass each filter is convolved with input to produce a feature map. The weight vector that generates the feature map is shared to reduce the number of learned parameters. As the name feature map implies, the convolution computation between each filter and each receptive field captures the local feature in that particular area of the input. When all the convolution computations are finished, features and their locations across the complete visual input are captured and recorded.

The pooling layer takes small rectangular block areas from the output of the convolution layer and compute subsample values (maximum or average value of all the unit values in the block, etc.). This computation step greatly reduces the spatial size of the representation, thus reducing the parameters to be computed. CNN differs from the traditional multilayer perceptrons (MLP) in the sense that it provides certain level shift and distortion invariance [28]. Such property is mainly achieved with the pooling layer. Since even when images are shifted or distorted a bit, the subsample values taken by the pooling layer can still largely remain unchanged and thus preserve the commonalities between input data samples.

The second convolution layer captures features from the output of the previous pooling layer. These features are from feature space that is different from the feature space of the input layer. Then the second pooling layer continues to reduce the dimensionality of the input from the second convolution layer. Due to its powerful capability of discovering features in different dimensionality space, such repeated stack-up pattern have been widely used in many different CNNs.

While locality information is important for the visual input, the global information or the latent relation between different local blocks is also very important. This information is captured by the fully connected layer. The fully connected layer connects all neurons from the previous layer to each neuron of it to compute the global semantic information.

Finally, depending on the output needs, an activation function (e.g. ReLU) may be applied to the fully connected layer. This function is used to generate a boundary (linear or nonlinear) between the input samples.

2.2 Tensor

The core concept in DeepDSL is tensor, which is represented by a Scala type `Vec`. A `Vec` object has an array of dimensions of the type `Dim`. Each dimension object is either a dimension variable `DimVar` or a dimension expression. For example, a 4-D tensor `x` of the type `VecDec` can be declared using a call `T._new(N, C, X, Y)`, where `N, C, X, Y` are the dimensions of batch, channel, width, and height of an image. If `T._new(F, C, K1, K2)` is a convolution kernel `k`, then the convolution of `x` and `k` with stride 1 and padding 0 will result in a tensor of the dimension `F, C, X-K1+1, Y-K2+1`, where `X-K1+1` and `Y-K2+1` are dimension expressions.

The code snippet in Listing 1 defines 2-D tensor `w` and `x` and the sum of their product over the abstract index `k` of dimension `M1`, where `x(i,k)` and `w(j,k)` represent tensor elements of `x` at index `i, k` and `w` at index `j, k` respectively.

```
1 val x = T._new(N, M1)
2 val w = T._new(M2, M1)
3
4 T.sum(M1, k => x(i, k) * w(j, k))
```

Listing 1: Sum of tensor expression

The call `T.sum(M1, k => e)` returns the sum of the scalar `e` over the index `k` of dimension `M1`. Note that this expression does not compute a value but is an abstraction

that can be translated to code that does the computation in the code generation stage.

Using this, we can define a fully connected layer (an affine transformation) with weight `k` and bias `b` in List 2.

```
T.vec(N, M2, (i, j) =>
  T.sum(M1, k => x(i, k) * w(j, k)) + b(j)
)
```

Listing 2: Fully connected layer

The call `T.vec(N, M2, (i, j) => e)` returns a tensor expression defined by the scalar `e` over the index `i` and `j` of dimension `N` and `M2` respectively.

2.3 Tensor Function

In order to compose the fully connected layer with other layers, we can use the tensor function in Listing 3 to represent the layer, where expression of the form `VecFun(x, v)` represents a function that takes input tensor `x` and returns a tensor represented by `v`.

```
VecFun(x,
 T.vec(N, M2,
  (i, j) => T.sum(M1, k => x(i, k) * w(j, k)) + b(j)
 )
)
```

Listing 3: Tensor function for fully-connected layer

Putting everything together, the Scala method in Listing 4 takes weight and bias tensor as parameters and return a tensor function that represents a fully connected layer, where `w.dim(0)` returns the first dimension of the tensor `w` and `T.dim` creates a new dimension variable.

```
// x: N x M1    w: M2 x M1    b: M2
def full(w: VecDec, b: VecDec) = {
 val N = T.dim; val M2 = w.dim(0); val M1 = w.dim(1)
 val x = T._new(N, M1)

 VecFun(x,
  T.vec(N, M2,
   (i, j) => T.sum(M1, k => x(i, k) * w(j, k)) + b(j)
  )
 )
}
```

Listing 4: Method that returns a fully-connected layer

If we represent the type of a tensor using its dimension list, then the tensor function returned by the method `full` has the type of `List(N, M1) -> List(N, M2)`. Note that for this type, the only dimension variable that must have concrete binding is `M2`, while `N` and `M1` can remain abstract since we can find concrete binding for them when connecting this layer with its previous layers in a network.

2.4 Fixed Tensor

Common layers in DL networks such as convolution and activation have sophisticated implementation in dedicated libraries such as Cudnn. To represent this kind of implementation, DeepDSL uses fixed tensors of the type `FixVec`. An expression of the form `FixVec(layer, param, dim)` represents a fixed implementation for some `layer` type that takes a parameter list `param` and returns tensor of dimensions `dim`.

For example, using this construct, we can define a Scala method `relu` in Listing 5 that returns a ReLU activation layer as a tensor function, where the parameter `n` of the method specifies the number of dimensions of the input `x` and `T._new(n)` creates a tensor variable of dimension `n`.

```
def relu(n: Int) = {
  val x = T._new(n)
  VecFun(x, FixVec(ReLU(), List(x), x.dim))
}
```

Listing 5: Function for ReLU

The reason that we define the fully connected layer differently from the ReLU layer is that the former will be translated to calls to more fine-grained Cuda operations such as matrix product and sum while the latter will be translated to a cross-grained Cudnn call for activation layer. Despite the difference, the two forms of tensor expressions are treated uniformly by DeepDSL during the process of gradient derivation and optimization. They only differ during code generation stage.

Of course, like how we encode the fully connected layer, we can also have direct encoding of ReLU such that the generated Java code will call the more basic Cuda functions instead of direct Cudnn functions.

2.5 Function Application and Composition

Just like the tensors and tensor functions, the function applications are also abstract as shown in Listing 6.

```
val x = T._new(2)

val M1 = T.dim
val M2 = T.dim(10)  // dimension of size 10

// w is named "W" and initialized as Gaussian variable
val w = T._new(Param.gaussian, "W", M2, M1)
// b is named "B" and initialized as constant 0
val b = T._new(Param.const(0), "B", M2)

val f = full(w, b)
val activate = relu(2)

activate(f(x))
```

Listing 6: Function application

In the example above, the function call `f(x)` does not directly compute a value since the tensor function `f` is abstract and so is `x`. Instead, `f(x)` reduces to a DSL expression of the type `VecApp`, which is a subtype of `Vec`. This is expected since the application of a tensor function to a tensor argument should result in a tensor as well. An expression of the form `VecApp(fun, arg)` represents the application of the tensor function `fun` to the tensor argument `arg`. For example, `f(x)` reduces to `VecApp(f, x)` and `activate(f(x))` reduces to `VecApp(activate, VecApp(f, x))`.

Now we can define a tensor function that represents the composition of `activate` and `f` as:

```
val x = T._new(2)
VecFun(x, activate(f(x)))
```

Since functions represent layers and function compositions represent DL networks, DeepDSL includes an operator `o` to simplify function composition so that the composition of `activate` with `f` can be written as:

```
activate o f
```

2.6 Network as Function Composition

Using some helper functions, we can define the Lenet network as in Listing 7.

```
val cv1 = CudaLayer.convolv("cv1", 5, 20)
val cv2 = CudaLayer.convolv("cv2", 5, 50)
val mp = CudaLayer.max_pool(2)
val relu = CudaLayer.relu(2)
val f = Layer.full("fc1", 500)
val f2 = Layer.full("fc2", 10)
val flat = Layer.flatten(4, 1)

val network = f2 o relu o f o flat o
              mp o cv2 o mp o cv1
```

Listing 7: Network definition of Lenet

In this example, `CudaLayer.convolv("cv1", 5, 20)` returns a tensor function representing convolution layer with 5 by 5 kernel and output channel size 20, with default stride 1 and padding 0. The name `cv1` are used to distinguish the weight and bias parameters of the convolution layer since these parameters need to be distinct from other parameters in the network. `CudaLayer.max_pool(2)` returns a max pooling layer that down-samples its input by a factor of 2. `Layer.full("fc1", 500)` returns a fully connected layer with output size 500. `Layer.flatten(4, 1)` returns a tensor function that flattens a 4-D tensor into a 2-D tensor by collapsing the 2nd, 3rd, and 4th dimension of the input tensor into the 2nd dimension of the output tensor (Note here 4 is the number of dimensions and 1 is the index where the collapsing starts). The last line defines the Lenet network as function composition, where `o` is left associative. For example, `f2 o relu o f` should read as `(f2 o relu) o f`.

`List(N,C,N1,N2)->List(N,10)` is `network`'s type, where the input is a 4-D tensor, output is a 2-D tensor, `N`, `C`, `N1`, `N2` are dimension variables, and the fixed dimension, 10, is the number of classes of the training data.

2.7 Training

The loss expression `c` of Lenet (line 16 in Listing 8) can be defined as the application of a tensor function `loss o softmax o network` to the input `x.asCuda`, where `x` represents training images, `x.asCuda` represents copying `x` to GPU memory, and `loss` is a tensor to scalar function that represents the loss of `softmax o network` applying to `x.asCuda`.

```
// batch size, channel, width, and height
val N = 500; val C = 1; val N1 = 28; val N2 = 28
val dim = List(N,C,N1,N2)

val y = T._new("Y", List(N)) // image class labels
val x = T._new("X", dim)     // training images

val y1 = y.asIndicator(10).asCuda
val x1 = x.asCuda            // load to GPU memory

val softmax = CudaLayer.log_softmax
val loss = Layer.loss(y1)

val p = network(x1)  // p is the prediction
                     // c is loss of training
val c = (loss o softmax o network) (x1)
```

Listing 8: Loss expression of Lenet

The variable `y` represents class labels of the training data, which are one-hot encoded as indicator vectors using the call `y.asIndicator(10)`, where 10 refers to the number of classes. The variable `p` represents the forward inference of the input `x`.

With forward inference expression and the loss expression, the code to generate Java source code can be defined in Listing 9.

```
val param = c.freeParam

// name, train/test iterations, learn rate, momentum
// weight decay, gradient clipping bound (0 means none)
val solver =
    Train("lenet", 100, 10, 0.01f, 0.9f, 0.0005f, 0)

val mnist = Mnist(dim)       // training dataSet
val loop = Loop(c, p, mnist, (x, y), param, solver)

// generate training and testing file
CudaCompile("path").print(loop)
```

Listing 9: Compilation of Lenet

We first extract the network parameters using the call `c.freeParam`. The variable `solver` is simply a collection of parameters that include output file name, train iteration, test iteration, learning rate, momentum, weight decay, gradient clipping bound (0 means no clipping). The variable `mnist` refers to the Mnist dataset. Finally, we put everything together in the variable `loop` and pass it to `CudaCompile` to generate the Java source code for training and testing, where the path string indicates in which the Java source code should be generated.

Note that gradient derivation and optimization occur inside the class `Loop` where it takes the gradients of the loss expression `c` against the parameters `param`, optimizes the gradients, and transforms them to IR expressions. The code generator `CudaCompile("path").print(loop)` performs a single-pass translation of the IR expressions to Java source code.

3 Formalization

In this section, we present a formal definition of DeepDSL that includes an abstract syntax, an operational semantics, a type system, and examples of their application to some important CNN building blocks.

3.1 Syntax

An abstract syntax for DeepDSL is shown in Figure 3, which defines two types of expressions: tensors (denoted by t) and scalars (denoted by s). A tensor is either a variable, a function application, a tensor expression, a tensor addition, a scalar-tensor product, or a cast. A scalar is either a constant, a function application, a tensor element, the sum of a tensor, or an arithmetic expression such as exponentiation and logarithm.

Dimension The symbol d represents a dimension and D represents a list of dimensions, which can be written as $d_1 \cdots d_k$. A dimension can be a constant, a dimension variable d_x, a dimension expression d/n, $d-d_x$, $d+n$, or a dimension product $d_x^1 \times \cdots \times d_x^k$. Dimension variables are used for defining functions polymorphic in dimensions such as the fully-connected layer in Section 2.3. Dimension expressions are used for defining the dimensions of operations such as convolution and pooling. For example, the dimension of a 1-D convolution between a vector of dimension d_1 and a kernel of dimension d_2 with stride 1 and padding 0 is $d_1 - d_2 + 1$. Dimension products are used for defining the flattened dimension of a tensor. For example, if we flatten the last 3 axis of a 4-D tensor of dimensions $d_1 \cdot d_2 \cdot d_3 \cdot d_4$, the dimension becomes $d_1 \cdot (d_2 \times d_2 \times d_4)$.

Types The type of a tensor is its dimension D while the type of a scalar is $\star$, which represents a real number type. For example, a DL network can represent its input images as a 4-D tensor with the type $d_n \cdot d_c \cdot d_h \cdot d_w$, where d_n, d_c, d_h, d_w are the dimensions of the images' batch size, channel, height, and width respectively.

Function A function has the form of $x \Rightarrow e$, where x is a tensor variable and the e is either a tensor or a scalar. The builtin functions (denoted by $\mathscr{F}$) always return tensors and they represent fixed DL layers in libraries.

Note that DeepDSL supports functions of the form $(x_1, \ldots x_k) \Rightarrow e$ that takes multiple parameters. We only consider single-parameter function here for simplicity.

Variable Each tensor variable x has an implicitly labeled type D. We assume an auxiliary function $\mathscr{T}(x)$ that returns the type of variable x. We overload it so that $\mathscr{T}(\mathscr{F})$ returns that type of builtin function $\mathscr{F}$.

Index The symbol i represents an index and I represents a list of indices, which can be written as $i_1 \cdots i_k$. An index can be an index variable i_x or an index expression $i \times n$, $i + i'$, $i + n$. Each index variable i_x has an implicitly labeled dimension d and the value of i_x ranges from 0 to $d-1$. We assume an auxiliary function $\mathscr{D}(i_x)$ that returns the dimension of i_x.

Tensor expression and tensor element The tensor expression $I \Rightarrow s$ defines a tensor that has the scalar value of s over the domain of the index list I and the dimensions of this tensor is the dimensions of I. The tensor element $x(I)$ is the element of a tensor variable x at index list I.

Sum The expression $\sum_I(s)$ represents the summation of s over the index list I. If $I = i_1 \cdots i_k$, then $\sum_I(s)$ is equivalent to $\sum_{i_1} \sum_{i_2} \cdots \sum_{i_k}(s)$.

Example The index expressions are used in tensor expressions such as convolution. For example, the 1-D convolution of a vector x and a kernel w (stride 1 and padding 0) can be written as

$$i \Rightarrow \sum_{i'} x(i+i') \times w(i')$$

where $\mathscr{T}(x) = d_1$, $\mathscr{T}(w) = \mathscr{D}(i') = d_2$, $\mathscr{D}(i) = d_1 - d_2 + 1$.

3.2 Semantics

An operational semantics for tensors and scalars is shown in Figure 4. In this semantics, each tensor expression evaluates to a tensor value $\mathscr{V}$, which is a flat array of scalar values and a list of dimension values $n_1 \cdots n_k$. The size of the array must be equal to the products of the dimensions.

$$\mathscr{V} \quad ::= \quad (v_0, \ldots, v_m)^{n_1 \cdots n_k} \quad \text{Tensor value}$$

where $m = (n_1 \times \ldots \times n_k) - 1$. The function $\mathscr{D}$ also returns the dimensions of the tensor values.

$$\mathscr{D}((v_0, \ldots, v_m)^{n_1 \cdots n_k}) = n_1 \cdots n_k$$

A function $x \Rightarrow e$ is polymorphic in dimensions if the dimensions D of x contain variables. When the function is applied to an argument $\mathscr{V}$, we not only substitute x in e with $\mathscr{V}$ but also substitute the dimension variables of D in e with the matching dimensions of $\mathscr{V}$ by applying the substitution $\mathscr{U}(D, \mathscr{D}(\mathscr{V}))$ to $e[\mathscr{V}/x]$.

$$\begin{array}{lcl} \mathscr{U}(d,d) & = & \emptyset \\ \mathscr{U}(d \cdot D, d' \cdot D') & = & \mathscr{U}(d,d') \cup \mathscr{U}(D,D') \\ \mathscr{U}(d_x,d) & = & \{d_x \mapsto d\} \end{array}$$

e	::=	t	tensor
	|	s	scalar
t	::=	x, y, z, w	variables
	|	$(x \Rightarrow t)(t)$	application
	|	$\mathscr{F}(t)$	builtin fun. app.
	|	$(i_x^1 \cdots i_x^k) \Rightarrow s$	tensor expression
	|	$t_1 + t_2$	tensor addtion
	|	$s \cdot t$	scalar tensor prod.
	|	$(D)\ x$	cast
s	::=	v	constant
	|	$(x \Rightarrow s)(t)$	application
	|	$x(I)$	tensor element
	|	$\sum_{i_x^1 \cdots i_x^k}(s)$	sum
	|	$\log(s) \mid \exp(s) \mid s^n$	arith. exp.
	|	$s_1 + s_2 \mid s_1 \times s_2$	arith. exp.
f	::=	$x \Rightarrow e$	function
	|	$\mathscr{F}$	builtin tensor fun.
$\mathscr{F}$	::=	convolution	
	|	pooling	
	|	activation | ...	
I	::=	$i \mid i \cdot I$	index list
i	::=	$i \times n \mid i + i' \mid i - n$	index exp.
	|	i_x	index var.
D	::=	$d \mid d \cdot D$	dimension list
d	::=	$n \mid d/n \mid d - d' \mid d + n$	dimension exp.
	|	$d_x^1 \times \cdots \times d_x^k$	dimension prod.
	|	d_x	dimension var.
τ	::=	$\star \mid D$	types

Figure 3: A formal syntax for DeepDSL, where n is some positive integer and $f_2 \circ f_1$ is defined as $x \Rightarrow f_2(f_1(x))$

Two tensors with the same dimensions can be added element-wise. A scalar-tensor product $s \times t$ multiples s with each element of t. We assume that the application of builtin function $\mathscr{F}$ to a tensor value $\mathscr{V}$ will return a tensor value $\mathscr{V}'$ so that the types of $\mathscr{V}$ and $\mathscr{V}'$ match the the parameter and return type of $\mathscr{F}$.

A tensor expression $(i_1 \cdots i_k) \Rightarrow s$ evaluates to a tensor with the dimensions $n_1 \cdots n_k$, where $n_1 \cdots n_k$ are values of $\mathscr{D}(i_1) \cdots \mathscr{D}(i_k)$. Note that for a dimension to evaluate to a value, the dimension must not contain variables. The jth element of the tensor is the value of s when $i_1 = m_1, \ldots, i_k = m_k$ and $j = (\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k$. For example, consider the tensor $(i_1 \cdot i_2 \cdot i_3) \Rightarrow s$, where the dimensions of i_1, i_2, and i_3 are 4, 5, and 6 respectively. The tensor element when $i_1 = 1$, $i_2 = 2$, and $i_3 = 3$ is the 46th element of the flat array that stores the tensor since $((1 * 5) + 2) \times 6 + 3 = 45$.

A cast expression $(n_1' \cdots n_l')\ (v_0, \ldots, v_m)^{n_1 \cdots n_k}$ changes the dimensions of the tensor value $(v_0, \ldots, v_m)^{n_1 \cdots n_k}$ from $n_1 \cdots n_k$ to $n_1' \cdots n_l'$ if they have the same flattened size. For example, in LeNet-5, the 4-D tensor after the second pooling layer needs to be flattened to a 2-D tensor before it can be passed to the fully connected layer. This is a cast of $(n_1' \cdot n_2')\ (v_0, \ldots v_m)^{n_1 \cdot n_2 \cdot n_3 \cdot n_4}$. When computing backward gradient, such cast changes to $(n_1 \cdot n_2 \cdot n_3 \cdot n_4)\ (v_0, \ldots v_m)^{n_1' \cdot n_2'}$. In both cases, $n_1' = n_1$ and $n_2' = n_2 \times n_3 \times n_4$. In general, cast requires $n_1' \times n_2' \times \ldots \times n_l' = n_1 \times n_2 \times \ldots \times n_k$ regardless in which index axis flattening or unflattening occurs.

A tensor element expression $\mathscr{V}(m_1 \cdots m_k)$ evaluates to either jth element of $\mathscr{V}$, where $j = (\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k$ or 0 if one of the indices is out of bound. A sum expression $\sum_I(s)$ evaluates to the sum of the tensor value evaluated from $I \Rightarrow s$. Reduction of other types of scalar expressions is omitted.

3.3 Typing Rules

Type judgment has the form of $\Gamma \vdash e : \tau$, where Γ maps tensor variables to their types and it also maps index variables to their dimensions. The typing rules for tensor and scalar expressions are shown in Figure 6, where the rules for arithmetic scalar expressions such as $\log(s)$ and $\exp(s)$ are omitted.

The type of a function $x \Rightarrow e$ is $\tau_1 \to \tau_2$ where τ_1 is the labeled type of x, which we retrieve through $\tau(x)$ and τ_2 is the type of e. The type of a function call $f(t)$ is the return type τ_2 of f after applying the substitution $\mathscr{U}(\tau_1, \tau_1')$, where τ_1 is the parameter type of f and τ_1' is the type of t. That is, we instantiate the dimension parameters of f with the concrete dimensions of t. The type of the tensor expression $(i_1 \cdots i_k) \Rightarrow s$ is the dimensions of $i_1 \cdots i_k$. These indices must be index variables with labeled dimensions. The type of a cast expression $(D)\ x$ is D and also the flattened dimensions of $\Gamma(x)$ and D must match. For example, we can cast a variable of type $d_1 \cdot d_2 \cdot d_3 \cdot d_4$ to $d_1 \cdot (d_2 \times d_3 \times d_4)$. The flatten function `flat` is defined as follows.

$$
\begin{array}{lcl}
\texttt{flat}(d \cdot D) & = & \texttt{flat}(d) \cdot \texttt{flat}(D) \\
\texttt{flat}(d \times d') & = & \texttt{flat}(d) \cdot \texttt{flat}(d') \\
\texttt{flat}(d) & = & d
\end{array}
$$

The type of a tensor element $x(I)$ is always $\star$ and the type of x must be the same as the type of I, which is defined by the rules in Figure 5.

The type of an index list I is the list of types of each index in I. The type of an index variable must be added to the environment by either a tensor expression or a sum expression. Index expressions of the form of $i \times n$, $i_1 + i_2$, $i - n$ are used in describing the indices of convolution operation.

For example, an 1-D convolution between a vector x and a kernel w with stride n_1 and padding n_2 is defined

$$\frac{\sigma = \mathscr{U}(\mathscr{T}(x), \mathscr{D}(\mathscr{V}))}{(x \Rightarrow e)(\mathscr{V}) \to \sigma(e[\mathscr{V}/x])}$$

$$\frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2} \quad \frac{t_2 \to t_2'}{\mathscr{V} + t_2 \to \mathscr{V} + t_2'}$$

$$\frac{\mathscr{V} = (v_0 + v_0', \ldots, v_m + v_m')^{n_1 \cdots n_k}}{(v_0, \ldots, v_m)^{n_1 \cdots n_k} + (v_0', \ldots, v_m')^{n_1 \cdots n_k} \to \mathscr{V}}$$

$$\frac{s \to s'}{s \cdot t \to s' \cdot t} \quad \frac{t \to t'}{v \cdot t \to v \cdot t'}$$

$$v \cdot (v_0, \ldots, v_m)^{n_1 \cdots n_k} \to (v \times v_0, \ldots, v \times v_m)^{n_1 \cdots n_k}$$

$$\frac{\begin{array}{c}\mathscr{D}(i_1) \to n_1 \ldots \mathscr{D}(i_k) \to n_k \\ m_1 \in \{0 \ldots n_1 - 1\} \ldots m_k \in \{0 \ldots n_k - 1\} \\ s[m_1/i_1, \ldots, m_k/i_k] \to^* v_{(\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k}\end{array}}{(i_1 \cdots i_k) \Rightarrow s \to (v_0, \ldots, v_m)^{n_1 \cdots n_k}}$$

$$\frac{\mathscr{D}(\mathscr{F}) = \mathscr{D}(\mathscr{V}) \to \mathscr{D}(\mathscr{V}')}{\mathscr{F}(\mathscr{V}) \to \mathscr{V}'}$$

$$\frac{n_1' \times \ldots \times n_l' = n_1 \times \ldots \times n_k}{(n_1' \cdots n_l')\ (v_0, \ldots, v_m)^{n_1 \cdots n_k} \to (v_0, \ldots, v_m)^{n_1' \cdots n_l'}}$$

$$\frac{i_1 \to m_1, \ldots, i_k \to m_k}{\mathscr{V}(i_1 \cdots i_k) \to \mathscr{V}(m_1 \cdots m_k)}$$

$$\frac{\mathscr{V} = (v_0, \ldots, v_m)^{n_1 \cdots n_k} \quad 0 \le m_1 \le n_1 \ldots 0 \le m_k \le n_k}{\mathscr{V}(m_1 \cdots m_k) \to v_{(\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k}}$$

$$\frac{\mathscr{V} = (v_0, \ldots, v_m)^{n_1 \cdots n_k} \quad \exists j.\ m_j < 0 \ \vee m_j > n_j}{\mathscr{V}(m_1 \cdots m_k) \to 0}$$

$$\frac{I \Rightarrow s \to (v_0, \ldots, v_m)^{n_1 \cdots n_k}}{\sum_I(s) \to v_0 + \ldots + v_m}$$

Figure 4: Operational Semantics

as $(i_1) \Rightarrow \sum_{i_2} x(i_1 \times n_1 + i_2 - n_2) \times w(i_2)$, where $\mathscr{D}(i_1) = (\mathscr{T}(x) + 2n_2 - \mathscr{D}(i_2))/n_1 + 1$.

The type of the index expression $i_1 \times n_1 + i_2 - n_2$ is derived as follows based on the rules in Figure 5.

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma(i_1) = \mathscr{D}(i_1), \text{ where } \mathscr{D}(i_1) = (\mathscr{T}(x) + 2n_2 - \mathscr{D}(i_2))/n_1 + 1}{\Gamma \vdash i_1 :\ (\mathscr{T}(x) + 2n_2 - \mathscr{D}(i_2))/n_1 + 1}}{\Gamma \vdash i_1 \times n_1 :\ \mathscr{T}(x) + 2n_2 - \mathscr{D}(i_2) + 1} \quad \Gamma \vdash i_2 :\ \mathscr{D}(i_2)}{\Gamma \vdash i_1 \times n_1 + i_2 :\ \mathscr{T}(x) + 2n_2}}{\Gamma \vdash i_1 \times n_1 + i_2 - n_2 :\ \mathscr{T}(x)}$$

The type of the index expression is based on the type

$$\frac{\Gamma(i_x) = d}{\Gamma \vdash i_x :\ d} \quad \frac{\Gamma \vdash i :\ d \quad \Gamma \vdash I :\ D}{\Gamma \vdash i \cdot I :\ d \cdot D}$$

$$\frac{\Gamma \vdash i :\ (d-1)/n + 1}{\Gamma \vdash i \times n :\ d}$$

$$\frac{\Gamma \vdash i_1 :\ d_1 - d_2 + 1 \quad \Gamma \vdash i_2 :\ d_2}{\Gamma \vdash i_1 + i_2 :\ d_1} \quad \frac{\Gamma \vdash i :\ d + 2n}{\Gamma \vdash i - n :\ d}$$

Figure 5: Typing rule for index expressions

derivation of $(i_1) \Rightarrow \sum_{i_2} x(i_1 \times n_1 + i_2 - n_2) \times w(i_2)$.

$$\frac{\dfrac{\dfrac{\Gamma \vdash i_1 \times n_1 + i_2 - n_2 :\ \mathscr{T}(x)}{\Gamma \vdash x :\ \mathscr{T}(x) \quad \Gamma \vdash i_2 :\ \mathscr{D}(i_2) \quad \Gamma \vdash w : \mathscr{D}(i_2) \quad \text{where } \Gamma = \Gamma', i_1 : \mathscr{D}(i_1), i_2 : \mathscr{D}(i_2)}}{\Gamma', i_1 : \mathscr{D}(i_1) \vdash \sum_{i_2} x(i_1 \times n_1 + i_2 - n_2) \times w(i_2) :\ \star}}{\Gamma' \vdash (i_1) \Rightarrow \sum_{i_2} x(i_1 \times n_1 + i_2 - n_2) \times w(i_2) :\ \mathscr{D}(i_1)}$$

Theorem 1 *If $\emptyset \vdash t : \tau$, then there exists $\mathscr{V}$ such that $t \to^* \mathscr{V}$. If $\emptyset \vdash s : \tau$, then there exists v such that $s \to^* v$.*

3.4 Examples

Affine transformation A fully-connected layer in DNN is an affine transformation of a 2-D tensor x that multiplies it with a 2-D filter w and adds an 1-D bias b.

$$f_1 \triangleq x \Rightarrow (i_1 \cdot i_3) \Rightarrow \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2 \cdot i_3) + b(i_1)$$

where $\mathscr{D}(i_k) = d_k$ for $k \in \{1,2,3\}$. The expression $x(i_1 \cdot i_2)$ evaluates to an element of x at index i_1 at axis 1 and i_2 at axis 2. The type of f_1 is

$$\emptyset \vdash f_1 : d_1 \cdot d_2 \to d_1 \cdot d_3$$

where $\mathscr{T}(x) = d_1 \cdot d_2$, $\mathscr{T}(w) = d_2 \cdot d_3$, and $\mathscr{T}(b) = d_1$.

Flattening A flattening function turns a 4-D tensor into a 2-D tensor by collapsing the last 3 dimensions into 1. This is a common operation of DNNs that transforms inputs for a fully-connected layer, which applies to 2-D tensors. The function f_2 below uses a cast to turn a 4-D tensor x into a 2-D tensor.

$$f_2 \triangleq x \Rightarrow (d_1 \cdot (d_2 \times d_3 \times d_4))\ x$$

where $\mathscr{T}(x) = d_1 \cdot d_2 \cdot d_3 \cdot d_4$.

Here x is a 4-D tensor and it is cast to a 2-D tensor type where the type of x is $d_1 \cdot d_2 \cdot d_3 \cdot d_4$, which is the same as $\texttt{flat}(d_1 \cdot (d_2 \times d_3 \times d_4))$.

$$\emptyset \vdash f_2 : d_1 \cdot d_2 \cdot d_3 \cdot d_4 \to d_1 \cdot (d_2 \times d_3 \times d_4)$$

$$\Gamma \vdash v : \star \quad \text{T-Constant}$$

$$\Gamma \vdash x : \Gamma(x) \quad \text{T-Var}$$

$$\Gamma \vdash \mathscr{F} : \mathscr{T}(\mathscr{F}) \quad \text{T-Builtin}$$

$$\frac{\Gamma,\ x : \mathscr{T}(x) \vdash e : \tau}{\Gamma \vdash x \Rightarrow e : \mathscr{T}(x) \to \tau} \quad \text{T-Fun}$$

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash t : \tau_1' \quad \sigma = \mathscr{U}(\tau_1, \tau_1')}{\Gamma \vdash f(t) : \sigma(\tau_2)} \quad \text{T-App}$$

$$\frac{\Gamma,\ i_1 : \mathscr{D}(i_1), \ldots, i_k : \mathscr{D}(i_k) \vdash s : \star}{\Gamma \vdash (i_1 \cdots i_k) \Rightarrow s : \mathscr{D}(i_1) \cdots \mathscr{D}(i_k)} \quad \text{T-TensorExp}$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 + t_2 : \tau} \quad \text{T-TenorPlus}$$

$$\frac{\Gamma \vdash s : \star \quad \Gamma \vdash t : \tau}{\Gamma \vdash s \cdot t : \tau} \quad \text{T-TensorScale}$$

$$\frac{\texttt{flat}(\Gamma(x)) = \texttt{flat}(D)}{\Gamma \vdash (D)\, x : D} \quad \text{T-Cast}$$

$$\frac{\Gamma,\ i_1 : \mathscr{D}(i_1), \ldots, i_k : \mathscr{D}(i_k) \vdash s : \star}{\Gamma \vdash \sum_{i_1 \cdots i_k}(s) : \star} \quad \text{T-Sum}$$

$$\frac{\Gamma(x) = D \quad \Gamma \vdash I : D}{\Gamma \vdash x(I) : \star} \quad \text{T-Element}$$

Figure 6: Typing rules for tensor and scalar expressions

Convolution The most common operations in DNN are convolutions. For image classification, a convolution layer applies 2-D convolution to input images x (or feature maps) with a kernel w and adds a bias b. The input to the convolution layer is a 4-D tensor where the 1st axis is the number of images, the 2nd axis is the number of input channels, and the 3rd and 4th axes are height and width of images. The kernel is also a 4-D tensor, where the 1st and 2nd axes are the number of output and input channels while the last two axes are kernel width and height.

The equation below defines a convolution function f_3 with stride 1 and padding 0.

$$f_3 \triangleq x \Rightarrow (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow \textstyle\sum_{i_c} \sum_{i_u} \sum_{i_v} \\ x(i_n \cdot i_c \cdot (i_h + i_u) \cdot (i_w + i_v)) \times w(i_k \cdot i_c \cdot i_u \cdot i_v) + b(i_k)$$

where $\mathscr{D}(i_n) = d_n$, $\mathscr{D}(i_c) = d_c$, $\mathscr{D}(i_u) = d_u$, $\mathscr{D}(i_v) = d_v$, $\mathscr{D}(i_k) = d_k$, $\mathscr{D}(i_h) = d_r - d_u + 1$, $\mathscr{D}(i_w) = d_s - d_v + 1$.

Note that the dimension of i_h is $d_r - d_u + 1$ since the dimension of i_u is d_u and the dimension of $i_h + i_u$ must be the same as that of the 3rd axis of x, which is d_r.

There are efficient implementation of convolution in libraries and we can treat it as a builtin function. The type of f_3 is

$$\emptyset \vdash f_3 : d_n \cdot d_c \cdot d_r \cdot d_s \to d_n \cdot d_k \cdot (d_r - d_u + 1) \cdot (d_s - d_v + 1)$$

where $\mathscr{T}(x) = d_n \cdot d_c \cdot d_r \cdot d_s$, $\mathscr{T}(w) = d_k \cdot d_c \cdot d_u \cdot d_v$, and $\mathscr{T}(b) = d_k$.

Loss In supervised learning, loss functions like f_4 below are used to compute scalar values that measure the average error of predicted classification,

$$f_4 \triangleq x \Rightarrow \sum_{i_n} \sum_{i_k} y(i_n \cdot i_k) \times x(i_n \cdot i_k) \times (-\frac{1}{d_n})$$

where x is the log probability of the prediction and y is the ground truth, and $\mathscr{T}(x) = d_n \cdot d_k$. Note that y is a $d_n \times d_k$ matrix, where i_nth row is an indicator vector that represents the true class of the i_nth image. The variable x is also a $d_n \times d_k$ matrix and the i_kth value of its i_nth row is the log probability of the image i_n being in class i_k.

The type of f_4 is

$$\emptyset \vdash f_4 : d_n \cdot d_k \to \star$$

Log softmax A softmax function like f_5 below is used to normalize the results of the previous layers.

$$f_5 \triangleq x \Rightarrow x - (i_n \cdot i_k) \Rightarrow \log(\sum_{i_l} \exp(x(i_n \cdot i_l)))$$

where $\mathscr{T}(x) = d_n \cdot d_k$, $\mathscr{D}(i_n) = d_n$, $\mathscr{D}(i_k) = \mathscr{D}(i_l) = d_k$. The type of f_5 is

$$\emptyset \vdash f_5 : d_n \cdot d_k \to d_n \cdot d_k$$

Note that there are efficient implementation of softmax in libraries so that we can treat it as a builtin function.

4 Gradient derivation

The forward inference of a DNN computes a loss expression s that measures the error of the predicted classification of training data. To train a DNN, we update each network parameter w with a function of $\frac{\partial s}{\partial w}$, which is the gradient of the loss expression against w. However, these gradients should not be directly computed and they should be simplified into computable forms through symbolic derivation.

The gradient of a scalar with respect to a tensor variable $\frac{\partial s}{\partial x}$ is a tensor with the same type as x. The gradient of a tensor with respect to a tensor variable $\frac{\partial t}{\partial x}$ is defined

by tensors of the form $t' \cdot \frac{\partial t}{\partial x}$ with the typing rule:

$$\frac{\Gamma \vdash t' : D \quad \Gamma \vdash t : D \quad \Gamma \vdash x : D'}{\Gamma \vdash t' \cdot \frac{\partial t}{\partial x} : D'}$$

If we view DNN layers as functions and the loss expression as the application of the functions on the training data, then the gradient derivation follows Rule 1.

$$\frac{\partial f(t)}{\partial y} = f'(t) \cdot \frac{\partial t}{\partial y} + \frac{\partial f}{\partial y}(t) \tag{1}$$

where $f(t)$ is the application of function f to tensor t, $\frac{\partial f(t)}{\partial y}$ is the derivative of $f(t)$ with respect to y, $\frac{\partial t}{\partial y}$ is the derivative of t with respect to y, f' is the derivative of f with respect to its parameter, and $\frac{\partial f}{\partial y}$ is the derivative of f with respect to y.

Note that we write f' instead of $\frac{\partial f}{\partial x}$ when f has a single parameter x. If f has multiple parameters such as x_1 and x_2, we write $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$ instead.

In general, y may appear in both f and t. However, in case where y only appears in either f or t, either $f'(t)$ or $\frac{\partial t}{\partial y}$ is zero, which means that $\frac{\partial f(t)}{\partial y}$ equals to either $f'(t) \cdot \frac{\partial t}{\partial y}$ or $\frac{\partial f}{\partial y}(t)$. Note that $f'(t)$ is a tensor if f returns a scalar and it is a tensor gradient if f returns a tensor.

Application of Rule 1 The following is the gradient derivation of a scalar $s = f_4(f_3(f_2(f_1(x))))$ with respect to the parameters w_1, w_2, and w_3, where w_i is a parameter in function f_i, $\forall i \in \{1,2,3\}$.

$$\begin{aligned}
\frac{\partial s}{\partial w_1} &= z_3 \cdot \frac{\partial f_3(f_2(f_1(x)))}{\partial w_1} \\
&= z_3 \cdot z_2 \cdot \frac{\partial f_2(f_1(x))}{\partial w_1} \\
&= z_3 \cdot z_2 \cdot z_1 \cdot \frac{\partial f_1(x)}{\partial w_1} \\
&= z_3 \cdot z_2 \cdot z_1 \cdot \frac{\partial f_1}{\partial w_1}(x) \\
\frac{\partial s}{\partial w_2} &= z_3 \cdot z_2 \cdot \frac{\partial f_2(f_1(x))}{\partial w_2} \\
&= z_3 \cdot z_2 \cdot \frac{\partial f_2}{\partial w_2}(f_1(x)) \\
\frac{\partial s}{\partial w_3} &= z_3 \cdot \frac{\partial f_3(f_2(f_1(x)))}{\partial w_3} \\
&= z_3 \cdot \frac{\partial f_3}{\partial w_3}(f_2(f_1(x)))
\end{aligned}$$

where $z_3 = f_4'(f_3(f_2(f_1(x))))$, $z_2 = f_3'(f_2(f_1(x)))$, and $z_1 = f_2'(f_1(x))$. Note in this example w_1 appears in $f_1(x)$ but not in x, w_2 appears in $f_2(f_1(x))$ but not $f_1(x)$, and w_3 appears in $f_3(f_2(f_1(x)))$ but not in $f_2(f_1(x))$.

Direct computation of the above expressions is not efficient since they contain a lot of redundancies such as the repeated evalution of z_3. The redundancies can be removed using common subexpression elimination but it is slow for complex DNN. Fortunately, we can use the reverse accumulation method [31, 6] to derive parameter gradients efficiently without redundancy.

$$x \rightsquigarrow x$$

$$\frac{t \rightsquigarrow a\ \hat{t} \quad t_1[y/x] \rightsquigarrow a'\ \hat{t}_1}{(x \Rightarrow t_1)(t) \rightsquigarrow a\ \ y \leftarrow \hat{t};\ \ a'\ \hat{t}_1}$$

$$\frac{t \rightsquigarrow a\ \hat{t} \quad s[y/x] \rightsquigarrow a'\ \hat{s}}{(x \Rightarrow s)(t) \rightsquigarrow a\ \ y \leftarrow \hat{t};\ \ a'\ \hat{s}}$$

$$\frac{t \rightsquigarrow a\ \hat{t}}{\mathscr{F}(t) \rightsquigarrow a\ \ y \leftarrow \hat{t};\ \ \mathscr{F}(y)}$$

$$\frac{t_1 \rightsquigarrow a_1\ \hat{t}_1 \quad t_2 \rightsquigarrow a_2\ \hat{t}_2}{t_1 + t_2 \rightsquigarrow a_1\ \ y_1 \leftarrow \hat{t}_1;\ \ a_2\ \ y_2 \leftarrow \hat{t}_2;\ \ y_1 + y_2}$$

$$\frac{t \rightsquigarrow a\ \hat{t}}{s \cdot t \rightsquigarrow a\ \ y \leftarrow \hat{t};\ \ s \cdot y}$$

Figure 7: Static single assignment transformation, where y variables are fresh and we assume that scalar function application does not appear inside a tensor expression.

Reverse accumululation Before gradient derivation of a loss expression s, we first perform a static single assignment (SSA) transformation to s, which result in expression of the form

$$\begin{aligned}
&y_1 \leftarrow \hat{t}_1; \\
&\cdots \\
&y_n \leftarrow \hat{t}_n; \\
&\hat{s}
\end{aligned}$$

where $y_i \leftarrow \hat{t}_i$ is an assignment of $\hat{t}_i$ to y_i and $\hat{s}$ is s with its tensor subexpressions replaced by tensor variables y_i.

Let a represent a sequence of assignments.

$$\begin{aligned}
a \quad ::= \quad & \varepsilon \\
\mid \quad & y \leftarrow \hat{t};\ a
\end{aligned}$$

Figure 7 defines the SSA transformation rules of the form $e \rightsquigarrow a;\ \hat{e}$, which transforms expression e to a sequence of assignment a followed by $\hat{e}$. After transformation, the tensor and scalar has the following syntax.

$$
\begin{array}{rcll}
t & ::= & x,y,z,w & \text{variables} \\
 & | & \mathscr{F}(y) & \text{application} \\
 & | & I \Rightarrow s & \text{tensor exp} \\
 & | & y_1 + y_2 & \\
 & | & s \cdot y & \\
s & ::= & v & \text{constant} \\
 & | & x(I) & \text{tensor element} \\
 & | & \sum_I(s) & \text{scalar sum} \\
 & | & \log(s) \mid \exp(s) \mid s^n & \\
 & | & s_1 + s_2 \mid s_1 \times s_2 & \\
 & \ldots & &
\end{array}
$$

Parameter gradients Given a scalar expression s, the gradient of s with respect to a parameter w_k is defined by

$$\frac{\partial s}{\partial w_k} = \sum_i \frac{\partial s}{\partial t_i} \cdot \frac{\partial t_i}{\partial w_k}$$

where each t_i is a subexpression of s that contains w_k. The gradient $\dfrac{\partial e}{\partial t_j}$ defined by the equation

$$\frac{\partial e}{\partial t_j} = \sum_i \frac{\partial e}{\partial t_i} \cdot \frac{\partial t_i}{\partial t_j}$$

where each t_i is a subexpression of e that contains t_j.

After SSA transformation, each subexpression t_i in s and e is replaced by a variable y_i defined by the assignment $y_i \leftarrow \hat{t}_i$ and the scalar s becomes $\hat{s}$. Then, parameter gradients can be derived as follows:

1. Derive $\dfrac{\partial \hat{t}_i}{\partial y_j}$ for each $\hat{t}_i$ and for each y_j in $\hat{t}_i$.
2. Define the variable z_j with the assignments

$$z_j \leftarrow \begin{cases} \dfrac{\partial \hat{s}}{\partial y_j} & y_j \text{ appears in } \hat{s} \\ \displaystyle\sum_i z_i \cdot \frac{\partial \hat{t}_i}{\partial y_j} & y_j \text{ appears in } \hat{t}_i \end{cases}$$

   where for simplicity, we assume that variables in $\hat{s}$ do not appear elsewhere.
3. If w_k is a parameter in s, then

$$\frac{\partial s}{\partial w_k} = \sum_i z_i \cdot \frac{\partial \hat{t}_i}{\partial w_k}$$

   where $\hat{t}_i$ contains w_k.

For the last example $s = f_4(f_3(f_2(f_1(x))))$, applying SSA transformation to s, we obtain the following result.

$$\begin{aligned} y_1 &\leftarrow \hat{t}_1 \\ y_2 &\leftarrow \hat{t}_2 \\ y_3 &\leftarrow \hat{t}_3 \\ &\hat{s} \end{aligned}$$

where $\hat{t}_1 = f_1(x)$, $\hat{t}_2 = f_2(y_1)$, $\hat{t}_3 = f_3(y_2)$, $\hat{s} = f_4(y_3)$.

Using reverse accumulation method, we can obtain the following statements, where the local variables y_i and z_i ($i \in \{1,2,3\}$) hold intermediate results.

$$\begin{aligned} z_3 &\leftarrow f_4'(y_3) & \frac{\partial s}{\partial w_3} &= z_3 \cdot \frac{\partial f_3}{\partial w_3}(y_2) \\ z_2 &\leftarrow z_3 \cdot f_3'(y_2) & \frac{\partial s}{\partial w_2} &= z_2 \cdot \frac{\partial f_2}{\partial w_2}(y_1) \\ z_1 &\leftarrow z_2 \cdot f_2'(y_1) & \frac{\partial s}{\partial w_1} &= z_1 \cdot \frac{\partial f_1}{\partial w_1}(x) \end{aligned}$$

Gradients of builtin functions If the functions in the above example are all builtin functions, then the gradient derivation is complete and the resulting statements correspond to calls to forward inference and backward gradient calls of the builtin functions.

For example, if f_2 represents convolution function in Cudnn library, then we have the following correspondence between library functions and tensor expressions.

convolution_forward(y_1)	$f_2(y_1)$
convolution_backward_data(z_2, y_1)	$z_2 \cdot f_2'(y_1)$
convolution_backward_filter(z_2, y_1)	$z_2 \cdot \dfrac{\partial f_2}{\partial w_2}(y_1)$

Gradient of tensors and scalars We do not have high-level functions for all types of DNN layers and some of them have to be implemented using low-level functions. For example, affine transformation is implemented with a matrix product and a sum. For these functions, we use tensor/scalar gradient derivation rules to derive gradients of the form $f'(y)$ and $\dfrac{\partial f}{\partial w}(y)$.

Note that in the rules below, the partial-derivative operator ∂ extends to the rightmost expressions.

Gradient derivation rules

$$\frac{\partial\ t_1 + t_2}{\partial y} = \frac{\partial t_1}{\partial y} + \frac{\partial t_2}{\partial y} \tag{2}$$

$$\frac{\partial\ s \cdot t}{\partial y} = s \cdot \frac{\partial t}{\partial y} \tag{3}$$

$$\frac{\partial\ (D)\ x}{\partial y} = (D')\frac{\partial x}{\partial y} \text{ where } D' = \mathscr{T}(x) \tag{4}$$

$$\frac{\partial x}{\partial y} = \frac{\partial\ I \Rightarrow x(I)}{\partial y} \text{ where } \mathscr{D}(I) = \mathscr{T}(x) \tag{5}$$

$$\frac{\partial\ I \Rightarrow s}{\partial y} = I \Rightarrow \frac{\partial s}{\partial y} \tag{6}$$

$$\frac{\partial s}{\partial y} = I \Rightarrow \frac{\partial s}{\partial y(I)} \text{ where } \mathscr{D}(I) = \mathscr{T}(y) \tag{7}$$

The gradient derivation rules specify how gradient expressions are simplified. For example, the gradient of

$t_1 + t_2$ is the sum of the gradients of t_1 and t_2. The gradient of $s \cdot t$ is the product of s and the gradient of t, where we assume that s does not contain any tensor variables.

The gradient of a cast expression $(D)\ x$ with respect to y is $(D')\ \dfrac{\partial x}{\partial y}$, where $D' = \mathscr{T}(x)$. $(D')\ \dfrac{\partial x}{\partial y}$ is a tensor gradient such that the product of a tensor variable z and $(D')\ \dfrac{\partial x}{\partial y}$ results in a cast expression $(D')\ (z \cdot \dfrac{\partial x}{\partial y})$.

The gradient of a tensor variable x is the gradient of the tensor expression $I \Rightarrow x(I)$, where $\mathscr{D}(I) = \mathscr{T}(x)$. The gradient of $I \Rightarrow s$ is the scalar gradient $\dfrac{\partial s}{\partial x}$ indexed over the domain of I. The scalar gradient $\dfrac{\partial s}{\partial y}$ is the tensor $I \Rightarrow \dfrac{\partial s}{\partial y(I)}$, where I is a list of fresh indices and $\mathscr{D}(I) = \mathscr{T}(y)$.

Note that the more obvious reduction of $\dfrac{\partial x}{\partial y}$ is a gradient consisting of 1s if $x = y$ or a gradient consisting of 0s if $x \neq y$. However, for simplicity, we choose to handle them in the more general way through Rules 5, 6, and 7.

Scalar derivation rules The derivation of $\dfrac{\partial s}{\partial y(I)}$ is defined by Rules 8 through 15. Most of scalar derivation rules are standard except Rule 14, which says that the derivative of the sum $\sum_{I'} s$ is the sum of $\dfrac{\partial s}{\partial y(I)}$. Rule 15 says that the derivative of a tensor element $x(I)$ with respect to $x(I')$ is obtained through the auxiliary function $\mathsf{match}(I, I')$, which returns the products of some Kronecker deltas.

$$
\begin{aligned}
\mathsf{match}(i, i') &= \delta_{i\ i'} \\
\mathsf{match}(i \cdot I, i' \cdot I') &= \delta_{i\ i'} \times \mathsf{match}(I, I')
\end{aligned}
$$

The Kronecker delta $\delta_{i\ i'}$ reduces to 1 if i and i' evaluate to the same index value and it reduces to 0 otherwise.

$$
\begin{aligned}
\frac{\partial \log(s)}{\partial y(I)} &= s^{-1} \times \frac{\partial s}{\partial y(I)} && (8) \\
\frac{\partial \exp(s)}{\partial y(I)} &= \exp(s) \times \frac{\partial s}{\partial y(I)} && (9) \\
\frac{\partial\ s_1 + s_2}{\partial y(I)} &= \frac{\partial s_1}{\partial y(I)} + \frac{\partial s_2}{\partial y(I)} && (10) \\
\frac{\partial\ s_1 \times s_2}{\partial y(I)} &= s_2 \times \frac{\partial s_1}{\partial y(I)} + s_1 \times \frac{\partial s_2}{\partial y(I)} && (11) \\
\frac{s^n}{\partial y(I)} &= (n \times s^{n-1}) \times \frac{\partial s}{\partial y(I)} && (12) \\
\frac{\partial n}{\partial y(I)} &= 0 && (13) \\
\frac{\partial \sum_{I'}(s)}{\partial y(I)} &= \sum_{I'} \frac{\partial s}{\partial y(I)} && (14) \\
\frac{\partial x(I)}{\partial y(I')} &= \begin{cases} 0 & \text{if } x \neq y \\ \mathsf{match}(I, I') & \text{otherwise} \end{cases} && (15)
\end{aligned}
$$

Syntax of gradients The additional syntax for describing the gradients of scalars and tensors with respect to tensor variables can be summarized as follows, where the symbol g denotes tensor gradients.

t	::=	$\ldots$	
	$\mid$	$z \cdot g$	tensor gradient product
g	::=	$I \Rightarrow I' \Rightarrow s$	gradient expression
	$\mid$	$g_1 + g_2$	gradient sum
	$\mid$	$s \cdot g$	scalar gradient product
	$\mid$	$(D)\ g$	gradient cast
	$\mid$	$\dfrac{\partial \mathscr{F}}{\partial x}(y)$	gradient tensor application
s	::=	$\ldots$	
	$\mid$	$\delta_{i\ i'}$	Kronecker delta

5 High-level Optimization

Simplification of parameter gradients The derivation of parameter gradients $\dfrac{\partial s}{\partial w_i}$ results in tensors of the form $z \cdot g$, which should be simplified. After applying the reduction rules below, all forms of g except $\dfrac{\partial \mathscr{F}}{\partial x}(y)$ are eliminated from the parameter gradients.

$$x \cdot (I \Rightarrow I' \Rightarrow s) = I' \Rightarrow \sum_I x(I) \times s \quad (16)$$

$$x \cdot (g_1 + g_2) = x \cdot g_1 + x \cdot g_2 \quad (17)$$

$$x \cdot (s \cdot g) = s \cdot (x \cdot g) \quad (18)$$

$$x \cdot (D)\, g = (D)\, x \cdot g \quad (19)$$

The tensors can be further simplified with reduction rules below in order to remove the Kronecker deltas. These rules are designed to move the sum operator $\sum_{i_x}$ inwards as much as possible until it meets a Kronecker delta $\delta_{i\ i'}$ where i is a function f of i_x. By the syntax of index in Figure 3, f is an invertible function. Since $\delta_{i\ i'}$ equals to 1 iff i and i' reduce to the same value, which is when $i_x = f^{-1}(i')$, we can reduce $\sum_{i_x} \delta_{i\ i'} \times s$ to s with i_x in s replaced by $f^{-1}(i')$.

$$\sum_I s + s' = \sum_I s + \sum_I s' \quad (20)$$

$$\sum_{i_x} s \times s' = s \times \sum_{i_x} s' \quad i_x \text{ is not used in } s \quad (21)$$

$$\sum_{i_x} \delta_{f(i_x)\ i'} \times s = s[f^{-1}(i')/i_x] \quad (22)$$

$$\sum_{i_x} \delta_{f(i_x)\ i'} = 1 \quad (23)$$

$$s \times (s_1 + s_2) = s \times s_1 + s \times s_2 \quad (24)$$

$$s \cdot (s' \cdot t) = (s \times s') \cdot t \quad (25)$$

The reduction rules will eliminate all Kronecker deltas, which are reduced from the derivative of a tensor element with respect to another tensor element. The index variables in I in each tensor element $x(I)$ are bound by either a sum expression or a tensor expression. In the first case, the sum eliminates the deltas. In the second case, $x(I)$ appears in a tensor of the form $I_1 \Rightarrow s$, where the index variables in I are defined in I_1. The deltas are in tensor of the form $z \cdot \frac{\partial I_1 \Rightarrow s}{\partial x}$, which reduces to $z \cdot (I_1 \Rightarrow I_2 \Rightarrow \frac{\partial s}{\partial x(I_2)})$ by Rule 7, which reduces to $I_2 \Rightarrow \sum_{I_1} z(I_1) \times \frac{\partial s}{\partial x(I_2)}$ by Rule 16. After reduction, the sum operator will eliminate the deltas from $\frac{\partial s)}{\partial x(I_2)}$.

Syntax after simplification After symbolic reduction, the parameter gradients are reduced to tensors of the following syntax.

$$\begin{array}{rcll} t & ::= & \ldots & \\ & | & z \cdot \frac{\partial \mathscr{F}}{\partial x}(y) & \text{tensor gradient product} \\ i & ::= & \ldots & \\ & | & i_1 - i_2 \mid i + n & \end{array}$$

where $z \cdot \frac{\partial \mathscr{F}}{\partial x}(y)$ represents a backward gradient of $\mathscr{F}$.

Indices of the form of $i_1 - i_2$ and $i + n$ appear in the gradients of convolution after the simplification of Kronecker delta and they have the following typing rules.

$$\frac{\Gamma \vdash i_1 : d_1 \quad \Gamma \vdash i_2 : d_2}{\Gamma \vdash i_1 - i_2 : d_1 - d_2 + 1} \qquad \frac{\Gamma \vdash i : d}{\Gamma \vdash i + n : d + 2n}$$

We have defined typing rules for indices of the forms $i_1 + i_2$ and $i - n$, which are used in tensor convolution. The rule on $i_1 - i_2$ is the inverse of the rule on $i_1 + i_2$ while the rule on $i + n$ is the inverse of the rule on $i - n$.

In the current form, tensors and parameter gradients can be evaluated. However, the direct evaluation of tensor expressions of the form $I \Rightarrow s$ is inefficient. For example, to evaluate $(i_1 \cdot i_3) \Rightarrow \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2, i_3)$, we need to have a double loop (outer loop for index i_1 and inner loop for i_3) where the loop body evaluates $\sum_{i_2} x(i_1 \cdot i_2) \times w(i_2, i_3)$. A more efficient evaluation strategy is to further transform tensors into forms that can be mapped to functions in high-performance libraries.

Trivial simplification rules We also apply some obvious simplifcation rules to reduce tensors and scalar expressions that involve 1 and 0.

For example, $1 \cdot t = t$, $0 \cdot t = \mathbf{0}$, $1 \times s = s$, $0 \times s = 0$, $0 + s = s$, $s \cdot \mathbf{0} = \mathbf{0}$, and $\mathbf{0} + t = t$, where $\mathbf{0}$ represents a tensor of the form $I \Rightarrow 0$ or $(D)\, I \Rightarrow 0$.

Vectorization Tensors of the form of $\mathscr{F}(y)$ and $z \cdot \frac{\partial \mathscr{F}}{\partial x}(y)$ can be mapped to high-level functions in libraries such as Cudnn. Thus, we focus on transforming tensor expressions into the computation of vectors and matrices so that they can be mapped to low-level functions in libraries such as Cuda. We call this reduction step vectorization.

$$
\begin{array}{lcll}
I \Rightarrow s^n & = & (I \Rightarrow s)^n & & (26)\\
I \Rightarrow \exp(s) & = & \exp(I \Rightarrow s) & & (27)\\
I \Rightarrow \log(s) & = & \log(I \Rightarrow s) & & (28)\\
I \Rightarrow s_1 \times s_2 & = & s_1 \cdot (I \Rightarrow s_2) & \forall i \in \{I\}.\ i \text{ is not used in } s_1 & (29)\\
I \Rightarrow s_1 \times s_2 & = & (I \Rightarrow s_1) .* (I \Rightarrow s_2) & & (30)\\
I \Rightarrow s_1 + s_2 & = & (I \Rightarrow s_1) + (I \Rightarrow s_2) & & (31)\\
I \Rightarrow x(I) & = & x & \mathscr{T}(x) = \mathscr{D}(I) & (32)\\
\sum_I s_1 \times s_2 & = & s_1 \times \sum_I s_2 & \forall i \in \{I\}.\ i \text{ is not used in } s_1 & (33)\\
\sum_I s_1 \times s_2 & = & (I \Rightarrow s_1) \cdot (I \Rightarrow s_2) & & (34)\\
\sum_I s & = & \sum (I \Rightarrow s) & & (35)\\
I \Rightarrow \sum I' \Rightarrow s & = & \sum_{|I'|} (I \cdot I') \Rightarrow s & & (36)\\
I \Rightarrow (I' \Rightarrow s_1) \cdot (I' \Rightarrow s_2) & = & (I_1 \cdot I') \Rightarrow s_1 \ \times_{|I'|}\ (I_2 \cdot I') \Rightarrow s_2 & \begin{array}{l} I = I_1 \cdot I_2 \\ \forall i \in \{I_1\}.\ i \text{ is not used in } s_2 \\ \forall i \in \{I_2\}.\ i \text{ is not used in } s_1 \end{array} & (37)\\
I \Rightarrow s & = & I \Rightarrow (I' \Rightarrow s)(I') & \begin{array}{l} s \text{ is not a tensor element, } \{I'\} \subset \{I\} \\ \forall i \in \{I\} \backslash \{I'\}.\ i \text{ is not used in } s \end{array} & (38)
\end{array}
$$

Figure 8: Rules for vectorization, where $\{I\}$ represents the set of indices in I.

Syntax after vectorization

$$
\begin{array}{lcll}
t & ::= & \ldots & \\
 & | & z \cdot \dfrac{\partial \mathscr{F}}{\partial x}(y) & \text{tensor gradient product} \\
 & | & \exp(t) \mid \log(t) \mid t^n & \\
 & | & t_1 .* t_2 & \text{pointwise product} \\
 & | & t_1 \times_n t_2 & \text{tensor contraction} \\
 & | & \sum_n t & \text{partial sum of tensor} \\
s & ::= & \ldots & \\
 & | & t_1 \cdot t_2 & \text{inner product} \\
 & | & \sum t & \text{sum of tensor} \\
 & | & t(I) & \text{tensor element}
\end{array}
$$

The vectorization rules introduce a few types of expressions, some of which have direct correspondence to low-level functions in libraries. The expression $\exp(t)/\log(t)/t^n$ is an exponentiation/logarithm/power of tensor, which is the same as applying exponentiation/logarithm/power to the tensor elements. For these expressions, the tensor can be treated as a vector so that each of operations can be mapped to library function that applies exponentiation/logarithm/power to a vector.

The expression $t_1 .* t_2$ is a pointwise product of two tensors t_1 and t_2, which have the same type. This expression can be mapped to a library function for the inner product of two vectors.

The partial-sum expression $\sum_n t$ sums up the lower n axis of t and it has the typing rule:

$$\frac{\Gamma \vdash t : D_1 \cdot D_2 \quad |D_2| = n}{\Gamma \vdash \sum_n t : D_1}$$

The partial-sum of tensor can be mapped to the matrix product $A \times B$, where A is a $n_1 \times n_2$ matrix [4] converted from t, B is a $n_2 \times 1$ matrix consisted of 1s, and n_i is the flattened size of D_i, $\forall i \in \{1,2\}$, Converting the tensor value evaluated from t to A takes constant time since they have the same array representation.

For example, if t is a tensor with the type $3 \cdot 4 \cdot 5 \cdot 6$, then $\sum_2 t$ is a tensor of the type $3 \cdot 4$. We can convert t to a 12×30 matrix A and sum up each row of A to obtain a vector of size 12, which is the same as $\sum_2 t$ as an array.

The contraction expression $t_1 \times_n t_2$ performs inner products of the lower n axis of t_1 and t_2 and it has the typing rule:

$$\frac{\Gamma \vdash t_1 : D_1 \cdot D_3 \quad \Gamma \vdash t_2 : D_2 \cdot D_3 \quad |D_3| = n}{\Gamma \vdash t_1 \times_n t_2 : D_1 \cdot D_2}$$

[4] Matrices in this work are row-major.

The tensor contraction can be mapped to matrix product $A \times B^T$, where A is a n_1 by n_3 matrix converted from t_1, B is a n_2 by n_3 matrix converted from t_2, n_i is the flattened size of D_i for $i \in \{1,2,3\}$. Converting the tensor values evaluated from t_1 and t_2 to A and B respectively also takes constant time.

For example, if t_1 has the type $2 \cdot 3 \cdot 5 \cdot 6$ and t_2 has the type $4 \cdot 5 \cdot 6$, then $t_1 \times_2 t_2$ has the type $2 \cdot 3 \cdot 4$. We can convert t_1 to a 6×30 matrix and convert t_2 to a 4×30 matrix. Then $A \times B^T$ is a 6×4 matrix, which is the same as $t_1 \times_2 t_2$ as an array.

The expression $t_1 \cdot t_2$ is the inner product of t_1 and t_2, which have the same type. The expression $\sum t$ sums over the tensor t. The expression $t(I)$ represents the element of the tensor t.

A tensor of the form $(I_1 \cdot I_2) \Rightarrow t(I_1)$, where indices in I_2 do not appear in t, is a tensor where each element of t is replicated n_2 times and n_2 is the flattened size of $\mathscr{D}(I_2)$. This can be implemented as a matrix product $A \times B$, where A is a $n_1 \times 1$ matrix converted from t, n_1 is the flattened size of $\mathscr{D}(I_1)$, and B is a $1 \times n_2$ matrix of 1s.

A tensor of the form $(I_1 \cdot I_2) \Rightarrow t(I_2)$, where indices in I_1 do not appear in t, is a tensor that is n_1 consecutive copies of t, where n_1 is the flattened size of $\mathscr{D}(I_1)$. This can be implemented as a matrix product $A \times B$, where A is a $n_1 \times 1$ matrix of 1s, B is a $1 \times n_2$ matrix converted from t, and n_2 is the flattened size of $\mathscr{D}(I_2)$.

Vectorization rules Figure 8 shows the vectorization rules, where Rule 26 to 31 lift operators on scalars in tensor expressions to the outside so that they become operators on tensors. For example, $I \Rightarrow \exp(s)$ is a tensor expression where each element is an exponentiation of a scalar expression s. By Rule 27, this is reduced to $\exp(I \Rightarrow s)$, which is the exponentiation of the tensor $I \Rightarrow s$. In Rule 30, the product of scalars becomes the pointwise product $.*$ of tensors.

Rule 32 simplifies $I \Rightarrow x(I)$ to just x if the dimensions of I are the same as the type of x. Rule 33 factors out scalars independent of the sum indices. Rule 34 reduces the sum of scalar products to the inner product of two tensors. The sum of scalar is reduced to sum of tensor by Rule 35. Rule 36 turns a tensor expression that contains a tensor sum into the partial sum of a tensor.

Rule 37 converts $I \Rightarrow (I' \Rightarrow s_1) \cdot (I' \Rightarrow s_2)$ into tensor contraction if I can be divided into I_1 and I_2 without changing order of indices so that indices in I_1 are not used in s_2 and indices in I_2 are not in s_1. Note that convolution can be converted to matrix product through Rule 34 and 37. However, there are more efficient implementation in existing libraries such as Cudnn so that convolution should be treated as builtin functions.

In an tensor expression $I \Rightarrow s$, not all indices in I are used in s. Rule 38 factors out the indices not used in s and reduces the tensor expression to $I \Rightarrow (I' \Rightarrow s)(I')$ so that all indices in I' are used in s. $I' \Rightarrow s$ may be reduced further and evaluated separately.

For example,

$$
\begin{aligned}
& (i_1 \cdot i_2) \Rightarrow \sum i_3 \Rightarrow x(i_1 \cdot i_3) \\
= \;& (i_1 \cdot i_2) \Rightarrow (i_1 \Rightarrow \sum i_3 \Rightarrow x(i_1 \cdot i_3))(i_1) && \text{by Rule 38} \\
= \;& (i_1 \cdot i_2) \Rightarrow (\sum_1 (i_1 \cdot i_3) \Rightarrow x(i_1 \cdot i_3))(i_1) && \text{by Rule 36} \\
= \;& (i_1 \cdot i_2) \Rightarrow (\sum_1 x)(i_1) && \text{by Rule 32}
\end{aligned}
$$

where $\sum_1 (i_1 \cdot i_3) \Rightarrow x(i_1 \cdot i_3)$ partially sums up $(i_1 \cdot i_3) \Rightarrow x(i_1 \cdot i_3)$ with its lower 1 axis, which is i_3.

5.1 Examples

In this section, we explain the application of rules for the gradient derivation, symbolic reduction, and vectorization using examples from Section 3.4.

Affine transformation In Figure 9, the tensor t_y is the output of affine transformation with input x, weight w, and bias b. Figure 9 also shows the reduction of $\frac{\partial t_y}{\partial x}$ and the backward gradient of x, which is $z_y \cdot \frac{\partial t_y}{\partial x}$, where z_y is the backward gradient of t_y.

Flattening Below t_y is the result of flattening the lower 3 axis of the tensor x, z_y is the backward gradient of t_y, and z_x is the backward gradient of x.

$$
\begin{aligned}
t_y &= (d_1 \cdot (d_2 \times d_3 \times d_4))\, x \\
z_x &= z_y \cdot \frac{\partial t_y}{\partial x} \\
&= z_y \cdot \frac{(d_1 \cdot (d_2 \times d_3 \times d_4))\, x}{\partial x} \\
&= z_y \cdot (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\, \frac{\partial x}{\partial x} && \text{by Rule 4} \\
&= (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\, z_y \cdot \frac{\partial x}{\partial x} && \text{by Rule 19} \\
&= (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\, z_y
\end{aligned}
$$

where $\mathscr{D}(x) = d_1 \cdot d_2 \cdot d_3 \cdot d_4$.

The last reduction step, though obvious, takes a few steps to complete by following the rules:

$$
\begin{aligned}
z_y \cdot \frac{\partial x}{\partial x} &= z_y \cdot (I \Rightarrow I' \Rightarrow \frac{\partial x(I)}{\partial x(I')}) && \text{by Rule 5, 6, 7} \\
&= I' \Rightarrow \textstyle\sum_I (z_y(I) \times \frac{\partial x(I)}{\partial x(I')}) && \text{by Rule 16} \\
&= I' \Rightarrow z_y(I') && \text{by Rule 15, 22} \\
&= z_y && \text{by Rule 32}
\end{aligned}
$$

where $\mathscr{D}(I) = \mathscr{D}(I') = \mathscr{T}(x)$.

$$
\begin{aligned}
t_y &= (i_1 \cdot i_3) \Rightarrow \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2 \cdot i_3) + b(i_1) \\
\\
\frac{\partial t_y}{\partial x} &= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \frac{\partial \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2 \cdot i_3) + b(i_1)}{\partial x(i'_1 \cdot i'_2)} && \text{by Rule 6, 7} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \sum_{i_2} \frac{\partial x(i_1 \cdot i_2)}{\partial x(i'_1 \cdot i'_2)} \times w(i_2 \cdot i_3) && \text{by Rule 10, 11, 15} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \sum_{i_2} \delta_{i_1\ i'_1} \times \delta_{i_2\ i'_2} \times w(i_2 \cdot i_3) && \text{by Rule 15} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times \sum_{i_2} \delta_{i_2\ i'_2} \times w(i_2 \cdot i_3) && \text{by Rule 21} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3) && \text{by Rule 22} \\
\\
z_x = z_y \cdot \frac{\partial t_y}{\partial x} &= z_y \cdot ((i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3)) \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_1} \sum_{i_3} z_y(i_1 \cdot i_3) \times \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3) && \text{by Rule 16} \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_1} \delta_{i_1\ i'_1} \times \sum_{i_3} z_y(i_1 \cdot i_3) \times w(i'_2 \cdot i_3) && \text{by Rule 21} \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_3} z_y(i'_1 \cdot i_3) \times w(i'_2 \cdot i_3) && \text{by Rule 22} \\
&= (i'_1 \cdot i'_2) \Rightarrow (i_3 \Rightarrow z_y(i'_1 \cdot i_3)) \cdot (i_3 \Rightarrow w(i'_2 \cdot i_3)) && \text{by Rule 34} \\
&= ((i'_1 \cdot i_3) \Rightarrow z_y(i'_1 \cdot i_3)) \times_1 ((i'_2 \cdot i_3) \Rightarrow w(i'_2 \cdot i_3)) && \text{by Rule 37} \\
&= z_y \times_1 w && \text{by Rule 32}
\end{aligned}
$$

Figure 9: Gradient derivation of affine transformation

Convolution Let t_y be the result of a convolution layer with stride 1 and padding 0 defined in Figure 10, where x is input, w is weight, and b is bias. If z_y is the backward gradient of t_y, then the gradient of t_y against x is $\frac{\partial t_y}{\partial x}$ and the backward gradient of x is $z_y \cdot \frac{\partial t_y}{\partial x}$, which can be derived as in Figure 10.

Loss The loss expression s and its backward gradient z_x are defined in Figure 11, where y is the ground truth and x is the predicted classification.

Log softmax The gradient derivation of log softmax is shown in Figure 12. After common subexpression elimination of both t_y and z_x, we can have the following:

$$
\begin{aligned}
x_1 &\leftarrow \exp(x) \\
x_2 &\leftarrow \sum_1 x_1 \\
t_y &= x - f(\log(x_2)) \\
z_x &= z_y - f(\sum_1 z_y) .* x_1 .* (f(x_2))^{-1}
\end{aligned}
$$

where $f(t)$ is defined as $(i_n \cdot i_k) \Rightarrow t(i_n)$.

Note that in most cases, z_y is the backward gradient of the loss expression, which is $(-\frac{1}{d_n}) \cdot y$ and each row of y is a unit vector. With this knowledge, we can reduce z_x to $\frac{1}{d_n} \cdot (x_1 .* (f(x_2))^{-1} - y)$, which is $\frac{1}{d_n} \cdot (exp(t_y) - y)$. However, this reduction is based on domain knowledge and is out of the scope of rule-based reduction.

6 Low-level Optimization

Inlining Since the generated code calls functions in libraries such as Cuda and Cudnn, there are opportunities to take advantage of the library functions that perform multiple computation the same time.

A Cuda or Cudnn function takes a number of tensor and scalar parameters and returns a number that either means success or is a failure code. The tensor parameters can be for input, for output, or both. For instance, a Cudnn function f_{cudnn} for computing backward gradient

$$t_y = (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow \sum_{i_c}\sum_{i_u}\sum_{i_v} x(i_n \cdot i_c \cdot i_h + i_u \cdot i_w + i_v) \times w(i_k \cdot i_c \cdot i_u \cdot i_v) + b(i_k)$$

$$\frac{\partial t_y}{\partial x} = (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow \frac{\partial \sum_{i_c}\sum_{i_u}\sum_{i_v} x(i_n \cdot i_c \cdot i_h + i_u \cdot i_w + i_v) \times w(i_k \cdot i_c \cdot i_u \cdot i_v) + b(i_k)}{\partial x} \quad \text{by Rule 6}$$

$$= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_c}\sum_{i_u}\sum_{i_v} \delta_{i_n\ i'_n} \times \delta_{i_c\ i'_c} \times \delta_{i_h+i_u\ i_r} \times \delta_{i_w+i_v\ i_s} \times w(i_k \cdot i_c \cdot i_u \cdot i_v) \quad \text{by Rule 7, 10, 11, 15}$$

$$= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times \sum_{i_c}\sum_{i_u}\sum_{i_v} \delta_{i_c\ i'_c} \times \delta_{i_h+i_u\ i_r} \times \delta_{i_w+i_v\ i_s} \times w(i_k \cdot i_c \cdot i_u \cdot i_v) \quad \text{by Rule 21}$$

$$= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times w(i_k \cdot i_c \cdot i_u \cdot i_v)[i'_c/i_c,\ i_r - i_h/i_u,\ i_s - i_w/i_v] \quad \text{by Rule 22}$$

$$= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)$$

$$z_x = z_y \cdot \frac{\partial t_y}{\partial x}$$

$$= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_n}\sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i_n \cdot i_k \cdot i_h \cdot i_w) \times \delta_{i_n\ i'_n} \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) \quad \text{by Rule 16}$$

$$= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_n}(\sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i_n \cdot i_k \cdot i_h \cdot i_w) \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)) \times \delta_{i_n\ i'_n} \quad \text{by Rule 21}$$

$$= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i'_n \cdot i_k \cdot i_h \cdot i_w) \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) \quad \text{by Rule 22}$$

$$= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow ((i_k \cdot i_h \cdot i_w) \Rightarrow z_y(i'_n \cdot i_k \cdot i_h \cdot i_w)) \cdot ((i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)) \quad \text{by Rule 34}$$

$$= (i'_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow z_y(i'_n \cdot i_k \cdot i_h \cdot i_w) \times_3 (i'_c \cdot i_r \cdot i_s \cdot i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) \quad \text{by Rule 37}$$

$$= z_y \times_3 (i'_c \cdot i_r \cdot i_s \cdot i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) \quad \text{by Rule 32}$$

Figure 10: Gradient derivation of convolution

f has the form:

$$f_{cudnn}(x_1, \ldots, x_n, z_y, \alpha, \beta)$$

which computes

$$z_y \leftarrow \alpha \times f(x_1, \ldots, x_n) + \beta \times z_y$$

where $x_1, \ldots, x_n$ are input tensors, z_y is the input/output tensor, and α, β are scaling factors.

To call this function, we pass 1 to α and 0 to β so that it effectively computes:

$$z_y \leftarrow f(x_1, \ldots, x_n)$$

However, z_y is often used in gradient update of the form:

$$y \leftarrow \alpha' \times z_y + \beta' \times y$$

Running the two statements separately not only consumes more time but also extra memory to hold z_y. In this case, it is more efficient to inline the update in the backward gradient computation to save time and space.

$$f_{cudnn}(x_1, \ldots, x_n, y, \alpha', \beta')$$

which computes

$$y \leftarrow \alpha' \times f(x_1, \ldots, x_n) + \beta' \times y$$

Update statements can also be inlined into other GEMM (general matrix multiplication) calls such as matrix product. In general, inlining an update statement such as $y = \alpha' \times z_y + \beta' \times y$ is possible only if z_y is not used in other computation.

Other than updates, we can also inline plus operations for some computation. For example, statements like $y \leftarrow y_1 + y_2$ and $y_2 \leftarrow f(x_1, \ldots, x_2)$ can be merged into

$$y_1 \leftarrow 1 \times f(x_1, \ldots, x_2) + 1 \times y_1$$

if y_2 are not used in other computation and y_1 is not used in subsequent computation since it will be overwritten. Also, any occurrences of y is replaced by y_1. Note that $y_1 \leftarrow 1 \times f(x_1, \ldots, x_2) + 1 \times y_1$ corresponds to the library call $f_{cudnn}(x_1, \ldots, x_2, y_1, 1, 1)$, where y_1 is the input/output parameter.

In-place computation The results of the operations such as tensor sum, tensor scalar product, point-wise tensor products, and the forward inference and backward gradient of activation layers can be stored in the memory of their input tensors if the inputs are not used in subsequent computation. In-place updates like these are

$$
\begin{aligned}
s &= \sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times x(i_n \cdot i_k) \times (-\frac{1}{d_n}) \\
z_x = \frac{\partial s}{\partial x} &= (i'_n \cdot i'_k) \Rightarrow \sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times \delta_{i_n\, i'_n} \times \delta_{i_k\, i'_k} \times (-\frac{1}{d_n}) && \text{by Rule 7, 11, 15} \\
&= (i'_n \cdot i'_k) \Rightarrow y(i'_n \cdot i'_k) \times (-\frac{1}{d_n}) && \text{by Rule 22} \\
&= (-\frac{1}{d_n}) \cdot (i'_n \cdot i'_k) \Rightarrow y(i'_n \cdot i'_k) && \text{by Rule 29} \\
&= (-\frac{1}{d_n}) \cdot y && \text{by Rule 32}
\end{aligned}
$$

Figure 11: Gradient derivation of loss expression

possible since the input tensor has the same size as the output tensor.

For example, if $y \leftarrow f(x)$ computes the forward inference of an activation layer with input x, then the actual call to the Cudnn function has the form of

$$y \leftarrow 1 \times f(x) + 0 \times y$$

We can avoid allocating memory for y by rewriting it as

$$x \leftarrow 1 \times f(x) + 0 \times x$$

For tensor sum, $y \leftarrow x_1 + x_2$, we write it as

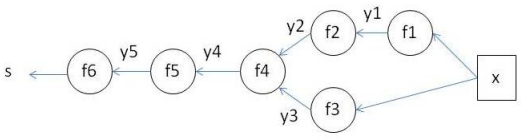
$$x_1 \leftarrow x_1 + x_2$$

To perform in-place computation, the overwritten tensor x must not be used in later statements. Alternatively, we can always use in-place computation for expressions such as tensor sum and if the overwritten tensor x_1 is used in a later statement, we make a copy of x_1 in that statement.

7 Code Scheduling

Next step is to schedule computation based on def-use dependency (i.e. the use of a variable must follow its definition) and also use heuristics to reduce peak memory usage. There are many possible schedules that satisfy the def-use dependency requirement. Since most of the statements allocate memory to store temporary result, some scheduling will result in higher peak memory usage than necessary.

To illustrate the scheduling process, consider the following network.



where f_1 to f_5 are tensor to tensor functions and f_6 is a tensor to scalar function. Also, w_1, w_2, w_3, and w_5 are the weight parameters in f_1, f_2, f_3, and f_5 respectively. The scalar expression s represents the loss of a network with input x, where

$$
\begin{array}{llllll}
y_1 & \leftarrow & f_1(x) & y_2 \leftarrow f_2(y_1) & y_3 \leftarrow f_3(x) \\
s & \leftarrow & f_6(y_5) & y_5 \leftarrow f_5(y_4) & y_4 \leftarrow f_4(y_2, y_3)
\end{array}
$$

Direct gradient derivation of the loss expression against each parameter results in the following equations, which contain multiple redundant computation steps.

$$
\begin{aligned}
\frac{\partial s}{\partial w_1} &= f'_6(y_5) \cdot f'_5(y_4) \cdot \frac{\partial f_4}{\partial x_1}(y_2, y_3) \cdot f'_2(y_1) \cdot \frac{\partial f_1}{\partial w_1}(x) \\
\frac{\partial s}{\partial w_2} &= f'_6(y_5) \cdot f'_5(y_4) \cdot \frac{\partial f_4}{\partial x_1}(y_2, y_3) \cdot \frac{\partial f_2}{\partial w_2}(y_1) \\
\frac{\partial s}{\partial w_3} &= f'_6(y_5) \cdot f'_5(y_4) \cdot \frac{\partial f_4}{\partial x_2}(y_2, y_3) \cdot \frac{\partial f_3}{\partial w_3}(x) \\
\frac{\partial s}{\partial w_5} &= f'_6(y_5) \cdot \frac{\partial f_5}{\partial w_5}(y_4)
\end{aligned}
$$

Note that the function f_4 has two parameters, which we assume to be x_1 and x_2. While for functions of one parameter such as f_2, we write its derivative as f'_2, we write the derivatives of f_4 as $\frac{\partial f_4}{x_1}$ and $\frac{\partial f_4}{x_2}$.

If we use reverse accumulation method, we can obtain

$$
\begin{array}{rcll}
t_y & = & x-(i_n\cdot i_k)\Rightarrow \log(\sum_{i_l}\exp(x(i_n\cdot i_l))) & = & x-f(i_n\Rightarrow \log(\sum_{i_l}\exp(x(i_n\cdot i_l)))) & \text{by Rule 38}\\
 & = & x-f(\log(i_n\Rightarrow \sum(i_l\Rightarrow \exp(x(i_n\cdot i_l))))) & = & x-f(\log(\sum_1(i_n\cdot i_l)\Rightarrow \exp(x(i_n\cdot i_l)))) & \text{by Rule 28, 36}\\
 & = & x-f(\log(\sum_1\exp((i_n\cdot i_l)\Rightarrow x(i_n\cdot i_l)))) & = & x-f(\log(\sum_1\exp(x))) & \text{by Rule 27, 32}
\end{array}
$$

$$
\begin{aligned}
\frac{\partial t_y}{\partial x} &= \frac{\partial(x-(i_n\cdot i_k)\Rightarrow \log(\sum_{i_l}\exp(x(i_n\cdot i_l))))}{\partial x}\\
&= ((i_n\cdot i_k)\Rightarrow (i'_n\cdot i'_k)\Rightarrow \frac{\partial x(i_n\cdot i_k)}{\partial x(i'_n\cdot i'_k)}) - ((i_n\cdot i_k)\Rightarrow (i'_n\cdot i'_k)\Rightarrow \frac{\sum_{i_l}\exp(x(i_n\cdot i_l))\times \frac{\partial x(i_n\cdot i_l)}{\partial x(i'_n\cdot i'_k)}}{\sum_{i_l}\exp(x(i_n\cdot i_l))}) && \text{by Rule 6, 7, 8, 9}\\
&= ((i_n\cdot i_k)\Rightarrow (i'_n\cdot i'_k)\Rightarrow \delta_{i_n\ i'_n}\times\delta_{i_l\ i'_k}) - ((i_n\cdot i_k)\Rightarrow (i'_n\cdot i'_k)\Rightarrow \frac{\exp(x(i_n\cdot i'_k))\times\delta_{i_n\ i'_n}}{\sum_{i_l}\exp(x(i_n\cdot i_l))}) && \text{by Rule 15, 22}\\
z_x &= z_y\cdot\frac{\partial t_y}{\partial x}\\
&= ((i'_n\cdot i'_k)\Rightarrow z_y(i'_n\cdot i'_k)) - ((i'_n\cdot i'_k)\Rightarrow \sum_{i_n}\sum_{i_k} z_y(i_n\cdot i_k)\times\frac{\exp(x(i_n\cdot i'_k))\times\delta_{i_n\ i'_n}}{\sum_{i_l}\exp(x(i_n\cdot i_l))}) && \text{by Rule 16, 22}\\
&= z_y-(i'_n\cdot i'_k)\Rightarrow \sum_{i_k} z_y(i'_n\cdot i_k)\times\frac{\exp(x(i'_n\cdot i'_k))}{\sum_{i_l}\exp(x(i'_n\cdot i_l))} && \text{by Rule 32, 21, 22}\\
&= z_y-(i'_n\cdot i'_k)\Rightarrow \sum_{i_k} z_y(i'_n\cdot i_k)\ .*\ (i'_n\cdot i'_k)\Rightarrow \exp(x(i'_n\cdot i'_k))\ .*\ (i'_n\cdot i'_k)\Rightarrow (\sum_{i_l}\exp(x(i'_n\cdot i_l)))^{-1} && \text{by Rule 30}\\
&= z_y-(i'_n\cdot i'_k)\Rightarrow \sum_{i_k} z_y(i'_n\cdot i_k)\ .*\ \exp((i'_n\cdot i'_k)\Rightarrow x(i'_n\cdot i'_k))\ .*\ ((i'_n\cdot i'_k)\Rightarrow \sum_{i_l}\exp(x(i'_n\cdot i_l)))^{-1} && \text{by Rule 27, 26}\\
&= z_y-((i'_n\cdot i'_k)\Rightarrow \sum i_k\Rightarrow z_y(i'_n\cdot i_k))\ .*\ \exp(x)\ .*\ ((i'_n\cdot i'_k)\Rightarrow \sum i_l\Rightarrow \exp(x(i'_n\cdot i_l)))^{-1} && \text{by Rule 32, 35}\\
&= z_y-f(i'_n\Rightarrow \sum i_k\Rightarrow z_y(i'_n\cdot i_k))\ .*\ \exp(x)\ .*\ (f(i'_n\Rightarrow \sum i_l\Rightarrow \exp(x(i'_n\cdot i_l))))^{-1} && \text{by Rule 38}\\
&= z_y-f(\sum_1(i'_n\cdot i_k)\Rightarrow z_y(i'_n\cdot i_k))\ .*\ \exp(x)\ .*\ (f(\sum_1(i'_n\cdot i_l)\Rightarrow \exp(x(i'_n\cdot i_l))))^{-1} && \text{by Rule 36}\\
&= z_y-f(\sum_1 z_y).*\exp(x).*(f(\sum_1\exp((i'_n\cdot i_l)\Rightarrow x(i'_n\cdot i_l))))^{-1} && \text{by Rule 32, 27}\\
&= z_y-f(\sum_1 z_y).*\exp(x).*(f(\sum_1\exp(x)))^{-1} && \text{by Rule 32}
\end{aligned}
$$

Figure 12: Gradient derivation of log softmax, where $f(t)$ is defined as $(i_n\cdot i_k)\Rightarrow t(i_n)$.

the following equations without redundancy.

$$
\begin{aligned}
z_1 &\leftarrow z_2\cdot f'_2(y_1)\\
z_2 &\leftarrow z_4\cdot\frac{\partial f_4}{\partial x_1}(y_2,y_3)\\
z_3 &\leftarrow z_4\cdot\frac{\partial f_4}{\partial x_2}(y_2,y_3)\\
z_4 &\leftarrow z_5\cdot f'_5(y_4)\\
z_5 &\leftarrow f'_6(y_5)\\
z_{w_1} &\leftarrow \frac{\partial s}{\partial w_1}=z_1\cdot\frac{\partial f_1}{\partial w_1}(x)\\
z_{w_2} &\leftarrow \frac{\partial s}{\partial w_2}=z_2\cdot\frac{\partial f_2}{\partial w_2}(y_1)\\
z_{w_3} &\leftarrow \frac{\partial s}{\partial w_3}=z_3\cdot\frac{\partial f_3}{\partial w_3}(x)\\
z_{w_5} &\leftarrow \frac{\partial s}{\partial w_5}=z_5\cdot\frac{\partial f_5}{\partial w_5}(y_4)
\end{aligned}
$$

For this example, the definition of z_{w_5} is ready to compute after z_5 is available. However, if we schedule the definition of z_{w_5} after z_{w_1}, then some dependencies of the definition of z_{w_5} (i.e. z_5 and y_4) will be held in memory longer than necessary. To reduce peak memory usage, it is preferable to schedule a definition closer to its uses to reduce the overlap between holding the result of the definition in memory and other computation.

However, the dependency relations of the statements form a *directed acyclic graph* (DAG). The problem of finding an optimal schedule of computation in terms of minimizing the memory consumption in a DAG is NP-complete even when all graph nodes have the same size [41]. Since a DL network can have thousands of statements, it would be too time-consuming to find the

most memory efficient schedule. Note that there exists a polynomial time solution [4] for optimal scheduling if the dependency relation is a tree, which unfortunately is not the case here. Therefore, we adopt a simple heuristics by scheduling statements based on a definition of height in the dependency graph.

1. Let statements form DAG (V,E) based on dependency – $(s_1,s_2) \in E$ iff s_2 depends on s_1. We say s_1 is the parent of s_2 and s_2 is the child of s_1 in this DAG.
2. Height of a node is initialized to 0 if it has no parent. For other node, initialize its height as the maximum height of its parents + 1.
3. For each parameter gradient z_{w_i}, in increasing order of its height,
   (a) schedule the ancestor nodes of z_{w_i} that have not been computed in increasing order of their heights.
   (b) schedule to compute z_{w_i}

The purpose of this modified definition of height is to make sure that a statement is scheduled to run only if its result is ready for use by at least one statement. In other words, we will not hold results in memory before they are ready for use.

Finally, tensors that hold intermediate results can be deallocated at the earliest point that it is no longer used. In the scheduled statements in Figure 13, we show the tensors that are alive after executing each statement, where we do not list parameter gradients such as z_{w_5} since they will be used for updating parameters. In our empirical evaluation, this algorithm is fast and yields memory-efficient scheduling.

Statement	Height	Live Variables
$y_1 \leftarrow f_1(x)$	0	y_1
$y_3 \leftarrow f_3(x)$	0	y_1, y_3
$y_2 \leftarrow f_2(y_1)$	1	y_1, y_2, y_3
$y_4 \leftarrow f_4(y_2, y_3)$	2	y_1, y_2, y_3, y_4
$y_5 \leftarrow f_5(y_4)$	3	y_1, y_2, y_3, y_4, y_5
$z_5 \leftarrow f_6'(y_5)$	4	y_1, y_2, y_3, y_4, z_5
$z_{w_5} \leftarrow z_5 \cdot \frac{\partial f_5}{\partial w_5}(y_4)$	5	y_1, y_2, y_3, y_4, z_5
$z_4 \leftarrow z_5 \cdot f_5'(y_4)$	5	y_1, y_2, y_3, z_4
$z_3 \leftarrow z_4 \cdot \frac{\partial f_4}{\partial x_2}(y_2, y_3)$	6	y_1, y_2, y_3, z_4, z_3
$z_{w_3} \leftarrow z_3 \cdot \frac{\partial f_3}{\partial w_3}(x)$	7	y_1, y_2, y_3, z_4
$z_2 \leftarrow z_4 \cdot \frac{\partial f_4}{\partial x_1}(y_2, y_3)$	6	y_1, z_2
$z_{w_2} \leftarrow z_2 \cdot \frac{\partial f_2}{\partial w_2}(y_1)$	7	y_1, z_2
$z_1 \leftarrow z_2 \cdot f_2'(y_1)$	7	z_1
$z_{w_1} \leftarrow z_1 \cdot \frac{\partial f_1}{\partial w_1}(x)$	8	

Figure 13: Scheduled statements, their heights, and live variables at each statement.

8 Code Generation and Runtime

For each DNN, we generate a Java class that contains a method to compute the forward inference and a method to compute the backward gradient by calling Cuda and Cudnn functions. Users can modify the generated Java source to implement any learning strategies. While the backward gradient method contains all the computation of the forward inference method, the former does not call the latter since many local variables of the forward inference computation are used in the backward gradient computation and these local variables must be deallocated as soon as possible.

The generated class stores the network parameters as fields, which are either initialized by specified strategies or loaded from disk. The backward gradient method updates these parameter fields each time it is called. When training is completed, the parameter tensors are saved to disk through Java class serialization.

We invoke Cuda and Cudnn functions through some wrapper classes. For instance, the wrapper class for convolution contains calls for convolution forward, backward gradient of data and filter. The instances of these wrapper classes can be reused so that they are stored in the fields of the generated class as well. Some wrapper classes for layers such as batch-norm contain persistent states that can be saved for later use.

Java API The generated Java source program uses the DeepDSL Java API to call Cuda/Cudnn functions through JCuda. Two of the main classes in the Java API are `JTensorFloat` and `JCudaTensor`, which implement tensor computation in CPU and GPU respectively. The `JTensorFloat` class is responsible for storing training data, initializing network parameters, loading saved parameters from files, and saving trained parameters into files.

The generated Java program automatically saves trained network parameters into files by serializing the `JTensorFloat` objects that store these parameters in a

designated directory. When the user restarts the same Java program, the program will first attempt to load network parameters from the files in the same directory and initialize the parameters as specified if the files are not found. The saved network parameters can also be used for inference.

A `JTensorFloat` object can be converted to a `JCudaTensor` object by copying to GPU memory. `JCudaTensor` is used for GPU computation and it can be converted to `JTensorFloat` by copying to CPU memory. The `JCudaTensor` class also manages GPU memory usage in two modes. In the memory efficient mode, tensors (and convolution workspace) are dynamically allocated and deallocated in GPU memory. In the runtime efficient mode, tensors are stored in a reusable tensor memory pool and the convolution workspace is shared and always in memory. The tensors and the convolution workspace are deallocated at the end of the program. In memory efficient mode, less GPU memory is required but with the cost of higher runtime overhead.

DL network layers supported by Cudnn library are accessed through a small set of classes such as `JCudnnConvolution`, which calls Cudnn functions through JCuda. Users can change parameters of these classes directly for low-level control. For example, users can set a limit on the total convolution workspace by modifying a field in `JCudnnConvolution`. Users can also modify fields in `JCudnnBatchNorm` class to change how the running mean and variance are computed.

Runtime memory management Allocating and deallocating memory in Cuda can incur significant overhead. Therefore, it is preferable to avoid repeated allocation and deallocation by reusing existing tensor memory. To this end, we can cache tensor memory by maintaining a pool of allocated memory segments with known sizes. Each time a tensor is freed, its memory is returned to the pool and each time a tensor is allocated, the pool is checked for memory segment of sufficient size. New memory is allocated only when the pool does not have memory segment of suitable size. Using this strategy, user can observe GPU memory increases during the first iteration of a training loop and the memory stabilizes once it reached its peak. The memory segments in the pool are freed at the end of the program.

To reduce peak memory consumption, we can stop using tensor caching and allow tensor memory be dynamically allocated and deallocated.

Other than tensor objects, another major source of memory consumption is convolution workspace, which can be very large. Since the convolution operations run sequentially, we can make them share a cached workspace. The runtime efficient method is to first find out the largest convolution workspace and allocate that much memory so that it can be used by any convolution operations. A more memory-efficient but slower method is to dynamically allocate convolution workspace before it is needed and deallocate it right after.

Static memory analysis An advantage of DeepDSL is that it can analyze the memory usage at each computation step statically. Once the statements from backward gradients are derived and scheduled, we can calculate the tensor memory and workspace required for running each statements. Based on this, we can find out the current memory consumption at each statement depending on whether tensor memory and workspace memory is cached. Since we can determine the peak memory consumption for a DNN based on runtime memory management strategy, we can statically decide whether it is possible to run a training program on a particular GPU or use more memory efficient runtime strategy.

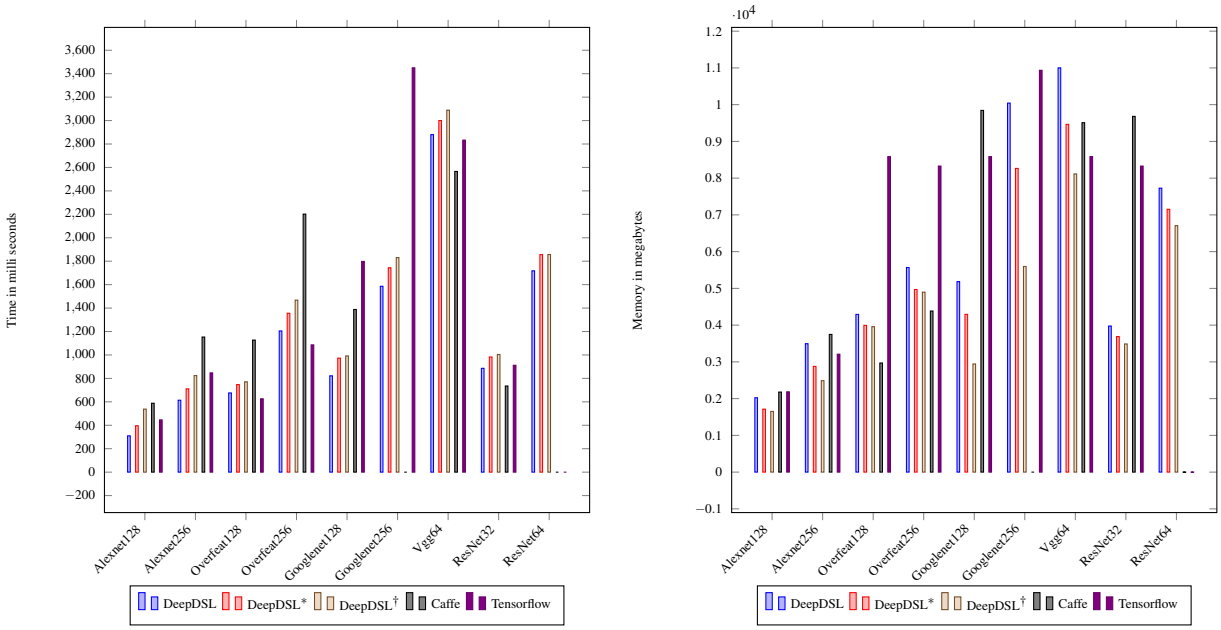
9 Performance

We compared the runtime (Figure 14a) and memory performance (Figure 14b) of DeepDSL with Caffe and Tensorflow by running several well-known DL networks on a Linux server with a single Nvidia K40c GPU. The networks include Alexnet [26], Overfeat [40], Googlenet [44], Vgg [42], and Deep residual network (Resnet) [18]. We ran DeepDSL in both runtime efficient mode (denoted as DeepDSL) and memory efficient mode (denoted as DeepDSL* and DeepDSL†) to compare the tradeoff between time and space.

When compared with Caffe, DeepDSL is 88% faster in Alexnet, 77% faster in Overfeat, and 69% faster in Googlenet; DeepDSL is 11% slower in Vgg and 17% slower in Resnet. When compared with Tensorflow, DeepDSL is 41% faster in Alexnet, 118% faster in Googlenet, and 3% faster in ResNet; DeepDSL is 8% slower in Overfeat and 2% slower in Vgg. Caffe ran out of memory for Googlenet of batch size 256 and for Resnet of batch size 64. Tensorflow ran out of memory for Resnet of batch size 64.

When compared with Caffe, DeepDSL uses 8% less memory in Alexnet, 48% less in Googlenet, and 59% less in Resnet; DeepDSL uses 44% more memory in Overfeat and 16% more in Vgg. Note that while DeepDSL uses 44% more memory in Overfeat than Caffe, DeepDSL is also 77% faster. DeepDSL is more memory efficient than Tensorflow in all cases except Vgg.

When compared with DeepDSL in runtime efficient mode, DeepDSL* (no tensor caching) saves 17% memory for Alexnet, 10% for Overfeat, 18% for Googlenet, 14% for Vgg, and 9% for Resnet. The runtime overhead of DeepDSL* is 18% for Alexnet, 10% for Over-



(a) Runtime cost (One forward/backward iteration)

(b) Peak GPU memory consumption

Figure 14: Runtime and memory comparison of DeepDSL, Tensorflow, and Caffe, where the network names are followed by the batch size. DeepDSL* and DeepDSL† are performance without tensor cache and without tensor & workspace cache respectively. Caffe failed to run GoogleNet (batch 256) and both Caffe and Tensorflow failed to run ResNet (batch 64) due to GPU memory exhaustion.

feat, 12% for Googlenet, 4% for Vgg, and 9% for Resnet. DeepDSL† reduces memory further by allocating convolution workspace dynamically. Significant reduction is achieved for Googlenet (44% less than DeepDSL) and Vgg (17% less) with modest runtime overhead. Note that memory reduction is less significant if convolution workspace is a large portion of overall GPU memory (e.g. 63% of memory use of Overfeat (batch 128) is convolution workspace.)

10 Related work

In this section, we review popular DL frameworks as well as tools that exhibit some unique features by focusing on DL network representation, optimization, and job scheduling. We also review related work on automatic differentiation as well as DSLs on scientific computing with tensors.

10.1 DL frameworks

Caffe [21] implements a DNN as a DAG by connecting DL network layers with the "Blob" (a 4D tensor array) construct. Caffe uses the layer-wise coarse-grained tensor arithmetic computation approach and its node in the computational graph is the layer. It predicts the amount of memory that is needed for a layer computation and uses that information to reserve memory in host or GPU. Caffe's core implementation is written in pure C++ and achieves similar performance as that of native code in its CPU or GPU versions (this also means each new layer requires new function implementations for both CPU and GPU). The more current Caffe2 [15] improves Caffe in aspects such as first-class support for large-scale distributed training using Gloo [19], a communications library for multi-machine training and Nvidia's NCCL[35] for multi-GPU communications. Caffe2 also leverages Redis [38], an fast in-memory database that is often used as cache or message broker, to facilitate management of nodes in distributed training. Finally, Caffe2 support mobile deployment and can run models on lower powered devices.

Theano [8][46] is a software library typically compiles and optimizes the computation graph written in Python and generates C and/or Cuda code to perform the computation. Theano derives gradients on computation graphs that represent DNNs. It optimizes the computation graphs using pattern matching on subgraphs for redundancy removal, numerical stabilization, and code specialization. In comparison, DeepDSL represents DNNs at language level as DSL expressions, where gra-

dient derivation and optimization are based on recursive application of reduction rules, which is more systematic. In addition, the result of optimization for Theano is compiled program while the result of optimization for DeepDSL remains a DSL program that can be further optimized, analyzed, and transformed into target code. In practice, Theano has trouble handling complex or deep DNNs (long chain between inputs and outputs), which can lead to long compilation time. In comparison, the compilation of DeepDSL is efficient even for complex and deep DNNs. For instance, it just takes a few seconds (5 seconds on a laptop with i5-3200M CPU @ 2.6GHz.) for DeepDSL to finish the compilation for Googlenet and Resnet.

Theano was enhanced by wrapper tools such as Pylearn2[17], Blocks[48] (in parallel with Fuel [48] – a dataset processing framework), Lasagne[27], and Keras[24]. These tools improve user interface, add support to parameterized Theano operations, recurrent neural networks (RNN) such as *Long Short-Term Memory* (LSTM), multi-input and multi-output training, and training methods such as Nesterov momentum [34], RMSprop [23], and ADAM [25].

TensorFlow [1] shares some of the design paradigm as that of Caffe. Its core is also written in C++ and its computation graph is described as a DAG where tensors (similar as Caffe's blobs) and layers are alternatively arranged. However, TensorFlow is more fine-grained, where each node is a tensor operation such as matrix multiply and convolution. TensorFlow uses a parameter server to analyze and distribute the user defined computational graph. The parameter server performs some optimization before distributing the DAG and generates subgraphs from the DAG. The main optimization is at the operational aspects such as data communication and memory handling. TensorFlow also performs limited optimization on the DAG such as common subexpression elimination and deadcode elimination. Tensorflow can map computation to multiple devices using heuristics. It can impose control flow on the DAG and execute different subgraphs asynchronously. TensorFlow's tensors are persistent mutable, which allows them to be reused across executions of a graph but it also increases the difficulty of code optimization and debugging.

In comparison, DeepDSL lacks the rich set of features of Tensorflow and does not support multi-GPU training. However, DeepDSL has superior memory and runtime performance on single GPU entirely due to its language-based optimization.

MXNet [10] provides flexible frontends in multiple programming languages. It uses multi-output symbolic expressions to declare the computation graph. The symbolic expressions are composed by operators such as matrix operation or convolution. An operator can take several input variables, produce more than one output variables, and have internal state variables. Before evaluation, MXNet transforms the graph to optimize the efficiency and allocates memory to internal variables. MXNet reuses memory for variables with non-intersecting lifetimes within a computation graph. In particular, MXNet employs scheduling heuristics for memory optimization. For example, it uses a reference counting to determine when the memory of a variable can be recycled and it allows two nodes to share a piece of memory if they do not run in parallel. It uses multiple threads to schedule the operations for better resource utilization and parallelization. Like Tensorflow, it can schedule the execution of computation graph on multiple devices.

In comparison, DeepDSL shares some similar approaches in memory optimization by reusing tensor memory of variables that are no longer needed and by inlining and in-place computation. However, DeepDSL's approach is based on static program transformation, where each type of optimization is implemented as a separate transformation pass. The runtime memory (for GPU) of DeepDSL program is managed as a pool of tensors that are allocated once and then reused by the program with minimal overhead.

CNTK (Microsoft Cognitive Toolkit [2]) provides a set of pre-defined *Network Definition Language* (NDL) functions (e.g. Convolution, MaxPooling) that are internally implemented as computational nodes. Users can also write their own NDL to describe the particular DL network in consideration. Its computation graph is a DAG with two types of vertices. The first type represents basic computation such as add and times, while the second type holds operands and has edges towards a computation node. Such low level computation scheme enables CNTK to encode arbitrary computational network and its core can assign each computation node to a particular CPU/GPU device. CNTK provides both C++ and Python API interfaces to define models, learning algorithms, data reading and distributed training. It decides its computation order via depth-first traversal of the computation graph. It optimizes memory usage by using the same memory across mini-batches and by sharing memory across computation nodes when possible. The latter is achieved by analyzing the execution plan and releasing the memory back to a pool to be reused whenever possible. For example, when a node finished computing all its children's gradients, the matrices owned by that node can all be released. Finally, CNTK leverages a technology named 1-Bit Quantized SGD [39] to quantize gradients with just 1 bit so that the communication cost between

computation nodes is reduced.

In comparison, DeepDSL define DL networks as DSL expressions and memory optimization of DeepDSL is language-based, where memory reuse and sharing are simply the consequence of compiler-based optimization and code scheduling.

Torch [11] leverages the Lua language to provide easy integration with C to achieve C-like performance through JIT compiler and it has a large set of optimized routines. In addition to CPU/GPU, it also supports mobile and FPGA backends. Torch's core is a N-dimensional array called Torch tensor and a comprehensive set of routines such as indexing, slicing, and transposition that operate on the tensors, Torch supports automatic differentiation and many existing neural network models. Torch does not have built-in optimization for memory and runtime or job scheduling. However, a derivative of Torch, PyTorch [36] supports Python frontend, distributed computation with message passing, custom memory allocators for the GPU to better memory efficiency.

BigDL [20], modeled after Torch, is the latest DNN framework from Intel. The main focus of BigDL is to be a deep learning library for Apache Spark. Therefore, the user's DL algorithm written with the BigDL library works as a standard Spark program. BigDL allows the user to load pre-trained Caffe or Torch models into Spark programs. BigDL also claims that it can achieve magnitude faster performance than out-of-box open source Caffe, Torch or TensorFlow on the intel-based system with the Intel Math Kernel Library. In other words, BigDL achieves comparable performance in intel CPU as that of the mainstream GPU.

DeepLearning4j [45] is a Java-based DL library, which extends its tensor functionalities from its n-dimensional array class that is similar to Numpy's NDArray, It provides distributed computation through MapReduce [13] framework Spark [51] and Hadoop [49]. In addition, DeepLearning4j leverages OpenMP for better parallel performance on CPUs. It improves GPU memory usage by allocating each GPU memory chunk once and cache it for further reuse.

Though DeepDSL compiles to Java program, the computation models of DeepDSL and DeepLearning4J are entirely different. DeepLearning4J implements DL networks as graphs, except in Java. Its runtime performance in our benchmark testing appears to be rather poor on single GPU due to lack of efficient optimization.

Chainer [47] provides more control-flow flexibility than other DL libraries. Unlike libraries such as Caffe and Theano, where computation graph is fixed after construction, Chainer follows a *define-by-run* approach, which allows dynamic changes to the control flow of a computation graph between training iteration. This is achieved by storing the order of operations during the graph construction.

10.2 Automatic differentiation

The computation of derivatives may refer to numerical differentiation, symbolic differentiation (of mathematical expressions), or automatic differentiation (of mathematical programs) [6].

Numerical differentiation estimates the derivative value from the mathematical definition. For example, the derivative of a function $f(x)$ can be defined by $f'(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$. Numerical differentiation is suitable when the function is unknown and can only be sampled.

Symbolic differentiation manipulates mathematical expressions. For example, $\frac{d}{dx}(x^2\cos(x))$ reduces to $x(2\cos(x) - x\sin(x))$. Rules such as product rule and chain rule are applied to calculate the derivative for each math expression and the result is simplified to achieve the final result. Symbolic differentiation may lead to inefficient code without sufficient optimization and it may not be applicable to computation problems that cannot be expressed as mathematical expressions.

Automatic differentiation [33] manipulates mathematical programs with control flow logic. Central to automatic differentiation is the application of chain rules to break down complex expressions into simpler ones and apply either forward or reverse accumulation [31] to obtain the final result. Forward accumulation directly applies chain rule to expressions. For example,

$$\begin{aligned}
& \frac{df_1(f_2(f_3(x)))}{dx} \\
= \;& f_1'(f_2(f_3(x))) \times \frac{df_2(f_3(x))}{dx} \\
= \;& f_1'(f_2(f_3(x))) \times f_2'(f_3(x)) \times \frac{df_3(x)}{dx} \\
= \;& f_1'(f_2(f_3(x))) \times f_2'(f_3(x)) \times f_3'(x)
\end{aligned}$$

Reverse accumulation first computes $f_1'(f_2(f_3(x)))$, then $f_1'(f_2(f_3(x))) \times f_2'(f_3(x))$, and finally $f_1'(f_2(f_3(x))) \times f_2'(f_3(x)) \times f_3'(x)$. Temporary variables are needed to hold intermediate results. For example, we need to have the assignments $y_1 = f_1'(f_2(f_3(x)))$ and $y_2 = y_1 \times f_2'(f_3(x))$ so that the final result is $y_2 \times f_3'(x)$.

DeepDSL implements automatic differentiation by applying the derivation rules to the tensor expression that encodes DL network. DeepDSL uses a variation of the reverse accumulation method, which first transforms the tensor expressions into SSA (static single assignment) form and then derives all parameter gradients together without redundant computation.

10.3 DeepDSL as a deeply embedded DSL

One important aspect of the DSL design is whether the DSL will be external or internal, as this has impacts on other aspects of the language [12, 5]. An external DSL typically develops its own parser to recognize the DSL's syntax and builds the related infrastructure to form an interpreter or a compiler for the DSL. In contrast, an internal (or embedded) DSL is implemented on top of a general purpose language (i.e. the host language) to reuse the host language's infrastructure, such as its parser, syntax, type system, and runtime support. The main advantages of the former include the ability for the developers to tune the execution process and to improve some features such as error control or performance, as well as the ease to adapt to new concrete syntax requirements (sometimes an external DSL is the only choice if the desired syntax cannot be expressed using the existing host language syntax). The disadvantages of external DSLs include the requirements for large implementation effort in the details of the execution semantics for each language construct, and usually the development of an IDE from scratch, although this can be greatly alleviated by tools like xText [14] or Spoofax [22]. The main advantages of an internal DSL include the reuse of the infrastructures of the host language (e.g. syntax, semantics, and runtime support) and internal DSL supports rapid-prototyping when different domain-specific constructs can be experimented without involving substantial infrastructure changes. The disadvantages of internal DSLs include that sometimes implementing domain-specific optimizations is difficult and there may be performance penalties when it is necessary to change the host language to obtain some DSL syntax [12].

DeepDSL is designed to be embedded in Scala for a number of reasons. Firstly, the main focus of DeepDSL is to encode a deep learning network and its related mathematical computation in the form of a set of domain-specific constructs with flexibility for future extension such as adding support for feedback networks. As a host language, Scala satisfies this need since it supports functional programming style, which offers facilities for defining different levels of expressions, functions, and function compositions. Secondly, DeepDSL transforms the DSL code to an intermediate representation (IR) and performs multiple optimization steps with the IR. Scala embedding allows the DeepDSL optimizations that consume an input IR and produce the output IR to be implemented in a clean and compact way, where both types of IR are abstractions of computation steps – a technique similar to the meta programming of lightweight modular staging [37] Finally, Scala is seamlessly integrated with Java in both compilation and runtime stage. This allows directly invocations of the Java-based Cuda wrapper code in the interpreter mode without any other dependencies.

DeepDSL is embedded in Scala with its own complete set of syntax (Figure 3), defined using Scalar classes and methods, and operational semantics (Figure 4). After evaluation, DeepDSL programs are de-sugared to a form of abstract syntax tree (AST). Therefore DeepDSL falls in the deep embedding category [16, 43]. Unlike the shallow embedding DSL implementation, terms in DeepDSL do not represent semantics (such as DL convolution function), these terms instead are used to construct an AST, which is later transformed for optimization and traversed for evaluation.

Many problems can be represented in either a generic or domain-specific way. A generic solution usually requires less constraints and offers maximum flexibility for its applicable domains. Contrarily, a domain specific design distills the jargons or dialects for some specific domain to provide some advantages for that domain problems, such as ease of understanding and use, the best possible performance, and quick development cycle, while sacrificing the flexibility, adaptability, or generality of the design (such as the domain-specific vs. domain-agnostic UML models shown in [32]). While deep learning and the DL network are complicated in aspects such as its deep system architecture, the sophisticated learning algorithms, and the difficult parameter tuning, the fundamental DL building blocks such as network construction, the execution workflow, and the related computations are all suitable for abstractions and can be represented with a few carefully designed constructs (e.g. Scalar, Tensor, and Tensor function) and their composition, assembly, and interaction. In addition, some low-level executions, such as Cuda or Cudnn library calls can also be abstracted to free the users from the complex and yet error-prone details. A generic solution will not allow us to handle DL specific problems with focused efforts like what we can achieve with the domain specific design in DeepDSL. For example, it would be very difficult to apply the execution scheduling technique that we employ in DeepDSL for memory optimization inherited from the knowledge of execution dependencies without the analysis of the computation orders of our domain-specific constructs. While DeepDSL constructs, syntax, and semantics are open for extensions such as the support for recurrent neural network, these building blocks have been proved to be highly expressive as shown in the experiments (Section 9) to support a wide range of DNNs.

10.4 DSL on scientific computing with tensors

A number of DSLs have emerged in recent years for scientific computing applications such as TCE (Tensor Contraction Engine) [7], UFL (Unified Form Language) [3], SPL [50] (DSL for signal processing).

TCE [7] is a high-level Mathematica-like DSL for implementing scientific computation in areas such as quantum chemistry, which involves the contractions of multi-dimensional arrays (or tensors). The objective is to improve the runtime and memory efficiency of tensor contractions on parallel platforms.

The TCE compiler searches for an optimal implementation and generates FORTRAN code accordingly. TCE performs a sequence of steps to achieve such goal. First, algebraic transformations are used to reduce the number of operations. Second, loop fusion is conducted to minimize the storage requirements. For the intermediate arrays that are allocated dynamically, TCE provides an algorithm to search the optimized evaluation order. TCE also provides support for re-computation for a reduction in storage requirements when the computation fail to fit within the disk limits and optimize the communication cost together with finding a fusion configuration for minimizing storage when the target machine has multi-processor. DeepDSL also performs tensor contraction operation by translating it to matrix product though it is simpler compared with TCE, which optimizes multiple contraction operations. The computation dependency of TCE forms a tree that can have optimal schedule for memory while the computation dependency of DeepDSL forms a graph that is scheduled by heuristics.

UFL [3] provides a DSL to express variational statements of partial differential equations (PDEs) in near-mathematical notation. Instead of providing a problem solving environment, UFL generates abstract representations of problems that can be used by form compilers to create concrete code implementations in general programming languages. As a general purpose DSL for partial differential equations, UFL offers complete notations for the arithmetic operations in terms of tensor. UFL's support of tensor and tensor algebra is similar to DeepDSL in that both define tensors in terms of their indices and related dimensions. UFL is solely a set of abstractions that can be used to represent partial differentiation equations or formulas (it relies on separate form compilers to provide different concrete language bindings), while DeepDSL not only abstract out the core deep learning concepts but also provide complete support for all the most important deep learning aspects, such as gradient computation, optimization, and code generation.

11 Conclusion

We have designed and implemented DeepDSL for encoding deep learning networks to achieve training and testing goals. This DSL is simple for the users to use and modify, is portable for the DL code to run across platforms, and has competitive runtime and memory efficiency.

DeepDSL generates Java source code. This provides several advantages compared to other DL libraries. The generated Java program is already optimized so that it does not incur lengthy startup time of other libraries, which must repeat the same preprocessing steps each time a DL program is launched. Also, the generated Java program has minimal dependencies and can run on all major operation systems such as Windows, OS X, and Linux, which make it far more portable than other DL libraries. Since it is compilation-based, DeepDSL can also statically detect programming errors and analyze memory consumption so that users can determine whether a DL network can be run on a platform before actually run it. The generated Java program is easy to debug using an IDE such as Eclipse and IntelliJ, where users can set break points and inspect intermediate results, which is very difficult for other DL libraries.

The development of DeepDSL also demonstrated the utility of rule-based symbolic reduction in mathematical computation. The DeepDSL programs are encoded in objects that represent mathematical abstractions. The high-level optimization process, which includes gradient derivation, simplification, and vectorization, is entirely based on rule-based symbolic reduction, which is is easy to understand, implement, and enhance. The result of the high-level optimization remains a sequence of abstract computation steps, which is further improved upon by compilation-based optimization such as common subexpression elimination and by the low-level optimization such as inlining and in-place computation. The final result of optimization is a sequence of statements that directly correspond to function calls of the underlying libraries. The sequence of statements are scheduled based on their dependencies to achieve better memory efficiency before they are mapped to Java source code.

DeepDSL is evaluated on convolutional neural networks. As future directions, we plan to evaluate this approach on other types of neural networks such as generative adversary network, reinforcement learning networks, and recurrent neural networks. The static analysis of DeepDSL may also be used for supporting GPU memory virtualization, where tensors can be temporarily moved from GPU memory to the main memory when they are not used and be copied back when they are needed in later computation.

References

[1] ABADI, M., AGARWAL, A., BARHAM, P., ET AL. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv e-prints* (Mar. 2016).

[2] AGARWAL, A., AKCHURIN, E., BASOGLU, C., ET AL. An introduction to computational networks and the computational network toolkit. Tech. Rep. MSR-TR-2014-112, August 2014.

[3] ALNAES, M. S., LOGG, A., OLGAARD, K. B., ROGNES, M. E., AND WELLS, G. N. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw. 40*, 2 (Mar. 2014), 9:1–9:37.

[4] APPEL, A., AND SUPOWIT, K. J. Generalizations of the sethi-ullman algorithm for register allocation. *Software – Practice and Experience 17* (1987), 417–421.

[5] BARRINGER, H., AND HAVELUND, K. Internal versus external dsls for trace analysis. In *International Conference on Runtime Verification* (2011), Springer, pp. 1–3.

[6] BARTHOLOMEW-BIGGS, MICHAEL ADN BROWN, S., CHRISTIANSON, B., AND DIXON, L. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics 124(1-2)* (2000), 171–190.

[7] BAUMGARTNER, G., AUER, A., BERNHOLDT, D. E., BIBIREATA, A., CHOPPELLA, V., COCIORVA, D., GAO, X., HARRISON, R. J., HIRATA, S., KRISHNAMOORTHY, S., ET AL. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE 93*, 2 (2005), 276–292.

[8] BERGSTRA, J., BREULEUX, O., BASTIEN, F., ET AL. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (June 2010).

[9] CHEN, S.-C., JAIN, R., TIAN, Y., AND WANG, H. Guest editorial multimedia: The biggest big data. *Multimedia, IEEE Transactions on 17*, 9 (2015), 1401–1403.

[10] CHEN, T., LI, M., LI, Y., ET AL. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of Workshop on Machine Learning Systems (LearningSys)* (2015).

[11] COLLOBERT, R., KAVUKCUOGLU, K., AND FARABET, C. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop* (2011).

[12] CUADRADO, J. S., IZQUIERDO, J. L. C., AND MOLINA, J. G. Comparison between internal and external dsls via rubytl and gra2mol. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments: Recent Developments* (2012), 109.

[13] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[14] EYSHOLDT, M., AND BEHRENS, H. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (2010), ACM, pp. 307–309.

[15] FACEBOOK, I. platoon: Multi-gpu mini-framework for theano. `https://caffe2.ai/`, 2017.

[16] GIBBONS, J., AND WU, N. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 339–347.

[17] GOODFELLOW, I. J., WARDE-FARLEY, D., LAMBLIN, P., DUMOULIN, V., MIRZA, M., PASCANU, R., BERGSTRA, J., BASTIEN, F., AND BENGIO, Y. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214* (2013).

[18] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[19] INCUBATOR, F. Gloo: Collective communications library with various primitives for multi-machine training. `https://github.com/facebookincubator/gloo`, 2017.

[20] INTEL. Bigdl: Distributed deep learning library for apache spark, 2016.

[21] JIA, Y., SHELHAMER, E., DONAHUE, J., ET AL. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[22] KATS, L. C., AND VISSER, E. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM sigplan notices* (2010), vol. 45, ACM, pp. 444–463.

[23] KENNEY, J. F., AND KEEPING, E. S. Root mean square. *Mathematics of Statistics, Pt. 1, 3rd edition 4.15* (1962), 59–60.

[24] KERAS TEAM. Keras: Deep learning library for theano and tensorflow, 2015.

[25] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[26] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[27] LASAGNE. Lasagne: A lightweight library to build and train neural networks in theano, 2014.

[28] LECUN, Y., AND BENGIO, Y. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995), 1995.

[29] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[30] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[31] LINNAINMAA, S. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics 16(2)* (1976), 146–160.

[32] MIKHAIEL, R., TSANTALIS, N., NEGARA, N., STROULIA, E., AND XING, Z. Differencing uml models: a domain-specific vs. a domain-agnostic method. In *Generative and Transformational Techniques in Software Engineering IV*. Springer, 2013, pp. 159–196.

[33] NEIDINGER, R. D. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM Rev. 52*, 3 (Aug. 2010), 545–563.

[34] NESTEROV, Y. A method of solving a convex programming problem with convergence rate o (1/k2).

[35] NVIDIA. NCCL: Optimized primitives for collective multi-gpu communication. `https://github.com/NVIDIA/nccl`, 2016.

[36] PYTORCH COMMUNITY. PyTorch: Tensors and dynamic neural networks in python with strong gpu acceleration. `http://pytorch.org/`, 2016.

[37] ROMPF, T., AND ODERSKY, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices* (2010), vol. 46, ACM, pp. 127–136.

[38] SALVATORE, S., PIETER, N., AND MATT, S. Redis: an in-memory database that persists on disk. `https://redis.io/`, 2011.

[39] SEIDE, F., FU, H., DROPPO, J., LI, G., AND YU, D. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014* (September 2014).

[40] SERMANET, P., EIGEN, D., ZHANG, X., MATHIEU, M., FERGUS, R., AND LECUN, Y. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR abs/1312.6229* (2013).

[41] SETHI, R. Complete register allocation problems. *SIAM journal on Computing 4*, 3 (1975), 226–248.

[42] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[43] SVENNINGSSON, J., AND AXELSSON, E. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures 44* (2015), 143–165.

[44] SZEGEDY, C., LIU, W., JIA, Y., ET AL. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 1–9.

[45] TEAM, D. D. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0.

[46] THEANO DEVELOPMENT TEAM. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints abs/1605.02688* (May 2016).

[47] TOKUI, S., OONO, K., HIDO, S., AND CLAYTON, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys)* (2015).

[48] VAN MERRIËNBOER, B., BAHDANAU, D., DUMOULIN, V., SERDYUK, D., WARDE-FARLEY, D., CHOROWSKI, J., AND BENGIO, Y. Blocks and fuel: Frameworks for deep learning. *arXiv preprint arXiv:1506.00619* (2015).

[49] WHITE, T. *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[50] XIONG, J., JOHNSON, J., JOHNSON, R. W., AND PADUA, D. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)* (2001), pp. 298–308.

[51] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[52] ZHAO, T., XIAOBING, H., AND CAO, Y. Deepdsl: A compilation-based domain-specific language for deep learning. In *Proceedings of the 5th International Conference on Learning Representations* (Toulon, France, 2017).