May 2018

Design and Implementation of a Domain Specific Language for Deep Learning

Xiao Bing Huang
University of Wisconsin-Milwaukee

DESIGN AND IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR DEEP LEARNING

by

Xiao Bing Huang

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Engineering

at

The University of Wisconsin-Milwaukee

May 2018

ABSTRACT

DESIGN AND IMPLEMENTATION OF A DOMAIN SPECIFIC LANGUAGE FOR DEEP LEARNING

by

Xiao Bing Huang

The University of Wisconsin-Milwaukee, 2018
Under the Supervision of Professor Tian Zhao

Deep Learning (DL) has found great success in well-diversified areas such as machine vision, speech recognition, big data analysis, and multimedia understanding recently. However, the existing state-of-the-art DL frameworks, e.g. Caffe2, Theano, TensorFlow, MxNet, Torch7, and CNTK, are programming libraries with fixed user interfaces, internal representations, and execution environments. Modifying the code of DL layers or data structure is very challenging without in-depth understanding of the underlying implementation. The optimization of the code and execution in these tools is often limited and relies on the specific DL computation graph manipulation and scheduling that lack systematic and universal strategies. Furthermore, most of these tools demand many dependencies beside the tool itself and require to be built to some specific platforms for DL training or inference.

This dissertation presents *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 the Intermediate Representation (IR), optimizes the IR expressions, and compiles the optimized IR expressions to cross-platform Java code that is easily modifiable and de-

buggable. Also, the code directly runs on GPU without additional dependencies except a small set of *JNI* (Java Native Interface) wrappers for invoking the underneath GPU libraries. Moreover, DeepDSL provides static analysis for memory consumption and error detection.

DeepDSL[1] has been evaluated with many current state-of-the-art DL networks (e.g. Alexnet, GoogleNet, VGG, Overfeat, and Deep Residual Network). While the DSL code is highly compact with less than 100 lines for each of the network, the Java source code generated by the DeepDSL compiler is highly efficient. Our experiments show that the output java source has very competitive runtime performance and memory efficiency compared to the existing DL frameworks.

[1] Our previous results are reported in [115]; design and implementation details are summarized in [114].

TABLE OF CONTENTS

1 Introduction **1**

1.1 Deep learning overview 2

1.1.1 Shallow and Deep learning 2

1.1.2 The deep learning process 4

1.2 Convolutional neural networks 10

1.2.1 Deep neural networks examples 12

1.3 DL challenges 19

1.4 Contributions 24

2 DeepDSL Overview **26**

2.1 DeepDSL 26

2.2 DeepDSL core constructs 29

2.2.1 Tensor 29

2.2.2 Tensor Function 30

2.2.3 Fixed Tensor 31

2.2.4 Function Application and Composition 32

2.2.5 Network as Function Composition 33

2.2.6 Training 34

3 Formalization **37**

3.1 Syntax 37

3.2 Semantics 40

3.3 Typing Rules 42

3.4 Examples 45

3.5 Gradient derivation 47

4 Optimization 55

4.1 High-level Optimization 55

4.1.1 Simplification of parameter gradients 55

4.1.2 Vectorization 57

4.1.3 Examples 61

4.2 Low-level Optimization 66

4.3 Code scheduling 68

4.4 Code generation and runtime 71

5 Experiments 75

5.1 DL network configuration standards 75

5.2 Experiment environment description 76

5.3 GPU-based runtime & memory performance 77

5.4 CPU-based Lenet experiments 79

6 Related work 81

6.1 DL frameworks 81

6.2 Core feature comparison between DeepDSL and existing frameworks . . . 93

6.3 Automatic differentiation 96

6.4 DSL on scientific computing with tensors 97

7 Conclusion 99

7.1 Upcoming releases and future directions 100

References 103

Appendix Proof of Theorem 3.1 110

1 Scalar *s* type soundness 111

1.1 Progress 111

1.2 Preservation 113

2 Tensor t type soundness 115

2.1 Progress 115

2.2 Preservation 117

3 Scalar $\hat{s}$ type soundness 121

3.1 Progress 121

3.2 Preservation 121

4 Proof conclusion 122

CURRICULUM VITAE **123**

LIST OF FIGURES

1.1 Complex function in multi-layer representation 3

1.2 Representations expressed with simpler representations of previous layer [37] 3

1.3 A DNN neuron 4

1.4 A sample DNN 5

1.5 The conceptual model of convolutional neural network 10

1.6 Reduced number of connections between adjacent layers 11

1.7 LeNet-5 network structure 13

1.8 Alexnet network structure [60] 15

1.9 Vgg-16 network structure [90] 16

1.10 Inception module with dimensionality reduction [96] 17

1.11 *GoogLeNet* network structure [96] 18

1.12 A residual learning block [44] 19

1.13 Resnet network structure [44] 20

2.1 An overview of Deep DSL 28

3.1 A formal syntax for DeepDSL, where n is some positive integer and $f_2 \circ f_1$ is defined as $x \Rightarrow f_2(f_1(x))$. Note that the shaded syntax is for intermediate representation at runtime. 38

3.2 Operational Semantics where $m = (n_1 \times \ldots \times n_k) - 1$ 40

3.3 Operational Semantics - other types of scalar expressions 41

3.4 Typing rules for tensor and scalar expressions 43

3.5 Typing rules for arithmetic scalar expressions 44

3.6 Typing rule for index expressions 44

3.7 Static single assignment transformation, where y variables are fresh and we assume that scalar function application does not appear inside a tensor expression. 50

4.1 Rules for vectorization, where $\{I\}$ represents the set of indices in I. . . . 58

4.2 Syntax after vectorization 60

4.3 Gradient derivation of affine transformation 62

4.4 Gradient derivation of convolution 63

4.5 Gradient derivation of loss expression 64

4.6 Gradient derivation of log softmax, where $f(t)$ is defined as $(i_n \cdot i_k) \Rightarrow t(i_n)$. 65

4.7 Sample network 68

4.8 Scheduled statements, their heights, and live variables at each statement. . 71

5.1 Runtime performance of DeepDSL, Tensorflow, and Caffe (1 forward/backward iteration), where the network names are followed by the batch size. DeepDSL* and DeepDSL† are performance for DeepDSL 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. 78

5.2 Peak GPU memory use of DeepDSL, Tensorflow, and Caffe during training, 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. 79

5.3 Runtime CPU performance of DeepDSL, Theano, and Caffe (1 forward/backward iteration) 80

LIST OF TABLES

5.1 Computer system configuration details 77

6.1 Core feature comparison between DeepDSL and existing frameworks . . . 95

ACKNOWLEDGEMENTS

Academic research is a, in many cases, painful and yet fun and rewarding journey. It repeatedly gives you the hope and motivation to continue every morning and then tells you that is just a false alarm at night. On the journey, there is satisfaction, excitement; there is also frustration and many sleepless nights. The most important thing I learned from this journey, though, is the spirit of the perseverance and strong belief of the inexhaustible possibilities.

First and foremost, I want to full-heartedly thank my advisor, Professor Tian Zhao. Needless to say the countless enlightening suggestions and helps he provided in the past years in my research, to me, he is not only my advisor but also my lifetime friend. He showed extreme patience when I was addicted to my seemingly good but in reality "dead-end" ideas. He always encouraged me to find out answers to difficult problems with hints that kept me out of the comfort zone and yet with sufficient information for me to move forward. In fact, he is very detail-oriented and never mind going through the painful points in tough problems together with me. The question "Can we do better?" becomes the defacto standard of any staged results in our research.

I would also like to extend my sincere thanks to Professor Yu Cao, who has been providing numerous great suggestions and ideas throughout my research adventure. Professor Cao also showed me a great research should not just stay theoretical but also be linked to its own real-world applications whenever possible. Also, I cannot imagine how our experiment results will be achieved without the fast hardware and software environment help from Professor Cao and his group.

I want to extend my inexpressible thanks to my parents, who have been always there whenever I am in need and have been always proud of me no matter what.

I want to wholeheartedly thank my wife, who takes care of all other aspects of life details in my long journey so that I can focus. More importantly, she gives me great understanding and space whenever I need dedicated time for concentration. The persistent love gives me courage to work out the balance between life, work, and research.

Finally, I want to thank all other professors in my thesis committee, Professor Ethan Munson, Professor Susan McRoy, and Professor Xiangming Mu, who have taken great efforts and time to go through my thesis and provided important suggestions and comments so that I can keep improving the quality of the thesis.

Chapter 1

Introduction

Multimedia has become the most valuable resource for insights and information [14] in recent years. Deep learning, which aims at learning multiple levels of representation and abstraction to help infer knowledge from the multimedia data, such as text, audio, image, and video, is making astonishing gains in well-diversified areas such as machine vision, speech recognition, big data analysis, and multimedia understanding.

While various types of DL networks are being rapidly applied to different research and industrial domains, the approaches that current DL frameworks adopt to address the key DL needs remain largely similar. These frameworks usually leverage some general-purpose language (e.g. C/C++, Python, and etc.) to represent the DL network structures with purposely designed language constructs in the format of programming libraries, in order to drive the DL training, validation, and inference or application steps using these programming constructs. While such method gives the user large maximum freedom in their DL tasks, it often requires the user to have in-depth knowledge of the programming language in use, in addition to thorough understanding of the meaning of these constructs. Also, since the programming constructs are fixed and their internal representation and transformation are opaque to the user, it is usually very difficult for the user to make changes or perform optimization. Moreover, since these DL constructs are provided via the format of programming libraries, the implementation of DL applications requires the presence of the DL framework and all related dependencies in not only compile-time but also runtime. In addition, error messages during the DL application execution are often not straightforward and sometimes misleading. Consequently, debugging and testing are usually difficult due to the loose connection between the exception and the actual cause.

This dissertation addresses these problems with a domain specific design, DeepDSL, to offer the user a hight level abstraction of the deep learning domain with simple, flexible, and powerful DSL constructs, while keeping the internal representation fully accessible. Since this domain specific approach remains fully abstract before the final source code generation, optimization can be achieved automatically by the DSL compiler without modifying a single line of the DSL code. In addition, the generated source code is in the familiar imperative style and very easy to modify for different training and inference needs. Debugging and testing also become much easier because the entire DL workflow is captured before the final code is generated so that problems such as incorrect tensor dimension values can be detected at compile-time.

The rest of this chapter is organized as follows. Section 1.1 provides an overview of the deep learning process. Section 1.2 describes one of the most important DNNs, the *convolutional neural network* (CNN) and surveys some widely used DNN examples. Section 1.3 discusses the challenges today's DL frameworks face and introduces DeepDSL to answer these challenges. Section 1.4 summarizes the main contributions of DeepDSL.

1.1 Deep learning overview

1.1.1 Shallow and Deep learning

Many traditional machine learning models, such as SVM [19], Boosting [34], *Logistic Regression* (LR) [20, 107, 33], or *back propagation* (BP) neural network [37], are sometimes categorized as *shallow learner* (SL) [86], in the sense that these models are trained with some domain specific features based on the user's prior knowledge[1]. While widely applied for decades, the performance of these SLs depends largely on the provided representation of the data [37]. These SL methods have limited power to represent complex functions

[1] The network structures of these models usually have either none (e.g. LR), or one hidden layer (e.g. SVM, Boosting, BP).

and hard to be generalized for difficult problems such as multi-class classification. Deep learning, also known as representation learning, distinguishes itself from shallow learning in that DL can learn features or representations directly from the input with little to no prior knowledge. Representations learned from one DL layer are expressed in terms of the simpler representations of its previous layer (as illustrated in Figure 1.1).

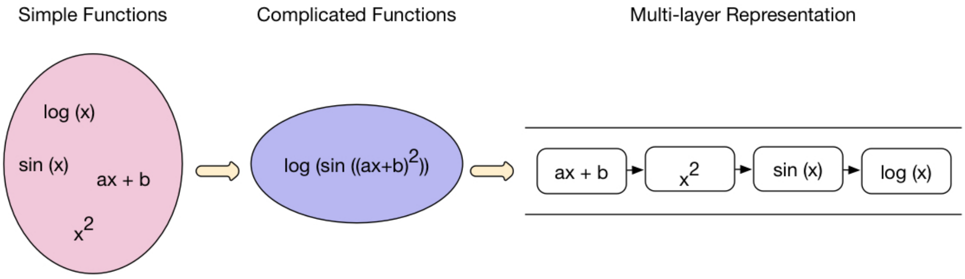

Fig. 1.1 Complex function in multi-layer representation

DL can approximate complex functions and characterize the distribution of input data with a deep-layered nonlinear structure, and exhibit strong learning power of inherent features from small input datasets [35].

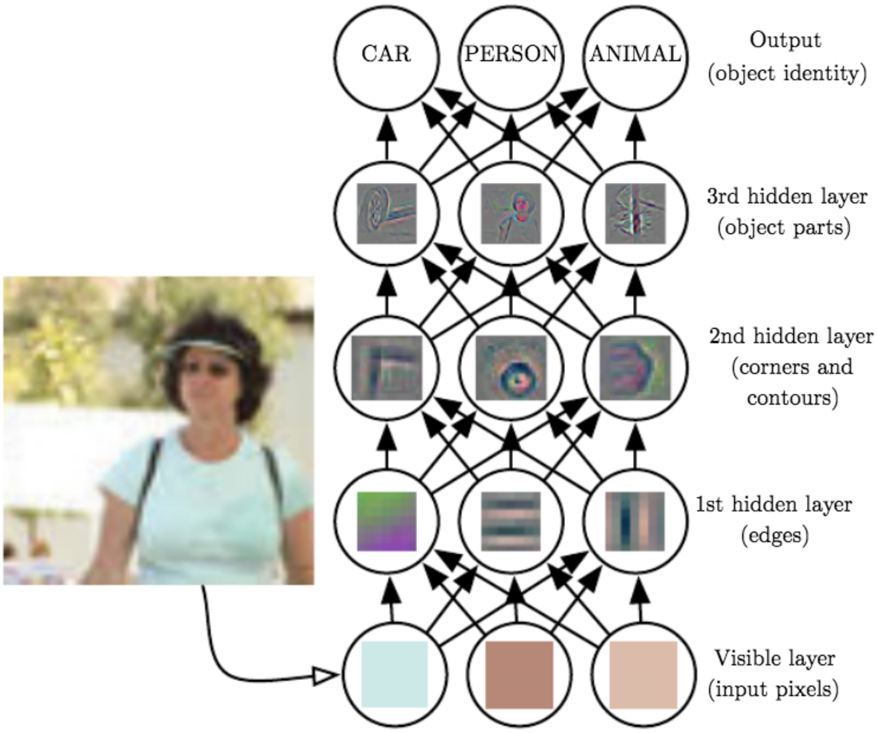

Fig. 1.2 Representations expressed with simpler representations of previous layer [37]

For example, Figure 1.2 shows a DL structure that has three hidden layers in between the input and output layer for an image recognition task. Although the input only contains the raw image pixels, the first hidden layer can identify edges and generate edge descriptions by comparing the brightness of neighboring pixels. With these edge descriptions, the second hidden layer can search for corners and extended contours using the collections of edges. In turn, the third hidden layer can detect entire parts of specific objects using specific collections of contours and corners. These object parts can then be used to recognize the objects appear in the image. Specifically, the original features of the input are transformed onto some new feature spaces through the layer-wise feature extraction and the importance of feature learning is explicitly emphasized in DL.

1.1.2 The deep learning process

As we mentioned in Section 1.1.1, DL models are capable of learning the right features by itself and require little to no guidance from the model designer/developer. DL is a form of machine learning that is inspired by the structure of our brain and the network structure of a DL model usually contains multiple layers and thus is often denoted as deep neural network (DNN). The fundamental building element of a DNN is called "neuron", which essentially inherits the name neuron in biology with much simpler function. Given a set of inputs and one associated weight per input, a neuron applies some function, either linear or nonlinear, to the weighted inputs to generate an output. This is illustrated in Figure 1.3.

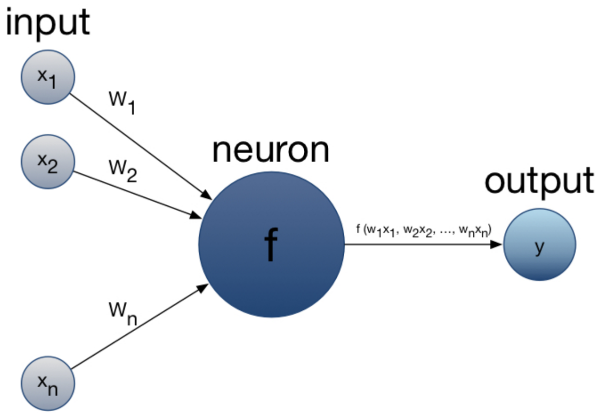

Fig. 1.3 A DNN neuron

Neural network is formed when we feed the output of a set of neurons as the input to another set of neurons. DNN is formed if we repeat such steps a few times. That is, DNN can be constructed with multiple interconnected layers of neurons. Given K the number of layers and w_{ij}^k the weight for the link from the i^{th} neuron in layer k to the j^{th} neuron in layer $k+1$, where $k = 0, \ldots, K-1$, a sample DNN is illustrated in Figure 1.4. Note there are no connections between the neurons in the same layer in the Figure. This type of networks is known as feedforward neural network and is the focus of this thesis[2]. The basic DNN network structure is formed by three types of layers: the input layer, the hidden layer(s) (the layer(s) between the input and output layer that is(are) responsible for capturing the latent/hidden features), and the output layer. Also note:

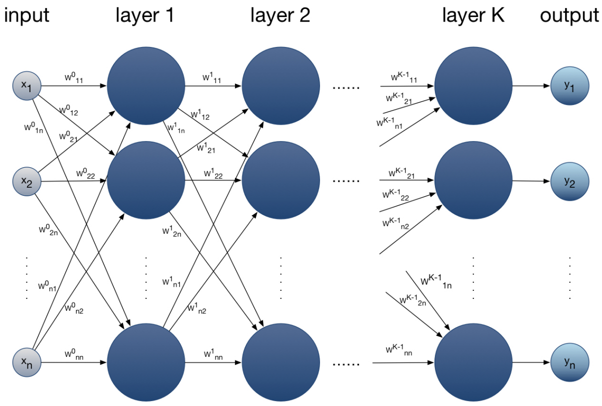

Fig. 1.4 A sample DNN

- the number of neurons in different layers may or may not be the same;
- although Figure 1.4 shows every neuron in the k layer connects to every neuron in the $k+1$ layer, this is not necessarily so; it can well be the case a neuron in one layer connects to only a subset of neurons in the next layer[3].

The input data X for any DNN is usually a multi-dimension tensor. For example, in Figure 1.4, X consists of a number of samples x_i, where $i = 0, \ldots, N-1$ and each sample can be a multidimensional tensor, such as the common 3-dimensional tensor representation

[2] There is another type of neural network named “feedback” or recurrent neural network (RNN) [83] where the output of some layer may be fed back into the network. This will essentially give the network some memory and such feedback makes RNN dynamic until it reaches some equilibrium. Changes on the input will cause the network to move towards another equilibrium state, if any.

[3] Figure 1.6 demonstrates this in details.

(channel, height, and width) for the color image. In fact, all the data involved in a DNN can be represented as tensors, including the input, the data in between layers and the output. DeepDSL provides first class support for tensors (See section 2.2 for details).

The input of layer $k + 1$ is the output of its previous layer k, which is obtained by applying some function f_k to the input of layer k, where f_k is the collective effect of each neuron function in the same layer. Hence the output Y can be represented in terms of the input X in Equation 1.1.

$$Y = f_k(f_{k-1}(...f_1(X)...)) \tag{1.1}$$

The output Y generally represents some objective of the learning process. For instance, the output may represent the labels for different image classes (e.g. cars, animals, mountains, or city skyscrapers) in image classification. The computed output may or may not match the true training image class label. The DNN training objective is to minimize the number the mismatched cases, usually represented with some error function E, by optimizing the parameters of the neurons in all DNN layers. This can be achieved with a backpropagation process [83]. Specifically, optimizing the parameters is equivalent to move through the "space" formed by all the possible parameter values for neurons, along the the steepest descent path of the gradient of the error function E towards minimum. As a result the changes on the weights should be proportion to the gradient. This is defined as "gradient descent", also known as steepest descent in literatures [77]. Mathematically, the value of gradient descent is calculated by taking the partial derivative of the error function E of the output layer with respect to its input. Such calculating can be propagated backwards from the output to the input layer by applying the gradient calculation chain rule. DeepDSL captures the entire backpropagation process (detailed in Section 3.5) and supports automatic derivation (described in Section 6.3) during the compilation stage.

Recall in Figure 1.4 we show three types of layers in a typical DNN, besides the input and output layers, we also have some hidden layers in between. Each hidden layer is formed with multiple single neurons; the input of neurons of a layer i is the output of neurons of the previous layer $i - 1$ and the output of the neurons of layer i is the input of neurons of the next layer $i+1$, where $i \geq 2$. This structure enables us to train the multilayer neuron network in a layer-by-layer manner. Section 3.5 shows the details of how the gradient derivation is supported in DeepDSL.

Generally, the output of a neuron can take any values in range $[-\infty, \infty]$. We need to decide

1. a way to determine when a neuron should be turned on (this is inspired by how the brain neuron works biologically) to contribute to the DL computation;
2. how much contribution a neuron's output offers to the overall computation.

In neural network, the control mechanism of turning a neuron on and off is called the "activation function". Several activation functions (and their variants) are widely used in today's DNN applications, such as the Sigmoid[4], Tanh[5], and *rectified linear unit* (ReLU) [71] function. These activation functions are nonlinear in nature and the combination of these functions is also nonlinear. This is important as otherwise the gradient of the activation function will be a constant to cause the neural network delta rule [84] calculation and further the entire training process to fail. ReLU function takes a simple form of $max(0, x)$, hence any negative values will have a zero output. ReLU has the nice property of sparsity. When other activation functions, such as Sigmoid, turn on almost all the neurons in an analog way (which results in huge computation cost), ReLU simply turns all the neurons with negative value input off[6], which saves the computation substantially.

The selection of which action function to use largely relies on the function to be approximated. If the nature of this function is well understood, then using an appropriate activation function will help approximate the function faster and result in a faster training process; if this function is unknown, ReLU is often recommended as it consistently offers

[4] Sigmoid function, a special case of the logistic function, and namely a "S" shape function that returns value in $[0, 1]$ or $[-1, 1]$ depending on convention.

[5] Tanh function returns the hyperbolic tangent value of input.

[6] This benefit does not come without cost. In fact this may cause some of the neurons stop to respond to variations in input at all, named dying ReLU problem. Mitigation of this problem includes ReLU variants like leaky ReLU e.g. $y = 0.01x$ for $x < 0$ to achieve nonzero gradient and recover during training eventually.

promising results. DeepDSL provides direct DSL construct that represents the high-level built-in GPU support for activation functions like ReLU, see Section 2.2.3 for details.

In practice, the gradient descent method frequently does not result in the global optimum output when we use nonlinear neurons. Specifically, directly applying the gradient descent method may cause the objective searching to be trapped in some local optimum. *Stochastic Gradient Descent* (SGD) [93] is designed to overcome this drawback to improve the probability of finding a global optimum. The standard gradient descent method updates the parameter θ of the objective function $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$ in iterations over the entire training set of size n.

$$\theta = \theta - \epsilon \nabla J(\theta) = \theta - \epsilon \sum_{i=1}^{n} \nabla J_i(\theta) \tag{1.2}$$

where ϵ is denoted as learning rate or step size. SGD, while updating the parameter θ similarly as the standard gradient descent method, only samples a single or subset (a.k.a. mini-batch) of examples at each iteration (Equation 1.3 where i will take its value from 1 to the mini-batch size in each iteration).

$$\theta = \theta - \epsilon \nabla J_i(\theta; (x_i, y_i)) \tag{1.3}$$

This small change improves the standard gradient descent algorithm in two ways. It reduces the variance during the parameter update which leads to more stable convergence; in addition it also enables us to take advantage of the existing highly optimized matrix operations[7] for the vectorized computation of the cost and gradient.

Nevertheless, the standard SGD still faces challenges [82] even with careful design:

1. Selecting a proper learning rate and modifying it throughout the learning process is indeed very difficult and application domain specific. A learning rate that is too small results in slow convergence, while a learning rate that is too large can hinder convergence or even diverge results. The adjustment rule on learning rate though

[7] There are many existing matrix computation routine libraries designed for different system architectures that can be invoked via their standard API.

generally entails checking the performance after one or several epochs[8] and reduce it accordingly. The performance checking schedule and thresholds, however, have to be defined in advance and are thus unrelated to the dataset's characteristics;

2. Another issue is using the same learning rate for all parameter updates. This does not incorporate the fact that some datasets are sparse and the features have different frequencies so that parameter updates require different frequencies accordingly;

3. Also, it is argued [21] that the difficulty of minimizing the non-convex error function E is not due to the local optima trap but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it extremely difficult for SGD to escape, as the gradient is close to zero in all dimensions.

The standard SGD in Equation 1.3 often leads to very slow convergence after initial steep gains due to the fact that the SGD results may oscillate among some local optima as the objective function of deep architecture have the form close to local optima. One improvement is to set the learning rate ϵ adaptively to help the algorithm converges. That is, the algorithm selects some initial value of learning rate and then decreases the initial value with some appropriate rate (the decreasing rate may be fixed in terms of the iteration progress or may be computed using some function). The momentum [80] method is used to accelerate gradient descent that accumulates a velocity vector across iterations in directions of persistent reduction of the objective function [95]; in other words, momentum is used to force SGD progressing faster along the path towards the global optimum. We can rewrite Equation 1.3 with Equation 1.4 and 1.5 to add the momentum factor. In Equation 1.4 $\epsilon > 0$ is the weight for the negative gradient and $\mu \in (0, 1]$ is the weight for the previous update. Essentially the momentum value determines how much the previous gradients are incorporated into the current network parameter update.

$$V_{t+1} = \mu V_t - \epsilon \nabla J_i(\theta_t; (x_i, y_i)) \tag{1.4}$$

[8] In DNN, an epoch is a full pass through over the entire training set.

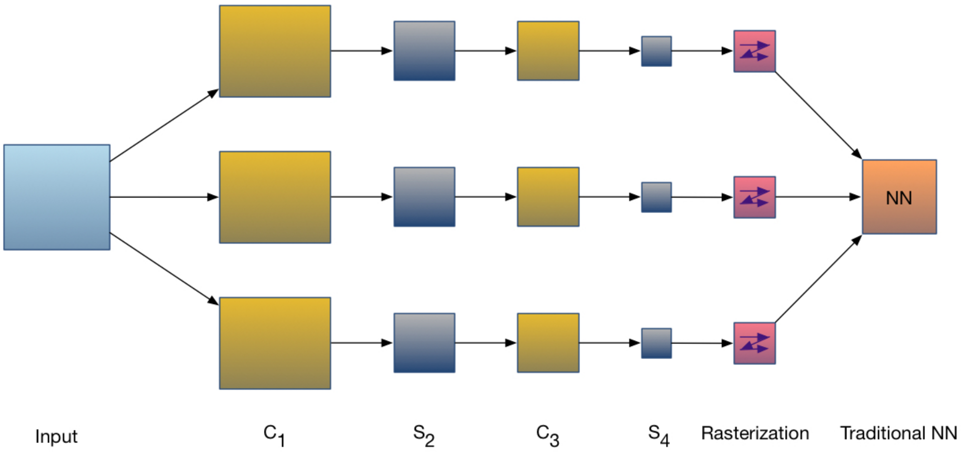

Fig. 1.5 The conceptual model of convolutional neural network

$$\theta_{t+1} = \mu\theta_t + V_{t+1} \tag{1.5}$$

Many other variants of the gradient descent methods have been designed to tackle the aforementioned SGD challenges as well, such as Adagrad [28], Adadelta [113], Adam [56], Nesterov accelerated gradient (NAG) [75], Nadam [27], and the latest AMSGrad [81]. Detailed information is referred to the nice summary in [82].

DeepDSL provides a convenience DSL training construct that can take parameters like learning rate or momentum directly, see Listing 2.2.6 for an example.

1.2 Convolutional neural networks

In this section, we introduce one of the most popular DL networks, the convolutional neural network (CNN) [63], to illustrate how the DL network works in general. CNN is a multilayer neural network, in which each layer is formed by several 2D planes and each plane has multiple independent neurons. Figure 1.5 demonstrates the simplified conceptual model of a CNN. The input images are convoluted with 3 trainable filters and addable biases. The convolution (filtering) process results in 3 feature maps in the C_1 layer. The pixels in the feature map are grouped and each group of the pixels are summed, weighted, and bias added. A sigmoid function is applied at the end to generate the 3 new feature maps in the S_2 layer. C_3 and S_4 layers are formed with the same procedure.

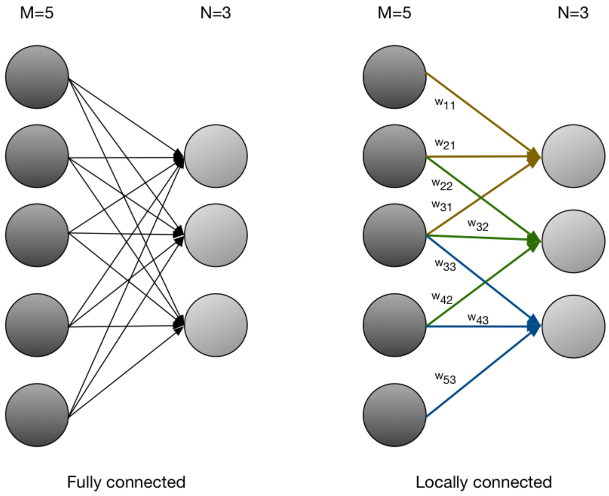

Fig. 1.6 Reduced number of connections between adjacent layers

Finally, the pixels from S_4 are rasterized to be connected to form a vector. This vector enters a traditional neural network to produce the final result.

The C layer works as a feature extraction layer. That is, the input of each neuron is connected to the local receptive field of its previous layer and extracts that local feature. The spatial relation between the local feature and other features is fixed after the feature extraction. The S layer works as a feature projection layer. Specifically, each feature projection produces a plane and the weights of all neurons in the plane share the same value. Sharing the weights for neurons on the same plane helps reduce the number of free variables and the complexity of the layer parameter selection. The reduction of parameter numbers in CNN is indeed the most critical idea for its success. Figure 1.6 illustrates how this is achieved conceptually. There are two adjacent layers in this figure. The input layer has $M = 5$ nodes and the output layer has $N = 3$ nodes. The left-hand side is the fully connected case where all the $M = 5$ nodes in the input layer are connected to all the nodes in the output layer. This generates $M * N = 5 * 3 = 15$ connections. The right-hand side shows the locally connected case where each node in the output layer only connects to $P = 3$ (here P is the size of the local receptive field) adjacent nodes in the input layer. This reduces the connection needed to $P * N = 3 * 3 = 9$.

In detail, suppose we have an image with 1000 * 1000 pixels and 1 million hidden neurons in the first layer. If the layers are fully connected, i.e., each hidden neuron is connected with a pixel in the image, we will have $1000 * 1000 * 1000000 = 10^{12}$ connections. The weights and biases associated with this huge number is formidable to train. Fortunately we can effectively reduce the data dimensionality by leveraging the simple fact that the spatial relations between pixels for an image is local. Just as our eyes sense the image via local receptive field, each neuron does not need to sense the entire image but only the local image area it is close to. All the sensed results can be summarized in higher layer to construct the global image information. For example, if we have the receptive field as $10 * 10$, each hidden neuron only needs to connect to the $10 * 10$ local image and thus 1 million neurons will produce $1000000 * 10 * 10 = 10^8$ connections, which is significantly smaller than 10^{12}. The result 10^8 number of connections is still huge that demands further reduction. Since each neuron connects to the $10 * 10 = 100$ image area, if we use the same 100 weights for all neurons, we will only have 100 parameters between layers no matter how many neurons we have. Note since we apply the same filter, the result will contain one type of feature. This is not a problem since we can apply different kind of filters to produce different types of feature projections. For example, We have 3 filters in Figure 1.5 which will generate 3 different types of features for further processing. Suppose we have 100 feature maps, we still only need to learn $100 * 100 = 10000$ weights. Finally, how many neurons we need in one hidden layer is determined by the image size, the filter size, and the shift-window of the filter inside the image. For example, given the $1000 * 1000$ pixel image with the filter size $10 * 10$ that has no overlapping between filter, the number of neurons in the hidden layer is $\frac{1000*1000}{10*10} = 10000$. If we have 100 filters, we will have 100 feature maps. The number of neurons we need will be $10000 * 100$ neurons[9].

1.2.1 Deep neural networks examples

This section we describe several deep neural networks that have been widely applied in a various areas such as image classification, video categorization, weather forecasting, self-driving vehicles, robotics, real-time behavior analysis, and etc[10].

[9] Since each type of filter has a bias parameter, we also need to add another 100 to the final number of parameters that require training.

[10] Note all the diagrams in this section are either modified or ported from the original publication with citation for convenience.

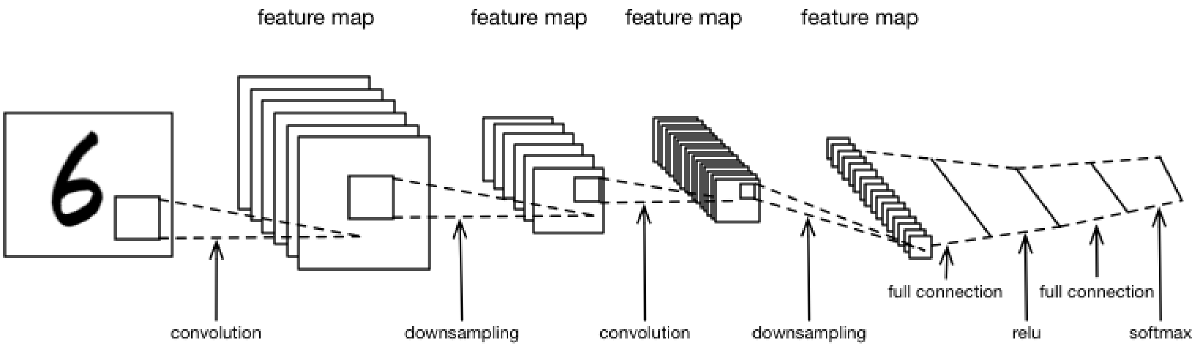

Fig. 1.7 LeNet-5 network structure

LeNet Figure 1.7 shows the network structure of a variant[11] of the classic *LeNet-5* [64]. This network has 2 convolution and downsampling/pooling layer alternatively arranged for 2 times, followed by 2 fully-connected layers with a ReLU activation layer in between. 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 having extreme variability, such as handwritten characters, with robustness to distortions and simple geometric transformations. Due to its simplicity and latent feature capturing power, the variations of *LeNet-5* have been applied to domains such as facial recognition, scene labeling, image classification, and etc.

The parameters of the convolution layer consist of learnable kernels or filters. Each unit of this layer receives input from a set of units located in a small neighboring area in the previous layer, the neighboring area is called the receptive field. During the forward pass each filter is convolved with the input to produce a feature map. The weight vector that generates the feature map is shared to reduce the number of learnable 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 downsampling layer takes small rectangular block areas from the output of the convolution layer and computes subsample values (maximum or average value of all the

[11] This is the network we use for experiments.

unit values in the block, etc.). This computation step reduces the spatial size of the representation, and in turn reduces the number of learnable parameters. CNN differs from the traditional multilayer perceptrons (MLP) in the sense that it provides certain level shift and distortion invariance [63]. Such property is mainly achieved with the downsampling layer, since the subsample values taken by the downsampling layer can still remain largely unchanged even with some level of shift and distortion on the input images.

The second convolution layer captures features from the output of the previous pooling layer. These features are learned from a feature space that is different from the feature space of the input layer. The second pooling layer continues to reduce the dimensionality of the input from the second convolution layer. In fact, such repeated stack-up pattern is very powerful in discovering features from different dimensionality spaces and has been widely used in many successor CNNs.

While locality information is critical 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 in it to compute the global semantic information.

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 nonlinear boundary between the input samples.

Alexnet Figure 1.8 shows the popular Alexnet [60][12]. Alexnet is the first DNN that achieves very good result in the hard ImageNet [25] dataset challenge that consist of millions images for the classification task with 1000 image categories. The core of Alexnet consists of eight layers with weights. The first five layers are convolutional (note there are also several max-pooling layers in between) and the rest three are fully-connected. The last fully-connected layer outputs to a 1000-way softmax which produces a distribution over the 1000 class labels.

[12] Note we also have used Overfeat [88] in our experiments but we will omit the detailed summary of it since Overfeat largely conforms to the Alexnet network design except a few small tweaks for adding support for localization and detection in addition to classification.

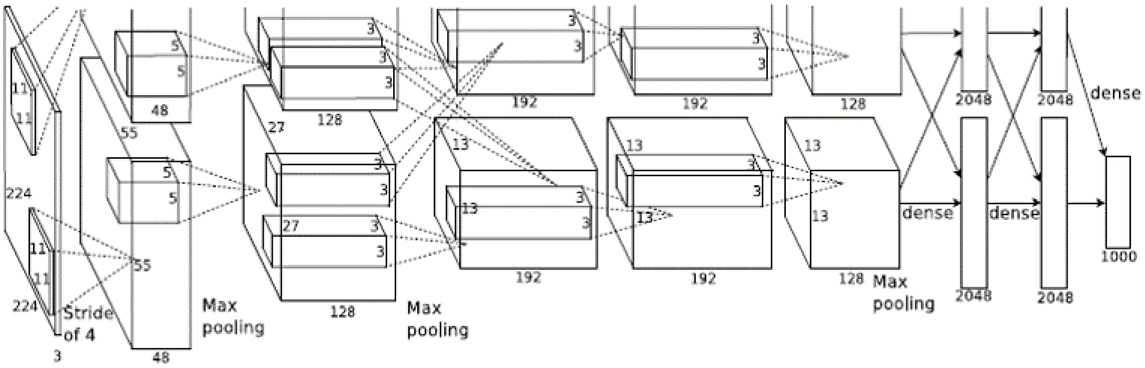

Fig. 1.8 Alexnet network structure [60]

To achieve high computational efficiency, as depicted in Figure 1.8, the kernels of the second, fourth, and fifth convolution layers are connected only to the kernel maps of the previous layer that reside on the same GPU. The kernels of the third convolutional layer are connected to all kernel maps in the second layer. Response-normalization layers are applied after the first and second convolutional layers. Max-pooling layers[13], are used after both response-normalization layers and fifth convolutional layer. The ReLU activation function is applied to the output of every convolutional and fully-connected layer. Finally, the dropout layer is also applied to two of the first fully-connected layers[14].

Vgg Vgg [90] addresses an important aspect, the depth, of convolutional networks. Particularly, will the result precision be improved by increasing the depth of the convolutional networks while fixing other parameters of the architecture? Vgg answers the question with an universally used 3×3 convolution layer[15]. Authors of Vgg reason the stack of two 3×3 convolution layers (without pooling layer in between) has an effective receptive field of 5×5; similarly, three convolution layers back to back have an effective receptive field of 7×7. In other words, the benefits of both a larger filter and smaller filter sizes are

[13] Alexnet adopts a modified max pooling strategy to reduce overfitting. A pooling layer consists the grid of pooling units that are s pixels apart from each other, each summarizing a neighborhood of size z x z centered at the location of the pooling unit. The traditional local pooling is obtained when $s = z$; overlapping pooling is obtained when $s < z$. Alexnet adopts overlapping pooling with $s = 2$ and $z = 3$.

[14] Since "dropped out" neurons will not be used for forward pass or backpropagation, the neural network samples a different architecture every time an input is presented, but all these architectures share weights. A neuron can no longer rely on a particular subset of other neurons; instead, more robust features will be learned with different random subsets of other neurons.

[15] The choice of 3×3 is because this is the smallest size to capture the notion of left/right, up/down, and center.

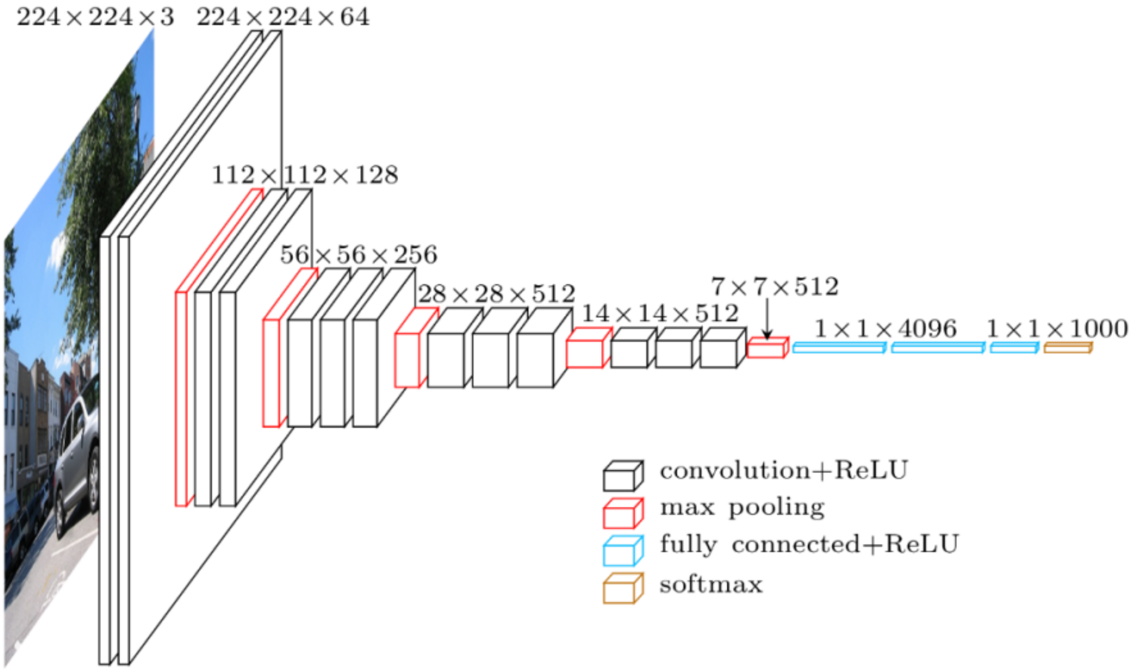

Fig. 1.9 Vgg-16 network structure [90]

obtained. The main benefit of such setting is the decrease in the number of parameters. Furthermore, with more convolution layers stacked up, more ReLU layers instead of one can be used (e.g. three non-linear rectification layers is incorporated instead of a single one) and in turn the decision function is made more discriminative. Figure 1.9 shows Vgg-16. In Vgg-16, the aforementioned two or three 3 × 3 convolution layers are stacked up and each stack is followed by a max pooling layer, ending with three fully-connected layers: the first two have 4096 channels each, the third contains 1000 channels and the final layer is the soft-max layer.

GoogLeNet Different from the common CNN pattern that stacks convolution and pooling layers on top of each other sequentially, *GoogLeNet* [96] leverages a structure noted as "inception module" (Figure 1.10) to solve the inevitable growing output dimension problem due to the increase of the number of output layer by layer. Specifically, there are two two main drawbacks in the traditional CNN network design:

- larger number of parameters of larger size DNN, which makes the enlarged network more prone to overfitting and;

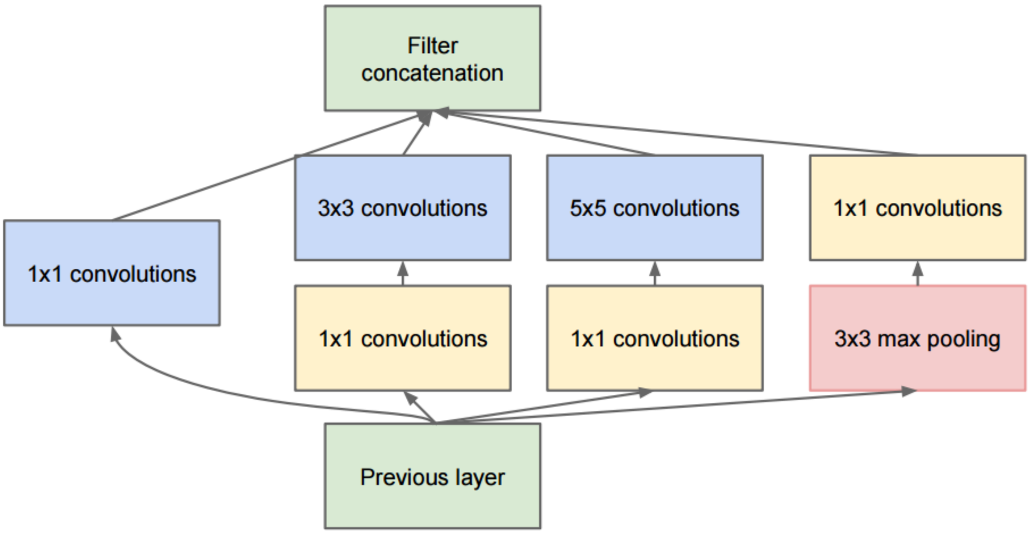

Fig. 1.10 Inception module with dimensionality reduction [96]

- dramatically increased use of computational resources.

GoogLeNet tackles both issues using a small convolution filter and concatenation of different convolutional and pooling layer outputs to essentially introduce sparsity. As a result, *GoogLeNet* has a deep network structure of 22 layers. In inception module, the 1×1 convolution layer is used for the dimension reduction. On top of the 1×1 convolution, the rectified linear activation is used to add further sparsity and efficient gradient propagation, etc. Figure 1.11 shows the full *GoogLeNet*. It is obvious to see that the "inception module" is the building block of *GoogLeNet*. The other unique aspect is the 2 auxiliary classifiers (i.e. Softmax0 and Softmax1). The purpose of the auxiliary classifier is to blend in features produced by the layers in the middle of the network. These classifiers are connected to the intermediate layers that stem from earlier stage of the *GoogLeNet* DNN. The losses of the 2 classifiers are added to the total loss of the network with a discounted weight during training and the classifiers are discarded at inference time.

Resnet Resnet [44] concerns the degradation problem in DNN where accuracy gets saturated with the network depth increasing and degrades rapidly after that. Early work [43] has shown that such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error. The main idea behind Resnet is demonstrated in the residual block as shown in Figure 1.12. In the residual

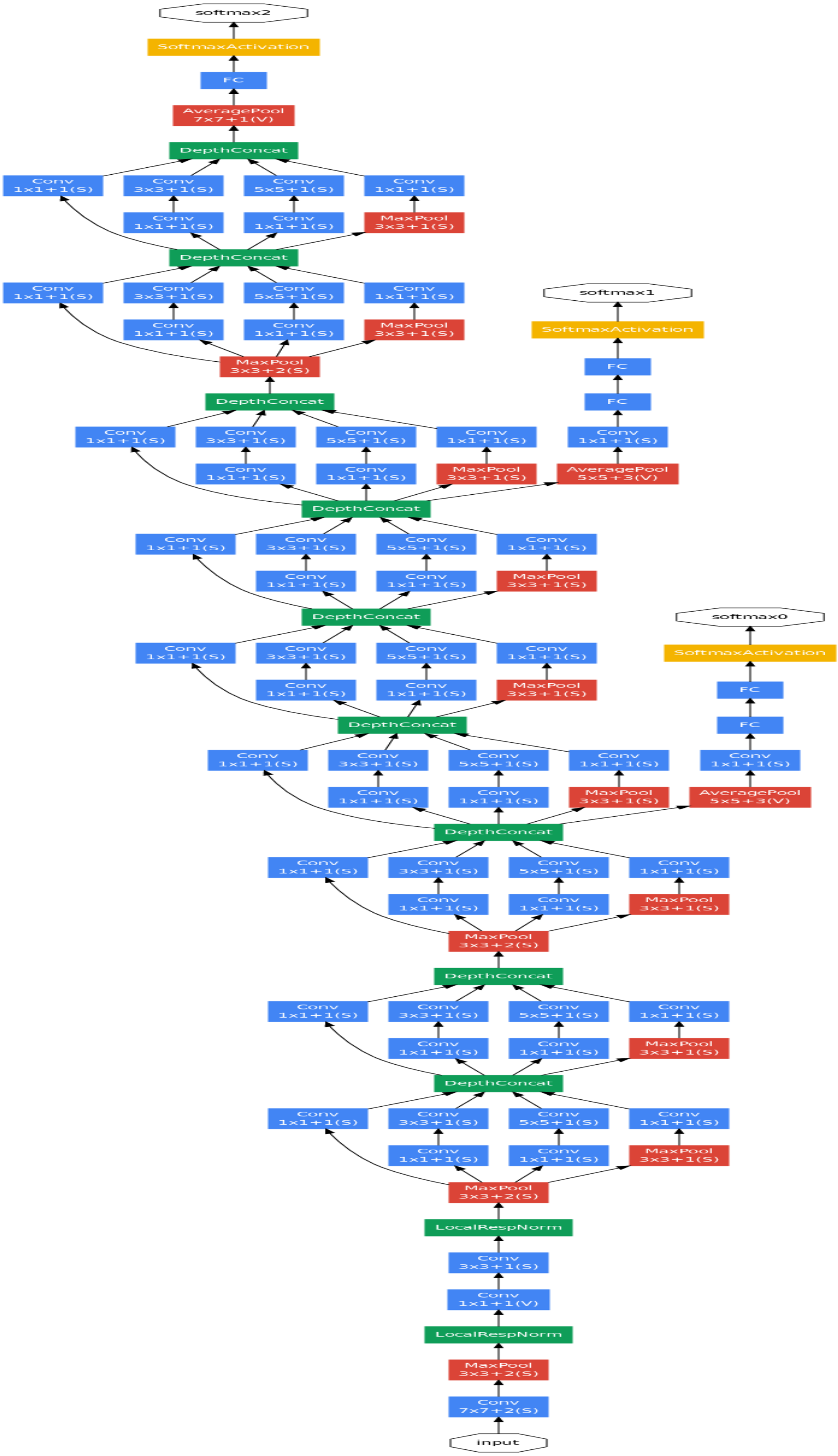

Fig. 1.11 *GoogLeNet* network structure [96]

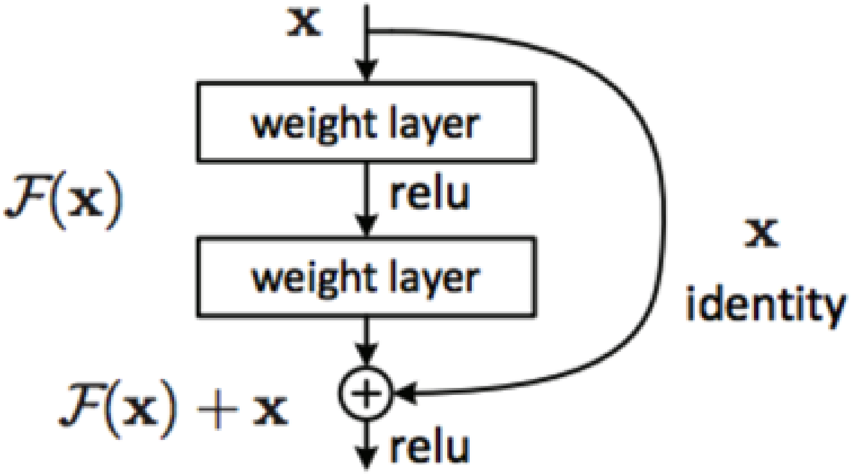
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Fig. 1.12 A residual learning block [44]

block, the input x go through *convolution* → *relu* → *convolution* stack, which can be thought as some function $\mathscr{F}(x)$. That result is added to the original input x. If the desired underlying mapping is denoted as as $\mathscr{H}(x)$, instead of hoping every few stacked layers directly fit a desired underlying mapping, i.e. $\mathscr{H}(x) = \mathscr{F}(x)$, Resnet lets these layers fit the residual mapping: $\mathscr{H}(x) = \mathscr{F}(x) + x$. In details, instead of computing that transformation from x to $\mathscr{F}(x)$, Resnet only computes the mapping that is added, $\mathscr{F}(x)$ to input x. In other words, the residual learning block computes the "delta" to the original input x to get a slightly altered representation in comparison to going from x to $\mathscr{F}(x)$ which is a brand new representation that does not keep any information about the original x in traditional CNNs. The reasoning of such setting is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. Figure 1.13 shows a 34-layer residual network in details.

1.3 DL challenges

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

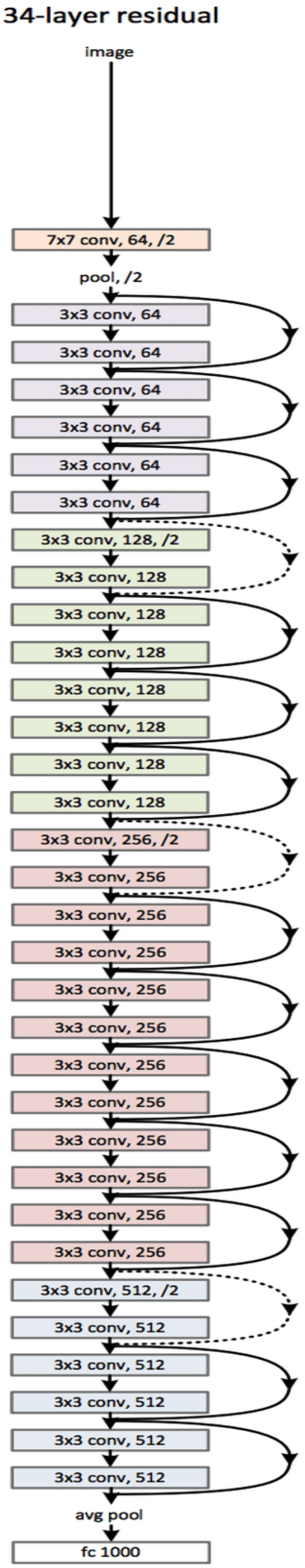

Fig. 1.13 Resnet network structure [44]

value from the training samples. 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 the 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, CNN applies multiple convolution operations (among others) to training data such as images, where the input image samples 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 for large dataset such as ImageNet. Memory efficiency is important since operations like convolution can use a great 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 computation 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 to debug.

The optimization of DL applications is domain specific since the meaning of the mathematical computation is not recognized by the programming language compiler. The optimization is also complicated in that the gradient-update step of the 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 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.

The current state-of-the-art frameworks, such as Caffe/Caffe 2 [51, 30], TensorFlow [1], MxNet [15], Torch7 [17], Theano/Pylearn2 [13, 39], and Computational Network Toolkit (CNTK) [111], are programming libraries with fixed bindings for key data structures such as tensors and opaque internal representation for control flow logic. Most of these frameworks represent the DL networks using some form of *directed acyclic graphs* (DAG), as known as computation graphs. The gradient derivation and optimization are based on graph transformation while runtime execution and memory management are based on the manipulation/optimization of the 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 the existing libraries through graph traversal and transformation are often less than optimal.

Computation graphs are not designed for user-level access, which makes it difficult to define customized DL applications and 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. Furthermore, these frameworks usually have numerous software dependencies and require platform dependent installation. Most of these frameworks directly or indirectly depend on languages such as

C/C++ that require to be compiled to specific platforms, which limits the portability of DL applications built with these frameworks.

To address these limitations, we developed DeepDSL, a domain specific language embedded in Scala, for encoding DL networks. DeepDSL differs from other DL frameworks in several aspects:

1. DeepDSL represents DL network as expressions where indexed scalars define tensor expressions, tensor functions define DL layers, and function compositions define DL networks. 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. Language-based representation is more flexible for optimization than the computation graph where parts of the graphs are tied to concrete data structures such as tensors and the graph's execution is by invocations of low-level code of a specific language such as C/C++.

2. Since DeepDSL program is compiled, it can statically detect errors such as incorrect network composition (as typing error) and report memory consumption at each computation step, the user can adjust the memory allocation strategy before actual execution.

3. DeepDSL program is compiled to Java source code[16] currently. Unlike other DL frameworks, 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.

[16] Generating target code in other languages is indeed relatively straightforward since the DeepDSL IR remains fully abstract before code generation. See Chapter 7 for the discussion of potential future support for target code in other language.

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 (Other DL frameworks usually have far more dependencies and are specific to language versions and operating systems).

1.4 Contributions

In this dissertation, we have designed and implemented a new domain specific language, DeepDSL, to facilitate the deep learning application development process. Our previous results have been published in the 5th International Conference on Learning Representations (ICLR) [115] and the full-length design and implementation details have been submitted to the Journal of Computer Languages, Systems & Structures [114][17]. The source code is now made open source[18]. Comparing with other current state-of-the-art frameworks, DeepDSL has the below key contributions:

- It directly encodes the mathematical representation of deep networks and stays fully abstract before code generation. Since the generated Java program has simple structure and is human readable, DeepDSL effectively provides two levels of interfaces: a high-level DSL interface for writing DL networks and a low-level Java interface for adjusting the training and the inference of DL networks;
- It provides programming language level optimization to improve memory and runtime efficiency;
- It provides a heuristic execution scheduler by analyzing dependencies statically to further optimize the memory usage;
- It supports static analysis for memory consumption and error detection, which allows the user to understand and analyze the exact memory usage at runtime before actual execution, as well as early bug detection in compile-time;

[17] Our previous DSL work for the multimedia information retrieval domain can be seen in [46].

[18] https://github.com/deepdsl/deepdsl.

- Unlike the existing DL frameworks, DeepDSL has minimum requirements on additional dependencies, besides a small JNI wrapper library for GPU operations;
- Unlike the existing DL frameworks, DeepDSL generates cross platform Java source code that runs independently without the existence of DeepDSL itself; this makes DeepDSL highly portable;
- DeepDSL generates highly efficient code that is both user friendly and easily debuggable.

Chapter 2

DeepDSL Overview

In this chapter we provide an overview of DeepDSL, the primary work of this dissertation. We introduce the main modules, workflow, core concepts of DeepDSL in Section 2.1, followed by the discussion of the principal DSL constructs using examples in Section 2.2. We will continue the extensive details with regard to DeepDSL's internals in Chapter 3 and 4.

2.1 DeepDSL

Domain specific language (DSL) A domain specific language (DSL) is, in contrast to a general-purpose language, a programming language or executable specification language that offers focused expressive power to a particular problem domain, via appropriate notations and abstractions [103]. The focused expressive power is the key characteristic in the definition that gives DSL its main benefits. Since DSLs are usually concise and only offer a restricted set of notations and abstractions, they are also denoted as micro-languages or little languages [12]. Since many DSLs are declarative and used to describe specifications, these declarative DSLs are also called application-specific language and their compilers are denoted as application generator [16].

The main feature of DSL is that it allows its program to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, users specialized in a domain understand and often develop DSL programs by themselves. DSL programs are concise, portable, and in many cases self-documenting. DSLs support fast prototyping and provide benefits such as productivity, reliability, usability, and maintainability [55]. The domain knowledge encoded in the DSL code is conserved and can be reused later. DSLs allow validation and optimization at the domain level [9]. DSLs can also improve testability by following approaches such as [91]. Four popular approaches to implement a DSL are listed below[1] [103]:

- Interpretation or compilation: This is the classic approach of implementing a new language. Either standard compilers [2] or tools dedicated to the implementation of DSLs (e.g. Draco [74] or ASF+SDF [102]) can be used.
- Embedded languages / domain-specific libraries: This approach defines the domain specific syntax, along with existing mechanisms such as definitions for functions or operators to build a library of domain-specific operations. The syntactic mechanism of the base (host) language is used to express the idiom of the domain. DeepDSL falls into this category where the base language is Scala.
- Preprocessing or macro processing: The domain specific constructs are translated to statements in the base language by a preprocessor in this approach (Examples include macro systems such as the Template Haskell or DSL embedding for the converge language [101]).
- Extensible compiler or interpreter: This approach is similar to the previous one except the preprocessing phase is integrated into the compiler and hence becomes part of the compile process.

DeepDSL DeepDSL is a DSL hosted in Scala and designed for deep learning. A high-level overview of DeepDSL is shown in Figure 2.1, 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 set of Java JNI interfaces (JCuda. JCudnn, and etc.) to invoke the underneath Cuda/Cudnn libraries.

[1] Other implementation techniques may also be used, e.g. aspect-oriented programming [54].

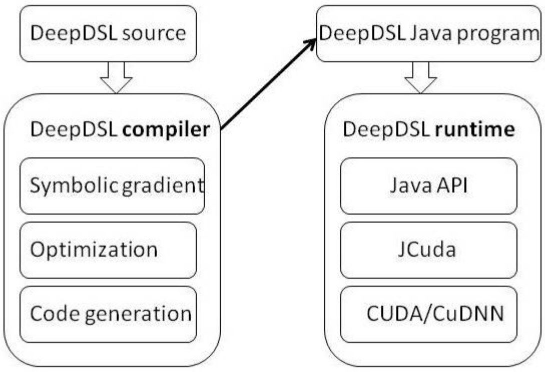

Fig. 2.1 An overview of Deep DSL

The core concepts of DeepDSL are abstract *tensor* and *scalar* expressions, as well as *tensor functions* that transform tensors to tensors and tensors to scalars. DeepDSL directly encodes the mathematical representation of deep networks, where each layer is represented as a tensor function and the entire network is represented as the 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 the 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.

Since DeepDSL programs are fully abstract, DeepDSL compiler can statically infer the dimensions of tensors in each layer, check whether the layers are properly connected, and automatically insert tensor reshaping operations as necessary. Errors caused by incorrect parameter dimensions are caught at compile-time before code generation.

Modification to the the network and layer definitions can be made with minimum effort. For example, instead of defining the dimensions of the input/output tensors of the convolution layer as $N \times C \times M1 \times M2$, where N is the batch size, C is the number of channels, and M_1/M_2 are image dimensions, we can switch to $N \times M1 \times M2 \times C$ by just changing a few of lines of DSL code, which slightly improves performance.

Changing execution strategy of DeepDSL is relatively painless as well. 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 the 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 supports a few computation mechanism and platforms. If JBLAS is supported, DSL compiler can generate calls to BLAS library for tensor products. If GPU is available, DSL compiler can generate CUDA calls to run on Nvidia GPU and OpenCL calls to run on AMD or Intel GPU for tensor products.

2.2 DeepDSL core constructs

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* (See Section 1.2.1 for details) is defined using DeepDSL.

2.2.1 Tensor

The core construct 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 size, channel, width, and height of input samples. If `T._new(F, C, K1, K2)`, where `F, C, K1, K2` are the dimensions of filter numbers, channel, kernel width, and kernel height, represents a convolution kernel `k`, 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[2].

The code snippet below 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.

```
val x = T._new(N, M1)
val w = T._new(M2, M1)

T.sum(M1, k => x(i, k) * w(j, k))
```

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` as follows.

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

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.

2.2.2 Tensor Function

In order to compose the fully connected layer with other layers, we can use the tensor function below 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)
```

[2] In general, given stride S and padding P, the last two dimensions are (X-K1+2P)/S+1 and (Y-K2+2P)/S+1.

```
 )
)
```

Putting everything together, the Scala method below 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)
  )
 )
}
```

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 DL network.

2.2.3 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` 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))
}
```

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 the activation layer. Despite the difference, the two forms of tensor expressions are treated uniformly in 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.

2.2.4 Function Application and Composition

Just like the tensors and tensor functions, the function applications are also abstract.

```
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))
```

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
```

2.2.5 Network as Function Composition

Using some helper functions, we can define the *LeNet* network as below.

```
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
```

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[3]. 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`.

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

2.2.6 Training

The loss expression `c` of *LeNet* (last line below) 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
```

[3] In Layer.flatten(4, 1), 4 is the number of dimensions and 1 is the index where the collapsing starts.

```
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)
```

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 as follows.

```
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)
```

We first extract the network parameters using the call `c.freeParam`. The variable `solver` is simply a collection of parameters that include the output file name, train iteration, test iteration, learning rate, momentum, weight decay, gradient clipping bound. The variable `mnist` refers to the MNIST[4] 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 where the Java source code should be generated.

[4] http://yann.lecun.com/exdb/mnist.

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.

Chapter 3

Formalization

In this chapter, we present the formal definition of DeepDSL that includes an abstract syntax in Section 3.1, an operational semantics in Section 3.2, a type system in Section 3.3, examples of the application of the typing rules to some important CNN building blocks in Section 3.4. We conclude this chapter by discussing our accommodation for gradient descent in Section 3.5.

3.1 Syntax

An abstract syntax for DeepDSL is shown in Figure 3.1, 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 add, a scalar-tensor product, or a cast. A scalar is either a function application, a constant, 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 2.2.2. Dimension expressions are used for defining the dimensions of operations such as convolution and pooling. For example, the dimension of

e	$::=$	t	tensor
	$\mid$	$\hat{s}$	scalar
t	$::=$	$\mathcal{V}$	tensor value
	$\mid$	x, y, z, w	variables
	$\mid$	$(x \Rightarrow t)(t)$	application
	$\mid$	$\mathcal{F}(t)$	built-in function application
	$\mid$	$(i_x^1 \cdots i_x^k) \Rightarrow s$	tensor expression
	$\mid$	$t_1 + t_2$	tensor add
	$\mid$	$s \cdot t$	scalar tensor product
	$\mid$	$(D)\ x$	cast
$\hat{s}$	$::=$	$(x \Rightarrow \hat{s})(t)$	application
	$\mid$	s	
s	$::=$	v	constant
	$\mid$	$x(I)$	tensor element
	$\mid$	$\mathcal{V}(I)$	tensor element
	$\mid$	$\sum_{i_x^1 \cdots i_x^k}(s)$	summation
	$\mid$	$\log(s) \mid \exp(s) \mid s^n$	arithmetic expression
	$\mid$	$s_1 + s_2 \mid s_1 \times s_2$	arithmetic expression
f	$::=$	$x \Rightarrow e$	tensor function
	$\mid$	$\mathcal{F}$	built-in tensor function
$\mathcal{F}$	$::=$	convolution $\mid$ pooling $\mid$ activation $\mid \ldots$	
I	$::=$	$i \mid i \cdot I$	index list
i	$::=$	m	index value
	$\mid$	$i \times n \mid i + i' \mid i - n$	index exppression
	$\mid$	i_x	index variable
D	$::=$	$d \mid d \cdot D$	dimension list
d	$::=$	n	dimension value
	$\mid$	$n \mid d/n \mid d - d' \mid d + n$	dimension expression
	$\mid$	$d_x^1 \times \cdots \times d_x^k$	dimension product
	$\mid$	d_x	dimension variable
τ	$::=$	$\star \mid D$	types

Fig. 3.1 A formal syntax for DeepDSL, where n is some positive integer and $f_2 \circ f_1$ is defined as $x \Rightarrow f_2(f_1(x))$. Note that the shaded syntax is for intermediate representation at runtime.

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 axes of a 4-D tensor of dimensions $d_1 \cdot d_2 \cdot d_3 \cdot d_4$, the dimension becomes $d_1 \cdot (d_2 \times d_3 \times d_4)$.

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 built-in functions (denoted by $\mathcal{F}$) 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 the single-parameter function here for simplicity.

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

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 $\mathcal{D}(i_x)$ 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. Note that by the definition of the syntax, the scalar expression s may not contain a function application or any tensor expressions other than tensor variables.

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

$$\frac{t \to t'}{(x \Rightarrow e)(t) \to (x \Rightarrow e)(t')} \quad \text{E-App}$$

$$\frac{\sigma = \mathcal{U}(\mathcal{T}(x), \mathcal{D}(\mathcal{V}))}{(x \Rightarrow e)(\mathcal{V}) \to \sigma(e[\mathcal{V}/x])} \quad \text{E-App2}$$

$$\frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2} \quad \text{E-Plus} \qquad \frac{t_2 \to t_2'}{\mathcal{V} + t_2 \to \mathcal{V} + t_2'} \quad \text{E-Plus2}$$

$$\frac{\mathcal{V} = (v_0 + v_0', \ldots, v_m + v_m')^{n_1 \cdots n_k}}{(v_0, \ldots, v_m)^{n_1 \cdots n_k} + (v_0', \ldots, v_m')^{n_1 \cdots n_k} \to \mathcal{V}} \quad \text{E-ValuePlus}$$

$$\frac{s \to s'}{s \cdot t \to s' \cdot t} \quad \text{E-ScalarTensorProd} \qquad \frac{t \to t'}{v \cdot t \to v \cdot t'} \quad \text{E-ScalarTensorProd2}$$

$$v \cdot (v_0, \ldots, v_m)^{n_1 \cdots n_k} \to (v \times v_0, \ldots, v \times v_m)^{n_1 \cdots n_k} \quad \text{E-ValueScale}$$

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

$$\frac{t \to t'}{\mathcal{F}(t) \to \mathcal{F}(t')} \quad \text{E-BuiltinApp}$$

$$\frac{\mathcal{D}(\mathcal{F}) = \mathcal{D}(\mathcal{V}) \to \mathcal{D}(\mathcal{V}')}{\mathcal{F}(\mathcal{V}) \to \mathcal{V}'} \quad \text{E-BuiltinApp2}$$

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

$$\frac{i_1 \to^* m_1, \ldots, i_k \to^* m_k}{\mathcal{V}(i_1 \cdots i_k) \to \mathcal{V}(m_1 \cdots m_k)} \quad \text{E-IndexSub}$$

$$\frac{\mathcal{V} = (v_0, \ldots, v_m)^{n_1 \cdots n_k} \quad 0 \leq m_1 \leq n_1 \ldots 0 \leq m_k \leq n_k}{\mathcal{V}(m_1 \cdots m_k) \to v_{(\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k}} \quad \text{E-TensorElement}$$

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

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

Fig. 3.2 Operational Semantics where $m = (n_1 \times \ldots \times n_k) - 1$

3.2 Semantics

An operational semantics for tensors and scalars is shown in Figure 3.2 and Figure 3.3[1]. In this semantics, each tensor expression evaluates to a tensor value $\mathcal{V}$, 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.

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

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

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

[1] Note we omit the straightforward operational semantics and typing rules for index and dimension expressions.

$$\frac{s_1 \to s_1'}{s_1 + s_2 \to s_1' + s_2} \quad \text{E-ScalarPlus}$$

$$\frac{s_2 \to s_2'}{v + s_2 \to v + s_2'} \quad \text{E-ScalarPlus2}$$

$$\frac{s_1 \to s_1'}{s_1 * s_2 \to s_1' * s_2} \quad \text{E-ScalarTimes}$$

$$\frac{s_2 \to s_2'}{v * s_2 \to v * s_2'} \quad \text{E-ScalarTime2}$$

$$\frac{s \to s'}{log(s) \to log(s')} \quad \text{E-ScalarLog}$$

$$\frac{s \to s'}{exp(s) \to exp(s')} \quad \text{E-ScalarNatExp}$$

$$\frac{s \to s'}{s^n \to (s')^n} \quad \text{E-ScalarExp}$$

Fig. 3.3 Operational Semantics - other types of scalar expressions

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

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

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 built-in function $\mathcal{F}$ to a tensor value $\mathcal{V}$ will return a tensor value $\mathcal{V}'$ so that the types of $\mathcal{V}$ and $\mathcal{V}'$ match the the parameter and return type of $\mathcal{F}$.

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 $\mathcal{D}(i_1) \cdots \mathcal{D}(i_k)$. 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 $\mathcal{V}(m_1 \cdots m_k)$ evaluates to either jth element of $\mathcal{V}$, 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 $I \Rightarrow s$.

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 3.4 and 3.5.

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 $\mathcal{U}(\tau_1, \tau'_1)$, 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.

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

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

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

$$\Gamma \vdash \mathcal{F} : \mathcal{T}(\mathcal{F}) \qquad \text{T-Builtin}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash f : \tau_1 \to \tau_2 \\ \Gamma \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')\end{array}}{\Gamma \vdash f(t) : \sigma(\tau_2)} \qquad \text{T-App}$$

$$\Gamma \vdash (v_0, \ldots, v_m)^{n_1 \cdots n_k} : n_1 \cdots n_k \qquad \text{T-TensorValue}$$

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

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

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

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

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

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

Fig. 3.4 Typing rules for tensor and scalar expressions

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 3.6.

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

$$\dfrac{\Gamma \vdash s_1 : \star \quad \Gamma \vdash s_2 : \star}{\Gamma \vdash s_1 + s_2 : \star} \qquad \text{T-ScalarPlus}$$

$$\dfrac{\Gamma \vdash s_1 : \star \quad \Gamma \vdash s_2 : \star}{\Gamma \vdash s_1 \times s_2 : \star} \qquad \text{T-ScalarTimes}$$

$$\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash log(s) : \star} \qquad \text{T-ScalarLog}$$

$$\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash exp(s) : \star} \qquad \text{T-ScalarExp}$$

$$\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash s^n : \star} \qquad \text{T-ScalarExp2}$$

Fig. 3.5 Typing rules for arithmetic scalar expressions

$$\Gamma \vdash m : d \qquad \text{T-IndexConstant}$$

$$\dfrac{\Gamma(i_x) = d}{\Gamma \vdash i_x :\ d} \qquad \text{T-IndexVar}$$

$$\dfrac{\Gamma \vdash i :\ d \quad \Gamma \vdash I :\ D}{\Gamma \vdash i \cdot I :\ d \cdot D} \qquad \text{T-IndexList}$$

$$\dfrac{\Gamma \vdash i :\ (d-1)/n+1}{\Gamma \vdash i \times n :\ d} \qquad \text{T-IndexTimes}$$

$$\dfrac{\Gamma \vdash i_1 :\ d_1 - d_2 + 1 \quad \Gamma \vdash i_2 :\ d_2}{\Gamma \vdash i_1 + i_2 :\ d_1} \qquad \text{T-IndexPlus}$$

$$\dfrac{\Gamma \vdash i :\ d + 2n}{\Gamma \vdash i - n :\ d} \qquad \text{T-IndexMinus}$$

Fig. 3.6 Typing rule for index expressions

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 3.6.

$$\Gamma(i_1) = \mathcal{D}(i_1), \text{ where}$$

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\mathcal{D}(i_1) = (\mathcal{T}(x) + 2n_2 - \mathcal{D}(i_2))/n_1 + 1}{\Gamma \vdash i_1 :\ (\mathcal{T}(x) + 2n_2 - \mathcal{D}(i_2))/n_1 + 1}}{\Gamma \vdash i_1 \times n_1 :\ \mathcal{T}(x) + 2n_2 - \mathcal{D}(i_2) + 1} \\ \Gamma \vdash i_2 :\ \mathcal{D}(i_2)\end{array}}{\Gamma \vdash i_1 \times n_1 + i_2 :\ \mathcal{T}(x) + 2n_2}}{\Gamma \vdash i_1 \times n_1 + i_2 - n_2 :\ \mathcal{T}(x)}$$

The type of the index expression is based on the type derivation of $(i_1) \Rightarrow \sum_{i_2} x(i_1 \times n_1 + i_2 - n_2) \times w(i_2)$.

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

Up until now, we have introduced a complete type system for DeepDSL. The soundness of this type system is described in Theorem 3.1. The proof of the theorem is provided in the Appendix chapter.

Theorem 3.1. *If $\emptyset \vdash t : \tau$, then there exists $\mathcal{V}$ such that $t \rightarrow^* \mathcal{V}$. If $\emptyset \vdash \hat{s} : \tau$, then there exists v such that $\hat{s} \rightarrow^* v$.*

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 $\mathcal{D}(i_k) = d_k$ 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

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

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

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

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 $\texttt{flat}(d_1 \cdot (d_2 \times d_3 \times d_4))$.

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

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.

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

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

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 built-in function. The type of f_3 is

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

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

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 $\mathcal{T}(x) = d_n \cdot d_k$. Note that y is a $d_n \times d_k$ matrix, where the 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 $\mathcal{T}(x) = d_n \cdot d_k$, $\mathcal{D}(i_n) = d_n$, $\mathcal{D}(i_k) = \mathcal{D}(i_l) = d_k$. 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 implementations of softmax in libraries so that we can treat it as a built-in function.

3.5 Gradient derivation

The forward inference of a DNN computes a loss expression s that measures the error of the predicted results against the ground truth values 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 $\frac{\partial t}{\partial x}$ appears in 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 3.1.

$$\frac{\partial f(t)}{\partial y} = f'(t) \cdot \frac{\partial t}{\partial y} + \frac{\partial f}{\partial y}(t) \tag{3.1}$$

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 $\frac{\partial f}{\partial y}(t)$, the first or second half of the right-hand side of Rule 3.1. 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 3.1 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 evaluation 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 [8, 66] to derive parameter gradients efficiently without redundancy.

Reverse accumulation Before computing gradient derivation of a loss expression s (assuming s contains a set of tensor subexpressions $\hat{t}_i$ where $i = 1, \cdots, n$), we first perform the static single assignment (SSA) transformation to s, which results 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.

$$x \rightsquigarrow x$$

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

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

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

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

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

Fig. 3.7 Static single assignment transformation, where y variables are fresh and we assume that scalar function application does not appear inside a tensor expression.

Let a represent a sequence of assignments.

$$\begin{array}{lcl} a & ::= & \epsilon \\ & | & y \leftarrow \hat{t};\ a \end{array}$$

Figure 3.7 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.

$$\begin{array}{lcll} t & ::= & \mathcal{V} & \text{tensor value} \\ & | & x, y, z, w & \text{variables} \\ & | & \mathcal{F}(y) & \text{built-in function application} \\ & | & I \Rightarrow s & \text{tensor exppression} \\ & | & y_1 + y_2 & \text{tensor expression} \\ & | & s \cdot y & \text{scalar tensor product} \\ & | & (D)\ x & \text{cast} \\ s & ::= & v & \text{constant} \\ & | & x(I) & \text{tensor element} \\ & | & \mathcal{V}(I) & \text{tensor element} \\ & | & \sum_I(s) & \text{summation} \\ & | & \log(s) \mid \exp(s) \mid s^n & \text{arithmetic expression} \\ & | & s_1 + s_2 \mid s_1 \times s_2 & \text{arithmetic expression} \\ & & \ldots & \end{array}$$

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. Similarly, the gradient $\frac{\partial e}{\partial t_j}$ 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 is 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 $\frac{\partial \hat{t}_i}{\partial y_j}$ for each $\hat{t}_i$ and for each y_j in $\hat{t}_i$.

2. Define the variable z_j with the assignments

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

   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 $s = f_4(f_3(f_2(f_1(x))))^2$, applying SSA transformation to s, we obtain the following result.

[2] Note in this example only one parameter exists for each function so that no summation as shown in item 2 is needed.

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

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

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

$$
\begin{aligned}
& \text{convolution\_forward}(y_1) && f_2(y_1) \\
& \text{convolution\_backward\_data}(z_2,\, y_1) && z_2 \cdot f_2'(y_1) \\
& \text{convolution\_backward\_filter}(z_2,\, y_1) && z_2 \cdot \frac{\partial f_2}{\partial w_2}(y_1)
\end{aligned}
$$

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 the following tensor and scalar gradient derivation rules to derive gradients of the form $f'(y)$ and $\dfrac{\partial f}{\partial w}(y)$ (Note the partial-derivative operator ∂ extends to the rightmost expressions that in these rules).

Gradient derivation rules

$$\frac{\partial\ t_1 + t_2}{\partial y} = \frac{\partial t_1}{\partial y} + \frac{\partial t_2}{\partial y} \tag{3.2}$$

$$\frac{\partial\ s \cdot t}{\partial y} = s \cdot \frac{\partial t}{\partial y} \tag{3.3}$$

$$\frac{\partial\ (D)\ x}{\partial y} = (D')\frac{\partial x}{\partial y} \text{ where } D' = \mathcal{T}(x) \tag{3.4}$$

$$\frac{\partial x}{\partial y} = \frac{\partial\ I \Rightarrow x(I)}{\partial y} \text{ where } \mathcal{D}(I) = \mathcal{T}(x) \tag{3.5}$$

$$\frac{\partial\ I \Rightarrow s}{\partial y} = I \Rightarrow \frac{\partial s}{\partial y} \tag{3.6}$$

$$\frac{\partial s}{\partial y} = I \Rightarrow \frac{\partial s}{\partial y(I)} \text{ where } \mathcal{D}(I) = \mathcal{T}(y) \tag{3.7}$$

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

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

Note that the more obvious reduction of $\frac{\partial x}{\partial y}$ 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 Rule 3.5.

Scalar derivation rules The derivation of $\frac{\partial s}{\partial y(I)}$ is defined in Rules 3.8 through 3.15. Most of scalar derivation rules are standard except (Note Rule 3.14 is a generic form of Rule 3.10, which says that the derivative of the sum $\sum_{I'} s$ is the sum of $\frac{\partial s}{\partial y(I)}$) Rule 3.15, which 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 [61].

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

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.

$$\frac{\partial \log(s)}{\partial y(I)} = s^{-1} \times \frac{\partial s}{\partial y(I)} \tag{3.8}$$

$$\frac{\partial \exp(s)}{\partial y(I)} = \exp(s) \times \frac{\partial s}{\partial y(I)} \tag{3.9}$$

$$\frac{\partial\ s_1 + s_2}{\partial y(I)} = \frac{\partial s_1}{\partial y(I)} + \frac{\partial s_2}{\partial y(I)} \tag{3.10}$$

$$\frac{\partial\ s_1 \times s_2}{\partial y(I)} = s_2 \times \frac{\partial s_1}{\partial y(I)} + s_1 \times \frac{\partial s_2}{\partial y(I)} \tag{3.11}$$

$$\frac{s^n}{\partial y(I)} = (n \times s^{n-1}) \times \frac{\partial s}{\partial y(I)} \tag{3.12}$$

$$\frac{\partial n}{\partial y(I)} = 0 \tag{3.13}$$

$$\frac{\partial \sum_{I'}(s)}{\partial y(I)} = \sum_{I'} \frac{\partial s}{\partial y(I)} \tag{3.14}$$

$$\frac{\partial x(I)}{\partial y(I')} = \begin{cases} 0 & \text{if } x \neq y \\ \mathsf{match}(I, I') & \text{otherwise} \end{cases} \tag{3.15}$$

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.

$$\begin{array}{rcll}
t & ::= & \ldots & \\
& | & z \cdot g & \text{tensor gradient product} \\
g & ::= & I \Rightarrow I' \Rightarrow s & \text{gradient expression} \\
& | & g_1 + g_2 & \text{gradient sum} \\
& | & s \cdot g & \text{scalar gradient product} \\
& | & (D)\ g & \text{gradient cast} \\
& | & \dfrac{\partial \mathcal{F}}{\partial x}(y) & \text{gradient tensor application} \\
s & ::= & \ldots & \\
& | & \delta_{i\ i'} & \text{Kronecker delta}
\end{array}$$

Chapter 4

Optimization

As discussed in Section 1.3, optimization is critical to the efficiency of DL applications, which can be implemented at high level or low level. In this section, we present the DeepDSL high- and low-level optimization strategy in Section 4.1 and Section 4.2, respectively. We describe our code scheduling strategy in Section 4.3 that offers further optimization with respect to memory efficiency. We conclude this chapter with the discussion for code generation and runtime in Section 4.4.

4.1 High-level Optimization

4.1.1 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 $\dfrac{\partial \mathcal{F}}{\partial x}(y)$ are eliminated from the parameter gradients.

$$x \cdot (I \Rightarrow I' \Rightarrow s) \quad = \quad I' \Rightarrow \sum_{I} x(I) \times s \tag{4.1}$$

$$x \cdot (g_1 + g_2) \quad = \quad x \cdot g_1 + x \cdot g_2 \tag{4.2}$$

$$x \cdot (s \cdot g) \quad = \quad s \cdot (x \cdot g) \tag{4.3}$$

$$x \cdot (D)\ g \quad = \quad (D)\ x \cdot g \tag{4.4}$$

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 index syntax in Figure 3.1, 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 $f^{-1}(i')$.

$$\sum_{I} s + s' = \sum_{I} s + \sum_{I} s' \tag{4.5}$$

$$\sum_{i_x} s \times s' = s \times \sum_{i_x} s' \quad i_x \text{ is not used in } s \tag{4.6}$$

$$\sum_{i_x} \delta_{f(i_x)\ i'} \times s = s[f^{-1}(i')/i_x] \tag{4.7}$$

$$\sum_{i_x} \delta_{f(i_x)\ i'} = 1 \tag{4.8}$$

$$s \times (s_1 + s_2) = s \times s_1 + s \times s_2 \tag{4.9}$$

$$s \cdot (s' \cdot t) = (s \times s') \cdot t \tag{4.10}$$

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 the 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 3.6 and 3.7, which reduces to $I_2 \Rightarrow \sum_{I_1} z(I_1) \times \frac{\partial s}{\partial x(I_2)}$ by Rule 4.1. 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.

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

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

Indices of the form $i_1 - i_2$ 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 simplification 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 $(D)\ I \Rightarrow 0$.

4.1.2 Vectorization

Tensors of the form of $\mathcal{F}(y)$ and $z \cdot \dfrac{\partial \mathcal{F}}{\partial x}(y)$ can be mapped to high-level functions in libraries such as Cudnn that work with vectors or matrices. This section we focus on

$$I \Rightarrow s^n = (I \Rightarrow s)^n \tag{4.11}$$

$$I \Rightarrow \exp(s) = \exp(I \Rightarrow s) \tag{4.12}$$

$$I \Rightarrow \log(s) = \log(I \Rightarrow s) \tag{4.13}$$

$$I \Rightarrow s_1 \times s_2 = s_1 \cdot (I \Rightarrow s_2) \qquad \forall i \in \{I\},\ i \text{ is not used in } s_1 \tag{4.14}$$

$$I \Rightarrow s_1 \times s_2 = (I \Rightarrow s_1) .* (I \Rightarrow s_2) \tag{4.15}$$

$$I \Rightarrow s_1 + s_2 = (I \Rightarrow s_1) + (I \Rightarrow s_2) \tag{4.16}$$

$$I \Rightarrow x(I) = x \qquad \mathcal{T}(x) = \mathcal{D}(I) \tag{4.17}$$

$$\sum_I s_1 \times s_2 = s_1 \times \sum_I s_2 \qquad \forall i \in \{I\},\ i \text{ is not used in } s_1 \tag{4.18}$$

$$\sum_I s_1 \times s_2 = (I \Rightarrow s_1) \cdot (I \Rightarrow s_2) \tag{4.19}$$

$$\sum_I s = \sum (I \Rightarrow s) \tag{4.20}$$

$$I \Rightarrow \sum I' \Rightarrow s = \sum_{|I'|} (I \cdot I') \Rightarrow s \tag{4.21}$$

$$I \Rightarrow (I' \Rightarrow s_1) \cdot (I' \Rightarrow s_2) = (I_1 \cdot I') \Rightarrow s_1 \ \times_{|I'|} \ (I_2 \cdot I') \Rightarrow s_2 \qquad \begin{array}{l} I = I_1 \cdot I_2 \\ \forall i \in \{I_1\},\ i \text{ is not used in } s_2 \\ \forall i \in \{I_2\},\ i \text{ is not used in } s_1 \end{array} \tag{4.22}$$

$$I \Rightarrow s = I \Rightarrow (I' \Rightarrow s)(I') \qquad \begin{array}{l} s \text{ is not a tensor element, } \{I'\} \subset \{I\} \\ \forall i \in \{I\} \backslash \{I'\},\ i \text{ is not used in } s \end{array} \tag{4.23}$$

Fig. 4.1 Rules for vectorization, where $\{I\}$ represents the set of indices in I.

transforming tensor expressions into the computation of vectors and matrices so that they can be directly mapped to low-level functions in high-performance libraries. We call this reduction step vectorization.

Vectorization rules Figure 4.1 shows the vectorization rules, where Rule 4.11 to 4.16 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 4.12, this is reduced to $\exp(I \Rightarrow s)$, which is the exponentiation of the tensor $I \Rightarrow s$. In Rule 4.15, the product of scalars becomes the pointwise product $.*$ of tensors.

Rule 4.17 simplifies $I \Rightarrow x(I)$ to just x if the dimensions of I are the same as the type of x. Rule 4.18 factors out scalars independent of the sum indices. Rule 4.19 reduces the sum of scalar products to the dot product of two tensors. The sum of scalars is reduced to sum of tensor Rule 4.20. Rule 4.21 turns a tensor expression that contains a tensor sum into the partial sum of a tensor.

Rule 4.22 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 4.19 and 4.22. However, there are more efficient implementation in existing libraries such as Cudnn so that convolution should be treated as built-in functions.

There are cases that not all indices in I are used in s in an tensor expression $I \Rightarrow s$. Rule 4.23 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,

$$\begin{aligned}
& (i_1 \cdot i_2) \Rightarrow \sum i_3 \Rightarrow x(i_1 \cdot i_3) & \\
= \ & (i_1 \cdot i_2) \Rightarrow (i_1 \Rightarrow \sum i_3 \Rightarrow x(i_1 \cdot i_3))(i_1) & \text{Rule 4.23} \\
= \ & (i_1 \cdot i_2) \Rightarrow (\sum_1 (i_1 \cdot i_3) \Rightarrow x(i_1 \cdot i_3))(i_1) & \text{Rule 4.21} \\
= \ & (i_1 \cdot i_2) \Rightarrow (\sum_1 x)(i_1) & \text{Rule 4.17}
\end{aligned}$$

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.

Syntax after vectorization The vectorization rules introduce a few types of expressions as shown in Figure 4.2, some of which have direct correspondence to low-level functions in libraries. The expression $\exp(t)$, $\log(t)$, or t^n is the exponentiation, logarithm, or power of tensor, which is the same as applying exponentiation, logarithm, or power to the tensor elements. For these expressions, the tensor can be treated as a vector so that each operation can be mapped to the corresponding exponentiation, logarithm, or power library function that is applied to the 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 dot product of two vectors.

$$
\begin{array}{rcll}
t & ::= & \ldots & \\
 & | & z \cdot \dfrac{\partial \mathcal{F}}{\partial x}(y) & \text{tensor gradient product} \\
 & | & \exp(t) \mid \log(t) \mid t^n & \\
 & | & t_1 .* t_2 & \text{pointwise product} \\
 & | & t_1 \times_n t_2 & \text{tensor contraction} \\
 & | & \sum_n t & \text{partial summation of tensor} \\
s & ::= & \ldots & \\
 & | & t_1 \cdot t_2 & \text{dot product} \\
 & | & \sum t & \text{summation of tensor} \\
 & | & t(I) & \text{tensor element}
\end{array} \tag{4.24}
$$

Fig. 4.2 Syntax after vectorization

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[1] 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 dot products of the lower n axis of t_1 and t_2 and it has the typing rule:

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

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

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

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 dot 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 $\mathcal{D}(I_2)$. 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 $\mathcal{D}(I_1)$, 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 $\mathcal{D}(I_1)$. 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 $\mathcal{D}(I_2)$.

4.1.3 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 4.3, the tensor t_y is the output of affine transformation with input x, weight w, and bias b. Figure 4.3 also shows the reduction of $\dfrac{\partial t_y}{\partial x}$ and the backward gradient of x z_x, which is $z_y \cdot \dfrac{\partial t_y}{\partial x}$, where z_y is the backward gradient of t_y.

$$
\begin{aligned}
t_y &= (i_1 \cdot i_3) \Rightarrow \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2 \cdot i_3) + b(i_1) \\
\frac{\partial t_y}{\partial x} &= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \frac{\partial \sum_{i_2} x(i_1 \cdot i_2) \times w(i_2 \cdot i_3) + b(i_1)}{\partial x(i'_1 \cdot i'_2)} && \text{Rule 3.6, 3.7} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \sum_{i_2} \frac{\partial x(i_1 \cdot i_2)}{\partial x(i'_1 \cdot i'_2)} \times w(i_2 \cdot i_3) && \text{Rule 3.10, 3.11, 3.15} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \sum_{i_2} \delta_{i_1\ i'_1} \times \delta_{i_2\ i'_2} \times w(i_2 \cdot i_3) && \text{Rule 3.15} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times \sum_{i_2} \delta_{i_2\ i'_2} \times w(i_2 \cdot i_3) && \text{Rule 4.6} \\
&= (i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3) && \text{Rule 4.7}
\end{aligned}
$$

$$
\begin{aligned}
z_x = z_y \cdot \frac{\partial t_y}{\partial x} &= z_y \cdot ((i_1 \cdot i_3) \Rightarrow (i'_1 \cdot i'_2) \Rightarrow \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3)) \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_1} \sum_{i_3} z_y(i_1 \cdot i_3) \times \delta_{i_1\ i'_1} \times w(i'_2 \cdot i_3) && \text{Rule 4.1} \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_1} \delta_{i_1\ i'_1} \times \sum_{i_3} z_y(i_1 \cdot i_3) \times w(i'_2 \cdot i_3) && \text{Rule 4.6} \\
&= (i'_1 \cdot i'_2) \Rightarrow \sum_{i_3} z_y(i'_1 \cdot i_3) \times w(i'_2 \cdot i_3) && \text{Rule 4.8} \\
&= (i'_1 \cdot i'_2) \Rightarrow (i_3 \Rightarrow z_y(i'_1 \cdot i_3)) \cdot (i_3 \Rightarrow w(i'_2 \cdot i_3)) && \text{Rule 4.19} \\
&= ((i'_1 \cdot i_3) \Rightarrow z_y(i'_1 \cdot i_3)) \times_1 ((i'_2 \cdot i_3) \Rightarrow w(i'_2 \cdot i_3)) && \text{Rule 4.22} \\
&= z_y \times_1 w && \text{Rule 4.17}
\end{aligned}
$$

Fig. 4.3 Gradient derivation of affine transformation

Flattening In flattening 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.

$$
\begin{aligned}
t_y &= (d_1 \cdot (d_2 \times d_3 \times d_4))\ x \\
z_x &= z_y \cdot \frac{\partial t_y}{\partial x} \\
&= z_y \cdot \frac{(d_1 \cdot (d_2 \times d_3 \times d_4))\ x}{\partial x} \\
&= z_y \cdot (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\ \frac{\partial x}{\partial x} && \text{Rule 3.4} \\
&= (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\ z_y \cdot \frac{\partial x}{\partial x} && \text{Rule 4.4} \\
&= (d_1 \cdot d_2 \cdot d_3 \cdot d_4)\ z_y
\end{aligned}
$$

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

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

$$\begin{aligned}
\frac{\partial t_y}{\partial x} &= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow \frac{\partial \sum_{i_c} \sum_{i_u} \sum_{i_v} x(i_n \cdot i_c \cdot i_h + i_u \cdot i_w + i_v) \times w(i_k \cdot i_c \cdot i_u \cdot i_v) + b(i_k)}{\partial x} && \text{Rule 3.6} \\
&= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_c}\sum_{i_u}\sum_{i_v} \delta_{i_n\ i'_n} \times \delta_{i_c\ i'_c} \times \delta_{i_h + i_u\ i_r} \times \delta_{i_w + i_v\ i_s} \times w(i_k \cdot i_c \cdot i_u \cdot i_v) && \text{Rule 3.7} \\
& && 3.10 \\
& && 3.11 \\
& && 3.15 \\
&= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times \sum_{i_c}\sum_{i_u}\sum_{i_v} \delta_{i_c\ i'_c} \times \delta_{i_h + i_u\ i_r} \times \delta_{i_w + i_v\ i_s} \times w(i_k \cdot i_c \cdot i_u \cdot i_v) && \text{Rule 4.6} \\
&= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times w(i_k \cdot i_c \cdot i_u \cdot i_v)[i'_c/i_c,\ i_r - i_h/i_u,\ i_s - i_w/i_v] && \text{Rule 4.7} \\
&= (i_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \delta_{i_n\ i'_n} \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)
\end{aligned}$$

$$\begin{aligned}
z_x &= z_y \cdot \frac{\partial t_y}{\partial x} \\
&= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_n}\sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i_n \cdot i_k \cdot i_h \cdot i_w) \times \delta_{i_n\ i'_n} \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) && \text{Rule 4.1} \\
&= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_n} \delta_{i_n\ i'_n} \times (\sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i_n \cdot i_k \cdot i_h \cdot i_w) \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)) && \text{Rule 4.6} \\
&= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow \sum_{i_k}\sum_{i_h}\sum_{i_w} z_y(i'_n \cdot i_k \cdot i_h \cdot i_w) \times w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) && \text{Rule 4.7} \\
&= (i'_n \cdot i'_c \cdot i_r \cdot i_s) \Rightarrow ((i_k \cdot i_h \cdot i_w) \Rightarrow z_y(i'_n \cdot i_k \cdot i_h \cdot i_w)) \cdot ((i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w)) && \text{Rule 4.19} \\
&= (i'_n \cdot i_k \cdot i_h \cdot i_w) \Rightarrow z_y(i'_n \cdot i_k \cdot i_h \cdot i_w) \times_3 (i'_c \cdot i_r \cdot i_s \cdot i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) && \text{Rule 4.22} \\
&= z_y \times_3 (i'_c \cdot i_r \cdot i_s \cdot i_k \cdot i_h \cdot i_w) \Rightarrow w(i_k \cdot i'_c \cdot i_r - i_h \cdot i_s - i_w) && \text{Rule 4.17}
\end{aligned}$$

Fig. 4.4 Gradient derivation of convolution

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

$$\begin{aligned}
z_y \cdot \frac{\partial x}{\partial x} &= z_y \cdot (I \Rightarrow I' \Rightarrow \frac{\partial x(I)}{\partial x(I')}) && \text{Rule 3.5, 3.6, 3.7} \\
&= I' \Rightarrow \textstyle\sum_I (z_y(I) \times \frac{\partial x(I)}{\partial x(I')}) && \text{Rule 4.1} \\
&= I' \Rightarrow z_y(I') && \text{Rule 3.15, 4.7} \\
&= z_y && \text{Rule 4.17}
\end{aligned}$$

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

Convolution Let t_y be the result of a convolution layer with stride 1 and padding 0 defined in Figure 4.4, where x is input, w is weight, and b is bias. If z_y is the backward gradient of t_y, similar like the the previous examples, 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 4.4.

$$
\begin{aligned}
s &= \sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times x(i_n \cdot i_k) \times (-\frac{1}{d_n}) \\
z_x = \frac{\partial s}{\partial x} &= (i'_n \cdot i'_k) \Rightarrow \frac{\partial(\sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times \delta_{i_n\ i'_n} \times \delta_{i_k\ i'_k} \times (-\frac{1}{d_n}))}{\partial x(i'_n \cdot i'_k)} && \text{Rule 3.7} \\
&= (i'_n \cdot i'_k) \Rightarrow \sum_{i_n}\sum_{i_k} \frac{\partial(y(i_n \cdot i_k) \times \delta_{i_n\ i'_n} \times \delta_{i_k\ i'_k} \times (-\frac{1}{d_n}))}{\partial x(i'_n \cdot i'_k)} && \text{Rule 3.14, 3.15} \\
&= (i'_n \cdot i'_k) \Rightarrow \sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times \frac{\partial x(i_n \cdot i_k)}{\partial x(i'_n \cdot i'_k)} \times (-\frac{1}{d_n}) && \text{Rule 3.11} \\
&= (i'_n \cdot i'_k) \Rightarrow \sum_{i_n}\sum_{i_k} y(i_n \cdot i_k) \times \delta_{i_n\ i'_n} \times \delta_{i_k\ i'_k} \times (-\frac{1}{d_n}) && \text{Rule 3.15} \\
&= (i'_n \cdot i'_k) \Rightarrow y(i'_n \cdot i'_k) \times (-\frac{1}{d_n}) && \text{Rule 4.7} \\
&= (-\frac{1}{d_n}) \cdot (i'_n \cdot i'_k) \Rightarrow y(i'_n \cdot i'_k) && \text{Rule 4.14} \\
&= (-\frac{1}{d_n}) \cdot y && \text{Rule 4.17}
\end{aligned}
$$

Fig. 4.5 Gradient derivation of loss expression

Loss The loss expression s and its backward gradient z_x are defined in Figure 4.5, where y is the ground truth and x is the predicted value vector.

Log softmax The gradient derivation of log softmax is shown in Figure 4.6. 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.

$$\begin{array}{llllll}
t_y & = & x-(i_n\cdot i_k)\Rightarrow\log(\sum_{i_l}\exp(x(i_n\cdot i_l))) & = & x-f(i_n\Rightarrow\log(\sum_{i_l}\exp(x(i_n\cdot i_l)))) & \text{Rule 4.23}\\
 & = & x-f(\log(i_n\Rightarrow\sum(i_l\Rightarrow\exp(x(i_n\cdot i_l))))) & = & x-f(\log(\sum_1(i_n\cdot i_l)\Rightarrow\exp(x(i_n\cdot i_l)))) & \text{Rule 4.13, 4.21}\\
 & = & x-f(\log(\sum_1\exp((i_n\cdot i_l)\Rightarrow x(i_n\cdot i_l)))) & = & x-f(\log(\sum_1\exp(x))) & \text{Rule 4.12, 4.17}
\end{array}$$

$$\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{Rule 3.10, 3.6, 3.7}\\
& && \text{Rule 3.8, 3.9}\\
&= ((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{Rule 3.15}\\
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{Rule 4.1, 4.7}\\
&= 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{Rule 4.17, 4.6, 4.7}\\
&= 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{Rule 4.15}\\
&= 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{Rule 4.12, 4.11}\\
&= 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{Rule 4.17, 4.20}\\
&= 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{Rule 4.23}\\
&= 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{Rule 4.21}\\
&= 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{Rule 4.17, 4.12}\\
&= z_y-f(\sum_1 z_y).*\exp(x).*(f(\sum_1\exp(x)))^{-1} && \text{Rule 4.17}
\end{aligned}$$

Fig. 4.6 Gradient derivation of log softmax, where $f(t)$ is defined as $(i_n\cdot i_k)\Rightarrow t(i_n)$.

4.2 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, output, or both. For instance, a Cudnn function f_{cudnn} for computing backward gradient 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 $y_2 \leftarrow f(x_1, \ldots, x_n)$ can be merged into

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

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. Here the $y_1 \leftarrow 1 \times f(x_1, \ldots, x_n) + 1 \times y_1$ corresponds to the library call $f_{cudnn}(x_1, \ldots, x_n, y_1, 1, 1)$, where y_1 is the input/output parameter.

In-place computation The results of the operations such as tensor summation, scalar tensor 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 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 summation, $y \leftarrow x_1 + x_2$, 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 summation and if the overwritten tensor x_1 is used in a later statement, we make a copy of x_1 in that statement.

4.3 Code scheduling

Up to this point, we have the optimized IR code that is nearly ready for the code generation. In fact, our further optimization will move to the execution order aspect. Specifically, the next step is to schedule computations based on the 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 results, some scheduling will result in higher peak memory usage than necessary.

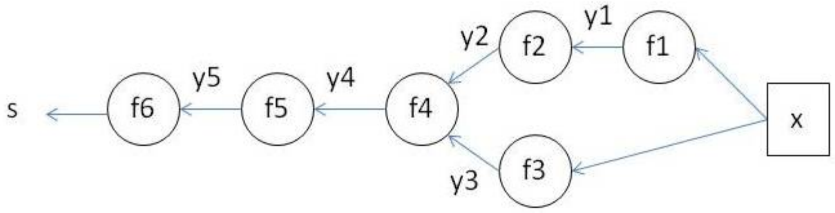

Fig. 4.7 Sample network

To illustrate the scheduling process, consider the sample network in Figure 4.7, 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

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

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 $\frac{\partial f_4}{\partial x_1}$ and $\frac{\partial f_4}{\partial x_2}$.

If we use reverse accumulation method, we can obtain 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 usages 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 [89]. 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 [5] 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 a DAG (V, E) where V is the set of statements and E is the set of edges. Each edge connects one statement to another. Specifically, $(s_1, s_2) \in E$ iff s_2 depends on s_1. We call 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 statement for computing 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 4.8, 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 sufficiently 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
$z_{w_5} \leftarrow z_5 \cdot \dfrac{\partial f_5}{\partial w_5}(y_4)$	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
$z_3 \leftarrow z_4 \cdot \dfrac{\partial f_4}{\partial x_2}(y_2, y_3)$	6	y_1, y_2, y_3, z_4, z_3
$z_{w_3} \leftarrow z_3 \cdot \dfrac{\partial f_3}{\partial w_3}(x)$	7	y_1, y_2, y_3, z_4
$z_2 \leftarrow z_4 \cdot \dfrac{\partial f_4}{\partial x_1}(y_2, y_3)$	6	y_1, z_2
$z_{w_2} \leftarrow z_2 \cdot \dfrac{\partial f_2}{\partial w_2}(y_1)$	7	y_1, z_2
$z_1 \leftarrow z_2 \cdot f_2'(y_1)$	7	z_1
$z_{w_1} \leftarrow z_1 \cdot \dfrac{\partial f_1}{\partial w_1}(x)$	8	

Fig. 4.8 Scheduled statements, their heights, and live variables at each statement.

4.4 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, e.g. Xavier initialization [36], 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 or 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 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 the 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 kept 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[2].

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 to be dynamically allocated and deallocated.

In addition to tensor objects, convolution workspace is also a major source of memory consumption, 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 determine the current memory consumption at each statement depending on whether the tensor and/or workspace memory is cached. Since we can determine the peak memory consumption for a DNN

[2] The advantage of batch normalization is described in [50].

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.

Chapter 5

Experiments

We summarize our experiments in this chapter. We start with describing the standards of the DL network definitions that our experiments conform to in Section 5.1, followed by the description of the computer system specifications that our experiments run on in Section 5.2. We summarize and discuss the GPU-based performance comparison experiments between DeepDSL, Caffe, and Tensorflow in Section 5.3 and conclude this chapter with the illustration of CPU-based Lenet runtime performance comparison experiments between DeepDSL, Theano, and Caffe in Section 5.4.

5.1 DL network configuration standards

DeepDSL primarily compiles to a GPU-based Java target program through Nvidia's CUDA/CuDNN library for various DNN applications. In addition, DeepDSL also has limited support for CPU-based computation, such as the classic DNN *LeNet* Java code generation against CPU cores.

To achieve consistent evaluation results, DeepDSL, Caffe, and Tensorflow tests all follow the same Caffe prototxt definitions in the experiments. Specifically, for Alexnet and GoogleNet, we follow the prototxt from Caffe's website[1]; for Vgg (Vgg-16), we follow prototxt from this link[2]; for Overfeat, we follow prototxt from IntelLabs[3]; and for Deep

[1] github.com/BVLC/caffe/tree/master/models.

[2] github.com/ruimashita/caffe-train/blob/master/vgg.train_val.prototxt.

[3] github.com/IntelLabs/Latte.jl/blob/master/benchmarks/overfeat/overfeat.prototxt.

residual network (ResNet-50), we follow the prototxt from the author's website[4]. The Tensorflow implementation of these networks are either modified from versions of convnet-benchmarks[5] or created from scratch. Finally, we follow Caffe's prototxt definition of *LeNet*[6] for CPU performance evaluation.

Note there are two differences between the tests of Tensorflow and those of DeepDSL and Caffe. First, the training data in the Tensorflow tests is generated from random data in memory while DeepDSL and Caffe tests load real images from the Lmdb database (Due to the lack of Lmdb data support in Tensorflow and the synthetic data usage in Tensorflow only favors Tensorflow in terms of runtime speed; that is, the actual runtime performance of Tensorflow will be worse with real data). Second, the GoogleNet test of Tensorflow only includes the main branch of the GoogleNet while DeepDSL and Caffe train with the full network (This again will favor Tensorflow in terms of runtime speed for GoogleNet tests). All our experiments are trained with ImageNet images that have been resized to 224 by 224 (though DeepDSL do support random cropping of images when their sizes are larger than specified dimensions).

5.2 Experiment environment description

All our experiments are performed on a server with configuration specified in Table 5.1. We execute all the DNNs in the standard batch mode. That is, the first dimension of the image data is always the size of the batch. Also, we compute the average time spent on a full pass of forward and backward across multiple iterations to minimize the random error in all the runtime evaluations. For the dynamic GPU memory usage check, we have automated the entire set of experiments with a script that leverages the standard NVIDIA system management interface with the memory probing interval at 100 milliseconds[7] to capture accurate dynamic memory consumption data.

[4] github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-50-deploy.prototxt.

[5] github.com/soumith/convnet-benchmarks.

[6] http://caffe.berkeleyvision.org/gathered/examples/mnist.html.

[7] Other smaller numbers have been tested and we observe 100 milliseconds probing internal is sufficient for all our tests.

Table 5.1 Computer system configuration details

CPU	Type Clock Number Cache	Intel(R) Xeon(R) CPU E5-2640 v2 2 GHz 32 20480 KB
Memory	Size	125GB
Hard Disk	Size	530GB
GPU	Type Clock Number Memory	NVIDIA Tesla K40c 3004 MHz 1 12GB
Operating System	Type Distribution	Linux CentOS 7
NVIDIA CUDA	Version	8.0.26
NVIDIA CUDNN	Version	5.1.3
GCC	Version	4.9.2
JCuda JNI Binding	Version	0.8.0RC

The primary experiments are performed using GPU. In addition, we also have experiments performed to compare DeepDSL's CPU runtime performance against Caffe and Theano using the *LeNet-5* CNN handwriting benchmark dataset Mnist[8]. We summarize our testing results on GPU in Section 5.3 and CPU in Section 5.4.

5.3 GPU-based runtime & memory performance

We compared the runtime (Figure 5.1) and memory performance (Figure 5.2) of DeepDSL with Caffe and Tensorflow[9] by running several well-known DL networks on a Linux server with a single Nvidia K40c GPU. The DL networks include Alexnet [60], Overfeat [88], Googlenet [96], Vgg [90], and Deep residual network (Resnet) [44]. 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. We also compare CPU performance of DeepDSL with Theano and Caffe using *LeNet* [64].

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.

[8] http://yann.lecun.com/exdb/mnist.

[9] We use the latest version of Caffe and Tensorflow we can obtain at the time of thesis writing.

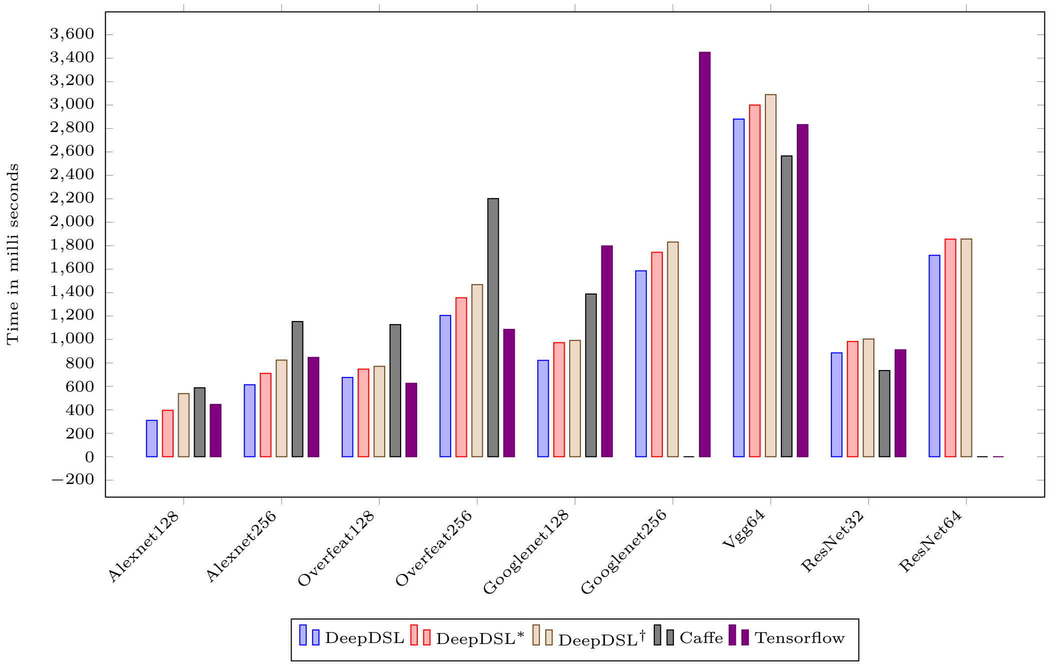

Fig. 5.1 Runtime performance of DeepDSL, Tensorflow, and Caffe (1 forward/backward iteration), where the network names are followed by the batch size. DeepDSL* and DeepDSL† are performance for DeepDSL without tensor cache and without tensor & workspace cache respectively. Caffe failed to run GoogleNet (batch 256) and both Caffe and Tensorflow failed to run ResNet (batch 64) due to GPU memory exhaustion.

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 Overfeat, 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) and Vgg (17% less) with modest runtime overhead. Note that

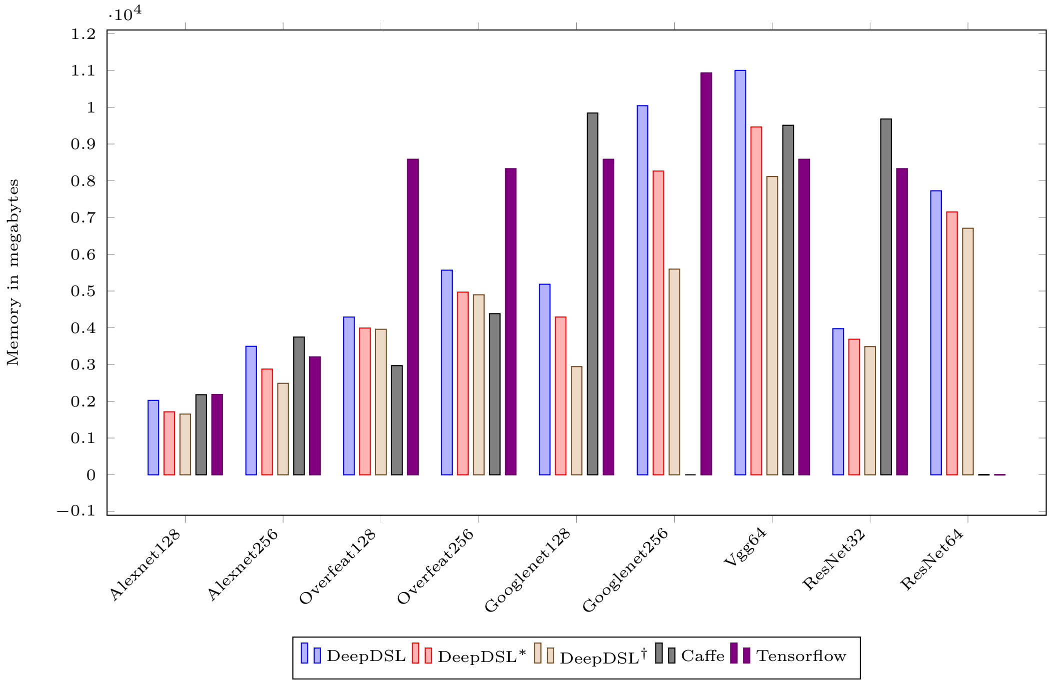

Fig. 5.2 Peak GPU memory use of DeepDSL, Tensorflow, and Caffe during training, where the network names are followed by the batch size. DeepDSL* and DeepDSL† are performance without tensor cache and without tensor & workspace cache respectively. Caffe failed to run GoogleNet (batch 256) and both Caffe and Tensorflow failed to run ResNet (batch 64) due to GPU memory exhaustion.

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.)

5.4 CPU-based Lenet experiments

DeepDSL currently provides limited CPU support to help the user achieve fast prototyping, such as encoding the complete *LeNet* network. In addition to applying extensive optimization on the internal representation (IR) before code generation, DeepDSL employs the standard copy-on-write optimization mechanism[10] to allow sharing on unmodified data resources. Furthermore, DeepDSL's CPU mode also supports leveraging OpenCL[11] for parallelism to gain further speedup. Such optimization results in a performance that's on par with Theano. Figure 5.3 shows our evaluation result of DeepDSL, Theano, and Caffe on the same machine as described in Section 5.2. The experiments are performed

[10] https://en.wikipedia.org/wiki/Copy-on-write.

[11] https://en.wikipedia.org/wiki/OpenCL.

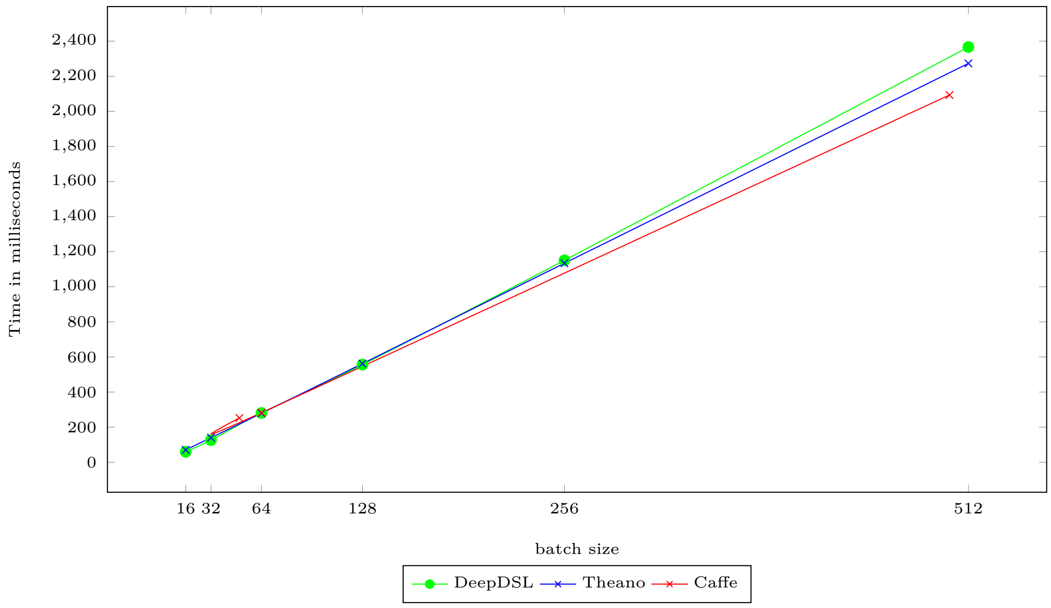

Fig. 5.3 Runtime CPU performance of DeepDSL, Theano, and Caffe (1 forward/backward iteration)

using different batch sizes ranging from 16 to 512 and the result is recorded as the time spending on a full pass (an iteration that contains a complete forward and backward pass), in milliseconds. We observe that DeepDSL is able to offer close performance as that of Theano and Caffe. This is indeed a very impressive result, considering that the generated source of DeepDSL is entirely Java code while both Theano and Caffe are C/C++ backed. Also notice that DeepDSL has some warm-up time period in the beginning some iterations when the internal data structure needs to be initialized at the start time. In fact, if we deduct the time expense on warm-up, DeepDSL achieves slightly better CPU performance than Theano.

Chapter 6

Related work

In this chapter, we review the current state-of-the-art DL frameworks in Section 6.1 and provide a comprehensive comparison between DeepDSL and these frameworks in Section 6.2. We also briefly describe the automatic differentiation that DeepDSL implements in Section 6.3. We conclude this chapter with discussions on several DSL researches in scientific computation using tensors in Section 6.4.

6.1 DL frameworks

In this section, we review some of the most widely accepted DL frameworks that achieve state-of-the-art results, e.g. Caffe2/Caffe, Theano, TensorFlow, MxNet, Microsoft Cognitive Toolkit (CNTK), and Torch/PyTorch. In addition, we will review other less popular but decent frameworks, such as deeplearning4j, Apache Singa, BigDL, and Chainer, that exhibit some unique or best-in-class features such as supporting dynamic network structure (e.g. Chainer) or asynchronous distributed computation (e.g. Apache Singa). Finally, as a comprehensive summary of these frameworks, we will also highlight some other small but elegant frameworks that either have coverage on specific application domain (e.g. MatConvNet, a MATLAB toolbox) or offer direct support for some particular model (e.g. RNNMELM for the hybrid RNN and ME model). These frameworks leverage one or more general-purpose programming languages to support the DL development by providing either direct encoding of the key DL data structures or abstractions of the coarse-grained DL network description. The majority of these frameworks have either comprehensive built-in C/C++ core API that encode the DL structure (e.g. TensorFlow and Caffe2/Caffe) or support the low level unit operations (e.g. CNTK) for arbitrary network structure. Some frameworks also leverage some C/C++ oriented APIs to achieve efficient unit mathematical computations. For instance, PyTorch and Torch both use the TH, THC, THNN, THCUNN C libraries, deeplearning4j uses OpenMP and ND4J for C/C++ backend code. While we will highlight the most important features for each of these frameworks, we will mainly focus on three critical aspects:

- DL network structure representation;
- DL computation optimization;
- computation job scheduling.

We will discuss the main differences between DeepDSL and these frameworks with respect to these three aspects and show the advantages that DeepDSL has over them in Section 6.2.

Caffe/Caffe2 Caffe [51] is one of the first few DL frameworks that uses the concept of tensor (Caffe's tensor is defined as a 4D array) and provides a modularized design. Caffe's DNN is a directed acyclic graph and constructed by connecting the uniformly defined DL network layers together using the "Blob" (a 4D tensor array) construct. Caffe separates its DL network model representation (defined with Protocol Buffers [40]) from the actual model parameter calculation implementation. 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 user's protobuf file layer definition is directly mapped to the internal API calls and thus there is no job scheduling concept in Caffe. Essentially, Caffe serves the purpose of library calls. The more current Caffe2 [30] improves Caffe in several few aspects. First, it provides first-class support for large-scale distributed training;

Specifically, Caffe2 uses Gloo [47], a communications library, for multi-machine training and NVIDIA's NCCL [18] for multi-GPU communications. Second, Caffe2 leverages Redis [85], an fast in-memory database that is often used as cache or message broker, to facilitate management of nodes in distributed training. Finally, Caffe2 supports mobile deployment and can run models on lower powered devices.

Theano Theano [13, 98] is a software library written in Python, C and CUDA[1]. It is a domain-specific compiler. Typically the user writes Python code to build a computation graph and lets Theano compile it. Theano optimizes the graph and generates C and/or CUDA code to perform the computation. In details, tensor.grad() traverses the graph from the output back towards the input through all apply nodes (apply nodes are those that define which computations the graph does). For each such apply node, its op construct defines how to compute the gradient of the node's output with respect to its input. If an op does not provide this information, it is assumed that the gradient is not defined. Using the chain rule these gradients can be composed in order to obtain the expression of the gradient of the graph's output with respect to the graph's input. Theano optimizes the graph computation by identifying and replacing certain patterns in the graph with other specialized patterns that produce the same results but are either faster or more stable. Optimizations can also detect identical subgraphs and ensure that the same values are not computed twice or reformulate parts of the graph to a GPU specific version. Theano normalizes mathematical expressions to remove redundancies and useless calculations; for instance, one simple optimization Theano uses is to replace the pattern $\frac{xy}{y}$ by x. Theano mainly supports computation on numpy.ndarray, but Theano also supports sparse matrices. It allows users to define symbolic variables and functions (mostly around its tensor objects from NumPy) to encode DNN and compile these symbolic expressions to C implementation via its symbolic expression compiler. There is no automatic job scheduling in Theano[2]. In practice, Theano has trouble handling large graphs (large number of nodes) and deep graphs (long chain between input and output), which can lead to crashes or

[1] there is also a incomplete support of an OpenCL back-end in the latest version.

[2] In the multiple-GPU case, platoon [31] can be leveraged to coordinate the results of workers by means of communication between the controller and the workers.

long compilation times. There have been a few frameworks evolved from Theano that have gained popularity. Pylearn2 [39] augments Theano to mainly focus on ease-of-use aspect. It adds support for reusable sub-component, cross-platform serialization of learned models by leveraging YAML to allow specifying an entire experiment without having to write any python code (this is similar to how Caffe leverages Protocol Buffers to describe the network model), Pylearn2 provides dataset interfaces for populate data types such as vector, images, or video. Pylearn2 also provides some limited visualization support such as its model monitor. Blocks [104] is developed in parallel with Fuel [104], a dataset processing framework. it expands Theano to support parametrized Theano operations (This is named "Brick") and it uses pattern matching to select variables and bricks in large models. It provides monitoring and analyzing facilities during training progress for both the training and test datasets. Lasagne [62] is a lightweight library to build and train neural networks in Theano. It supports feed-forward networks as well as recurrent networks such as *Long Short-Term Memory* (LSTM). it allows architectures of multi-input and multi-output, including auxiliary classifiers. It has built-in support for a few activation functions including Nesterov momentum [75], RMSprop [52] and ADAM [57]. Keras [53] was initially built on top of Theano and later added TensorFlow as a backend engine. Like Lasagne, Keras supports convolutional networks and recurrent networks, or the combination of the two. Like Lasagne, it also supports multi-input and multi-output training. The difference between Lasagne and Keras is that Keras adds a higher level wrapper on Theano and recently TensorFlow. The wrapper in Keras is able to describe the DNN graph structure in an abstract way. Keras adopts an object-oriented design, namely, everything in Keras, e.g. layers, models, optimizers, is an object. cudamat [70], while does not have direct support for DNN data structures, is worth mentioning as it provides some matrix facilities that allows easy CUDA-enabled GPU mathematical calculations. Specifically, It supports conversions against the Numpy ndarray[3] data structure and the basic GPU matrix calculations. It also maps CUDA errors into Python exceptions which

[3] A Numpy multidimensional array.

eases the user's troubleshooting. Gnumpy [99], built on top of cudamat, is a Python module that works in a similar way like Numpy, but does its computations on GPU.

TensorFlow TensorFlow [1] shares many common paradigms as that of Caffe. Its core is also written in C++ and its computation graph is described with a DAG where tensors (similar as Caffe's blobs) and layers are alternatively arranged. However, there are also a lot of important differences between TensorFlow and Caffe in the core design. In TensorFlow, each node is a tensor operation (e.g. matrix add/multiply, convolution, and etc.). That is, the building brick of TensorFlow is in unit computation granularity while the building block of Caffe is the coarse-grained layers. Such design enables TensorFlow to provide better flexibility. Specifically, unlike Caffe, TensorFlow no longer needs to define the full forward, backward, and gradient update functions for new layer types. Also due to the finer-granularity of the node, there is no need in TensorFlow to provide extra functions when both CPU and GPU computation need to be supported. TensorFlow use a PS (parameter server) to analyze and distribute the user defined computational graph, i.e. the DAG. PS performs some optimization before distributing the DAG and generates subgraphs from the DAG. The main optimization is done on the operational aspects such as data communication and memory usage handling. TensorFlow also does some optimization on the DAG with common subexpression elimination. In summary, TensorFlow's global optimization is against the DAG. When the worker node gets the subgraph for computation, it can perform partial gradient along the backwards path using the chain rule and only computes the necessary gradient along the way (This means if an operation has two outputs and the gradient computation for a node only needs one output, then the gradient function of the other one will be set the zero to avoid unnecessary calculation). Like Theano and Caffe, TensorFlow separates the execution from the layer construction. It goes one step further to enable mapping the computation not only to single device but also multiple devices. In TensorFlow, such mapping is named node placement. TensorFlow runs a simulated execution using a heuristic to decide which device a specific branch of the DAG should run on. Like Theano, TensorFlow provides additional supports such as allowing control flow to be imposed on the DAG, queues that allow different subgraph to execute asynchronously. Finally, unlike Caffe's 4D array limitation, TensorFlow's tensor is a typed (including integer, float, double, complex number, and string) multi-dimensional array. TensorFlow's tensor is also different from that of Theano in that it is persistent mutable. This design consideration has pros and cons. Having persistent mutable tensor allows the tensor to be carried across executions of a graph and reused. However, it also increases the difficulty of code optimization and debugging.

MXNet MXNet [15] uses multi-output symbolic expressions, denoted as Symbols, to declare the computation graph. Symbols are connected by operators, such as simple matrix operations (e.g. $x.\_\_mul\_\_(y)$ for $x * y$), or a complex neural network layer (e.g. convolution layer). An operator can take several input variables, produce more than one output variables, and have internal state variables. A variable can be either free, which can be bound with value later, or an output of another symbol. MXNet leverages lazy evaluation. Before evaluation, MXNet transforms the graph to optimize the efficiency and allocate memory to internal variables. Since each variable's lifetime, namely the period between the creation and the last time used, is known for a computation graph, MXNet can reuse memory for non-intersected variables. With respect to job scheduling, MXNet employs two heuristics strategies for memory optimization. The first, in-place, simulates the procedure of traversing the graph, and keeps a reference counter of depended nodes that are not used so far. If the counter reaches zero, the memory is recycled. The second, co-share, allows two nodes to share a piece of memory if and only if they cannot be run in parallel. Exploring co-share imposes one additional dependency constraint. In particular, each time upon scheduling, among the pending paths in the graph, the longest path is searched and required memory allocations are performed. The dependency engine uses multiple threads to schedule the operations for better resource utilization and parallelization. Furthermore, MXNet provides both declarative and imperative programming styles and multiple language supports (by embedding into multiple host languages and unifying the execution with one backend engine). MXNet is also based on a multi-dimensional array for tensor computation. Like TensorFlow and CNTK, it resolves the graph dependencies and schedules the execution against multiple devices (Like CNTK, can be on a different machine and a 2-level data communication mechanism is employed for the intra- and inter-machine synchronization).

CNTK As a relatively new system, Computational Network Toolkit (CNTK) [111][4] tackles the DL computations in a different way from others. CNTK exposes to the user a set of pre-defined *Network Definition Language* (NDL) functions (e.g. Convolve, MaxPooling) that are internally implemented as computational nodes. Users can also write their own NDL to describe the particular DL network in consideration. CN is a DAG with two types of vertices. The first type represents some basic computation (e.g. Add, Times, Plus) while the second type represents vertices holding operands and have edges towards a computation node. Such low level computation scheme enables CNTK to encode arbitrary computational network and the CNTK core can assign each computation node to a particular CPU/GPU device (these devices may be from different machines as well). CNTK provides both C++ and Python API interfaces to define models, learning algorithms, data reading and distributed training. Recent versions also support Protocol Buffers for these definitions. CNTK decides its computation order via depth-first traversal of the directed acyclic graph (DAG). CNTK optimize memory usage by 1. using the same memory across mini-batches (i.e. do not destroy and reallocate memory at each mini-batch); 2. sharing memory across computation nodes when possible (This is achieved by analyzing the execution plan and release 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 [87] to quantize gradients with just 1 bit so that the communication cost between computation nodes is reduced.

Torch/PyTorch Torch [17] leverages the Lua programming language to provide easy integration with C (It achieves C-like performance for most of the pure Lua code using the Lua *Just In Time* (JIT) compiler). Also, as a relatively long endeavor in fields like machine learning and neural network, it has accumulated a large set of optimized routines

[4] Recently the name was modified to Microsoft Cognitive Toolkit but the acronym CNTK remains.

to help users gain results faster. In addition to CPU/GPU, it also supports mobile and FPGA backends. Torch's core is a N-dimensional array, i.e. Torch Tensor, as well as a comprehensive set of routines operating on the Tensor, such as indexing, slicing, transposing, and etc. Torch 7 now supports automatic differentiation by wrapping the tensor class into a new thin-layer Variable class and performing backward computation against it. Torch provides a "nn" package that supports many existing neural network models that work directly against the Tensor. There is neither built-in optimization support in terms of memory usage or computational speed nor job scheduling in Torch. Torch is provided rather as a handy toolbox with many out of box routines ready to be used, given the user understands the neural network forward / backward computation well and can program in Lua to some extent. As a more recent effort, PyTorch [79], improves Torch in a few aspects. First, PyTorch supports the more popular Python language instead of Lua by means of integration with the Numpy package, a Python version of N-dimensional tensor. Second, it supports distributed computation via its new torch.distributed package. PyTorch does the distribution using message passing (coordinate with a master node) between worker nodes, either in a point-to-point manner or collectively (i.e. all individual processes in a set can communicate with each other). Finally, it adds custom memory allocators for the GPU to gain better memory usage.

deeplearning4j deeplearning4j [97] is developed mainly with Java and serves as a distributed neural net library. Similar to Numpy ndarray, deeplearning4j extends its functionalities from a n-dimensional array class defined in its ND4J module. The most noticeable feature of deeplearning4j is its integration with the MapReduce [22][5] framework Spark [112][6] and Hadoop [108][7], through the *deeplearning4j-scaleout* module. In addition, deeplearning4j contains a *deeplearning4j-modelimport* module to import models from other frameworks for convenience. deeplearning4j leverages OpenMP for better parallel

[5] MapReduce is a distributed cluster programming model that can process and generate large data sets in parallel.

[6] Spark is a MapReduce based distributed computation framework that perform computational tasks in memory parallelly.

[7] Hadoop is another MapReduce based framework that supports distributed processing of large data sets across clusters of computer nodes.

performance on CPUs. deeplearning4j improves GPU memory usage by allocating each GPU memory chunk once and cache it for further reuse. The GPU device constant memory is used for ShapeInfo and TAD (Tensor along dimension) cache for faster access from kernel context. deeplearning4j improves the computation efficiency via two parallelism models: a. Element-level parallelism: threads in grid use the same linear buffer; b. TAD-level parallelism: grid is split into blocks, each block is linked to one TAD. Finally, in the Multi-GPU environment, one Java thread is attached to one GPU at any given moment.

Apache Singa Apache Singa [76] defines two core abstractions, Tensor and Device. The Tensor abstraction encapsulates a multi-dimensional array and its related functions as well as model variables. The Device manages the memory for a tensor and executes tensor operations on CPU, Cuda GPU, or OpenCL-based[8] generic GPU interfaces. It then provides a set of higher level data structures, e.g. Layers (for neural network), Loss (defines the training objective loss functions), or Optimizer (updates the parameters using methods such as SGD). This design makes it very close to Caffe framework. In fact, Singa also requires explicit layer definition as well as the associated forward and backward functions definition and coding. Singa supports a variety of popular deep learning models, i.e., feed-forward models including convolutional neural networks (CNN), energy models like restricted Boltzmann machine (RBM), and recurrent neural networks (RNN). Many built-in layers are provided for users. Singa architecture is sufficiently flexible to run synchronous, asynchronous and hybrid training frameworks. In addition, Singa supports different neural net partitioning schemes to parallelize the training of large models, i.e., partitioning on batch dimension, feature dimension or hybrid partitioning. Singa uses ZooKeeper [32] to coordinate the training, and uses ZeroMQ [78] for transferring messages.

BigDL BigDL [49], 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

[8] https://en.wikipedia.org/wiki/OpenCL.

using BigDL. BigDL also claims it can achieve magnitude faster performance than out-of-box open source Caffe, Torch or TensorFlow on the intel-based system with the Intel MKL. Specifically, BigDL achieves comparable performance in intel CPU as that of the mainstream GPU.

Chainer Chainer [100] concerns the forward (input processing) and backward (gradient calculation) computation. Comparing to the fixed "define-and-run" paradigm, which constructs a computational graph and then fetch mini-batch info the graph periodically to do forward and backward (adopted by many frameworks such as Torch, Theano, and Caffe), Chainer follows a "define-by-run" pattern which essentially allows modifying the control flow (such as changing the computation graph in each iteration) during the execution of a computational graph (achieved by storing the order of operations during the graph contruction).

MShadow MShadow [26] is a Matrix/Tensor Template Library in C++/CUDA. It supports writing lazily evaluated expressions and compile-time optimization. The MShadow code can run on both CPU and GPU without changes, the details are handled internally by the compilation process. Users of MShadow do not need to preallocate memory for the Tensor defined, however, they need to specify the details such as shape and stride of the Tensor for MShadow to allocate the memory at compile-time. The “mshadow-ps” interface in MShadow allows the user to write multi-gpu and distributed programs in an unified way by implementing a two-level asynchronous parameter server where the intra-machine communication is at level 1 and inter-machine communication is at level 2 [65]. It can apply different consistency models at different levels to trade-off the algorithm efficiency and system performance as the data communication at level 1 is normally as much as 10X faster than at level 2.

Other C/C++ based frameworks, such as CUV [48], OpenNN [7], cuda-convnet2 [58], and neuralnetworks [4], while only provide limited functions (e.g. OpenNN requires non-trivial C++ efforts to implement even a simple and straightforward neural network) on limited environments (e.g. CUV can only run on certain Linux distributions), are worth mentioning as they provide supports for popular DL computations needs with special focuses, in the format of programming libraries. For example, CUV supports the python bindings for easy use of Nvidia CUDA functions on matrices. It also has an *Restricted Boltzmann machine* (RBM) [92] [45] and annealed importance sampling [72] implementation as well as library code to calculate the partition function exactly. OpenNN supports neural networks parallelization with OpenMP on CPU and CUDA on GPU. It provides a GUI Neural Designer to support data entry and interpretation of results[9]. cuda-convnet2 implements the feedforward neural networks and can model arbitrary layer connectivity and network depth. Its multi-gpu training supports data parallelism, model parallelism, and a hybrid approach [59] for cross GPU parallelization. neuralnetworks provides support for GPU-based computation for training deep neural networks via OpenCL and Aparapi. It is one of the earliest Java libraries that allows the user to program the DNN tasks in Java. Its OpenCL interfaces offer the flexibility to run code on more GPU machines but also restrict it to achieve the best performance comparing to specialized GPU APIs, such as Nvidia CUDA.

As the major tool for numerical computing, MATLAB [67] has been widely used in machine learning. Although all are still in early stage at the time of this thesis writing, there have been some efforts to try integrating deep learning support into Matlab platform lately, in the format of MATLAB toolboxes[10]. MatConvNet [106] is a MATLAB toolbox implementing CNN for computer vision applications. it relies heavily on the VLFeat library [105][11] internally for the CNN computations. One convenient highlight is that MatConvNet provides many pre-trained CNNs for image classification, segmentation, face recognition, and text detection. Like Caffe, ConvNet [23] operates on 4-dimensional tensors. Two noticeable features are: 1. ConvNet can import and use pre-trained MatConvNet models; 2. ConvNet implements the *Invariant Backpropagation* (IBP) [24] and *Adversarial Training* (AT) [38] algorithms. LightNet [110] provides MLP, CNN and RNN deep learning

[9] https://www.neuraldesigner.com.

[10] The MATLAB toolbox extends the out-of-box MATLAB functionalities to meet some special domain needs, such as MATLAB machine learning toolbox.

[11] VLFeat is a C-based library that implements a set of computer vision algorithms for image understanding and local features extraction and matching.

support purely in the MATLAB environment. Since LightNet relies on the underneath MATLAB computational power, its major computations are vectorized and is implemented in just hundreds of lines of code. Simplicity is the biggest advantage of LightNet for the mathematically oriented users.

There are also some frameworks that are specialized in some certain application domains. Nengo [11] is a graphical and scripting based Python library for simulating large-scale neural systems. It is based on the *Neural Engineering Framework* (NEF) [29] to simulate the mechanism of the neural structure. NEF is based on three principles: 1. Representation: A group of neurons represents a vector of a specific length; 2. Transformation: A connection from one neural group to another computes a function on the represented value; Dynamics: Recurrent connections generates the complex dynamical models. This essentially matches with the classic DNN representation. The goal of Neogo, quite different from implementing existing popular DNNs (such as CNN), is to create biologically plausible models of cognition [94]. The reason is two-fold. First, it tries to use realistic neurons to better evaluate neural network theories; second, the generated neural network based on the three NEF principles can suggest new types of algorithms. Specifically, the neurons approximate the algorithm during the learning process. There are not only no predefined neural parameters but also no sophisticated pre-defined functions. The learning process is forced to use the basic operations that are available for neurons.

NeuralDesigner [6] is a DL tool to solve predictive analytics and data mining applications. It was originally developed from OpenNN and is mainly for commercial usages (e.g. business intelligence, health care applications, and etc.). The advantage of NeuralDesigner is that it provides a complete suite of GUIs during the learning work flow with its Neural Editor and Neural Designer packages. In other words, it allows the user to perform visual parameter settings, model selection, training action, result analytics, and model deployment.

A few C-based libraries also exist. RNNLM [69] is a RNN-based Language models Toolkit. It leverages RNN to train RNN-based language model and can also train hash-based maximum entropy model (ME) and RNNMELM [68]: the hybrid of RNN and

ME models. RNNLM can help improve existing systems for speech recognition and machine translation. RNNLIB [42] is another recurrent neural network library focusing on sequence learning problems. It leverages *Long short-term memory* (LSTM) [41] to generate complex sequences with long-range structure by predicting one data point at a time. RNNLIB predicts in a "fuzzy" way by synthesizing and reconstituting the training data using RNN's internal representation to perform a high-dimensional interpolation between training examples and rarely generating the same thing twice. It avoids the curse of dimensionality and thus has better performance at modeling real-valued or multivariate data than exact matches.

6.2 Core feature comparison between DeepDSL and existing frameworks

There are several unique features that differ DeepDSL from other frameworks. In this section, we discuss these features and compare DeepDSL with other frameworks in details.

Like a few other mainstream frameworks (e.g. TensorFlow, MXNet, and etc.), DeepDSL supports automatic differentiation. Unlike the other frameworks, the output expressions of intermediate representation (IR) remain fully abstract after symbolic gradient, which allows the DeepDSL compiler to statically infer the dimensions of tensors in each layer, check whether the layers are properly connected, and automatically insert tensor reshaping operations as necessary. Errors caused by incorrect parameter dimensions are caught before code generation.

DeepDSL program 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 result intermediate representation (IR) remains abstract and human readable. The biggest advantage of such design is it allows the DeepDSL compiler to handle the abstract syntax tree (AST) rather than the computation graph, and thus the processing steps are standardized regardless the structure and complexity of the computation graph. i.e. DeepDSL compiler can apply a series standard compiler technologies, i.e. simplification, optimization, and transformation to the DeepDSL program, which largely helps computation optimization, such as redundant expression calculation removal via common subexpression elimination, code specialization via vectorization; moreover the IR can be compiled to any target environments with great flexibility for desired customization.

For automatic task scheduling, DeepDSL analyzes the dependencies of the DSL expressions during the optimization stage to determine when each DSL expression is ready to run. Such information is obtained by directly 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 also provides statement by statement memory usage analysis against the IR, which gives the user great knowledge of exactly what to expect before executing the program. The output Java code also runs independently without the presence of DeepDSL itself. In addition, The java code generated by the DeepDSL compilation can be debugged easily in any mainstream IDEs that support Java development.

One main drawback of the other framework is the long list of required dependencies before getting things up and running. DeepDSL, on the contrary, is designed to be lightweight and portable from the very beginning. In fact, only one major external dependency group, a set of JNI wrappers used to interface with CUDA/CUDNN APIs, are needed for the computation. This gives the user the convenience of fast installation and prototyping. Such "minimalism" design will also simplify efforts for the upcoming DeepDSL releases that aim at supporting distributed computation and mobile environment.

Table 6.1 summarizes the detailed comparisons between DeepDSL and the other mainstream frameworks.

Table 6.1 Core feature comparison between DeepDSL and existing frameworks

Features	**DeepDSL**	**Caffe Caffe2**	**Theano Keras Lasagne Pylearn2**	**TensorFlow**	**MXNet**	**CNTK**	**Torch / PyTorch**	**deeplearning4j**
computation optimization	compiler level optimization on IR	hardcode / partial optimization	symbolic gradient optimization	computation graph-based optimization	computation graph-based optimization	computation graph-based optimization	N.A.	threading-based optimization
memory optimization	compile-time dependency analysis and heuristic for earliest unused memory recycling	shared memory and 3rd party memory optimizer with multiloading: reuse memory from previous blobs in later blobs	expression graph-based optimization	default: map all GPU visible memory to process; allow_growth: allocate only as much GPU memory based on runtime allocations; per process gpu memory fraction: determine the fraction of overall memory that each visible GPU should be allocated	computation graph-based optimization	parameter sharing of the same model between multiple threads	3rd party memory optimizer	“Off-heap” (allocate memory outside of the JVM) for NDArrays
automatic differentiation	yes	no	yes	yes	yes	yes	yes	no
automatic job/task scheduling	yes	no	no (can be partially support with 3rd party package)	yes	yes	yes	no	no (can partially support using YARN[12])
debuggability	very easy (can directly debug in any mainstream IDEs / insert any log statements in generated code)	rather difficult (requires certain level understanding of C++ debugging skill and in-depth Caffe DL implementation)	difficult (requires certain level Python knowledge and theano details)	difficult (requires in-depth understanding of TensorFlow and its tool tfdbg)	rather difficult (require certain skills for debugging C/C++ libraries called by Python)	difficult (can become easier if leverage Microsoft’s proprietary IDE Visual Studio)	difficult (requires in-depth Lua debugging skills)	difficult (requires C++/C debugger such as gdb[13])
Modifiability	easy (can modify both the DSL and generated code)	easy for network definition; difficult for internal functions	easy for surface code, difficult for internal functions	easy for frontend code, difficult for internal core	easy for wrapper code, difficult for C++/C library code	easy for surface code, difficult for core	intermediary difficulty for frontend Lua code, difficult for library code	easy for wrapper code, difficult for C++/C native code
Complexity of dependencies	low (only one major JNI library dependency to call GPU)	high	high	medium to high	medium	medium	medium	high
Run independently	yes (can run without the presence of DeepDSL tool)	no	no	no	no	no	no	no
memory usage analysis	compile-time memory usage analysis in statement by statement granularity	no	limited with profiler	very limited		no	no	no
mobile environment support	no (plan in future versions)	supported in Caffe2	no	yes	yes	no	no	very limited (does not support real-world dataset/DL structure)
distributed computation	no (plan in next version)	supported in Caffe2	supported in Keras	yes	yes	yes	yes (via torch.distributed package)	yes (via integration with Spark[14])
dynamic computation graph	no (plan in future versions)	no	no	yes (via Tensorflow Fold)		no	supported in PyTorch	no

6.3 Automatic differentiation

The computation of derivatives may be processed via numerical differentiation, symbolic differentiation (of mathematical expressions), or automatic differentiation (of mathematical programs) [8].

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 output. 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 [73] 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 [66] 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 a SSA (static single assignment) form and then derives all parameter gradients together without redundant computation.

6.4 DSL on scientific computing with tensors

A number of DSLs have emerged recent years in the scientific computing applications such as TCE (Tensor Contraction Engine) [10], UFL (Unified Form Language) [3], SPL [109] (DSL for signal processing).

TCE [10] 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 dependencies of TCE form 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 abstracts out the core deep learning concepts but also provides complete support for all the most important deep learning aspects, such as gradient computation, optimization, and code generation.

Chapter 7

Conclusion

In this thesis research, we have designed and implemented an end-to-end DSL solution, DeepDSL, for encoding deep learning networks to achieve the training and inference goals. Both the Code authored with the DSL constructs defined in DeepDSL and the generated source are easy to understand and modify. The generated source is very portable to run across platforms, and highly optimized to offer very competitive runtime and memory efficiency.

The development of DeepDSL also demonstrated the utility of rule-based symbolic reduction in mathematical computation. 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 with the compilation-based optimization such as common subexpression elimination and 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 the output Java source code.

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 like 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, macOS, and Linux without any changes or rebuild; these advantages make DeepDSL far more portable than other DL libraries.

Since it is compilation-based, DeepDSL can statically detect programming errors and analyze memory consumption so that users can determine whether a DL network can run on a computer platform at compile-time without actually deploy the code. The generated Java program is easy to debug with an IDE such as Eclipse or IntelliJ where users can set break points and inspect intermediate results, which is very difficult for most other DL libraries.

7.1 Upcoming releases and future directions

Upcoming releases Our current interpreter mode in DeepDSL is only partially implemented to support simple DNNs like em LeNet. We plan to make it full-fledged as the compiler mode. This will bring benefit for the user who wants to understand the DeepDSL internal execution of a DL workflow and also help DeepDSL developers to verify things like tensor computations or internal optimization. In addition, although our main efforts are on GPU-based code generation and optimization, we will also provide enough support for CPU-based code generation. This in practice allows the user to quickly examine their DL network design and verify results on a machine that is not equipped with the Nvidia GPU.

DeepDSL provides its DSL constructs for encoding a DL network. Most of the existing DL libraries leverage some external standard description file to define the network structure and connections between nodes. For example, the prototxt-based configuration file format in Caffe has been used widely in some benchmarking (e.g. convnet-benchmarks[1]). We

[1] https://github.com/soumith/convnet-benchmarks.

plan to add support for converting/generating these files automatically to/from the DSL constructs as input to the DeepDSL compiler. This should help greatly reduce the user's time for authoring the DSL code and comparing the DSL code with other frameworks. This will also increase the interoperability of DeepDSL so that a working network that runs with other frameworks can also be verified and executed in DeepDSL quickly, and vise versa.

Future directions 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.

We have leveraged JCuda to fully support the current version of CUDA API. Also the CuDNN interface is used to run some complete DL functions right within GPU. This proves to have superior performance than that of the pieced JCuda API calls due to tremendously reduced CPU ↔ GPU data transfer time. However, such approach relies on using a generic interface to handle all possible DL input and output which is not optimal in terms of memory usage in many cases. We plan to add certain support to allow users to customize and control the API library usages based on their needs.

The current version of DeepDSL only supports generated code to run on a single GPU. In many advanced hardware environments, multiple GPUs are available on a given server. Since DeepDSL compiler is capable of analyzing the dependencies in the tensor computation process to collect the tensor functions that can run in parallel at runtime, we plan to extend DeepDSL for supporting the multi-GPU environment.

It may be beneficial to leverage some specially designed hardware power with native languages in some application domains. Generating target code in other languages is indeed relatively straightforward since the DeepDSL IR remains fully abstract before code generation; we plan to add the support for target code in other language when such need arises.

Many current mobile devices already come equipped with capable GPUs. With these increasingly capable hardwares, researchers have started investing more efforts for DNN in the mobile environment. The Java source code output makes DeepDSL a good candidate in this emerging research domain. Mobile environment has its unique constraints such as limited battery life and frequently lost signal. These constraints entail new requirements to the DeepDSL code execution, for example more balanced computation (for slower battery draining) and data stored locally vs. remote data loading.

In summary, the aforementioned directions are the few we are considering at the present stage. Nevertheless the compiler-based lightweight and portable essence of DeepDSL should make it flexible in many other potential future expansions.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv e-prints.*

[2] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, Principles, Techniques.* Addison wesley.

[3] Alnaes, M. S., Logg, A., Olgaard, K. B., Rognes, M. E., and Wells, G. N. (2014). Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37.

[4] AMD (2011). Aparapi: Opencl gpu and multi-core cpu heterogeneous computing for java.

[5] Appel, A. and Supowit, K. J. (1987). Generalizations of the sethi-ullman algorithm for register allocation. *Software – Practice and Experience*, 17:417–421.

[6] Artelnics (2014). Neural designer for data mining using neural networks.

[7] Artelnics (2015). Opennn: Advanced analytics library.

[8] Bartholomew-Biggs, Michael adn Brown, S., Christianson, B., and Dixon, L. (2000). Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1-2):171–190.

[9] Basu, A., Hayden, M., Morrisett, G., and von Eicken, T. (1997). A language-based approach to protocol construction. *Kamin [43]*, pages 1–15.

[10] Baumgartner, G., Auer, A., Bernholdt, D. E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R. J., Hirata, S., Krishnamoorthy, S., et al. (2005). Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292.

[11] Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A. R., and Eliasmith, C. (2013). Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7.

[12] Bentley, J. (1986). Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721.

[13] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy).* Oral Presentation.

[14] Chen, S.-C., Jain, R., Tian, Y., and Wang, H. (2015a). Guest editorial multimedia: The biggest big data. *Multimedia, IEEE Transactions on*, 17(9):1401–1403.

[15] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015b). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Neural Information Processing Systems, Workshop on Machine Learning Systems.*

[16] Cleaveland, J. C. (1988). Building application generators. *IEEE Softw.*, 5(4):25–33.

[17] Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop.*

[18] Corporation, N. (2016). NCCL: Optimized primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl.

[19] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

[20] Cox, D. R. (1958). The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242.

[21] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941.

[22] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[23] Demyanov, S. (2013). Matlab: Cnns for matlab for classification and segmentation.

[24] Demyanov, S., Bailey, J., Kotagiri, R., and Leckie, C. (2015). Invariant back-propagation: how to train a transformation-invariant neural network. *arXiv preprint arXiv:1502.04434.*

[25] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE.

[26] dmlc (2013). Matrix shadow:lightweight cpu/gpu matrix and tensor template library in c++/cuda for (deep) machine learning.

[27] Dozat, T. (2016). Incorporating nesterov momentum into adam.

[28] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

[29] Eliasmith, C. and Anderson, C. H. (2004). *Neural engineering: Computation, representation, and dynamics in neurobiological systems.* MIT press.

[30] Facebook, I. (2017). platoon: Multi-gpu mini-framework for theano. https://caffe2.ai/.

[31] for Learning Algorithms, M. M. I. (2016). Caffe2: A new lightweight, modular, and scalable deep learning framework. https://github.com/mila-udem/platoon.

[32] Foundation, T. A. S. (2014). Apache ZooKeeper: A centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. https://zookeeper.apache.org/.

[33] Freedman, D. A. (2009). *Statistical models: theory and practice.* cambridge university press.

[34] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.

[35] Geoffrey, H. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

[36] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

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

[38] Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572.*

[39] Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013). Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214.*

[40] Google (2008). Protocol buffers. google's data interchange format.

[41] Graves, A. (2013a). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

[42] Graves, A. (2013b). Rnnlib.

[43] He, K. and Sun, J. (2015). Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5353–5360.

[44] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[45] Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

[46] Huang, X., Zhao, T., and Cao, Y. (2014). Pir: A domain specific language for multimedia information retrieval. *Int. J. Multimed. Data Eng. Manag.*, 5(3):1–27.

[47] Incubator, F. (2017). Gloo: Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo.

[48] Institute for Computer Science VI, U. o. B. (2015). Cuv: A c++ template and python library.

[49] Intel (2016). Bigdl: Distributed deep learning library for apache spark.

[50] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456.

[51] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

[52] Kenney, J. F. and Keeping, E. S. (1962). Root mean square. *Mathematics of Statistics, Pt. 1, 3rd edition*, 4.15:59–60.

[53] keras team (2015). Keras: Deep learning library for theano and tensorflow.

[54] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

[55] Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. (1996). A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society.

[56] Kingma, D. P. and Ba, J. (2014a). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[57] Kingma, D. P. and Ba, J. (2014b). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[58] Krizhevsky, A. (2014a). cuda-convnet2.

[59] Krizhevsky, A. (2014b). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

[60] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[61] Kronecker, L. (1903). *Vorlesungen über die Theorie der Determinanten*.

[62] Lasagne (2014). Lasagne: A lightweight library to build and train neural networks in theano.

[63] LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995.

[64] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[65] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598.

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

[67] MathWorks (2016). Matlab: The language of technical computing.

[68] Mikolov, T., Deoras, A., Povey, D., Burget, L., and Černockỳ, J. (2011a). Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE.

[69] Mikolov, T., Kombrink, S., Deoras, A., Burget, L., and Cernocky, J. (2011b). Rnnlm-recurrent neural network language modeling toolkit. In *Proc. of the 2011 ASRU Workshop*, pages 196–201.

[70] Mnih, V. (2009). Cudamat: a cuda-based matrix class for python. *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, 4.

[71] Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

[72] Neal, R. M. (2001). Annealed importance sampling. *Statistics and Computing*, 11(2):125–139.

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

[74] Neighbors, J. M. (1986). The draco approach to constructing software from reusable components. In *Readings in artificial intelligence and software engineering*, pages 525–535. Elsevier.

[75] Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376.

[76] Ooi, B. C., Tan, K.-L., Wang, S., Wang, W., Cai, Q., Chen, G., Gao, J., Luo, Z., Tung, A. K., Wang, Y., et al. (2015). Singa: A distributed deep learning platform. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 685–688. ACM.

[77] Petrova, S. S. and Solov'ev, A. D. (1997). The origin of the method of steepest descent. *Historia Mathematica*, 24(4):361–375.

[78] project, T. Z. (2010). ZeroMQ: Distributed messagings. http://www.zeromq.org/.

[79] pytorch community (2016). PyTorch: Tensors and dynamic neural networks in python with strong gpu acceleration. http://pytorch.org/.

[80] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

[81] Reddi, S. J., Kale, S., and Kumar, S. (2018). On the convergence of adam and beyond. In *International Conference on Learning Representations*.

[82] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[83] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533.

[84] Russell, I. (2012). The delta rule. *University of Hartford, West Hartford. Accessed*, 5.

[85] Salvatore, S., Pieter, N., and Matt, S. (2011). Redis: an in-memory database that persists on disk. https://redis.io/.

[86] Schmidhuber, J. (2014). Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828.

[87] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. (2014). 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*.

[88] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*.

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

[90] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[91] Sirer, E. G. and Bershad, B. N. (1999). Using production grammars in software testing. In *ACM SIGPLAN Notices*, volume 35-1, pages 1–13. ACM.

[92] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE.

[93] Spall, J. C. (2003). *Introduction to Stochastic Search and Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition.

[94] Stewart, T. C. (2012). A technical overview of the neural engineering framework.

[95] Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. *ICML (3)*, 28:1139–1147.

[96] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[97] Team, D. (2016). Deeplearning4j: Open-source distributed deep learning for the jvm. *Apache Software Foundation License*, 2.

[98] Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

[99] Tieleman, T. (2010). Gnumpy: an easy way to use gpu boards in python. *Department of Computer Science, University of Toronto*.

[100] Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*.

[101] Tratt, L. (2008). Domain specific language implementation via compile-time metaprogramming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):31.

[102] Van Deursen, A., Heering, J., and Klint, P. (1996). *Language Prototyping: an algebraic specification approach*, volume 5. World Scientific.

[103] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36.

[104] Van Merriënboer, B., Bahdanau, D., Dumoulin, V., Serdyuk, D., Warde-Farley, D., Chorowski, J., and Bengio, Y. (2015). Blocks and fuel: Frameworks for deep learning. *arXiv preprint arXiv:1506.00619*.

[105] Vedaldi, A. and Fulkerson, B. (2008). VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/.

[106] Vedaldi, A. and Lenc, K. (2015). Matconvnet – convolutional neural networks for matlab. In *Proceeding of the ACM Int. Conf. on Multimedia*.

[107] Walker, S. H. and Duncan, D. B. (1967). Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1-2):167–179.

[108] White, T. (2009). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

[109] Xiong, J., Johnson, J., Johnson, R. W., and Padua, D. (2001). SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308.

[110] Ye, C., Zhao, C., Yang, Y., Fermüller, C., and Aloimonos, Y. (2016). Lightnet: A versatile, standalone matlab-based environment for deep learning. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1156–1159, New York, NY, USA. ACM.

[111] Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al. (2014). An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*.

[112] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

[113] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

[114] Zhao, T. and Huang, X. (2018). Design and implementation of deepdsl: A dsl for deep learning. *Computer Languages, Systems & Structures*.

[115] Zhao, T., Xiaobing, H., and Cao, Y. (2017). Deepdsl: A compilation-based domain-specific language for deep learning. In *Proceedings of the 5th International Conference on Learning Representations*, Toulon, France.

Appendix

Proof of Theorem 3.1

We provide the proof for Theorem 3.1 in this chapter. For convenience, we restate the theorem as below.

Theorem 1 (Scalar $\hat{s}$ type soundness). *If $\emptyset \vdash \hat{s} : \tau$, then there exists v such that $\hat{s} \rightarrow^* v$.*

Theorem 2 (Tensor t type soundness). *If $\emptyset \vdash t : \tau$, then there exists $\mathcal{V}$ such that $t \rightarrow^* \mathcal{V}$.*

We first state two type soundness lemmas, one for dimension and one for index.

Lemma 1 (Dimension type soundness). *If $\emptyset \vdash D$, then $\exists\ n_1 \cdots n_k$ such that $D \rightarrow^* n_1 \cdots n_k$.*

Proof: Straightforward by induction on dimension expression and product, omitted.

Lemma 2 (Index type soundness). *If $\emptyset \vdash I : D$, then $\exists\ m_1 \cdots m_k$ such that $I \rightarrow^* m_1 \cdots m_k$.*

Proof: Straightforward by induction on index expression, omitted.

Recall in the syntax definition (Figure 3.1) scalar $\hat{s}$ can be either a function application $(x \Rightarrow \hat{s})(t)$ or s. The former has the tensor as its parameter and the scalar $\hat{s}$ itself in the function body. Therefore, we proceed our type soundness lemma proofs in the order of scalar s, tensor t, and $(x \Rightarrow \hat{s})(t)$. The proofs of s and $(x \Rightarrow \hat{s})(t)$ together completes the proof of type soundness for scalar $\hat{s}$.

1 Scalar s type soundness

Theorem 3 (Scalar s type soundness)**.** *If $\emptyset \vdash s : \tau$, then there exists v such that $s \to^* v$.*

In this section, we prove the type soundness lemma for the scalar s; we prove the progress lemma in Section 1.1 and the preservation lemma in Section 1.2.

1.1 Progress

Lemma 3 (Scalar progress s)**.** *If $\emptyset \vdash s : \tau$, then either s is a value or $\exists\ s'$ with $s \to s'$. If $\emptyset \vdash I \Rightarrow s : \tau$, then $I \Rightarrow s \to \mathcal{V}$.*

Proof: By induction on a derivation of $\emptyset \vdash s : \tau$ (Note second half of the progress lemma for $I \Rightarrow s$ is straightforward by leveraging the index type soundness lemma and by induction hypothesis on s).

Case T-Constant: $\Gamma \vdash v : \star$

Immediate for scalar value.

Case T-Sum: $\dfrac{\Gamma,\ i_1 : \mathcal{D}(i_1), \ldots, i_k : \mathcal{D}(i_k) \vdash s : \star}{\Gamma \vdash \sum_{i_1 \cdots i_k}(s) : \star}$

The only evaluation rule applies is E-ScalarSum. From the premise of T-Sum, we have $\Gamma \vdash I \Rightarrow s : D \to \star$ and hence $I \Rightarrow s \to \mathcal{V}$; applying E-TensorElement to obtain each element $v_0, \cdot\cdot\cdot, v_m$ of $\mathcal{V}$. Applying E-ScalarSum, we have $\sum_{i_1 \cdots i_k}(s) = v_0 + \cdots + v_m$, thus $\sum_{i_1 \cdots i_k}(s)$ is a value.

Case T-Element: $\dfrac{\Gamma(x) = D \quad \Gamma \vdash I : D}{\Gamma \vdash t(I) : \star}$

If $I = m_1 \cdots m_k$, then $t(I) = \mathcal{V}(I)$ as $t(I)$ has type $\star$, Rule E-TensorElement or E-TensorInvalid applies; otherwise, we can achieve $I = m_1 \cdots m_k$ with E-IndexSub (premise is satisfied by leveraging the index type soundness lemma).

Case T-ScalarPlus: $\dfrac{\Gamma \vdash s_1 : \star \quad \Gamma \vdash s_2 : \star}{\Gamma \vdash s_1 + s_2 : \star}$

By the induction hypothesis, either s_1 is a value or $\exists\ s_1'$ with $s_1 \rightarrow s_1'$, and likewise s_2. If $\exists\ s_1'$ with $s_1 \rightarrow s_1'$, then E-ScalarPlus applies; if s_1 is a value and $s_2 \rightarrow s_2'$, E-ScalarPlus2 applies; if both s_1 and s_2 are values, then the conclusion vacuously holds.

Case T-ScalarTimes: $\dfrac{\Gamma \vdash s_1 : \star \quad \Gamma \vdash s_2 : \star}{\Gamma \vdash s_1 \times s_2 : \star}$

By the induction hypothesis, either s_1 is a value or $\exists\ s_1'$ with $s_1 \rightarrow s_1'$, and likewise s_2. If $\exists\ s_1'$ with $s_1 \rightarrow s_1'$, then E-ScalarTime applies; if $s_1 \rightarrow v$ and $s_2 \rightarrow s_2'$, then E-ScalarTime2 applies; if both s_1 and s_2 are values, then the conclusion vacuously holds.

Case T-ScalarLog: $\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash \mathit{log}(s) : \star}$

Proof is similar to the T-ScalarTimes case by leveraging evaluation rules E-ScalarLog and E-ScalarLog2.

Case T-ScalarExp: $\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash \mathit{exp}(s) : \star}$

Proof is similar to the T-ScalarTimes case by leveraging evaluation rules E-ScalarNatExp and E-ScalarNatExp2.

Case T-ScalarExp2: $\dfrac{\Gamma \vdash s : \star}{\Gamma \vdash s^n : \star}$

Proof is similar to the T-ScalarTimes case by leveraging evaluation rules E-ScalarExp and E-ScalarExp2.

1.2 Preservation

Scalar (s) preservation

Lemma 4 (Preservation). *If* $\Gamma \vdash s : \tau \wedge s \to s'$, *then* $\Gamma \vdash s' : \tau$.

By induction on a derivation of $\Gamma \vdash s : \tau$. We assume the desired properties holds for all subderivations ($s\_$) at each step of the induction. That is, if $\Gamma \vdash s\_ : \tau'$ and $s\_ \to s'\_$, then $\Gamma \vdash s'\_ : \tau'$, whenever $\Gamma \vdash s\_ : \tau'$ is proved by the subderivation of the present one. Our case analysis is proceeded on the last rule used in the derivation.

Case T-Constant: $s = v$

$\tau = \star$

Immediate for scalar value.

Case T-Sum: $s = \sum_I (s\_)$

$\Gamma,\ i_1 : \mathcal{D}(i_1), \ldots, i_k : \mathcal{D}(i_k) \vdash s\_ : \star$

$\tau = \star$

The only evaluation rule with sum operation on the left-hand side by which $s \to s'$ can be derived is E-ScalarSum.

Subcase E-ScalarSum: $s \to s'$ $\ s' = \sum_I (s')$ (If s can take a step).

By induction hypothesis, we have $\Gamma,\ i_1 : \mathcal{D}(i_1), \ldots, i_k : \mathcal{D}(i_k) \vdash s' : \star$, applying rule T-sum we obtain $\Gamma \vdash s' : \tau$.

Case T-Element: $s = x(I)$

$\Gamma(x) = D \quad \Gamma \vdash I : D$

$\tau = \star$

The evaluation rule that applies to $x(I)$ is either E-TensorElement or E-TensorInvalid; by leveraging the Index type soundness lemma, we have

Subcase E-TensorElement: $I = m_1 \cdots m_k \quad s' = v_{(\ldots(m_1 \times n_2 + m_2)\ldots) \times n_k + m_k}$

$\Gamma \vdash s' : \tau$ holds.

Subcase E-TensorInvalid: $I = m_1 \cdots m_k \quad s' = 0$.

$\Gamma \vdash s' : \tau$ holds.

Case T-ScalarPlus: $s = s_1 + s_2$

$\tau = \star$

Proof is by simple induction with 2 subcases E-ScalarPlus and E-ScalarPlus2.

Case T-ScalarTimes: $s = s_1 \times s_2$

$\tau = \star$

Proof is by simple induction with 2 subcases E-ScalarTimes and E-ScalarTimes2.

Case T-ScalarLog: $s = log(s)$

$\tau = \star$

Proof is by simple induction with the subcase E-ScalarLog.

Case T-ScalarExp: $s = exp(s)$

$\tau = \star$

Proof is by simple induction with the subcase E-ScalarNatExp.

Case T-ScalarExp2: $s = s^n$

$\tau = \star$

Proof is by simple induction with the subcase E-ScalarExp.

With the progress proof in Section 1.1 and preservation proof in Section 1.2, we conclude the type soundness proof for scalar s.

2 Tensor t type soundness

In this section, we prove the type soundness lemma for the tensor t; we prove the progress lemma in Section 2.1 and the preservation lemma in Section 2.2.

2.1 Progress

Lemma 5 (Tensor progress)**.** *If $\emptyset \vdash t : \tau$, then either t is a tensor value $\mathcal{V}$ or $\exists\ t'$ with $t \to t'$.*

Proof: By induction on a derivation of $\emptyset \vdash t : \tau$.

Case T-Var: $\Gamma \vdash x : \Gamma(x)$

Based on definition, this case cannot occur since our term definition is closed.

Case T-TensorValue: $\Gamma \vdash (v_0, \ldots, v_m)^{n_1 \cdots n_k} : n_1 \cdots n_k$

Immediate for tensor value.

Case T-Fun: $\dfrac{\Gamma,\ x : \mathcal{T}(x) \vdash e : \tau}{\Gamma \vdash x \Rightarrow e : \mathcal{T}(x) \rightarrow \tau}$

Immediate as $x \Rightarrow e$ is a value (analogous to lambda abstraction in the lambda calculus).

Case T-Builtin: $\Gamma \vdash \mathcal{F} : \mathcal{T}(\mathcal{F})$

Immediate; this case is similar to the T-Fun case.

Case T-App: $\dfrac{\begin{array}{c}\Gamma \vdash f : \tau_1 \rightarrow \tau_2 \\ \Gamma \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')\end{array}}{\Gamma \vdash f(t) : \sigma(\tau_2)}$

By the induction hypothesis, either t is a tensor value $\mathcal{V}$ or $\exists\ t'$ with $t \rightarrow t'$. If $\exists\ t'$ with $t \rightarrow t'$, then Rule E-App applies; if t is a tensor value $\mathcal{V}$, then Rule E-App2 applies.

Case T-TensorExp: $\dfrac{\Gamma,\ i_1 : \mathcal{D}(i_1), \ldots, i_k : \mathcal{D}(i_k) \vdash s : \star}{\Gamma \vdash (i_1 \cdots i_k) \Rightarrow s : \mathcal{D}(i_1) \cdots \mathcal{D}(i_k)}$

By the induction hypothesis, either s is a scalar value v or $\exists\ s'$ with $s \rightarrow s'$. If s is a value, then Rule E-Tensor applies; if $\exists\ s'$ with $s \rightarrow s'$, then the progress lemma for s we proved in Section 1.1 applies.

Case T-TenorPlus: $\dfrac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 + t_2 : \tau}$

By the induction hypothesis, either t_1 is a tensor value $\mathcal{V}$ or $\exists\ t_1'$ with $t_1 \rightarrow t_1'$, and likewise t_2. If $\exists\ t_1'$ with $t_1 \rightarrow t_1'$, then E-Plus applies; if t_1 is a tensor value and $t_2 \rightarrow t_2'$, E-Plus2 applies; if both t_1 and t_2 are values, then E-ValuePlus applies.

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

By the induction hypothesis, $\exists\ v$ such that s is a scalar value v or $\exists\ s'$ with $s \to s'$, and likewise for t. If $\exists\ s'$ with $s \to s'$, then E-ScalarTensorProd applies; if s is a scalar value v and $t \to t'$, then E-ScalarTensorProd2 applies; if both s and t are values, then E-ValueScale applies.

Case T-Cast: $\dfrac{\texttt{flat}(\Gamma(x)) = \texttt{flat}(D)}{\Gamma \vdash (D)\ x : D}$

Immediate follows by leveraging the Dimension type soundness lemma.

2.2 Preservation

Tensor (t) preservation of types under substitution

Lemma 6 (Tensor (t) preservation of types under substitution)**.** *If* $\Gamma, x : \tau' \vdash t : \tau \wedge \Gamma \vdash \mathcal{V} : \tau'$*, then* $\Gamma \vdash [x \mapsto \mathcal{V}]t : \tau$*.*

Proof: By induction on a derivation of the statement $\Gamma, x : \tau' \vdash t : \tau$.

Note we use y to replace x in the original rules to avoid name conflict with environment variable x whenever applicable; we also assume $x \neq y$ and $y \notin FV(t)$.

Case T-Var, T-Builtin, T-TensorValue:

Immediate for varible, built-in function, and tensor value types.

Case T-Fun: $t = y \Rightarrow t$

$\tau = \mathcal{T}(y) \to \tau$

$\Gamma, x : \tau', y : \mathcal{T}(y) \vdash t : \tau$

Using permutation on the given subderivation, we obtain $\Gamma, y : \mathcal{T}(y), x : \tau' \vdash e : \tau$. Using weakening on the other given derivation ($\Gamma \vdash \mathcal{V} : \tau'$), we obtain $\Gamma, y : \mathcal{T}(y) \vdash \mathcal{V} : \tau'$. By induction hypothesis, $\Gamma, y : \mathcal{T}(y) \vdash [x \mapsto \mathcal{V}]t : \tau$. By T-Fun, $\Gamma \vdash [x \mapsto \mathcal{V}]y \Rightarrow t : \mathcal{T}(y) \rightarrow \tau$.

Case T-App: $t = f(t)$

$\tau = \sigma(\tau_2)$

$\Gamma, x : \tau' \vdash f : \tau_1 \rightarrow \tau_2$

$\Gamma, x : \tau' \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$

By the induction hypothesis, $\Gamma, \vdash [x \mapsto \mathcal{V}]f : \tau_1 \rightarrow \tau_2$ and $\Gamma \vdash [x \mapsto \mathcal{V}]t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$. By T-App, $\Gamma \vdash [x \mapsto \mathcal{V}]f([x \mapsto \mathcal{V}]t) : \tau$, i.e., $\Gamma \vdash [x \mapsto \mathcal{V}](f(t)) : \tau$.

Case T-TenorPlus and T-TensorScale:

The proof is similar to the proof for T-Fun except we use the induction step twice (one on t_1 and another on t_2 for the T-TenorPlus rule; and one on s and another on t for the T-TensorScale rule) to achieve the conclusion.

Case T-Cast: $t : (D)\ x$

$\tau : D$

$\texttt{flat}(\Gamma(x)) = \texttt{flat}(D)$

Immediate as the requirements of the lemma is vacuously satisfied.

Tensor (t) preservation

Lemma 7 (Preservation). *If* $\Gamma \vdash t : \tau \wedge t \rightarrow t'$, *then* $\Gamma \vdash t' : \tau$.

Case T-Var:

Can't happen as there is no evaluation rules with a variable as the left-hand side.

Case T-Builtin: t is either $\mathcal{F}(t)$ or $\mathcal{F}(\mathcal{V})$

$$\tau = \mathcal{T}(\mathcal{F})$$

Proof is by simple induction with the subcases E-BuiltinApp and E-BuiltinApp2.

Case T-TensorValue: $t = (v_0, \ldots, v_m)^{n_1 \cdots n_k}$

$$\tau = n_1 \cdots n_k$$

Can't happen as there is no evaluation rules with a tensor variable as the left-hand side.

Case T-Fun:

Can't happen as there is no evaluation rules with a $x \Rightarrow e$ as the left-hand size.

Case T-App: $t = (x \Rightarrow t)(t)$

$$\tau = \sigma(\tau_2)$$

$$\Gamma \vdash (x \Rightarrow t) : \tau_1 \to \tau_2$$

$$\Gamma \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$$

Subcase E-App: $t \to t' \quad t' = (x \Rightarrow t')(t')$

Applying T-App we obtain $t' : \tau$.

Subcase E-App2: $t \to \mathcal{V} \quad t' = (x \Rightarrow t')(\mathcal{V})$

Applying the tensor substitution lemma we obtain $t' : \tau$.

Case T-TensorExp: $t = (i_1 \cdots i_k) \Rightarrow s$

$$\tau = \mathcal{D}(i_1) \cdots \mathcal{D}(i_k)$$

Proof is by simple induction with the subcase E-Tensor and leverage the substitution lemma for tensor t.

Case T-TenorPlus: $t = t_1 + t_2$

$$\tau = \tau$$

Proof is by simple induction with the subcases E-Plus, E-Plus2, and E-ValuePlus.

Case T-TensorScale: $t = s \cdot t$

$$\tau = \tau$$

Proof is by simple induction with the subcases E-ScalarTensorProd (Note for this case, we need to leverage the conclusion from the scalar preservation lemma we just proved in Section 1.2), E-ScalarTensorProd2, and E-ScalarTensorProd.

Case T-Cast: $t = (D)\ x$

$$\tau = D$$

Proof is by simple induction with the subcase E-TensorCast and leveraging the Dimension type soundness lemma.

With the progress proof in Section 2.1 and preservation proof in Section 2.2, we conclude the type soundness proof for tensor t.

3 Scalar $\hat{s}$ type soundness

We have proved the type soundness lemma for scalar s in Section 1, to complete the type soundness lemma proof for scalar $\hat{s}$, in this section we prove the progress lemma in Section 3.1 and the preservation lemma in Section 3.2 for the function application scalar $(x \Rightarrow \hat{s})(t)$.

3.1 Progress

Lemma 8 (Scalar progress $\hat{s}$). *If $\emptyset \vdash \hat{s} : \tau$, then either $\hat{s}$ is a value or $\exists\ \hat{s}'$ with $\hat{s} \to \hat{s}'$.*

$$\text{Case T-App: } \frac{\begin{array}{c}\Gamma \vdash (x \Rightarrow \hat{s}) : \tau_1 \to \tau_2 \\ \Gamma \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')\end{array}}{\Gamma \vdash (x \Rightarrow \hat{s})(t) : \sigma(\tau_2)}$$

By the induction hypothesis, either t is a tensor value $\mathcal{V}$ or $\exists\ t'$ with $t \to t'$. If $\exists\ t'$ with $t \to t'$, then Rule E-App applies; if t is a tensor value $\mathcal{V}$, then Rule E-App2 applies.

3.2 Preservation

Function application scalar ($\hat{s}$) preservation of types under substitution

Lemma 9 (Function application scalar ($\hat{s}$) preservation of types under substitution). *If $\Gamma, x : \tau' \vdash \hat{s} : \tau \land \Gamma \vdash \mathcal{V} : \tau'$, then $\Gamma \vdash [x \mapsto \mathcal{V}]\hat{s} : \tau$.*

Proof: By induction on a derivation of the statement $\Gamma, x : \tau' \vdash \hat{s} : \tau$.

Case T-App: $\hat{s} = f(\hat{s})$

$$\tau = \sigma(\tau_2)$$

$$\Gamma, x : \tau' \vdash f : \tau_1 \to \tau_2$$

$$\Gamma, x : \tau' \vdash \hat{s} : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$$

By the induction hypothesis, $\Gamma, \vdash [x \mapsto \mathcal{V}]f : \tau_1 \to \tau_2$ and $\Gamma \vdash [x \mapsto \mathcal{V}]t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$. By T-App, $\Gamma \vdash [x \mapsto \mathcal{V}]f([x \mapsto \mathcal{V}]\hat{s}) : \tau$, i.e., $\Gamma \vdash [x \mapsto \mathcal{V}](f(\hat{s})) : \tau$.

Function application scalar $\hat{s}$ preservation

Lemma 10 (Scalar preservation $\hat{s}$)**.** *If* $\Gamma \vdash \hat{s} : \tau \wedge \hat{s} \rightarrow \hat{s}'$*, then* $\Gamma \vdash \hat{s}' : \tau$*.*

Case T-App: $\hat{s} = (x \Rightarrow \hat{s})(t)$

$\tau = \sigma(\tau_2)$

$\Gamma \vdash (x \Rightarrow \hat{s}) : \tau_1 \rightarrow \tau_2$

$\Gamma \vdash t : \tau_1' \quad \sigma = \mathcal{U}(\tau_1, \tau_1')$

Subcase E-App: $t \rightarrow t' \quad \hat{s}' = (x \Rightarrow \hat{s})(t')$

Applying T-App we obtain $\hat{s}' : \tau$.

Subcase E-App2: $t \rightarrow \mathcal{V} \quad \hat{s}' = (x \Rightarrow \hat{s})(\mathcal{V})$

Applying the function application scalar substitution lemma we obtain $\hat{s}' : \tau$.

Combining proofs in Section 1, 3.1, and 3.2, we complete the type soundness lemma for $\hat{s}$.

4 Proof conclusion

With type soundness proof for tensor t in Section 2 and scalar $\hat{s}$ in Section 3, we conclude the type soundness proof for Theorem 3.1.

CURRICULUM VITAE

Xiao Bing Huang

Education

B.Sc., Tianjin University, China, 1994

Major: Computer Science

M.Sc., Memorial University of Newfoundland, Canada, 2006

Major: Computer Science

Dissertation Title: DEEPDSL: A DOMAIN SPECIFIC LANGUAGE FOR DEEP LEARNING