

An Introduction to Computational Networks and the Computational Network Toolkit

Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen,
Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter,
Mark Hillebrand, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov,
Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek,
Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro,
Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng,
Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney,
Andreas Stolcke, Huaming Wang, Kaisheng Yao, Dong Yu,
Yu Zhang, Geoffrey Zweig
(*authors are listed in alphabetical order)

MSR-TR-2014-112 (DRAFT v1.0: Feb 17, 2016)

Abstract

We introduce computational network (CN), a unified framework for describing arbitrary learning machines, such as deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short term memory (LSTM), logistic regression, and maximum entropy model, that can be illustrated as a series of computational steps. A CN is a directed graph in which each leaf node represents an input value or a parameter and each non-leaf node represents a matrix operation upon its children. We describe algorithms to carry out forward computation and gradient calculation in CN and introduce most popular computation node types used in a typical CN.

We further introduce the computational network toolkit (CNTK), an implementation of CN that supports both GPU and CPU. We describe the architecture and the key components of the CNTK, the command line options to use CNTK, and the network definition and model editing language, and provide sample setups for acoustic model, language model, and spoken language understanding. We also describe the Argon speech recognition decoder as an example to integrate with CNTK.

Contents

1	Introduction	7
1.1	Overview	7
1.2	A Simple Demo	8
1.3	A Speech Demo (TIMIT)	12
2	Computational Network	16
2.1	Computational Network	16
2.2	Forward Computation	19
2.3	Model Training	21
2.4	Typical Computation Nodes	26
2.4.1	Computation Node Types with No Operand	26
2.4.2	Computation Node Types with One Operand	26
2.4.3	Computation Node Types with Two Operands	32
2.4.4	Computation Node Types for Computing Statistics	40
2.5	Convolutional Neural Network	41
2.6	Recurrent Connections	45
2.6.1	Sample by Sample Processing Only Within Loops	46
2.6.2	Processing Multiple Utterances Simultaneously	47
2.6.3	Building Arbitrary Recurrent Neural Networks	47
3	Computational Network Toolkit	51
3.1	Run CNTK	51
3.2	Network Builders	54
3.2.1	The Simple Network Builder	54
3.2.2	The Network Description Language (NDL)	55
3.3	Learners	56
3.3.1	Stochastic Gradient Descent Learner	57
3.3.1.1	Training process control	57
3.3.1.2	Learning rate and momentum control	58

3.3.1.3	Gradient control	60
3.3.1.4	Adaptation	61
3.3.1.5	Information display	61
3.3.1.6	Gradient Check	61
3.4	Data Readers	61
3.4.1	UCIFastReader	62
3.4.2	HTKMLFReader	63
3.4.3	LMSequenceReader	68
3.4.4	LUSequenceReader	69
3.5	Top-Level Commands	71
3.5.1	Train Command	72
3.5.2	Adapt Command	72
3.5.3	Test or Eval Command	72
3.5.4	CV	73
3.5.5	Write Command	73
3.5.6	Edit Command	74
3.5.7	SVD Command	74
3.5.8	Dumpnode Command	75
3.5.9	WriteWordAndClass Command	75
3.5.10	CreateLabelMap Command	76
3.6	Plot Command	76
3.7	ConvertDBN Command	77
3.8	Additional Top-Level Configurations	77
3.8.1	Stderr	78
3.8.2	DeviceId	78
3.8.3	Precision	78
3.8.4	NumCPUThreads	79
3.8.5	TraceLevel	79
3.8.6	ShareNodeValueMatrices	79
3.9	Advanced Command Line Parsing Rules	79
3.9.1	Commands and Actions	81
3.9.2	Configuration Overloads In the Command Line	81
3.9.3	Layered Configuration Files	82
3.9.4	Stringize Variables	83
3.9.5	Default, Repeated Values, and Comments	84
4	Advanced Setups in Computational Network Toolkit	85
4.1	Network Definition Language	85
4.1.1	Basic Concepts	85
4.1.1.1	Variables	86

4.1.1.2	Inputs	87
4.1.1.3	Parameters	87
4.1.1.4	Functions	87
4.1.1.5	Training and Testing Criteria	88
4.1.1.6	Special Nodes	88
4.1.1.7	Comments	89
4.1.2	Macros	89
4.1.2.1	Defining Macros	89
4.1.2.2	Using Macros	90
4.1.3	Optional Parameters	91
4.1.3.1	Parameter initialization	91
4.1.3.2	Tagging special values	91
4.1.4	NDL Functions	92
4.1.4.1	InputValue or Input	92
4.1.4.2	ImageInput or Image	93
4.1.4.3	LearnableParameter or Parameter	93
4.1.4.4	Constant or Const	94
4.1.4.5	RectifiedLinear or ReLU, Sigmoid, Tanh, Log	95
4.1.4.6	Softmax	95
4.1.4.7	LogSoftmax	95
4.1.4.8	SumElements	96
4.1.4.9	SumColumnElements	96
4.1.4.10	Negate	96
4.1.4.11	Reshape	96
4.1.4.12	RowSlice	97
4.1.4.13	RowStack	97
4.1.4.14	Scale	97
4.1.4.15	Times	97
4.1.4.16	TransposeTimes	98
4.1.4.17	DiagTimes	98
4.1.4.18	Plus, Minus	98
4.1.4.19	ElementTimes, RowElementTimes, ColumnElementTimes,	98
4.1.4.20	SquareError or SE	99
4.1.4.21	CrossEntropyWithSoftmax or CEWithSM	99
4.1.4.22	ClassBasedCrossEntropyWithSoftmax or CBCEWithSM	99
4.1.4.23	ErrorPrediction or ClassificationError	100
4.1.4.24	CosDistance or CosDist	100

4.1.4.25	CosDistanceWithNegativeSamples or CosWith-NegSamples	101
4.1.4.26	MatrixL1Reg or L1Reg, MatrixL2Reg or L2Reg	101
4.1.4.27	Mean	101
4.1.4.28	InvStdDev	102
4.1.4.29	PerDimMeanVarNormalization or PerDimMVNorm	102
4.1.4.30	PerDimMeanVarDeNormalization or PerDimMV-DeNorm	102
4.1.4.31	Dropout	103
4.1.4.32	Convolution or Convolve	103
4.1.4.33	MaxPooling, AveragePooling	104
4.1.4.34	PastValue (or Delay) and FutureValue	105
4.2	Model Editing Language	105
4.2.1	Basic Features	106
4.2.1.1	Loading and Setting Default Models	107
4.2.1.2	Viewing a Model File	107
4.2.1.3	Copy Nodes	108
4.2.1.4	SetInput	109
4.2.1.5	Adding New Nodes: In-line NDL and NDL Snippets	109
4.2.1.6	SaveModel	110
4.2.1.7	Name Matching	111
4.2.2	MEL Command Reference	111
4.2.2.1	CreateModel, CreateModelWithName	112
4.2.2.2	LoadModel, LoadModelWithName	112
4.2.2.3	SaveDefaultModel, SaveModel	112
4.2.2.4	UnloadModel	113
4.2.2.5	LoadNDLSnippet	113
4.2.2.6	Dump, DumpModel	113
4.2.2.7	DumpNode	114
4.2.2.8	Copy, CopyNode	114
4.2.2.9	CopySubTree	115
4.2.2.10	SetInput or SetNodeInput	115
4.2.2.11	SetInputs or SetNodeInputs	116
4.2.2.12	SetProperty	116
4.2.2.13	SetPropertyForSubTree	117
4.2.2.14	Remove or RemoveNode, Delete or DeleteNode	117
4.2.2.15	Rename	117

5	Speech Recognition Decoder Integration	119
5.1	Argon Speech Recognition Decoder	119
5.1.1	Representing the Acoustic Search Space	120
5.1.1.1	From HTK Outputs to Acoustic Search Space	120
5.1.2	Representing the Language Model	121
5.1.3	Command-Line Parameters	121
5.1.3.1	WFST Compilation Parameters	121
5.1.3.2	Parameters specifying inputs	121
5.1.3.3	Parameters specifying outputs	122
5.1.3.4	Search Parameters	122
5.2	Constructing a Decoding Graph	122
5.2.1	Context-Independent Phonetic Recognition	123
5.2.2	Cross-Word Context-Dependent Large Vocabulary Decoding	123
5.2.2.1	Stage One: Extract structure from HTK donor model	123
5.2.2.2	Stage Two: Build and Compile Graph	124
5.2.3	Running Argon	124
6	Example Setups	126
6.1	Acoustic Model	126
6.1.1	Training a DNN with SimpleNetworkBuilder	126
6.1.2	Adapting the learning rate based on development data	128
6.1.3	Training a DNN with NDNetworkBuilder	129
6.1.4	Training an autoencoder	130
6.1.5	Using layer-by-layer discriminative pre-training	131
6.1.6	Training a network with multi-task learning	133
6.1.7	Training a network with multiple inputs	134
6.1.8	Evaluating networks and cross validation	136
6.1.9	Writing network outputs to files	136
6.2	RNN Language Model	138
6.2.1	Train	138
6.2.2	Test	140
6.3	LSTM Language Model	141
6.3.1	Training	141
6.3.2	Test	142
6.4	Spoken Language Understanding	142
6.4.1	Training	143
6.4.2	Test	144

Chapter 1

Introduction

1.1 Overview

The Computational Network Toolkit (CNTK) is a software package that makes it easy to design and test computational networks such as deep neural networks. A computational network is a style of computation where data flows through a graph and computations happen in the nodes of the graph. The goal of a computational network is to take feature data, transform the data through a network of simple computations, and then produce one or more outputs. The output is often some sort of decision based on the input features. A computational network can take many forms—such as feed-forward, recursive or convolutional—and includes various forms of computations and non-linearities. The network’s parameters are optimized to produce the “best” possible outcome for a given set of data and an optimization criteria.

In this chapter we can only give readers a taste for CNTK, and specific features are described in more detail later. With this purpose, CNTK has 5 primary features:

- CNTK is a general solution for training and testing many kinds of neural networks
- A user specifies a network using a simple text configuration file. The configuration file specifies the type of network, where to find the input data, and how to optimize the parameters. All of these design parameters are fixed in the configuration file.
- CNTK is designed to seamlessly put as many of the computations onto a GPU, if available, as possible. These types of computational networks are easy to vectorize and fit nicely onto GPUs. CNTK is compatible with GPUs that support the CUDA programming environment.

- CNTK automatically calculates the needed derivatives in order to efficiently perform the necessary optimizations. Networks are composed of many simple elements, and CNTK can keep track of the details to insure that the optimization is done correctly.
- CNTK can be extended by adding small amounts of C++ code to implement the necessary blocks. New data readers, non-linearities, and objective functions are easy to add.

Consider the work needed to build a non-standard neural network such as the variable parameter DNN [1]. Conventionally, one needs to design the network, derive the derivatives needed to optimize the network, implement the algorithm, and then run the experiments. These steps are error prone and time consuming. With CNTK, however in many cases, you only need to write a simple configuration file. The rest of this chapter describes the configuration file needed to implement two simple examples.

The purpose of this documentation is to describe how to use CNTK for your own experiments. The bulk of this documentation describes the definition and mathematics of computational networks (Chapter 2), how to run and configure CNTK (Chapter 3), how to design networks and edit the models (Chapter 4). Chapter 5 describes a modern speech decoder known as Argon, and Chapter 6 describes a number of CNTK examples. But first we start with a couple of simple examples to show the basics of CNTK.

1.2 A Simple Demo

We used a simple Matlab script (included in the distribution) to generate two-dimensional data with binary labels. The raw data is shown below in Figure 1.1. This data was stored as a text file, one sample per line, that can be read by the `UCIFastReader` method (Section 3.4.1) in CNTK.

The following configuration file tells CNTK how to process this data. There are three sections of this file:

- The preamble specifies which command blocks to run (`Simple_Demo`), which GPU to use, and how to log CNTK's output.
- The `Simple_Demo` block specifies one action for CNTK. The start of the block specifies the action to take (train) but more importantly specifies the actions needed to train the desired network. This block specifies three components in order to train a new network:

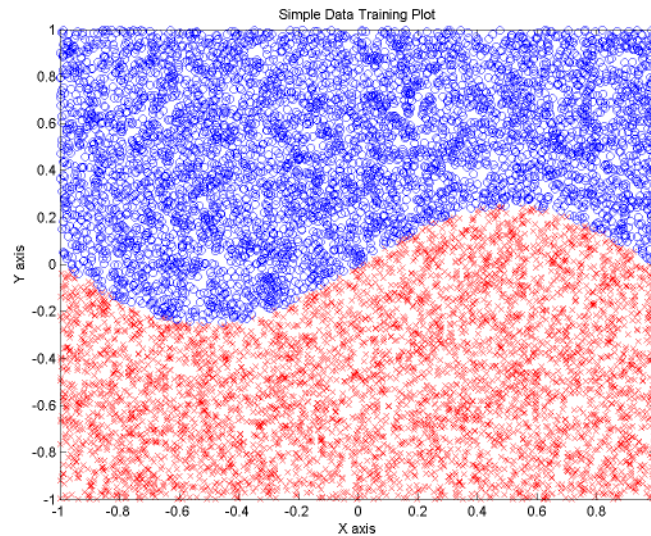


Figure 1.1: A simple test set to demonstrate the power and ease of CNTK.

- The SimpleNetworkBuilder block specifies the topology of the network, how to initialize the network, and how to evaluate its performance (cross-entry with softmax in this case).
 - The SGD (Stochastic Gradient Descent) block tells CNTK how to optimize the network to find the best parameters. This includes information about miniblock size (so the computation is more efficient), the learning rate, and how many epochs to train.
 - The reader block specifies from where to read the input data, how the features and labels are formatted, and how to randomize the order the data is applied to the network during training
- Finally, the Simple_Demo_Output block specifies how to test the network. Again we must specify the source of the data (we are training and testing on the same data to keep the example small) and how to evaluate the results. CNTK write the value of the two output nodes, for each test case, into a text file.

The entire script is shown below.

```
# command=Simple_Demo_Output
command=Simple_Demo:Simple_Demo_Output
```

```

# deviceId=-1 or "CPU" for CPU, >=0 for GPU devices
DeviceNumber=0
stderr="Demo"
precision="float"
modelPath="models/simple.dnn"
deviceId=$DeviceNumber$
outputNodeNames=ScaledLogLikelihood
traceLevel=1
#####
# TRAINING CONFIG (Simple, Fixed LR) #
#####
Simple_Demo=[
    action="train"
    # Notation xxx:yyy*n:zzz is equivalent to xxx,
    # then yyy repeated n times, then zzz
    # example: 10:20*3:5 is equivalent to 10:20:20:20:5
    SimpleNetworkBuilder = [
        # 2 input, 2 50-element hidden, 2 output
        layerSizes=2:50*2:2
        trainingCriterion="CrossEntropyWithSoftmax"
        evalCriterion="ErrorPrediction"
        layerTypes="Sigmoid"
        initValueScale=1.0
        applyMeanVarNorm=true
        uniformInit=true
        needPrior=true
    ]
    SGD = [
        # epochSize=0 means epochSize is the size of
        # the training set. Must be evenly divisible
        # into number of data frames.
        epochSize=0
        minibatchSize=25
        learningRatesPerMB=0.5:0.2*20:0.1
        momentumPerMB=0.9
        dropoutRate=0.0
        maxEpochs=500
    ]
    # Parameter values for the reader
    reader = [
        # reader to use
        readerType="UCIFastReader"
        file="SimpleDataTrain.txt"
        miniBatchMode="partial"
        randomize=Auto
    ]

```

```

    verbosity=1
    features=[
        dim=2 # two-dimensional input data
        start=0 # Start with first element on line
    ]
    labels=[
        start=2 # Skip two elements
        dim=1 # One label dimension
        labelDim=2 # Two labels possible
        labelMappingFile="SimpleMapping.txt"
    ]
]
]
#####
# OUTPUT RESULTS (Simple) #
#####
Simple_Demo_Output=[
    action="write"
    # Parameter values for the reader
    reader=[
        # reader to use
        readerType="UCIFastReader"
        file="SimpleDataTest.txt"
        features=[
            dim=2
            start=0
        ]
        labels=[
            start=2
            dim=1
            labelDim=2
            labelMappingFile="SimpleMapping.txt"
        ]
    ]
    outputPath="SimpleOutput" # Dump output as text
]

```

You can run this script using the following command on Windows

cntk configFile=Simple.cntk

or under Linux

cntk configFile=Simple.cntk

This CNTK job trains a feedforward network with 2-dimensional input (x and

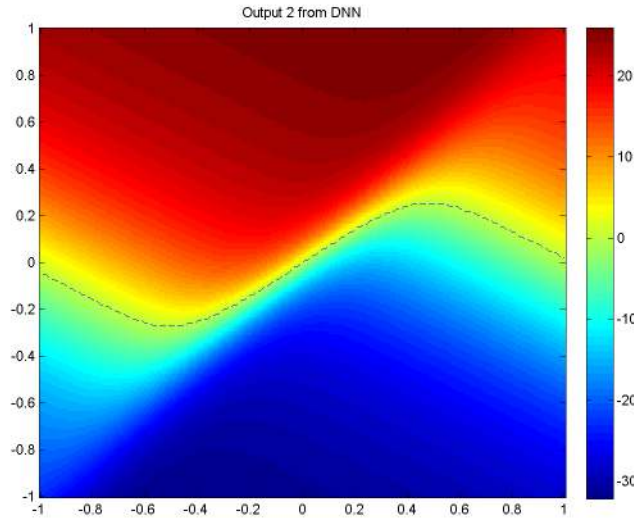


Figure 1.2: Trained output values over the region of interest for the first (of two) output neurons. The dashed line shows the position where the output is zero (the decision boundary).

y coordinates), and two hidden layers with 50 units each, and then with two output units (one for each decision). CNTK uses cross entropy with softmax to measure and optimize the network's performance.

Figure 1.2 shows the optimized output (before the sigmoid) of one of the two output nodes.

The output neuron successfully captures the shape of the decision boundary. The network's error rate, shown in Figure 1.3, over time is interesting.

On a 2.2GHz Xeon processor with a low-end Quadro 4000 GPU, the training and testing take 11 minutes 9 seconds (it is this long because the model size and minibatch size are both small, and the GPU is more efficient with larger amounts of data).

1.3 A Speech Demo (TIMIT)

A second example shows how to use CNTK to do a simple speech experiment. We use the LDC's TIMIT speech database as training and testing data. While this data is not free, it is relatively common and illustrates a real speech task. The data is available at <https://catalog.ldc.upenn.edu/LDC93S1>.

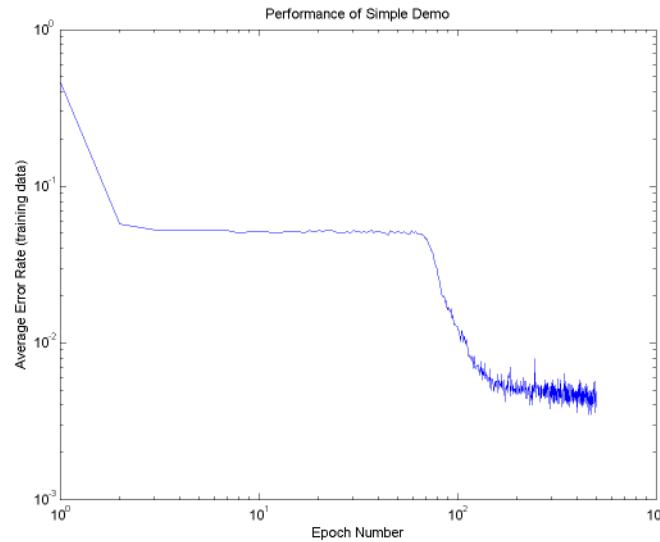


Figure 1.3: Training error rate vs. epoch number for the simple demo. The network quickly converges to a solution that gets most of the data correct, before finding the non-linearity that allows it to track the “bumps” in the decision boundary.

There are three steps in using TIMIT with CNTK: prepare the data, train the network, and finally test the network’s performance. We will describe each step in turn. We provide a (cygwin) shell script to perform the necessary actions, but the necessary steps can easily be replicated with other shells.

The first step is to prepare the data using HTK (Hidden Markov Toolkit). This involves creating a list of files (the HTK script) and then passing this script to HTK’s HCopy program, which calculates the filterbank features we use in our experiment. Not all the files in TIMIT are usable, so we start with a list of valid subjects (the text file TimitSubjects.txt). A python script uses this list of subjects, prepends the directory where the TIMIT distribution is found, and creates two script files. The first script file is used by HCopy to tell it where to read each audio file (in NIST’s wave format, which is different from Microsoft’s wave format) and where to put the output (in a directory called features.) We also take this opportunity to change the directory structure so the output feature files (____.zda) are stored in a flat directory called Features. The second script file is used as input to CNTK and tells CNTK where to find each input file, already processed into features. If you do not have python, or you prefer, you can also edit these two included .scp files by hand so each line contains the directory where your data is stored.

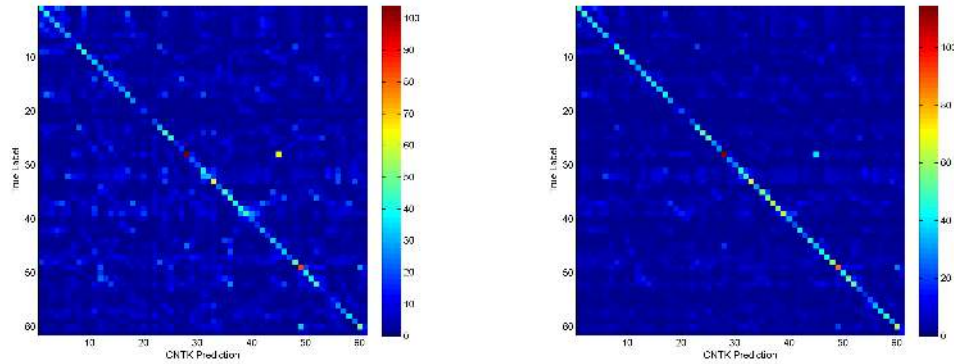


Figure 1.4: (Square root of the) phone confusion matrices after 1 (left) and 100 (right) epochs. To highlight the off-diagonal terms in these figures, we are displaying the square root of all the matrix elements. The single bright yellow off-diagonal pixel in the epoch 1 result corresponds to phones that are supposed to be “pau” (silence) but are confused with “h#” (the beginning/end of the word.).

The second step is to use HTK’s HCopy program to compute the necessary speech features. In this study the features are 24-dimensional filterbank features, plus delta and delta-delta features, and the results are stored in the Features/ directory. The original TIMIT database is supplied in multiple directories and subdirectories, but we’ve flattened the directory structure to make later processing easier. (No extra directories to create.)

The third and final step is to use CNTK to compute a model that predicts the correct tri-phone label. The neural network has 792 input nodes, 3 hidden layers with 512 nodes each, and 183 output nodes.

Input: The number of input nodes in the network divided by the dimensionality of the HTK feature vector determines the width of the context window. In this case 11 frames are used as input to the network.

Output: The TIMIT data is labeled with 61 different phonemes, and the network is trained to recognize tri-phones. Thus there are $61 \times 3 = 183$ output nodes.

We use a couple of Matlab scripts to analyze the output data and compute the phone confusion matrix. Figure 1.4 shows the confusion matrices after 1 and 100 epochs. As expected, there is much less energy in the off-diagonal terms as the number of epochs grows. This is also shown in Figure 1.5 which shows the training error rate calculated after each epoch.

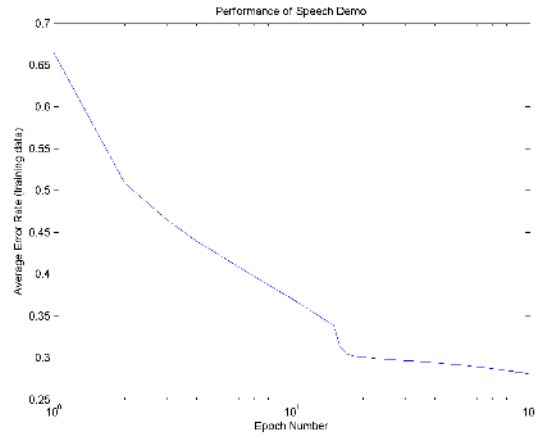


Figure 1.5: Training error rate as a function of epoch number for the TIMIT speech example.

On a 2.2GHz Xeon processor the training takes 167 minutes without a GPU, and 61 minutes with a low-end Quadro 4000 GPU.

Chapter 2

Computational Network

2.1 Computational Network

There is a common property in key machine learning models, such as deep neural networks (DNNs) [2, 3, 4, 5, 6, 7], convolutional neural networks (CNNs) [8, 9, 10, 11, 12, 13, 14, 15, 16, 17], and recurrent neural networks (RNNs) [18, 19, 20, 21, 22]. All these models can be described as a series of computational steps. For example, a one-hidden-layer sigmoid neural network can be described as the computational steps listed in Algorithm 2.1. If we know how to compute each step and in which order the steps are computed we have an implementation of the neural network. This observation suggests that we can unify all these models under the framework of computational network (CN), part of which has been implemented in toolkits such as Theano [23], CNTK [24] and RASR/NN [25].

Algorithm 2.1 Computational Steps Involved in an One-Hidden-Layer Sigmoid Neural Network

```
1: procedure ONEHIDDENLAYERNNCOMPUTATION(X)  
     $\triangleright$  Each column of X is an observation vector  
2:    $\mathbf{T}^{(1)} \leftarrow \mathbf{W}^{(1)}\mathbf{X}$   
3:    $\mathbf{P}^{(1)} \leftarrow \mathbf{T}^{(1)} + \mathbf{B}^{(1)}$   $\triangleright$  Each column of  $\mathbf{B}^{(1)}$  is the bias  $\mathbf{b}^{(1)}$   
4:    $\mathbf{S}^{(1)} \leftarrow \sigma(\mathbf{P}^{(1)})$   $\triangleright \sigma(\cdot)$  is the sigmoid function applied element-wise  
5:    $\mathbf{T}^{(2)} \leftarrow \mathbf{W}^{(2)}\mathbf{S}^{(1)}$   
6:    $\mathbf{P}^{(2)} \leftarrow \mathbf{T}^{(2)} + \mathbf{B}^{(2)}$   $\triangleright$  Each column of  $\mathbf{B}^{(2)}$  is the bias  $\mathbf{b}^{(2)}$   
7:    $\mathbf{O} \leftarrow \text{softmax}(\mathbf{P}^{(2)})$   $\triangleright$  Apply softmax column-wise to get output O  
8: end procedure
```

A computational network is a directed graph $\{\mathbb{V}, \mathbb{E}\}$, where \mathbb{V} is a set of vertices and \mathbb{E} is a set of directed edges. Each vertex, called a computation node,

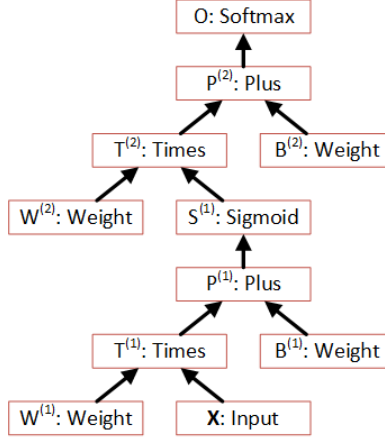


Figure 2.1: Represent the one-hidden-layer sigmoid neural network of Algorithm 2.1 with a computational network

represents a computation. Vertices with edges toward a computation node are the operands of the associated computation and sometimes called the children of the computation node. Here the order of operands matters for some operations such as matrix multiplication. Leaf nodes do not have children and are used to represent input values or model parameters that are not result of some computation. A CN can be easily represented as a set of computation nodes n and their children $\{n : c_1, \dots, c_{K_n}\}$, where K_n is the number of children of node n . For leaf nodes $K_n = 0$. Each computation node knows how to compute the value of itself given the input operands (children).

Figure 2.1 is the one-hidden-layer sigmoid neural network of Algorithm 2.1 represented as a CN in which each node n is identified by a $\{\text{nodename} : \text{operatortype}\}$ pair and takes its ordered children as the operator's inputs. From the figure, we can observe that in CN there is no concept of layers. Instead, a computation node is the basic element of operations. This makes the description of a simple model such as DNN more cumbersome, but this can be alleviated by grouping computation nodes together with macros. In return, CN provides us with greater flexibility in describing arbitrary networks and allows us to build almost all models we are interested in within the same unified framework. For example, we can easily modify the network illustrated in Figure 2.1 to use rectified linear unit instead of a sigmoid nonlinearity. We can also build a network that has two input nodes as shown in Figure 2.2 or a network with shared model parameters as shown in Figure 2.3.

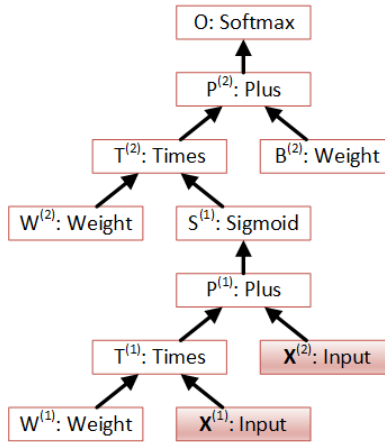


Figure 2.2: A computational network with two input nodes (highlighted).

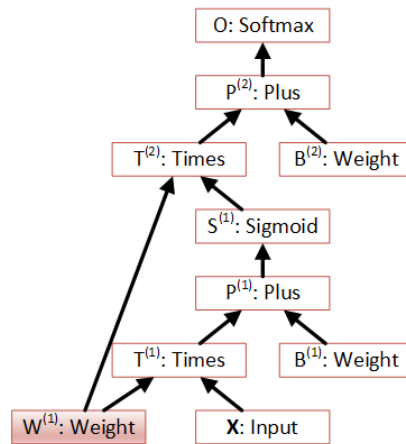


Figure 2.3: A computational network with shared model parameters (highlighted). Here we assume the input X , hidden layer $S^{(1)}$, and output layer O have the same dimension.

2.2 Forward Computation

When the model parameters (i.e., weight nodes in Figure 2.1) are known, we can compute the value of any node given the new input values. Unlike in the DNN case, where the computation order can be trivially determined as layer-by-layer computation from bottom up, in CN different network structure comes with a different computation order. When the CN is a directed acyclic graph (DAG) the computation order can be determined with a depth-first traverse over the DAG. Note that in a DAG there is no directed cycles (i.e., no recurrent loop). However, there might be loops if we don't consider edge directions, for which Figure 2.3 is an example. This is because the same computation node may be a child of several other nodes. Algorithm 2.2. determines the computation order of a DAG and takes care of this condition. Once the order is decided, it will remain the same for all the subsequent runs, regardless of the computational environment. In other words, this algorithm only needs to be executed per output node and then cache the computation order. Following the order determined by Algorithm 2.2, the forward computation of the CN is carried out synchronously. The computation of the next node starts only after the computation of the previous node has finished. It is suitable for environments where single computing device, such as one GPGPU or one CPU host, is used, or the CN itself is inherently sequential, e.g., when the CN represents a DNN.

Algorithm 2.2 Synchronous forward computation of a CN. The computation order is determined by a depth-first traverse over the DAG.

```

1: procedure DECIDEFORWARDCOMPUTATIONORDER(root, visited, order)
    ▷ Enumerate nodes in the DAG in the depth-first order.
    ▷ visited is initialized as an empty set. order is initialized as an empty
    queue
2:   if root  $\notin$  visited then    ▷ the same node may be a child of several nodes.
3:     visited  $\leftarrow$  visited  $\cup$  root
4:     for each c  $\in$  root.children do    ▷ apply to children recursively
5:       call DECIDEFORWARDCOMPUTATIONORDER(c, visited, order)
6:     end for
7:     order  $\leftarrow$  order + root    ▷ Add root to the end of order
8:   end if
9: end procedure

```

The forward computation can also be carried out asynchronously with which the order of the computation is determined dynamically. This can be helpful when the CN has many parallel branches and there are more than one computing device to compute these branches in parallel. Algorithm 2.3 shows an algorithm that car-

ries out the forward computation of a CN asynchronously. In this algorithm, all the nodes whose children have not been computed are in the waiting set and those whose children are computed are in the ready set. At the beginning, all non-leaf descendants of *root* are in the waiting set and all leaf descendants are in the ready set. The scheduler picks a node from the ready set based on some policy, removes it from the ready set, and dispatches it for computation. Popular policies include first-come/first-serve, shortest task first, and least data movement. When the computation of the node finishes, the system calls the `SIGNALCOMPLETE` method to signal to all its parent nodes. If all the children of a node have been computed, the node is moved from the waiting set to the ready set. The algorithm stops when all nodes are computed. Although not explicitly indicated in the Algorithm 2.3, the `SIGNALCOMPLETE` procedure is called under concurrent threads and should be guarded for thread safety. This algorithm can be exploited to carry out computation on any DAG instead of just a CN.

Algorithm 2.3 Asynchronous forward computation of a CN. A node is moved to the ready set when all its children have been computed. A scheduler monitors the ready set and decides where to compute each node in the set.

```

1: procedure SIGNALCOMPLETE(node, waiting, ready)
    ▷ Called when the computation of the node is finished. Needs to be
    thread safe.
    ▷ waiting is initialized to include all non-leaf descendants of root.
    ▷ ready is initialized to include all leaf descendants of root.
2:   for each  $p \in \text{node.parents} \wedge p \in \text{waiting}$  do
3:      $p.\text{numFinishedChildren}++$ 
4:     if  $p.\text{numFinishedChildren} == p.\text{numChildren}$  then
        ▷ all children have been computed
5:        $\text{waiting} \leftarrow \text{waiting} - \text{node}$ 
6:        $\text{ready} \leftarrow \text{ready} \cup \text{node}$ 
7:     end if
8:   end for
9: end procedure
10: procedure SCHEDULECOMPUTATION(ready)
    ▷ Called by the job scheduler when a new node is ready or computation
    resource is available.
11:   pick node  $\in \text{ready}$  according to some policy
12:    $\text{ready} \leftarrow \text{ready} - \text{node}$ 
13:   dispatch node for computation.
14: end procedure

```

In many cases, we may need to compute the value for a node with changing input values. To avoid duplicate computation of shared branches, we can add a time stamp to each node and only recompute the value of the node if at least one of the children has newer value. This can be easily implemented by updating the time stamp whenever a new value is provided or computed, and by excluding nodes whose children is older from the actual computation.

In both Algorithms 2.2 and 2.3 each computation node needs to know how to compute its value when the operands are known. The computation can be as simple as matrix summation or element-wise application of sigmoid function or as complex as whatever it may be. We will describes the evaluation functions for popular computation node types in Section 2.4.

2.3 Model Training

To train a CN, we need to define a training criterion J . Popular criteria include cross-entropy (CE) for classification and mean square error (MSE) for regression. Since the training criterion is also a result of some computation, it can be represented as a computation node and inserted into the CN. Figure 2.4 illustrates a CN that represents an one-hidden-layer sigmoid neural network augmented with a CE training criterion node. If the training criterion contains regularization terms the regularization terms can also be implemented as computation nodes and the final training criterion node is a weighted sum of the main criterion and the regularization term.

The model parameters in a CN can be optimized over a training set $\mathbb{S} = \{(\mathbf{x}^m, \mathbf{y}^m) | 0 \leq m < M\}$ using the minibatch based backpropagation (BP) algorithm. More specifically, we improve the model parameter \mathbf{W} at each step $t + 1$ as

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \varepsilon \Delta \mathbf{W}_t, \quad (2.1)$$

where

$$\Delta \mathbf{W}_t = \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla_{\mathbf{W}_t} J(\mathbf{W}; \mathbf{x}^m, \mathbf{y}^m), \quad (2.2)$$

and M_b is the minibatch size. The key here is the computation of $\nabla_{\mathbf{W}_t} J(\mathbf{W}; \mathbf{x}^m, \mathbf{y}^m)$ which we will simplify as $\nabla_{\mathbf{W}}^J$. Since a CN can have arbitrary structure, we cannot use the exact same BP algorithm described in Algorithm ?? to compute $\nabla_{\mathbf{W}}^J$.

A naive solution to compute $\nabla_{\mathbf{W}}^J$ is illustrated in Figure 2.5, in which $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters. In this solution, each edge is associated with a partial derivative, and

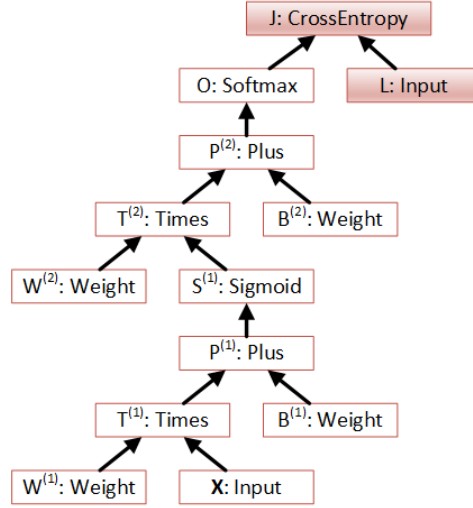


Figure 2.4: The one-hidden-layer sigmoid neural network augmented with a cross entropy training criterion node J and a label node L .

$$\nabla_{\mathbf{W}^{(1)}}^J = \frac{\partial J}{\partial \mathbf{V}^{(1)}} \frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}} \frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(1)}} + \frac{\partial J}{\partial \mathbf{V}^{(3)}} \frac{\partial \mathbf{V}^{(3)}}{\partial \mathbf{V}^{(4)}} \frac{\partial \mathbf{V}^{(4)}}{\partial \mathbf{W}^{(1)}} \quad (2.3)$$

$$\nabla_{\mathbf{W}^{(2)}}^J = \frac{\partial J}{\partial \mathbf{V}^{(1)}} \frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}} \frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(2)}}. \quad (2.4)$$

This solution comes with two major drawbacks. First, each derivative can have very high dimension. If $\mathbf{V} \in \mathbb{R}^{N_1 \times N_2}$ and $\mathbf{W} \in \mathbb{R}^{N_3 \times N_4}$ then $\frac{\partial \mathbf{V}}{\partial \mathbf{W}} \in \mathbb{R}^{(N_1 \times N_2) \times (N_3 \times N_4)}$. This means a large amount of memory is needed to keep the derivatives. Second, there are many duplicated computations. For example, $\frac{\partial J}{\partial \mathbf{V}^{(1)}} \frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}}$ is computed twice in this example, once for $\nabla_{\mathbf{W}^{(1)}}^J$ and once for $\nabla_{\mathbf{W}^{(2)}}^J$.

Fortunately, there is a much simpler and more efficient approach to compute the gradient as illustrated in Figure 2.6. In this approach, each node n keeps two values: the evaluation (forward computation) result \mathbf{V}_n and the gradient ∇_n^J . Note that the training criterion J is always a scalar, If $\mathbf{V}_n \in \mathbb{R}^{N_1 \times N_2}$ then $\nabla_n^J \in \mathbb{R}^{N_1 \times N_2}$. This requires significantly less memory than that required in the naive solution illustrated in Figure 2.5. This approach also allows for factorizing out the common prefix terms and making computation linear in the number of nodes in the graph. For example, $\frac{\partial J}{\partial \mathbf{V}^{(2)}}$ is computed only once and used twice when computing $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$ with this approach. This is analogous to com-

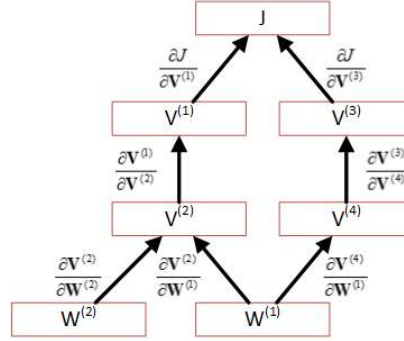


Figure 2.5: The naive gradient computation in CN. $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters and each edge is associated with a partial derivative.

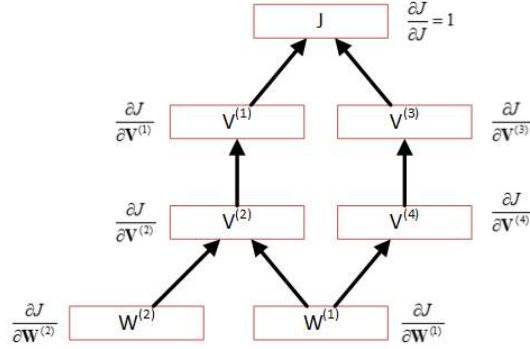


Figure 2.6: An efficient gradient computation algorithm in CN. $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters. Each node n stores both the value of the node v_n and the gradient ∇_n^J .

mon subexpression elimination in a conventional expression graph, only here the common subexpressions are the parents of the nodes, rather than the children.

Automatic differentiation has been an active research area for decades and many techniques have been proposed ([26]). A simple recursive algorithm for deciding the gradient computation order so that the gradient computation can be efficiently carried out is shown in Algorithm 2.4 to which a similar recursive algorithm for scalar functions has been provided [27, 28]. This algorithm assumes that each node has a `ComputePartialGradient(child)` function which computes the gradient of the training criterion with regard to the node's child *child* and is called in the order that is decided by the algorithm. Before the algorithm is computed, the gradient ∇_n^J at each node n is set to 0, the queue *order* is set to empty, and

$parentsLeft$ is set to the number of parents of each node. This function is then called on a criterion node that evaluates to a scalar. Similar to the forward computation, an asynchronous algorithm can be derived. Once the gradient is known, the minibatch based stochastic gradient descent (SGD) algorithm and other training algorithms that depend on gradient only can be used to train the model.

Alternatively, the gradients can be computed by following the reverse order of the forward computation and calling each node's parents' `ComputePartialGradient(child)` function and passing itself as the *child* argument. This approach, however, requires additional book keeping, e.g., keeping pointers to a node's parents which can introduce additional overhead when manipulating the network architecture.

Algorithm 2.4 Reverse automatic gradient computation algorithm. At the top level *node* must be a training criterion node that evaluates to a scalar.

```

1: procedure DECIDEGRADIENTCOMPUTATIONORDER(node, parentsLeft,
   order)
   ▷ Decide the order to compute the gradient at all descendants of node.
   ▷ parentsLeft is initialized to the number of parents of each node.
   ▷ order is initialized to an empty queue.

2:   if IsNotLeaf(node) then
3:     parentsLeft[node] ← —
4:     if parentsLeft[node] == 0 ∧ node ∉ order then ▷ All parents have
       been computed.
5:       order ← order + node ▷ Add node to the end of order
6:       for each c ∈ node.children do
7:         call DECIDEGRADIENTCOMPUTATIONORDER(c,
           parentsLeft, order)
8:       end for
9:     end if
10:  end if
11: end procedure

```

In many cases, not all the gradients need to be computed. For example, the gradient with regard to the input value is never needed. When adapting the model, some of the model parameters don't need to be updated and thus it is unnecessary to compute the gradients with regard to these parameters. We can reduce the gradient computation by keeping a *needGradient* flag for each node. Once the flags of leaf nodes (either input values or model parameters) are known, the flags of the non-leaf nodes can be determined using Algorithm 2.5, which is essentially a depth-first traversal over the DAG. Since both Algorithms 2.2 and 2.5 are essentially depth-first traversal over the DAG and both only need to be executed once they may be

combined in one function.

Algorithm 2.5 Update the *needGradient* flag recursively.

```

1: procedure UPDATENEEDGRADIENTFLAG(root, visited)
    ▷ Enumerate nodes in the DAG in the depth-first order.
    ▷ visited is initialized as an empty set.
2:   if root  $\notin$  visited then   ▷ The same node may be a child of several nodes
    and revisited.
3:     visited  $\leftarrow$  visited  $\cup$  root
4:     for each c  $\in$  root.children do
5:       call UPDATENEEDGRADIENTFLAG(c, visited, order)
6:       if IsNotLeaf(node) then
7:         if node.AnyChildNeedGradient() then
8:           node.needGradient  $\leftarrow$  true
9:         else
10:          node.needGradient  $\leftarrow$  false
11:        end if
12:      end if
13:    end for
14:  end if
15: end procedure

```

Since every instantiation of a CN is task dependent and different, it is critical to have a way to check and verify the gradients computed automatically. A simple technique to estimate the gradient numerically is:

$$\frac{\partial J}{\partial w_{ij}} \approx \frac{J(w_{ij} + \epsilon) - J(w_{ij} - \epsilon)}{2\epsilon}, \quad (2.5)$$

where w_{ij} is the (i, j) -th element of a model parameter \mathbf{W} , ϵ is a small constant typically set to 10^{-4} , and $J(w_{ij} + \epsilon)$ and $J(w_{ij} - \epsilon)$ are the objective function values evaluated with all other parameters fixed and w_{ij} changed to $w_{ij} + \epsilon$ and $w_{ij} - \epsilon$, respectively. In most cases the numerically estimated gradient and the gradient computed from the automatic gradient computation agree to at least 4 significant digits if double precision computation is used. Note that this technique works well with a large range of ϵ values, except extremely small values such as 10^{-20} which would lead to numerical roundoff errors.

2.4 Typical Computation Nodes

For the forward computation and gradient calculation algorithms described above to work we assume that each type of computation node implements a function *Evaluate* to evaluate the value of the node given the values of its child nodes, and the function *ComputePartialGradient(child)* to compute the gradient of the training criterion with regard to the child node *child* given the node value \mathbf{V}_n and the gradient ∇_n^J of the node *n* and values of all its child nodes. For simplicity we will remove the subscript in the following discussion.

In this section we introduce the most widely used computation node types and the corresponding *Evaluate* and *ComputePartialGradient(child)* functions. In the following discussion we use symbols listed in Table 2.1 to describe the computations. We treat each minibatch of input values as a matrix in which each column is a sample. In all the derivations of the gradients we use the identity

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}}, \quad (2.6)$$

where v_{mn} is the (m, n) -th element of the matrix \mathbf{V} , and x_{ij} is the (i, j) -th element of matrix \mathbf{X} .

2.4.1 Computation Node Types with No Operand

The values of a computation node that has no operand are given instead of computed. As a result both *Evaluate* and *ComputePartialGradient(child)* functions for these computation node types are empty.

- *Parameter*: Used to represent model parameters that need to be saved as part of the model.
- *InputValue*: used to represent features, labels, or control parameters that are provided by users at run time.

2.4.2 Computation Node Types with One Operand

In these computation node types, $\text{Evaluate} = \mathbf{V}(\mathbf{X})$ and $\text{ComputePartialGradient}(\mathbf{X}) = \nabla_{\mathbf{X}}^J$

- *Negate*: reverse the sign of each element in the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow -\mathbf{X} \quad (2.7)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J - \nabla_{\mathbf{n}}^J. \quad (2.8)$$

Symbols	Description
λ	a scalar
\mathbf{d}	column vector that represents the diagonal of a square matrix
\mathbf{X}	matrix of the first operand
\mathbf{Y}	matrix of the second operand
V	value of current node
∇_n^J (or ∇_V^J)	gradient of the current node
$\nabla_{\mathbf{X}}^J$	gradient of the child node (operand) \mathbf{X}
$\nabla_{\mathbf{Y}}^J$	gradient of the child node (operand) \mathbf{Y}
\bullet	element-wise product
\oslash	element-wise division
\circ	inner product of vectors applied on matrices column-wise
\odot	inner product applied to each row
$\delta(\cdot)$	Kronecker delta
$\mathbf{1}_{m,n}$	an $m \times n$ matrix with all 1's
\mathbf{X}^α	element-wise power
$\text{vec}(\mathbf{X})$	vector formed by concatenating columns of \mathbf{X}

Table 2.1: Symbols used in describing the computation nodes

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} -1 & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.9)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = -\frac{\partial J}{\partial v_{ij}}. \quad (2.10)$$

- *Sigmoid*: apply the sigmoid function element-wise to the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow \frac{1}{1 + e^{-\mathbf{X}}} \quad (2.11)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet [\mathbf{V} \bullet (1 - \mathbf{V})]. \quad (2.12)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} v_{ij}(1 - v_{ij}) & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.13)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} v_{ij} (1 - v_{ij}). \quad (2.14)$$

- *Tanh*: apply the tanh function element-wise to the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow \frac{e^{\mathbf{X}} - e^{-\mathbf{X}}}{e^{\mathbf{X}} + e^{-\mathbf{X}}} \quad (2.15)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet (1 - \mathbf{V} \bullet \mathbf{V}). \quad (2.16)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 1 - v_{ij}^2 & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.17)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} (1 - v_{ij}^2) \quad (2.18)$$

- *ReLU*: apply the rectified linear operation element-wise to the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow \max(0, \mathbf{X}) \quad (2.19)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet \delta(\mathbf{X} > 0). \quad (2.20)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} \delta(x_{ij} > 0) & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.21)$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} \delta(x_{ij} > 0). \quad (2.22)$$

- *Log*: apply the log function element-wise to the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow \log(\mathbf{X}) \quad (2.23)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet \frac{1}{\mathbf{X}}. \quad (2.24)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} \frac{1}{x_{ij}} & m = i \wedge n = j \\ 0 & else \end{cases} \quad (2.25)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} \frac{1}{x_{ij}}. \quad (2.26)$$

- *Exp*: apply the exponent function element-wise to the operand \mathbf{X} .

$$\mathbf{V}(\mathbf{X}) \leftarrow \exp(\mathbf{X}) \quad (2.27)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet \mathbf{V}. \quad (2.28)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} v_{ij} & m = i \wedge n = j \\ 0 & else \end{cases} \quad (2.29)$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} v_{ij}. \quad (2.30)$$

- *Softmax*: apply the softmax function column-wise to the operand \mathbf{X} . Each column is treated as a separate sample.

$$m_j(\mathbf{X}) \leftarrow \max_i x_{ij} \quad (2.31)$$

$$e_{ij}(\mathbf{X}) \leftarrow e^{x_{ij} - m_j(\mathbf{X})} \quad (2.32)$$

$$s_j(\mathbf{X}) \leftarrow \sum_i e_{ij}(\mathbf{X}) \quad (2.33)$$

$$v_{ij}(\mathbf{X}) \leftarrow \frac{e_{ij}(\mathbf{X})}{s_j(\mathbf{X})} \quad (2.34)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + [\nabla_{\mathbf{n}}^J - \nabla_{\mathbf{n}}^J \circ \mathbf{V}] \bullet \mathbf{V}. \quad (2.35)$$

The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} v_{ij} (1 - v_{ij}) & m = i \wedge n = j \\ -v_{mj} v_{ij} & m \neq i \wedge n = j \\ 0 & else \end{cases} \quad (2.36)$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \left(\frac{\partial J}{\partial v_{ij}} - \sum_m \frac{\partial J}{\partial v_{mj}} v_{mj} \right) v_{ij}. \quad (2.37)$$

- *LogSoftmax*: apply the log of softmax function column-wise to the operand \mathbf{X} . Each column is treated as a separate sample.

$$m_j(\mathbf{X}) \leftarrow \max_i x_{ij} \quad (2.38)$$

$$e_{ij}(\mathbf{X}) \leftarrow e^{x_{ij} - m_j(\mathbf{X})} \quad (2.39)$$

$$s_j(\mathbf{X}) \leftarrow \sum_i e_{ij}(\mathbf{X}) \quad (2.40)$$

$$v_{ij}(\mathbf{X}) \leftarrow \log \frac{e_{ij}(\mathbf{X})}{s_j(\mathbf{X})} \quad (2.41)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + [\nabla_{\mathbf{n}}^J - \exp(\mathbf{V}) \text{Diag}(\text{VectorSum}(\nabla_{\mathbf{n}}^J))] \quad (2.42)$$

where $\text{VectorSum}(\cdot)$ sums each column vector, and $\text{Diag}(\mathbf{x})$ is a diagonal matrix formed by vector \mathbf{x} . The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 1 - \exp(v_{ij}) & m = i \wedge n = j \\ -\exp(v_{ij}) & m \neq i \wedge n = j \\ 0 & else \end{cases} \quad (2.43)$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \left(\frac{\partial J}{\partial v_{ij}} - \sum_m \frac{\partial J}{\partial v_{mj}} \exp(v_{ij}) \right). \quad (2.44)$$

- *SumElements*: sum over all elements in the operand \mathbf{X} .

$$\mathbf{v}(\mathbf{X}) \leftarrow \sum_{i,j} x_{ij} \quad (2.45)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J. \quad (2.46)$$

The gradient can be derived by noting that v and $\nabla_{\mathbf{n}}^J$ are scalars,

$$\frac{\partial v}{\partial x_{ij}} = 1 \quad (2.47)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial x_{ij}} = \frac{\partial J}{\partial v}. \quad (2.48)$$

- *L1Norm*: take the matrix L_1 norm of the operand \mathbf{X} .

$$\mathbf{v}(\mathbf{X}) \leftarrow \sum_{i,j} |x_{ij}| \quad (2.49)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \text{sgn}(\mathbf{X}). \quad (2.50)$$

The gradient can be derived by noting that v and $\nabla_{\mathbf{n}}^J$ are scalars,

$$\frac{\partial v}{\partial x_{ij}} = \text{sgn}(x_{ij}) \quad (2.51)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial x_{ij}} = \frac{\partial J}{\partial v} \text{sgn}(x_{ij}). \quad (2.52)$$

- *L2Norm*: take the matrix L_2 norm (Frobenius norm) of the operand \mathbf{X} .

$$\mathbf{v}(\mathbf{X}) \leftarrow \sqrt{\sum_{i,j} (x_{ij})^2} \quad (2.53)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \frac{1}{v} \nabla_{\mathbf{n}}^J \mathbf{X}. \quad (2.54)$$

The gradient can be derived by noting that v and $\nabla_{\mathbf{n}}^J$ are scalars,

$$\frac{\partial v}{\partial x_{ij}} = \frac{x_{ij}}{v} \quad (2.55)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial x_{ij}} = \frac{1}{v} \frac{\partial J}{\partial v} x_{ij}. \quad (2.56)$$

•

2.4.3 Computation Node Types with Two Operands

In these computation node types, $Evaluate = V(a, \mathbf{Y})$, where a can be \mathbf{X} , λ or \mathbf{d} , and $ComputePartialGradient(\mathbf{b}) = \nabla_{\mathbf{b}}^J$ where \mathbf{b} can be \mathbf{X} , \mathbf{Y} or \mathbf{d} .

- *Scale*: scale each element of \mathbf{Y} by λ .

$$V(\lambda, \mathbf{Y}) \leftarrow \lambda \mathbf{Y} \quad (2.57)$$

$$\nabla_{\lambda}^J \leftarrow \nabla_{\lambda}^J + \text{vec}(\nabla_{\mathbf{n}}^J) \circ \text{vec}(\mathbf{Y}) \quad (2.58)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \lambda \nabla_{\mathbf{n}}^J. \quad (2.59)$$

The gradient ∇_{λ}^J can be derived by observing

$$\frac{\partial v_{mn}}{\partial \lambda} = y_{mn} \quad (2.60)$$

and

$$\frac{\partial J}{\partial \lambda} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial \lambda} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} y_{mn}. \quad (2.61)$$

Similarly to derive the gradient $\nabla_{\mathbf{Y}}^J$, we note that

$$\frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} \lambda & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.62)$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \lambda \frac{\partial J}{\partial v_{ij}} \quad (2.63)$$

- *Times*: matrix product of operands \mathbf{X} and \mathbf{Y} . Must satisfy $\mathbf{X.cols} = \mathbf{Y.rows}$.

$$V(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X}\mathbf{Y} \quad (2.64)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \mathbf{Y}^T \quad (2.65)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \mathbf{X}^T \nabla_{\mathbf{n}}^J. \quad (2.66)$$

The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} y_{jn} & m = i \\ 0 & \text{else} \end{cases} \quad (2.67)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \sum_n \frac{\partial J}{\partial v_{in}} y_{jn}. \quad (2.68)$$

Similarly to derive the gradient $\nabla_{\mathbf{Y}}^J$, we note that

$$\frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} x_{mi} & n = j \\ 0 & \text{else} \end{cases} \quad (2.69)$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \sum_m \frac{\partial J}{\partial v_{mj}} x_{mi} \quad (2.70)$$

- *ElementTimes*: element-wise product of two matrices. Must satisfy $\mathbf{X.rows} = \mathbf{Y.rows}$ and $\mathbf{X.cols} = \mathbf{Y.cols}$.

$$v_{ij}(\mathbf{X}, \mathbf{Y}) \leftarrow x_{ij} y_{ij} \quad (2.71)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \bullet \mathbf{Y} \quad (2.72)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \nabla_{\mathbf{n}}^J \bullet \mathbf{X}. \quad (2.73)$$

The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} y_{ij} & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.74)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} y_{ij}. \quad (2.75)$$

The gradient $\nabla_{\mathbf{Y}}^J$ can be derived exactly the same way due to symmetry.

- *Plus*: sum of two matrices \mathbf{X} and \mathbf{Y} . Must satisfy $\mathbf{X.rows} = \mathbf{Y.rows}$. If $\mathbf{X.cols} \neq \mathbf{Y.cols}$ but one of them is a multiple of the other, the smaller matrix needs to be expanded by repeating itself.

$$\mathbf{V}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} + \mathbf{Y} \quad (2.76)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \begin{cases} \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J & \mathbf{X.rows} = \mathbf{V.rows} \wedge \mathbf{X.cols} = \mathbf{V.cols} \\ \nabla_{\mathbf{X}}^J + \mathbf{1}_{1, \mathbf{V.rows}} \nabla_{\mathbf{n}}^J & \mathbf{X.rows} = 1 \wedge \mathbf{X.cols} = \mathbf{V.cols} \\ \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V.cols}, 1} & \mathbf{X.rows} = \mathbf{V.rows} \wedge \mathbf{X.cols} = 1 \\ \nabla_{\mathbf{X}}^J + \mathbf{1}_{1, \mathbf{V.rows}} \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V.cols}, 1} & \mathbf{X.rows} = 1 \wedge \mathbf{X.cols} = 1 \end{cases} \quad (2.77)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \begin{cases} \nabla_{\mathbf{Y}}^J + \nabla_{\mathbf{n}}^J & \mathbf{Y.rows} = \mathbf{V.rows} \wedge \mathbf{Y.cols} = \mathbf{V.cols} \\ \nabla_{\mathbf{Y}}^J + \mathbf{1}_{1, \mathbf{V.rows}} \nabla_{\mathbf{n}}^J & \mathbf{Y.rows} = 1 \wedge \mathbf{Y.cols} = \mathbf{V.cols} \\ \nabla_{\mathbf{Y}}^J + \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V.cols}, 1} & \mathbf{Y.rows} = \mathbf{V.rows} \wedge \mathbf{Y.cols} = 1 \\ \nabla_{\mathbf{Y}}^J + \mathbf{1}_{1, \mathbf{V.rows}} \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V.cols}, 1} & \mathbf{Y.rows} = 1 \wedge \mathbf{Y.cols} = 1 \end{cases} \quad (2.78)$$

The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing that when \mathbf{X} has the same dimension as \mathbf{V} , we have

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 1 & m = i \wedge n = j \\ 0 & \text{else} \end{cases} \quad (2.79)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{ij}} \quad (2.80)$$

If $\mathbf{X.rows} = 1 \wedge \mathbf{X.cols} = \mathbf{V.cols}$ we have

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} 1 & n = j \\ 0 & \text{else} \end{cases} \quad (2.81)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \sum_m \frac{\partial J}{\partial v_{mj}} \quad (2.82)$$

We can derive $\nabla_{\mathbf{X}}^J$ and $\nabla_{\mathbf{Y}}^J$ under other conditions similarly.

- *Minus*: difference of two matrices \mathbf{X} and \mathbf{Y} . Must satisfy $\mathbf{X.rows} = \mathbf{Y.rows}$. If $\mathbf{X.cols} \neq \mathbf{Y.cols}$ but one of them is a multiple of the other, the smaller matrix needs to be expanded by repeating itself.

$$V(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} - \mathbf{Y} \quad (2.83)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \begin{cases} \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla_{\mathbf{X}}^J + \mathbf{1}_{1, \mathbf{V}.rows} \nabla_{\mathbf{n}}^J & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V}.cols, 1} & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = 1 \\ \nabla_{\mathbf{X}}^J + \mathbf{1}_{1, \mathbf{V}.rows} \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V}.cols, 1} & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = 1 \end{cases} \quad (2.84)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \begin{cases} \nabla_{\mathbf{Y}}^J - \nabla_{\mathbf{n}}^J & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla_{\mathbf{Y}}^J - \mathbf{1}_{1, \mathbf{V}.rows} \nabla_{\mathbf{n}}^J & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla_{\mathbf{Y}}^J - \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V}.cols, 1} & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = 1 \\ \nabla_{\mathbf{Y}}^J - \mathbf{1}_{1, \mathbf{V}.rows} \nabla_{\mathbf{n}}^J \mathbf{1}_{\mathbf{V}.cols, 1} & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = 1 \end{cases} \quad (2.85)$$

The derivation of the gradients is similar to that for the *Plus* node.

- *DiagTimes*: the product of a diagonal matrix (whose diagonal equals to \mathbf{d}) and an arbitrary matrix \mathbf{Y} . Must satisfy $\mathbf{d}.rows = \mathbf{Y}.rows$.

$$v_{ij}(\mathbf{d}, \mathbf{Y}) \leftarrow d_i y_{ij} \quad (2.86)$$

$$\nabla_{\mathbf{d}}^J \leftarrow \nabla_{\mathbf{d}}^J + \nabla_{\mathbf{n}}^J \odot \mathbf{Y} \quad (2.87)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \text{DiagTimes}(\mathbf{d}, \nabla_{\mathbf{n}}^J). \quad (2.88)$$

The gradient $\nabla_{\mathbf{d}}^J$ can be derived by observing

$$\frac{\partial v_{mn}}{\partial d_i} = \begin{cases} y_{in} & m = i \\ 0 & else \end{cases} \quad (2.89)$$

and

$$\frac{\partial J}{\partial d_i} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial d_i} = \sum_n \frac{\partial J}{\partial v_{in}} y_{in} \quad (2.90)$$

Similarly to derive the gradient $\nabla_{\mathbf{Y}}^J$ we note that

$$\frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} d_i & m = i \wedge n = j \\ 0 & else \end{cases} \quad (2.91)$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \frac{\partial J}{\partial v_{ij}} d_i \quad (2.92)$$

- *Dropout*: randomly set λ percentage of values of \mathbf{Y} to be zero and scale the rest so that the expectation of the sum is not changed:

$$m_{ij}(\lambda) \leftarrow \begin{cases} 0 & rand(0,1) \leq \lambda \\ \frac{1}{1-\lambda} & else \end{cases} \quad (2.93)$$

$$v_{ij}(\lambda, \mathbf{Y}) \leftarrow m_{ij} y_{ij} \quad (2.94)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \begin{cases} \nabla_{\mathbf{n}}^J & \lambda = 0 \\ \nabla_{\mathbf{n}}^J \bullet \mathbf{M} & else \end{cases} \quad (2.95)$$

Note that λ is a given value instead of part of the model. We only need to get the gradient with regard to \mathbf{Y} . If $\lambda = 0$ then $\mathbf{V} = \mathbf{X}$ which is a trivial case. Otherwise it's equivalent to the *ElementTimes* node with a randomly set mask \mathbf{M} .

- *KhatriRaoProduct*: column-wise cross product of two matrices \mathbf{X} and \mathbf{Y} . Must satisfy $\mathbf{X}.cols = \mathbf{Y}.cols$. Useful for constructing tensor networks.

$$\mathbf{v}_{.j}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{x}_{.j} \otimes \mathbf{y}_{.j} \quad (2.96)$$

$$[\nabla_{\mathbf{X}}^J]_{.j} \leftarrow [\nabla_{\mathbf{X}}^J]_{.j} + \left[[\nabla_{\mathbf{n}}^J]_{.j} \right]_{\mathbf{X}.rows, \mathbf{Y}.rows} \mathbf{Y} \quad (2.97)$$

$$[\nabla_{\mathbf{Y}}^J]_{.j} \leftarrow [\nabla_{\mathbf{Y}}^J]_{.j} + \left[\left[[\nabla_{\mathbf{n}}^J]_{.j} \right]_{\mathbf{X}.rows, \mathbf{Y}.rows} \right]^T \mathbf{X}, \quad (2.98)$$

where $[\mathbf{X}]_{m,n}$ reshapes \mathbf{X} to become an $m \times n$ matrix. The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} y_{kj} & n = j \wedge i = m / \mathbf{Y}.rows \wedge k = modulus(m, \mathbf{Y}.rows) \\ 0 & else \end{cases} \quad (2.99)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \sum_{i,k} \frac{\partial J}{\partial v_{i \times \mathbf{Y}.rows + k, j}} y_{kj}. \quad (2.100)$$

The gradient $\nabla_{\mathbf{Y}}^J$ can be derived similarly.

- *Cos*: column-wise cosine distance of two matrices \mathbf{X} and \mathbf{Y} . Must satisfy $\mathbf{X}.cols = \mathbf{Y}.cols$. The result is a row vector. Frequently used in natural language processing tasks.

$$v_{.j}(\mathbf{X}, \mathbf{Y}) \leftarrow \frac{\mathbf{x}_{.j}^T \mathbf{y}_{.j}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} \quad (2.101)$$

$$[\nabla_{\mathbf{X}}^J]_{.j} \leftarrow [\nabla_{\mathbf{X}}^J]_{.j} + [\nabla_{\mathbf{n}}^J]_{.j} \bullet \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} v_{.j}}{\|\mathbf{x}_{.j}\|^2} \right] \quad (2.102)$$

$$[\nabla_{\mathbf{Y}}^J]_{.j} \leftarrow [\nabla_{\mathbf{Y}}^J]_{.j} + [\nabla_{\mathbf{n}}^J]_{.j} \bullet \left[\frac{x_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{y_{ij} v_{.j}}{\|\mathbf{y}_{.j}\|^2} \right]. \quad (2.103)$$

The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing

$$\frac{\partial v_{.n}}{\partial x_{ij}} = \begin{cases} \frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} (\mathbf{x}_{.j}^T \mathbf{y}_{.j})}{\|\mathbf{x}_{.j}\|^3 \|\mathbf{y}_{.j}\|} & n = j \\ 0 & else \end{cases} \quad (2.104)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_n \frac{\partial J}{\partial v_{.n}} \frac{\partial v_{.n}}{\partial x_{ij}} = \frac{\partial J}{\partial v_{.j}} \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} (\mathbf{x}_{.j}^T \mathbf{y}_{.j})}{\|\mathbf{x}_{.j}\|^3 \|\mathbf{y}_{.j}\|} \right] \quad (2.105)$$

$$= \frac{\partial J}{\partial v_{.j}} \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} v_{.j}}{\|\mathbf{x}_{.j}\|^2} \right]. \quad (2.106)$$

The gradient $\nabla_{\mathbf{Y}}^J$ can be derived similarly.

- *ClassificationError*: compute the total number of columns in which the indexes of the maximum values disagree. Each column is considered as a sample and δ is the Kronecker delta. Must satisfy $\mathbf{X}.cols = \mathbf{Y}.cols$.

$$a_j(\mathbf{X}) \leftarrow \arg \max_i x_{ij} \quad (2.107)$$

$$b_j(\mathbf{Y}) \leftarrow \arg \max_i y_{ij} \quad (2.108)$$

$$v(\mathbf{X}, \mathbf{Y}) \leftarrow \sum_j \delta(a_j(\mathbf{X}) \neq b_j(\mathbf{Y})) \quad (2.109)$$

This node type is only used to compute classification errors during the decoding time and is not involved in the model training. For this reason, calling *ComputePartialGradient*(**b**) should just raise an error.

- *SquareError*: compute the square of Frobenius norm of the difference $\mathbf{X} - \mathbf{Y}$. Must satisfy $\mathbf{X}.rows = \mathbf{Y}.rows$ and $\mathbf{X}.cols = \mathbf{Y}.cols$.

$$v(\mathbf{X}, \mathbf{Y}) \leftarrow \frac{1}{2} \text{Tr} \left((\mathbf{X} - \mathbf{Y})(\mathbf{X} - \mathbf{Y})^T \right) \quad (2.110)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \nabla_{\mathbf{n}}^J (\mathbf{X} - \mathbf{Y}) \quad (2.111)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J - \nabla_{\mathbf{n}}^J (\mathbf{X} - \mathbf{Y}). \quad (2.112)$$

Note that v is a scalar. The derivation of the gradients is trivial given

$$\frac{\partial v}{\partial \mathbf{X}} = \mathbf{X} - \mathbf{Y} \quad (2.113)$$

$$\frac{\partial v}{\partial \mathbf{Y}} = -(\mathbf{X} - \mathbf{Y}). \quad (2.114)$$

- *CrossEntropy*: compute the sum of cross entropy computed column-wise (over samples) where each column of \mathbf{X} and \mathbf{Y} is a probability distribution. Must satisfy $\mathbf{X}.\text{rows} = \mathbf{Y}.\text{rows}$ and $\mathbf{X}.\text{cols} = \mathbf{Y}.\text{cols}$.

$$\mathbf{R}(\mathbf{Y}) \leftarrow \log(\mathbf{Y}) \quad (2.115)$$

$$v(\mathbf{X}, \mathbf{Y}) \leftarrow -\text{vec}(\mathbf{X}) \circ \text{vec}(\mathbf{R}(\mathbf{Y})) \quad (2.116)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J - \nabla_{\mathbf{n}}^J \mathbf{R}(\mathbf{Y}) \quad (2.117)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J - \nabla_{\mathbf{n}}^J (\mathbf{X} \oslash \mathbf{Y}). \quad (2.118)$$

Note that v is a scalar. The gradient $\nabla_{\mathbf{X}}^J$ can be derived by observing

$$\frac{\partial v}{\partial x_{ij}} = -\log(y_{ij}) = -r_{ij}(\mathbf{Y}) \quad (2.119)$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial x_{ij}} = -\frac{\partial J}{\partial v} r_{ij}(\mathbf{Y}) \quad (2.120)$$

Similarly to derive the gradient $\nabla_{\mathbf{Y}}^J$ we note that

$$\frac{\partial v}{\partial y_{ij}} = -\frac{x_{ij}}{y_{ij}} \quad (2.121)$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial y_{ij}} = -\frac{\partial J}{\partial v} \frac{x_{ij}}{y_{ij}}. \quad (2.122)$$

- *CrossEntropyWithSoftmax*: same as *CrossEntropy* except that \mathbf{Y} contains values before the softmax operation (i.e., unnormalized).

$$\mathbf{P}(\mathbf{Y}) \leftarrow \text{Softmax}(\mathbf{Y}) \quad (2.123)$$

$$\mathbf{R}(\mathbf{Y}) \leftarrow \log(\mathbf{P}(\mathbf{Y})) \quad (2.124)$$

$$v(\mathbf{X}, \mathbf{Y}) \leftarrow \text{vec}(\mathbf{X}) \circ \text{vec}(\mathbf{R}(\mathbf{Y})) \quad (2.125)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J - \nabla_{\mathbf{n}}^J \mathbf{R}(\mathbf{Y}) \quad (2.126)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \nabla_{\mathbf{n}}^J (\mathbf{P}(\mathbf{Y}) - \mathbf{X}) \quad (2.127)$$

The gradient $\nabla_{\mathbf{X}}^J$ is the same as in the *CrossEntropy* node. To derive the gradient $\nabla_{\mathbf{Y}}^J$ we note that

$$\frac{\partial v}{\partial y_{ij}} = \mathbf{p}_{ij}(\mathbf{Y}) - x_{ij} \quad (2.128)$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \frac{\partial J}{\partial v} \frac{\partial v}{\partial y_{ij}} = \frac{\partial J}{\partial v} (\mathbf{p}_{ij}(\mathbf{Y}) - x_{ij}). \quad (2.129)$$

- *ClassBasedCrossEntropyWithSoftmax*: same as *CrossEntropyWithSoftmax* except that label \mathbf{Y} is a 4 byte T dense matrix. The first row is the word id, the second row is the class id of this word id, the third row is the first word id in this class, and the last row is the first word id of the next class. The last two rows provide the left and right boundary of a slice that correspond to softmax of label given its observation \mathbf{X} . It also takes a weight matrix \mathbf{W} for computing with observation \mathbf{X} . It also takes outputs \mathbf{Q} from Times node that provides inputs for computing posterior probability given class.

$$\text{for } t = 1 : T \quad (2.130)$$

$$z_t \leftarrow Y(0, t) \quad (2.131)$$

$$\mathbf{Z}_t \leftarrow [0 \cdots \delta(i = z_t) \cdots 0]^T \quad (2.132)$$

$$c_t \leftarrow Y(1, t) \quad (2.133)$$

$$\mathbf{C}_t \leftarrow [0 \cdots \delta(i = c_t) \cdots 0]^T \quad (2.134)$$

$$l_t \leftarrow Y(2, t) \quad (2.135)$$

$$r_t \leftarrow Y(3, t) \quad (2.136)$$

$$\mathbf{W}_t \leftarrow W[:, l_t : r_t - 1] \quad (2.137)$$

$$\mathbf{P}_t \leftarrow \text{Softmax}(\mathbf{W}_t^T \mathbf{X}) \quad (2.138)$$

$$\mathbf{R}_t \leftarrow \log(\mathbf{P}_t) \quad (2.139)$$

$$v(\mathbf{X}, \mathbf{Y}) \quad + = \quad \mathbf{Z}_t \circ \mathbf{R}_t \quad (2.140)$$

$$\mathbf{U}_t \leftarrow \log \text{Softmax}(\mathbf{Q}_t) \quad (2.141)$$

$$v(\mathbf{X}, \mathbf{Y}) \quad + = \quad \mathbf{C}_t \circ \mathbf{U}_t \quad (2.142)$$

$$\nabla_{\mathbf{Q}}^J \leftarrow \nabla_{\mathbf{Q}}^J + \nabla_{\mathbf{n}}^J(\exp(\mathbf{U}_t) - \mathbf{C}_t) \quad (2.143)$$

$$\nabla_{\mathbf{P}}^J \leftarrow \nabla_{\mathbf{n}}^J(\mathbf{P}_t - \mathbf{Z}_t) \quad (2.144)$$

$$\nabla_{\mathbf{W}}^J \leftarrow \nabla_{\mathbf{W}}^J + \mathbf{X}_t \nabla_{\mathbf{P}}^{JT} \quad (2.145)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \mathbf{W}_t \nabla_{\mathbf{P}}^J \quad (2.146)$$

•

2.4.4 Computation Node Types for Computing Statistics

Sometimes we only want to get some statistics of the input values (either input features or labels). For example, to normalize the input features we need to compute the mean and standard deviation of the input feature. In speech recognition we need to compute the frequencies (mean) of the state labels to convert state posterior probability to the scaled likelihood. Unlike other computation node types we just described, computation node types for computing statistics do not require a gradient computation function (i.e., this function should not be called for these types of nodes) because they are not learned and often need to be precomputed before model training starts. Here we list the most popular computation node types in this category.

- *Mean*: compute the mean of the operand \mathbf{X} across the whole training set. When the computation is finished, it needs to be marked so to avoid recomputation. When a minibatch of input \mathbf{X} is fed in

$$k \leftarrow k + \mathbf{X}.cols \quad (2.147)$$

$$\mathbf{v}(\mathbf{X}) \leftarrow \frac{1}{k} \mathbf{X} \mathbf{1}_{\mathbf{X}.cols,1} + \frac{k - \mathbf{X}.cols}{k} \mathbf{v}(\mathbf{X}) \quad (2.148)$$

Note here $\mathbf{X}.cols$ is the number of samples in the minibatch.

- *InvStdDev*: compute the invert standard deviation of the operand \mathbf{X} element-wise across the whole training set. When the computation is finished, it needs to be marked so to avoid recomputation. In the accumulation step

$$k \leftarrow k + \mathbf{X}.cols \quad (2.149)$$

$$\mathbf{v}(\mathbf{X}) \leftarrow \frac{1}{k} \mathbf{X} \mathbf{1}_{\mathbf{X}.cols,1} + \frac{k - \mathbf{X}.cols}{k} \mathbf{v}(\mathbf{X}) \quad (2.150)$$

$$\omega(\mathbf{X}) \leftarrow \frac{1}{k} (\mathbf{X} \bullet \mathbf{X}) \mathbf{1} + \frac{k - \mathbf{X}.cols}{k} \omega(\mathbf{X}) \quad (2.151)$$

When the end of the training set is reached,

$$\mathbf{v} \leftarrow (\omega - (\mathbf{v} \bullet \mathbf{v}))^{1/2} \quad (2.152)$$

$$\mathbf{v} \leftarrow \mathbf{1} \oslash \mathbf{v}. \quad (2.153)$$

- *PerDimMeanVarNorm*: compute the normalized operand \mathbf{X} using mean \mathbf{m} and invert standard deviation \mathbf{s} for each sample. Here \mathbf{X} is matrix whose number of columns equals to the number of samples in the minibatch and \mathbf{m} and \mathbf{s} are vectors that needs to be expanded before element-wise product is applied.

$$\mathbf{V}(\mathbf{X}) \leftarrow (\mathbf{X} - \mathbf{m}) \bullet \mathbf{s}. \quad (2.154)$$

2.5 Convolutional Neural Network

A Convolutional neural network (CNN) [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 29] provides shift invariance over time and space and is critical to achieve state-of-the-art performance on image recognition. It has also been shown to improve speech recognition accuracy over pure DNNs on some tasks [12, 16, 17, 14, 15]. To support CNN we need to implement several new computation nodes.

- *Convolution*: convolve element-wise products of a kernel to an image. An example of a convolution operation is shown in Figure 2.7, where the input to

the convolution node has three channels (represented by three 3×3 matrices) and the output has two channels (represented by two 2×2 matrices at top). A channel is a view of the same image. For example, an RGB image can be represented with three channels: R, G, B. Each channel is of the same size.

There is a kernel for each output and input channel pair. The total number of kernels equals to the product of the number of input channels C_x and the number of output channels C_v . In Figure 2.7 $C_x = 3$, $C_v = 2$ and the total number of kernels is 6. Each kernel $\mathbf{K}_{k\ell}$ of input channel k and output channel ℓ is a matrix. The kernel moves along (and thus shared across) the input with strides (or subsampling rate) S_r and S_c at the vertical (row) and horizontal (column) direction, respectively. For each output channel ℓ and input slice (i, j) (the i -th step along the vertical direction and j -th step along the horizontal direction)

$$v_{lij}(\mathbf{K}, \mathbf{Y}) = \sum_k \text{vec}(\mathbf{K}_{k\ell}) \circ \text{vec}(\mathbf{Y}_{kij}), \quad (2.155)$$

where \mathbf{Y}_{kij} has the same size as $\mathbf{K}_{k\ell}$.

This evaluation function involves many small matrix operations and can be slow. Chellapilla et al. [9] proposed a technique to convert all these small matrix operations to a large matrix product as shown at the bottom of Figure 2.7. With this trick all the kernel parameters are combined into a big kernel matrix \mathbf{W} as shown at left bottom of Figure 2.7. Note that to allow for the output of the convolution node to be used by another convolution node, in Figure 2.7 we have organized the conversion slightly differently from what proposed by Chellapilla et al. [9] by transposing both the kernel matrix and the input features matrix as well as the order of the matrices in the product. By doing so, each sample in the output can be represented by $O_r \times O_c$ columns of $C_v \times 1$ vectors which can be reshaped to become a single column, where

$$O_r = \begin{cases} \frac{I_r - K_r}{S_r} + 1 & \text{no padding} \\ \frac{(I_r - \text{mod}(K_r, 2))}{S_r} + 1 & \text{zero padding} \end{cases} \quad (2.156)$$

is the number of rows in the output image, and

$$O_c = \begin{cases} \frac{I_c - K_c}{S_c} + 1 & \text{no padding} \\ \frac{(I_c - \text{mod}(K_c, 2))}{S_c} + 1 & \text{zero padding} \end{cases} \quad (2.157)$$

is the number of rows in the output image, where I_r and I_c are, respectively, the number of rows and columns of the input image, K_r and K_c are, respectively, the number of rows and columns in each kernel. The combined kernel matrix

is of size $C_v \times (O_r \times O_c \times C_x)$, and the packed input feature matrix is of size $(O_r \times O_c \times C_x) \times (K_r \times K_c)$.

With this conversion the related computations of the convolution node with operands \mathbf{W} , \mathbf{Y} become

$$\mathbf{X}(\mathbf{Y}) \leftarrow \text{Pack}(\mathbf{Y}) \quad (2.158)$$

$$\mathbf{V}(\mathbf{W}, \mathbf{Y}) \leftarrow \mathbf{W}\mathbf{X} \quad (2.159)$$

$$\nabla_{\mathbf{W}}^J \leftarrow \nabla_W^J + \nabla_{\mathbf{n}}^J \mathbf{X}^T \quad (2.160)$$

$$\nabla_{\mathbf{X}}^J \leftarrow \mathbf{W}^T \nabla_{\mathbf{n}}^J \quad (2.161)$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_Y^J + \text{Unpack}(\nabla_{\mathbf{X}}^J). \quad (2.162)$$

Note that this technique enables better parallelization with large matrix operations but introduces additional cost to pack and unpack the matrices. In most conditions, the gain outweighs the cost. By composing convolution node with plus nodes and element-wise nonlinear functions we can add bias and nonlinearity to the convolution operations.

- *MaxPooling*: apply the maximum pooling operation to input \mathbf{X} inside a window with size $K_r \times K_c$ for each channel. The operation window moves along the input with strides (or subsampling rate) S_r and S_c at the vertical (row) and horizontal (column) direction, respectively. The pooling operation does not change the number of channels and so $C_v = C_x$. For each output channel ℓ and the (i, j) -th input slice $\mathbf{X}_{\ell ij}$ of size $K_r \times K_c$ we have

$$v_{\ell ij}(\mathbf{X}) \leftarrow \max(\mathbf{X}_{\ell ij}) \quad (2.163)$$

$$[\nabla_{\mathbf{X}}^J]_{\ell, i_m, j_n} \leftarrow \begin{cases} [\nabla_{\mathbf{X}}^J]_{\ell, i_m, j_n} + [\nabla_{\mathbf{n}}^J]_{\ell, i_m, j_n} & (m, n) = \arg \max_{m, n} x_{\ell, i_m, j_n} \\ [\nabla_{\mathbf{X}}^J]_{\ell, i_m, j_n} & \text{else} \end{cases} \quad (2.164)$$

where $i_m = i \times S_r + m$, and $j_n = j \times S_c + n$.

- *AveragePooling*: same as *MaxPooling* except average instead of maximum is applied to input \mathbf{X} inside a window with size $K_r \times K_c$ for each channel. The operation window moves along the input with strides (or subsampling rate) S_r and S_c at the vertical (row) and horizontal (column) direction, respectively. For each output channel ℓ and the (i, j) -th input slice $\mathbf{X}_{\ell ij}$ of size $K_r \times K_c$ we have

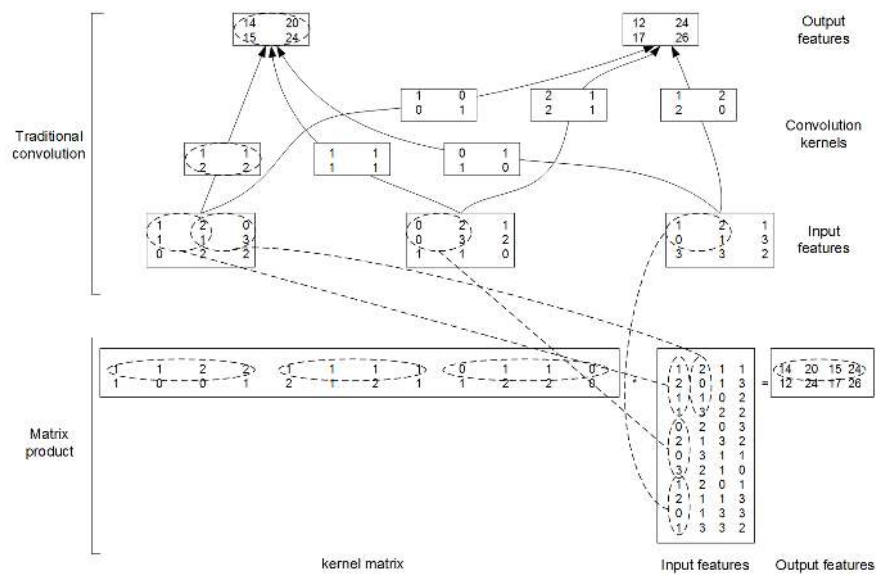


Figure 2.7: Example convolution operations. Top: the original convolution operations. Bottom: the same operations represented as a large matrix product. Our matrix organization is different from what proposed by Chellapilla et al. [9] to allow for stacked convolution operations. (Modified from the figure in Chellapilla et al. [9], permitted to use by Simard)

$$v_{lij}(\mathbf{X}) \leftarrow \frac{1}{K_r \times K_c} \sum_{m,n} x_{\ell, i_m, j_n} \quad (2.165)$$

$$[\nabla_{\mathbf{X}}^J]_{lij} \leftarrow [\nabla_{\mathbf{X}}^J]_{lij} + \frac{1}{K_r \times K_c} [\nabla_{\mathbf{n}}^J]_{lij}, \quad (2.166)$$

where $i_m = i \times S_r + m$, and $j_n = j \times S_c + n$.

2.6 Recurrent Connections

In the above sections, we assumed that the CN is a DAG. However, when there are recurrent connections in the CN, this assumption is no longer true. The recurrent connection can be implemented using a *Delay* node that retrieves the value λ samples to the past where each column of \mathbf{Y} is a separate sample stored in the ascending order of time.

$$v_{.j}(\lambda, \mathbf{Y}) \leftarrow \mathbf{Y}_{.(j-\lambda)} \quad (2.167)$$

$$[\nabla_{\mathbf{Y}}^J]_{.j} \leftarrow [\nabla_{\mathbf{Y}}^J]_{.j} + [\nabla_{\mathbf{n}}^J]_{.j+\lambda}. \quad (2.168)$$

When $j - \lambda < 0$ some default values need to be set for $\mathbf{Y}_{.(j-\lambda)}$. The gradient can be derived by observing

$$\frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} 1 & m = i \wedge n = j + \lambda \\ 0 & \text{else} \end{cases} \quad (2.169)$$

and

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \frac{\partial J}{\partial v_{ij+\lambda}}. \quad (2.170)$$

An example CN that contains a delay node is shown in Figure 2.8. Different from the CN without a directed loop, a CN with a loop cannot be computed for a sequence of samples as a batch since the next sample's value depends on the previous samples. A simple way to do forward computation and backpropagation in a recurrent network is to unroll all samples in the sequence over time. Once unrolled, the graph is expanded into a DAG and the forward computation and gradient calculation algorithms we just discussed can be directly used. This means, however, all computation nodes in the CN need to be computed sample by sample and this significantly reduces the potential of parallelization.

There are two approaches to speedup the computation of a CN with directed loops. In the next two subsections, we will discuss them.

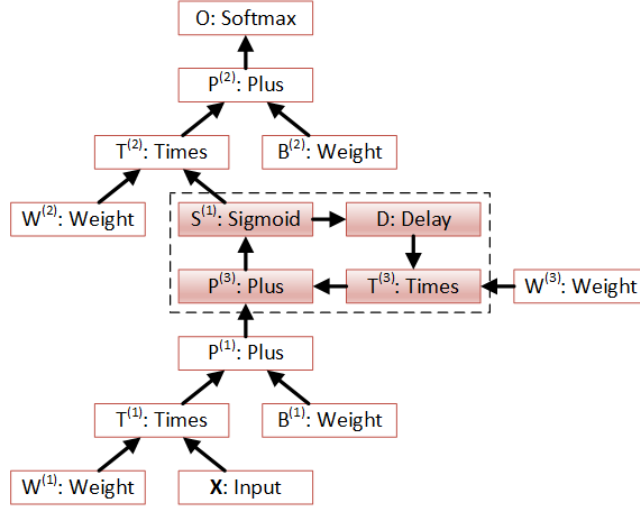


Figure 2.8: An example CN with a delay node. The shaded nodes form a recurrent loop which can be treated as a composite node.

2.6.1 Sample by Sample Processing Only Within Loops

The first approach identifies the loops in the CN and only applies the sample-by-sample computation for nodes inside the loops. For the rest of the computation nodes all samples in the sequence can be computed in parallel as a single matrix operation. For example, in Figure 2.8 all the nodes included in the loop of $T^{(3)} \rightarrow P^{(3)} \rightarrow S^{(1)} \rightarrow D \rightarrow T^{(3)}$ need to be computed sample by sample. All the rest of the nodes can be computed in batches. A popular technique is to identify the strongly connected components (SCC) in the graph, in which there is a path between each pair of vertices and adding any edges or vertices would violate this property, using algorithms such as Tarjan's strongly connected components algorithm [30]¹. Once the loops are identified, they can be treated as a composite node in the CN and the CN is reduced to a DAG. All the nodes inside each loop (or composite node) can be unrolled over time and also reduced to a DAG. For all these DAGs the forward computation and backpropagation algorithms we discussed in the previous sections can be applied. The detailed procedure in determining the forward computation order in the CN with arbitrary recurrent connections is described in Algorithm 2.6. Since the input to the delay nodes are computed in the

¹Tarjan's algorithm is favored over others such as Kosaraju's algorithm [31] since it only requires one depth-first traverse, has a complexity of $O(|V| + |E|)$, and does not require reversing arcs in the graph.

past they can be considered as the leaf if we only consider one time slice. This makes the order decision inside loops much easier.

2.6.2 Processing Multiple Utterances Simultaneously

The second approach to speed up the processing in the recurrent CN is to process multiple sequences at a time. To implement this, we need to organize sequences in a way that the frames with the same frame id from different sequences are grouped together as shown in Figure 2.9. By organizing sequences in this way we can compute frames from different sequences in batch when inside a loop and compute all samples in all utterances in one batch outside loops. For example, in Figure 2.9 we can compute 4 frames together for each time step. If sequences have different lengths, we can truncate them to the same length and save the final state of the sequences that are not finished yet. The remaining frames can be grouped with other sequences for further processing.

2.6.3 Building Arbitrary Recurrent Neural Networks

With the inclusion of delay nodes, we can easily construct complicated recurrent networks and dynamic systems. For example, the long-short-term-memory (LSTM) [18, 32] neural network that is widely used to recognize and generate hand-written characters involves the following operations:

$$\mathbf{i}_t = \sigma \left(\mathbf{W}^{(xi)} \mathbf{x}_t + \mathbf{W}^{(hi)} \mathbf{h}_{t-1} + \mathbf{W}^{(ci)} \mathbf{c}_{t-1} + \mathbf{b}^{(i)} \right) \quad (2.171)$$

$$\mathbf{f}_t = \sigma \left(\mathbf{W}^{(xf)} \mathbf{x}_t + \mathbf{W}^{(hf)} \mathbf{h}_{t-1} + \mathbf{W}^{(cf)} \mathbf{c}_{t-1} + \mathbf{b}^{(f)} \right) \quad (2.172)$$

$$\mathbf{c}_t = \mathbf{f}_t \bullet \mathbf{c}_{t-1} + \mathbf{i}_t \bullet \tanh \left(\mathbf{W}^{(xc)} \mathbf{x}_t + \mathbf{W}^{(hc)} \mathbf{h}_{t-1} + \mathbf{b}^{(c)} \right) \quad (2.173)$$

$$\mathbf{o}_t = \sigma \left(\mathbf{W}^{(xo)} \mathbf{x}_t + \mathbf{W}^{(ho)} \mathbf{h}_{t-1} + \mathbf{W}^{(co)} \mathbf{c}_t + \mathbf{b}^{(o)} \right) \quad (2.174)$$

$$\mathbf{h}_t = \mathbf{o}_t \bullet \tanh(\mathbf{c}_t), \quad (2.175)$$

where $\sigma(\cdot)$ is the logistic sigmoid function, \mathbf{i}_t , \mathbf{f}_t , \mathbf{o}_t , \mathbf{c}_t and \mathbf{h}_t are vectors with same size (since there is one and only one of each in every cell) to represent values at time t of the input gate, forget gate, output gate, cell, cell input activation, and hidden layer, respectively, \mathbf{W} 's are the weight matrices connecting different gates, and \mathbf{b} 's are the corresponding bias vectors. All the weight matrices are full except the weight matrix $\mathbf{W}^{(ci)}$ from the cell to gate vectors which is diagonal. It is obvious that the whole LSTM can be described as a CN with following node

Algorithm 2.6 Forward computation of an arbitrary CN

```

1: procedure DECIDEFORWARDCOMPUTATIONORDERWITHRECCURENT-
   LOOP( $G = (V, E)$ )
2:   StronglyConnectedComponentsDetection( $G, G'$ )  $\triangleright G'$  is a DAG of
   strongly connected components (SCC)
3:   Call DecideForwardComputationOrder on  $G' \rightarrow order$  for DAG
4:   for  $v \in G, v \in V$  do
5:     Set the order of  $v$  equal the max order of the SCC  $V \triangleright$  This guarantee
     the forward order of SCC is correct
6:   end for
7:   for each SCC  $V$  in  $G$  do
8:     Call GetLoopForwardOrder(root of  $V$ )  $\rightarrow order$  for each SCC
9:   end for
10:  return order for DAG and order for each SCC (loop)
11: end procedure
12: procedure GETLOOPFORWARDORDER(root)
13:  Treat all the delayNode as leaf and Call DecideForwardComponen-
   tionOrder
14: end procedure
15: procedure STRONGLYCONNECTEDCOMPONENTSDETECTION( $G = (V, E), DAG$ )
16:   $index = 0, S = empty$ 
17:  for  $v \in V$  do
18:    if  $v.index$  is undefined then StrongConnectComponents( $v, DAG$ )
19:    end if
20:  end for
21: end procedure
22: procedure STRONGCONNECTCOMPONENT( $v, DAG$ )
23:   $v.index = index, v.lowlink = index, index = index + 1$ 
24:   $S.push(v)$ 
25:  for  $(v, w) \in E$  do
26:    if  $w.index$  is undefined then
27:      StrongConnectComponent( $w$ )
28:       $v.lowlink = \min(v.lowlink, w.lowlink)$ 
29:    else if  $w \in S$  then
30:       $v.lowlink = \min(v.lowlink, w.index)$ 
31:    end if
32:  end for
33:  if  $v.lowlink = v.index$  then  $\triangleright$  If  $v$  is a root node, pop the stack and
   generate an SCC
34:    start a new strongly connected component
35:    repeat
36:       $w = S.pop()$ 
37:      add  $w$  to current strongly connected component
38:    until  $w == v$ 
39:    Save current strongly connected component to  $DAG$ 
40:  end if
41: end procedure

```

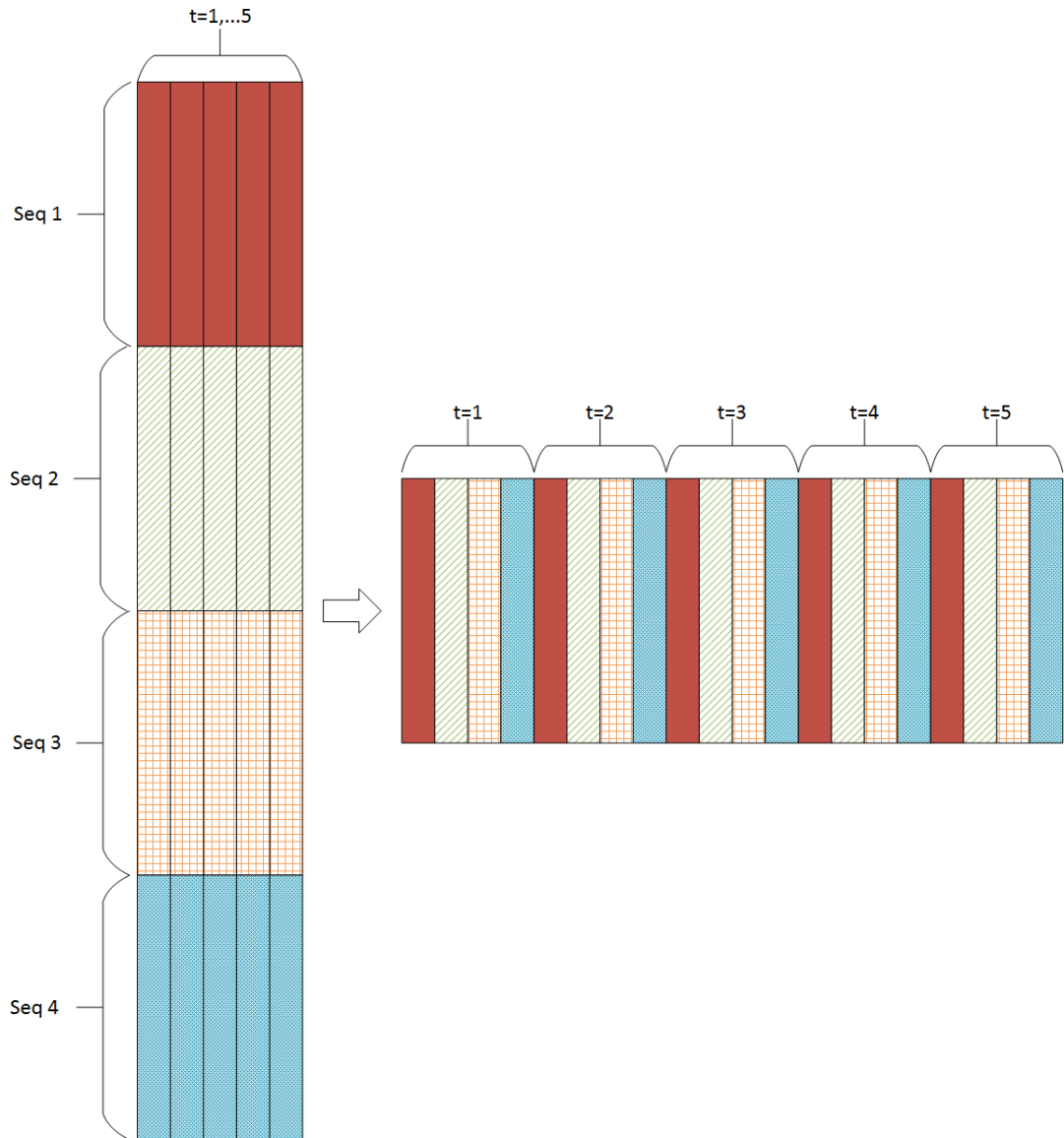


Figure 2.9: Process multiple sequences in a batch. Shown in the figure is an example with 4 sequences. Each color represents one sequence. Frames with the same frame id from different sequences are grouped together and computed in batch. The right matrix is a reshape of the left matrix.

types: *Times*, *Plus*, *Sigmoid*, *Tanh*, *DiagTimes*, *ElementTimes* and *Delay*. More specifically, using these computational nodes, the LSTM can be described as:

$$\mathbf{H}^{(d)} = \text{Delay}(\mathbf{H}) \quad (2.176)$$

$$\mathbf{C}^{(d)} = \text{Delay}(\mathbf{C}) \quad (2.177)$$

$$\mathbf{T}^{(1)} = \text{Macro2W1b}(\mathbf{W}^{(xi)}, \mathbf{X}, \mathbf{W}^{(hi)}, \mathbf{H}^{(d)}, \mathbf{b}^{(i)}) \quad (2.178)$$

$$\mathbf{I} = \text{Sigmoid}\left(\text{Plus}\left(\mathbf{T}^{(1)}, \text{DiagTimes}\left(\mathbf{d}^{(ci)}, \mathbf{C}^{(d)}\right)\right)\right) \quad (2.179)$$

$$\mathbf{T}^{(2)} = \text{Macro3W1b}\left(\mathbf{W}^{(xf)}, \mathbf{X}, \mathbf{W}^{(hf)}, \mathbf{H}^{(d)}, \mathbf{W}^{(cf)}\mathbf{C}^{(d)}, \mathbf{b}^{(f)}\right) \quad (2.180)$$

$$\mathbf{F} = \text{Sigmoid}\left(\mathbf{T}^{(2)}\right) \quad (2.181)$$

$$\mathbf{T}^{(3)} = \text{Macro2W1b}\left(\mathbf{W}^{(xc)}, \mathbf{X}, \mathbf{W}^{(hc)}, \mathbf{H}^{(d)}, \mathbf{b}^{(c)}\right) \quad (2.182)$$

$$\mathbf{T}^{(4)} = \text{ElementTime}\left(\mathbf{F}, \mathbf{C}^{(d)}\right) \quad (2.183)$$

$$\mathbf{T}^{(5)} = \text{ElementTimes}\left(\mathbf{I}, \text{Tanh}\left(\mathbf{T}^{(3)}\right)\right) \quad (2.184)$$

$$\mathbf{C} = \text{Plus}\left(\mathbf{T}^{(4)}, \mathbf{T}^{(5)}\right) \quad (2.185)$$

$$\mathbf{T}^{(6)} = \text{Macro3W1b}\left(\mathbf{W}^{(xo)}, \mathbf{X}, \mathbf{W}^{(ho)}, \mathbf{H}^{(d)}, \mathbf{W}^{(co)}\mathbf{C}, \mathbf{b}^{(o)}\right) \quad (2.186)$$

$$\mathbf{O} = \text{Sigmoid}\left(\mathbf{T}^{(6)}\right) \quad (2.187)$$

$$\mathbf{H} = \text{ElementTime}\left(\mathbf{O}, \text{Tanh}(\mathbf{C})\right), \quad (2.188)$$

where we have defined the macro $\text{Macro2W1b}(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}, \mathbf{W}^{(2)}, \mathbf{I}^{(2)}, \mathbf{b})$ as

$$\mathbf{S}^{(1)} = \text{Plus}\left(\text{Times}\left(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}\right), \text{Times}\left(\mathbf{W}^{(2)}, \mathbf{I}^{(2)}\right)\right) \quad (2.189)$$

$$\text{Macro2W1b} = \text{Plus}\left(\mathbf{S}^{(1)}, \mathbf{b}\right) \quad (2.190)$$

and macro $\text{Macro3W1b}(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}, \mathbf{W}^{(2)}, \mathbf{I}^{(2)}, \mathbf{W}^{(3)}, \mathbf{I}^{(3)}, \mathbf{b})$ as

$$\mathbf{S}^{(1)} = \text{Macro2W1b}\left(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}, \mathbf{W}^{(2)}, \mathbf{I}^{(2)}, \mathbf{b}\right) \quad (2.191)$$

$$\text{Macro3W1b} = \text{Plus}\left(\mathbf{S}^{(1)}, \text{Times}\left(\mathbf{W}^{(3)}, \mathbf{I}^{(3)}\right)\right) \quad (2.192)$$

Chapter 3

Computational Network Toolkit

The computational network toolkit (CNTK) implements CNs. It is written in C++ and supports both GPU (CUDA) and CPU execution. CNTK supports arbitrary valid computational networks and makes building DNNs, CNNs, RNNs, LSTMs, and other complicated networks as simple as describing the operations of the networks. The toolkit is implemented with efficiency in mind. It removes duplicated computations in both forward and backward passes, uses minimal memory needed and reduces memory reallocation by reusing them. It also speeds up the model training and evaluation by doing batch computation whenever possible. In this chapter we introduce how to exploit CNTK to accelerate your research.

3.1 Run CNTK

To run CNTK you use a command line similar to

```
cntk configFile=yourExp.cntk
```

where yourExp.cntk is a CNTK configuration file, which typically contains several command blocks. A command block is a top level block of the configuration. Each command block must specify what action to be carried out with related information. To illustrate configuration and command blocks, we use a simple example below.

```
command=mnistTrain

mnistTrain=[
    action="train"

    NDINetworkBuilder=[
```

```

    networkDescription="c:\cntk\config\sample.ndl"
    run=ndlMacroUse
]

SGD=[
    modelPath="c:\cntk\model\mnist.cn"
    learningRatesPerMB=0.001
    minibatchSize=32
    epochSize=60000
    maxEpochs=50
]

reader=[
    readerType="UCIFastReader"
    file="c:\cntk\data\mnist\mnist_train.txt"

    features=[
        dim=784
        start=1
    ]

    labels=[
        dim=1
        start=0
        labelDim=10
        labelMappingFile="c:\cntk\data\mnist\mnistlabels.txt"
        "
    ]
]
]

```

In this example, you can notice that all configuration values are specified as a name-value pair. A value can be a numeric, a string, a list, or even a block of configurations.

The top-level configuration parameter *command* determines what command blocks are to be executed and in what order they are executed if more than one command block is specified. In this example, the `mnistTrain` command block will be executed. This command block specifies the action, which is *train* in this case, to execute. There are often three parameter blocks associated with the *train* action: a network builder block, which specifies how to build a network from scratch and how to load a model from an existing model file, a learner block, which specifies what training algorithm to use, and a reader block, which specifies where and how to load features and labels. In this specific example, the NDL (network definition

language) network builder indicated by the `NDLNetworkBuilder` block is used to define the network, the stochastic gradient descent learning algorithm as indicated by the `SGD` block is used to train the model, and the `UCIFastReader` is used to load the feature and labels from file in UCI format. Note that readers are implemented as separate DLLs, and the name of the reader is also the name of the DLL file that will be loaded to read data.

The most frequently used configuration blocks are:

- Network Builders
 - `SimpletNetworkBuilder`: creates one of the predefined networks with limited customization.
 - `NDLNetworkBuilder`: creates a network defined using the network description language (NDL). It provides full flexibility in designing your own network operations and structure.
- Learners
 - `SGD` : uses the stochastic gradient descent algorithm to train the model. It is the desired trainer for most applications.
- Data Readers
 - `UCIFastReader`: reads the text-based UCI format, which contains labels and features combined in one file.
 - `HTKMLFReader`: reads the HTK/MLF format files, often used in speech recognition applications.
 - `BinaryReader`: reads files in a CNTK binary format. It is also used by `UCIFastReader` to cache the dataset in a binary format for faster processing.
 - `LMSequenceReader`: reads text-based files that contain word sequences, for predicting word sequences. This is often used in language modeling.
 - `LUSequenceReader`: reads text-based files that contain word sequences and their labels. This is often used for language understanding.

In the following, we will describe CNTK configuration and the above blocks in detail.

3.2 Network Builders

Computational networks are constructed using either the network description language (NDL) builder or the simple network builder.

3.2.1 The Simple Network Builder

The behavior of the simple network builder is controlled by the SimpleNetworkBuilder block of the options. When an option is omitted the default value is assumed. Here we list all the control parameters available.

- *initValueScale*: the value for scaling the range of the random numbers used for initialization. Default is 1. If the model parameters are initialized using the uniform distribution, the random number range will be adjusted to $[-0.05 \times \text{initValueScale}, 0.05 \times \text{initValueScale}]$. If the model parameters are initialized using the Gaussian distribution, the standard deviation will be adjusted to $0.2 \times \text{initValueScale} / \sqrt{\text{fanout}}$.
- *layerTypes*: the type of nonlinear operation in hidden layers. Valid values are sigmoid (default), Tanh, and RectifiedLinear.
- *uniformInit*: determines whether to use uniform distribution to initialize model parameters. Valid values are true (default) and false (using Gaussian distribution to initialize model parameters).
- *applyMeanVarNorm*: whether to apply mean/variance normalization on the input. Valid values are true and false (default)
- *addDropoutNodes*: whether to add drop-out nodes. The default is false. If specified to true, a drop-out node will be applied to the input node and the output of every hidden layer.
- *layerSizes*: specifies the dimensions of layers. For instance, `layerSizes=128:10:200:4000` describes a neural network with two hidden layers. The first hidden layer has a dimension of 10, and the second hidden layer has a dimension of 200. The input and output layers have a dimension of 128 and 4000, respectively.
- *trainingCriterion*: the criterion used for training. The default is CrossEntropyWithSoftmax. Alternatives are SquareError, CrossEntropy, and ClassBasedCrossEntropyWithSoftmax. The ClassBasedCrossEntropyWithSoftmax is for class-based training, which would be useful if the output dimension is large and therefore need to be split into classes to speed-up the training and evaluation.

- *evalCriterion*: the criterion for evaluation. The selection of values are the same as the *trainingCriterion*.
- *lookupTableOrder*: specifies the order of context expanding in the lookupNode. The default value is 1. Setting it to a value such as 3 would expand the input dimension in a context-dependent way by an order of 3. For example, if the input observation has a dimension of 20, setting this value to 3 would set the input node dimension to 60.

For recurrent neural networks (RNNs), there are additional parameters.

- *recurrentLayer*: specifies the layers that contain self recurrent connections. By default there is no recurrent layer. Use the syntax *n1:n2:n3* to specify that layers *n1*, *n2*, and *n3* have recurrent connections.
- *defaultHiddenActivity*: the default hidden layer activity value used by the delay node when accessing values before the first observation. The default value is 0.1.
- *rnnType*: the type of predefined networks. Valid values are
 - SIMPNET: the feed-forward neural network. This is the default network type.
 - SIMPLERNN: the simple RNN, which may be a deep RNN in which several layers have recurrent loops.
 - CLASSLM: the class-based simple RNN. It uses sparse input, sparse parameter and sparse output. This is often used for language modeling tasks.
 - LBLM: the log-bilinear neural network.
 - LSTM: the long short-term memory neural network.
 - CLASSLSTM: the class-based long short-term memory neural network. It uses sparse input, sparse parameter and sparse output. This is often used for language modeling tasks.

3.2.2 The Network Description Language (NDL)

The simple network builder only allows you to build one of the predefined network types. To build more complicated networks, you need to use the NDL network builder which has the following parameters. Detailed description on NDL can be found in Chapter 4.

- *networkDescription*: the file path of the NDL script, which will be described in Chapter 4, to execute. If there is no *networkDescription* file specified then the NDL is assumed to be in the same configuration file as the *NDLNetworkBuilder* subblock, specified with the *run* parameter. Note that only one file path may be specified via the *networkDescription* parameter. To load multiple files of macros, use the *ndlMacros* parameter.
- *run*: the block of the NDL that will be executed. If an external NDL file is specified via the *networkDescription* parameter, the *run* parameter identifies a block in that file. This parameter overrides any *run* parameters that may already exist in the file. If no *networkDescription* file is specified, the *run* parameter identifies a block in the current configuration file.
- *load*: the blocks of NDL scripts to load. Multiple blocks can be specified via a ":" separated list. The blocks specified by the *load* parameter typically contain macros, which will be described in Chapter 4, for use by the *run* block. Similar to the *run* parameter, the *load* parameter identifies blocks in an external NDL file and overrides any *load* parameters that may already exist in the file, if a file is specified by the *networkDescription* parameter. If no *networkDescription* file is specified, *load* identifies a block in the current configuration file.
- *ndlMacros*: the file path where NDL macros may be loaded. This parameter is usually used to load a default set of NDL macros that can be used by all NDL scripts. Multiple NDL files, each specifying different sets of macros, can be loaded by specifying a "+" separated list of file paths for this *ndlMacros* parameter. In order to share macros with other command blocks such as model editing language (MEL) blocks we will describe in Chapter 4, you should define it at the root level of the configuration file.
- *randomSeedOffset*: a non-negative random seed offset value in initializing the learnable parameters. The default value is 0. This allows users to run experiments with different random initialization.

3.3 Learners

Up to now, CNTK implements an SGD learner. New learners will be added in the future.

3.3.1 Stochastic Gradient Descent Learner

The behavior of the SGD algorithm is controlled by the SGD block of the options. When an option is omitted the default value is assumed. Here we list all the control parameters used.

3.3.1.1 Training process control

- *modelPath*: the full path to save the final model. Must be provided and points to a valid file name.
- *trainCriterionNodeName*: the name of the training criterion node. If not provided the default training criterion node in the network will be used.
- *evalCriterionNodeName*: the name of the evaluation criterion node. If not provided the default evaluation criterion node in the network will be used.
- *epochSize*: epoch size, i.e., the number of samples in each epoch. Often is but may be different from the dataset size. An intermediate model and other check point information is saved for each epoch. When set to 0 the whole dataset size is used.
- *keepCheckPointFiles*: whether you want to keep the check point file after a new epoch starts. Valid values are true and false (default).
- *maxEpochs*: maximum number of epochs to run.
- *minibatchSize*: minibatch size for each epoch. Default value is 256. Can use syntax such as 128*2:1024 which means using minibatch size 128 for 2 epochs and then 1024 for the rest.
- *dropoutRate*: dropout rate during the training procedure. Default is 0.0. Can use syntax such as 0.5*10:0.2 which means using dropout rate 0.5 for 10 epochs and then 0.2 for the rest.
- *maxTempMemSizeInSamplesForCNN*: maximum temporary memory used (in number of samples) when packaging and unpackaging input features. Default is 0, which means using any value as needed. Useful to control the memory foot print esp. when run under GPU.
- *executionEngine*: the execution engine to use. Valid values is synchronous (default).

3.3.1.2 Learning rate and momentum control

- *learningRatesPerMB*: learning rates per minibatch. Useful when you want to use the same learning rate while the minibatch size is changed. Can use syntax such as `0.8*10:0.2` which means using the learning rate 0.8 for 10 epochs and then 0.2 for the rest. *learningRatesPerMB* may be missing, for example, when *learningRatesPerSample* is provided or the automatic learning rate determination algorithm is used.
- *learningRatesPerSample*: learning rates per sample. Useful when you want to keep the learning rates per sample constant, i.e., automatically increases effective learning rate for the minibatch when the minibatch size is increased. Can use syntax such as `0.008*10:0.002` which means using the learning rate 0.008 for 10 epochs and then 0.002 for the rest. *learningRatesPerSample* may be missing, for example, when *learningRatesPerMB* is provided or the automatic learning rate determination algorithm is used.
- *momentumPerMB*: momentum per minibatch. Default is 0.9. Can use syntax such as `0.1*2:0.9` which means using momentum 0.1 for 2 epochs and then 0.9 for the rest.
- *momentumPerSample*: momentum per sample. Useful when you want to keep the momentum per sample constant, i.e., automatically scales effective momentum for the minibatch when the minibatch size is changed. Can use syntax such as `0.9996*10:0.998` which means using the per sample momentum 0.9996 for 10 epochs and then 0.998 for the rest. *momentumPerSample* may be missing, for example, when *momentumPerMB* is provided.
- *autoAdjust*: contains the information related to the automatic learning rate control. Default value is empty (""), which means no automatic learning rate control. Inside the block, there can be following values:
 - *autoAdjustLR*: the automatic learning rate adjustment algorithm to use. Valid values are `None` (default, don't auto adjust learning rate), `AdjustAfterEpoch` (check the training criterion after each epoch using the development set of the training set and decide whether to adjust the learning rate), and `SearchBeforeEpoch` (search the learning rate based on a small portion of the training set before each epoch starts).

used in the `AdjustAfterEpoch` mode:

- *reduceLearnRateIfImproveLessThan*: reduce the learning rate if the improvement is less than this value. Default is 0.

- *learnRateDecreaseFactor*: the learning rate decrease factor. Default value is 0.618.
- *increaseLearnRateIfImproveMoreThan*: increase the learning rate if the improvement is larger than this value. Default value is 1#INF (infinity) which means never increase.
- *learnRateIncreaseFactor*: the learning rate increase factor. Default value is 1.382.
- *loadBestModel*: whether to load the best model if the current model decreases the performance. Valid values are true (default) and false.
- *learnRateAdjustInterval*: determine the frequency of applying the learning rate adjustment check. Default is 1 epoch. If this value is set to a value larger than 1 the learning rate adjustment will be based on the average criterion computed from the last learnRateAdjustInterval epochs.

Used in the SearchBeforeEpoch mode.

- *numMiniBatch4LRSearch*: the number of minibatches used to search the learning rate. Default value is 500. It's typically set to 10-20% of the total minibatches in an epoch.
- *numPrevLearnRate*: number of previous learning rates used as a hint to the search range. Default value is 5.
- *numBestSearchEpoch*: number of epochs in which we use the best learning rate instead of the sufficient learning rate. Default value is 1.

Used in the Adaptive Minibatch Sizing mode.

- *numMiniBatch4LRSearch*: the number of minibatches used to search the minibatch size when in adaptive minibatch size mode. Default value is 500. It's typically set to 10-20% of the total minibatches in an epoch. This is shared with the search for learning rate in SearchBeforeEpoch mode.
- *autoAdjustMinibatch*: enable or disable whether minibatch size is adaptively adjusted. Default value is false. Adaptive minibatch sizing will begin on epochs starting after user minibatch sizes explicitly specified are complete. For example if the userspecified minibatchSize=256:1024, then 256 and 1024 are used in the first 2 Epochs and adaptive minibatch-sizing is used afterwards

- *minibatchSizeTuningFrequency*: The number of epochs to skip, on a periodic basis, before dynamically adjusting the minibatch size. Default value is 1.
- *minibatchSizeTuningMax*: The maximum size allowed for an adaptively adjusted minibatch size. Default value is 1048576.

3.3.1.3 Gradient control

- *gradientClippingWithTruncation*: whether to use the truncation based gradient clipping to control gradient explosion. Valid values are true (default) and false. If it is false the norm based clipping will be used instead which is more expensive.
- *clippingThresholdPerSample*: the clipping threshold for each sample. Default value is 1#INF which means infinity (i.e., clipping is turned off).
- *L2RegWeight*: the L2 regularization weight. Default is 0.
- *L1RegWeight*: the L1 regularization weight. Default is 0.
- *gradUpdateType*: gradient update type. Valid values are None (default, no special treatment to the gradient), AdaGrad, and RmsProp. When *gradUpdateType* equals to AdaGrad or RmsProp, you can control the behavior of the gradient update using following parameters:
 - *normWithAveMultiplier*: normalize the gradient with the average multipliers applied to the gradients by the AdaGrad/RmsProp algorithm. Default is true (default).

When *gradUpdateType* equals to RmsProp, you can control the behavior of the gradient update using following parameters:

- *rms_wgt_inc*: multiplicative increment of the learning rate scale. Default is 1.2.
- *rms_wgt_dec*: multiplicative decrement of the learning rate scale. Default is 0.75.
- *rms_wgt_max*: maximum learning rate scale allowed. A value closer to 1 makes the learning rate adjustment more stable but slower. Default is 10.
- *rms_wgt_min*: minimum learning rate scale allowed. A value closer to 1 makes the learning rate adjustment more stable but slower. Default is 0.1.

- *rms_gamma*: smoothing factor used to estimate the moving average of the variance. The smaller the value, the quicker it forgets the past information. Default is 0.99.
- *gaussianNoiseInjectStd*: the standard deviation of the Gaussian noise added when using the AdaGrad approach. Default is 0.

3.3.1.4 Adaptation

Only KL divergence regularization is directly supported. Other adaptation techniques can be easily implemented by adding computation nodes to the network using the model editing language (MEL) that we will describe in Chapter 4.

- *adaptationRegType*: adaptation regularization type. Valid values are None (default) and KL (for KL divergence based regularization)
- *adaptationRegWeight*: adaptation regularization weight. Default value is 0.

3.3.1.5 Information display

- *traceLevel*: trace level to decide what information to print out in the stderr. Valid values are 0 (default) and 1.
- *numMBsToShowResult*: display training statistics after how many minibatches. Default is 10.

3.3.1.6 Gradient Check

- *gradientCheck*: determines whether to use the gradient checker. The default value is false. When using the gradient checker you need to use a minibatch size that is larger than the sequence length for RNNs due to the truncated back-propagation through time (BPTT) algorithm used to train RNNs, and a smaller learning rate to prevent numerical issues caused by divergence. In addition, precision should be set to double.

3.4 Data Readers

The reader block is used for all types of readers and the *readerType* parameter determines which reader to use. Each reader implements the same *IDataReader* interface which will be described in detail in Chapter ???. Many parameters in the

reader block are shared across different types of readers. Some are specific to a particular reader. In this block we will introduce the main data readers implemented in CNTK.

3.4.1 UCIFastReader

UCIFastReader reads text-based UCI format data, in which each data record is a line of space-delimited floating point feature and label values. The label information is either at the beginning or the end of each line, if label information is provided. To use the UCIFastReader you set the readerType to UCIFastReader as in

```
reader=[
  readerType="UCIFastReader"

  file="c:\cntk\data\mnist\mnist_train.txt"

  features=[
    dim=784
    start=1
  ]

  labels=[
    dim=1
    start=0
    labelDim=10
    labelMappingFile="c:\cntk\data\mnist\mnistlabels.txt"
  ]
]
```

In this example you can also notice two sub-blocks named features and labels. These names are used by the data readers to match the computation node in your network and the data loaded from the files. If simple network builders are used to create your network, features and labels are the standard names of the feature and label nodes, respectively. If you defined your network using the NDL network builder you need to make sure these names matches the corresponding nodes in your network. The UCIFastReader has following parameters:

- *file*: the file that contains the dataset. This parameter has been moved up from the features and labels sub-blocks, because UCIFastReader requires the file be the same, and moving up a level is an excellent way to make sure this restriction is met.

- *dim*: the dimension of the input value. Note that each column in the UCI data file represents one dimension of the input data.
- *start*: the start column (zero-based) of the input data.
- *labelDim*: the number of possible label values. This parameter is required for categorical labels since the dimension of the label node will be determined by this value. Note that the label value itself is typically specified in one column in the UCI data file.
- *labelMappingFile*: the path to a file used to map from the label value to a numerical label identifier. The file typically lists all possible label values, one per line, which might be text or numeric. The zero-based line number is the identifier that will be used by CNTK to identify that label. It's important that the same label mapping file is used for training and evaluation. This can be done by moving the *labelMappingFile* parameter up so that it can be shared by both the training and evaluation blocks.
- *customDelimiter* : the customized delimiter. By default, spaces are used as the delimiter. With this parameter you can choose to use another delimiter such as comma or semicolon in addition to spaces. Note though, this reader does not handle empty fields, e.g., two commas in a row, with or without whitespace in-between.
- *customDecimalPoint*: the customized decimal point. By default, dot is used as the decimal point. With this parameter you can choose to use another decimal point such as those used in European countries.
- *labelType* : to indicate how the label columns should be interpreted. By default it is set to "Category" which means it requires a mapping table to map from values (can be string) to class IDs so that values are converted to 1-hot representation. It can be set to "Regression" to indicate that the values are used directly without any conversion or "None" to indicate no labels are used.

3.4.2 HTKMLFReader

HTKMLFReader is a data reader that reads files typically associated with speech recognition tasks, and specifically with the HTK suite of tools. The reader can take as input two types of files, a list of feature files known in HTK parlance as an SCP file ("script" file) and a label file known as a MLF file ("model label file"). Because CNTK can be used to build networks with arbitrary topology including networks

with multiple inputs and outputs, the HTKMLFReader can process one or multiple SCP and MLF files. A typical example of usage for the HTKMLFReader is as follows:

```
reader=[
  readerType="HTKMLFReader"
  readMethod="rollingWindow"
  miniBatchMode="partial"
  randomize="auto"

  features=[
    dim=792
    scpFile="$ScpDir$\TIMIT.train.scp.fbank.fullpath"
  ]

  labels=[
    mlfFile="$MlfDir$\TIMIT.train.align_cistate.mlf.cntk"
    labelDim=183
    labelMappingFile="$MlfDir$\TIMIT.statelist"
  ]
]
```

The HTKMLFReader has the following configuration parameters:

- *readMethod*: the method used to read the features files into memory for processing during network training. The “rollingWindow” option reads in all feature files and stores them on disk in one large temporary binary file. The data is randomized by running a large rolling window over the data in this file and randomizing the data within the window. This method produces more thorough randomization of the data but requires a large temporary file written to disk. The other option is “blockRandomize” which divides the data into large blocks and then randomizes the order of the blocks and the order of the data within the block. The data in each block is read directly from the feature files. Because this does not require any temporary feature files, this option is more appropriate for large corpora. The default is blockRandomize. Note that the blockRandomize option requires an archive format of SCP file, described below. Also, as described below, when used in conjunction with “frameMode=false” the blockRandomize read method will randomize over utterances, but not frames. This is appropriate for use with recurrent architectures when preserving the sequential nature of the training examples is desired.
- *pageFilePath*: specify where the temporary page file of the features should be stored. By default it will use system provided temp file.

- *randomize*: Auto or None if readMethod is rollingWindow, Auto, None, or a specified block size if readMethod is blockRandomize. It determines how the reader should randomize the data. When a block size is specified, only that much of data will be in memory at one time.
- *minibatchMode*: Partial or Full, this option decides how to handle the last minibatch if there are not enough frames to form a complete minibatch of the requested size. The default is Partial, which will use the remaining frames in a smaller final minibatch of the training epoch. The Full option will only process complete minibatches.

The above example has two sources of data being processed by the reader, *features*, in the form of a list of HTK feature files, and *labels*, which are in the form of an HTK MLF file. Both features and labels correspond to nodes in the computational network, in this case, the input and output nodes, respectively. Note that “features” and “labels” are the default names used by the SimpleNetworkBuilder but if the network is designed using the Network Description Language (NDL), then any names can be used, as long as they each have a corresponding node in the network.

To specify continuous-valued features, e.g. MFCC’s or log mel filterbank coefficients, the following parameters should be included in the a configuration block:

- *scpFile*: a list of files to be processed. The files should be HTK compatible files and can be specified in standard mode or “archive” model. The details of using an archive are described below. configuration parameter.
- *dim*: an integer that specifies the full feature vector dimension with the desired context window. For example, if you had 72-dimensional features (24-dimensional filterbank features plus delta and delta-delta coefficients) and the network is designed to process a context window of 11 frames, the specified dimension should be 792. Note that only symmetric context windows are supported.

To specify the corresponding labels, e.g. phoneme or senone labels, a configuration block should be used that specifies the following parameters:

- *mlfFile*: an HTK-style MLF file that contains the labels for all utterances specified in the SCP file.
- *labelDim*: the total cardinality of the label set
- *labelMappingFile*: a list of all the labels seen in the MLF file

Note that multiple inputs and outputs can be specified using additional reader blocks for either features read from files listed in an SCP file or labels read from an MLF file. For example, in a multi-task learning scenario, where the network was predicting both speaker identity and senone label, the user would specify an additional block that includes an MLF file that contains labels corresponding about speaker identity. Similarly, for a network with multiple inputs, e.g. both MFCC and PLP features, an additional feature block would be used.

For recurrent structures, such as an RNN or an LSTM, there are additional configuration options in the `HTKMLFReader`

- *frameMode*: true or false, whether the reader should randomize the data at the frame level or the utterance level. Setting *frameMode* to true is the default and is appropriate for training networks without any temporal connections. For networks that are designed to learn sequential information, e.g. RNN, *frameMode* should be set to false, which means utterances are randomized but proper sequence is maintained within an utterance. Note that this only works with the “*blockRandomize*” read method.
- *nbruttsineachrecurrentiter*: specifies the number of utterances to process together for efficient training of networks with recurrent structures. The default is 1.
- *truncated*: true or false

It is important to note that there are some small differences between the standard SCP and MLF files used in HTK and the ones used in CNTK.

Most notably, the MLF file must contain the actual symbols used for classification. For continuous speech recognition, this typically means labels corresponding to the senones (physical HMM states). However, `HVite` typically generates an MLF during forced alignment that includes only the logical HMM state names. Thus, to use this MLF in CNTK, either the MLF must be post-processed to replace the logical state names with the corresponding senone labels, or `HVite` must be modified so that it writes the senone labels directly.

The SCP files processed by the `HTKMLFReader` can be one of two varieties: either the standard format, where each line corresponds to a physical feature file, or the aliased format, where each line contains a logical name and refers to a segment of a possibly larger physical file, defined by start and end frames. The logical name is used to identify the corresponding labels in the MLF file. Even if the files are stored individually, as in the first case, the aliased format must always be used with the *blockRandomize* read method, as it uses the information about the starting and ending frames in the SCP file to determine the utterance lengths without having

to open the files themselves. In this case, the start frame should be 0 and the ending frame should be set appropriately based on the length of the utterance. In addition, for multiple inputs and/or outputs, the aliased format should also be used so that all SCP files and MLF files have their logical names in common. If the `rollingWindow` read method is used instead of `blockRandomize`, then the start and end frame information may be omitted.

Here are example snippets of SCP and MLF files for the TIMIT corpus for appropriate for a network with 2 feature inputs (in this case, MFCC and PLP features) and 1 output corresponding to phoneme states. The SCP file for the MFCC features contains entries such as these.

```
train-dr1-fcjf0-si1027.mfc=//hostname/data/TIMIT/mfc/train/
dr1/fcjf0/train-dr1-fcjf0-si1027.mfc[0,306]
train-dr1-fcjf0-si1657.mfc=//hostname/data/TIMIT/mfc/train/
dr1/fcjf0/train-dr1-fcjf0-si1657.mfc[0,281]
train-dr1-fcjf0-si648.mfc=//hostname/data/TIMIT/mfc/train/
dr1/fcjf0/train-dr1-fcjf0-si648.mfc[0,359]
```

The SCP file for the PLP features is be similar but point to different physical files. Note that the logical root name in both SCP files is the same.

```
train-dr1-fcjf0-si1027.plp=//hostname/data/TIMIT/plp/train/dr1/fcjf0/train-dr1-fcjf0-si1027.
plp[0,306]
train-dr1-fcjf0-si1657.plp=//hostname/data/TIMIT/plp/train/dr1/fcjf0/train-dr1-fcjf0-si1657.
plp[0,281]
train-dr1-fcjf0-si648.plp=//hostname/data/TIMIT/plp/train/dr1/fcjf0/train-dr1-fcjf0-si648.plp
[0,359]
```

The MLF file also shares the logical name with both SCP files. In this

```
#!MLF!#
"train-dr1-fcjf0-si1027.rec"
0 200000 h#_s2 -136.655975 h# -589.680481 h#
200000 400000 h#_s3 -145.780716
400000 800000 h#_s4 -307.243774
800000 1200000 q_s2 -349.529327 q -897.429504 q
1200000 1500000 q_s3 -280.568817
1500000 1800000 q_s4 -267.331390
1800000 1900000 iy_s2 -76.825096 iy -673.892883 iy
1900000 2400000 iy_s3 -305.832458
2400000 2800000 iy_s4 -291.235352
```

Finally, it is important to note that in most speech recognition applications, the continuous-valued features are used as the inputs discrete categorical labels are used as the output. However, the `HTKMLFReader` just associates data with node

names and is agnostic as to how this data is used. For example, an appropriate MLF file of speaker identity labels could be used to generate a one-hot vector of speaker identity features as an input to the network.

3.4.3 LMSequenceReader

LMSequenceReader is a reader that reads text string. It is mostly often used for language modeling tasks. An example of its setup is as follows

```
reader=[
  readerType="LMSequenceReader"
  randomize=false
  nbruttsineachrecurrentiter=10
  unk="<unk>"
  wordclass="$DataDir$\wordclass.txt"
  file="$DataDir$\penntreebank.train.txt"

  labelIn=[
    labelDim=10000

    beginSequence="</s>"
    endSequence="</s>"
  ]
]
```

In this example, the LMSequenceReader has following parameters:

- *randomize*: it is either None or Auto. This specifies the mode of whether doing sentence randomization of the whole corpus.
- *nbruttsineachrecurrentiter* : this specifies the limit of the number of sentences in a minibatch. The reader arranges same-length input sentences, up to the specified limit, into each minibatch. For recurrent networks, trainer resets hidden layer activities only at the beginning of sentences. Activities of hidden layers are carried over to the next minibatch if an end of sentence is not reached. Using multiple sentences in a minibatch can speed up training processes.
- *unk* : this specifies the symbol to represent unseen input symbols. Usually, this symbol is “<unk>”. Unseen words will be mapped to the symbol.
- *wordclass* : this specifies the word class information. This is used for class-based language modeling. An example of the class information is below.

The first column is the word index. The second column is the number of occurrences, the third column is the word, and the last column is the class id of the word.

```

0 42068 </s> 0
1 50770 the 0
2 45020 <unk> 0
3 32481 N 0
4 24400 of 0
5 23638 to 0
6 21196 a 0
7 18000 in 1
8 17474 and 1

```

- *file* : the file contains text strings. An example is below

```

</s> pierre <unk> N years old will join the board as a
nonexecutive director nov. N </s>
</s> mr. <unk> is chairman of <unk> n.v. the dutch
publishing group </s>

```

In this example you can also notice one sub-blocks named *labelIn*.

- *labelIn*: the section for input label. It contains the following setups
 - *beginSequence* – the sentence beginning symbol
 - *endSequence* – the sentence ending symbol
 - *labelDim* – the dimension of labels. This usually means the vocabulary size.

3.4.4 LUSequenceReader

LUSequenceReader is similar to SequenceReader. It however is used for language understanding tasks which have input and output strings that are different. An example of setting up LUSequenceReader is as follows

```

reader=[
  readerType="LUSequenceReader"
  randomize="none"
  wordContext=0:1:2

  nbrUttsInEachRecurrentIter=10

```

```

unk="<unk>"
wordMap="$DataDir$\inputmap.txt"
file="$DataDir$\atis.train.IOB"

labelIn=[
    useWordMap=true

    beginSequence="BOS"
    endSequence="EOS"
    token="$DataDir$\input.txt"
]

labels=[
    beginSequence="O"
    endSequence="O"
    token="$DataDir$\output.txt"
]
]

```

In this example, the `LUSequenceReader` has following parameters:

- *wordContext*: this specifies a context window. For example, `wordContext=0:1:2` specifies a context window of 3. In this context window, it reads input at a current time, the next time and the time after the next time. Another example would be `wordContext=0:-1`. In such case, `LUSequencReader` reads a context window of 2 that consist of the current input and the immediate last input. .
- *randomize*: it is either `None` or `Auto`. This specifies the mode of whether doing sentence randomization of the whole corpus.
- *nbrUttsInEachRecurrentIter* : this specifies the limit of the number of sentences in a minibatch. The reader arranges same-length input sentences, up to the specified limit, into each minibatch. For recurrent networks, trainer resets hidden layer activities only at the beginning of sentences. Activities of hidden layers are carried over to the next minibatch if an end of sentence is not reached. Using multiple sentences in a minibatch can speed up training processes.
- *unk* : this specifies the symbol to represent unseen input symbols. Usually, this symbol is "<unk>".
- *wordMap*: this specifies a file that maps inputs to other inputs. This is useful if the user wants to map some inputs to unknown symbols. An example of the word mapping file is as follows

```
buy buy
trans <unk>
```

- *file* : the file contains input and its labels. The last column is the label, and the other columns contain inputs. An example of training file is below.

```
BOS O
flight O
from O
charlotte B-fromloc.city_name
to O
las B-toloc.city_name
vegas I-toloc.city_name
EOS O
```

In this example you can also notice two sub-blocks named features and labels.

- *labelIn*: the section for input label. It contains the following setups
 - useWordMap – [True, False] specifies if using word map to map input words to other input words.
 - beginSequence – the sentence beginning symbol
 - endSequence – the sentence ending symbol
 - token – token file contains a list of input words. Their orders are not important.
- *labels*: the section for output label.
 - token – token file contains a list of output labels. Their order is not important as long as the tokens are unique.

3.5 Top-Level Commands

Besides the *train* command, CNTK supports a list of other top-level commands. We list all the top-level commands and corresponding parameters in this section.

3.5.1 Train Command

This command asks CNTK to train a model. The related parameters are:

- reader – the reader configuration block used to determine how to load input data. We have covered this parameter block in previous sections.
- SGD – the SGD training setup.
- NDNetworkBuilder – the NDL network builder configuration block.
- simpleNetworkBuilder – the simple network builder configuration block.
- cvReader – (optional) the reader configuration block for cross-validation data
- makeMode – if true (default) the training will continue from whatever epoch interrupted. If false the training will restart from scratch.
- numMBsToShowResult – indicates after how many minibatches the intermediate results should be shown.

3.5.2 Adapt Command

This command adapts an already trained model using KL divergence regularization. It is advised that all other adaptations should be carried out using MEL. The adapt command is very similar to the train command except that it carries two more parameters:

- originalModelFileName – the file name of the model that will be adapted.
- refNodeName – the name of the node in the computational network which will be used for KL divergence regularization.

3.5.3 Test or Eval Command

This command evaluate/test a model for accuracy, usually with a test dataset. The related parameters are

- reader – the reader configuration block to read the test data.
- modelPath – the path to the model to be evaluated.
- minibatchSize – the minibatch size to use when reading and processing the dataset.

- `epochSize` – the size of dataset. Default is 0. The entire dataset will be evaluated if it's set to 0.
- `numMBsToShowResult` – indicates after how many minibatches the intermediate results should be shown.
- `evalNodeNames` – an array of one or more node names to evaluate.

3.5.4 CV

This command evaluates a series of models from different epochs on a development (or cross validation) set and displays the information of the best model. Besides the parameters used for the test command, this command also use the parameters

- `crossValidationInterval` – the array of 3 integers identifying the starting epoch, epoch increment and final epoch to evaluate. For example, 3:2:9 means the models 3,5,7, and 9 will be evaluated.
- `sleepTimeBetweenRuns` – how many seconds to wait between runs. This is needed only if your GPU is too hot.

3.5.5 Write Command

This command writes the value of an output node to a file. The related parameters are

- `reader` – the reader configuration block to read the input data.
- `writer` – the writer configuration block to determine how to write the output data. If this value is not specified the `outputPath` parameter will be used.
- `minibatchSize` – the minibatch size to use when reading and processing the dataset.
- `epochSize` – the size of dataset. Default is 0. The entire dataset will be evaluated if it's set to 0.
- `modelPath` – the path to the model to be used to compute the output.
- `outputPath` – the path to the file to write the output in a text format. If the writer block exists this parameter will be ignored. Either `outputPath` or `writer` must exist.
- `outputNodeNames` – an array of one or more output node names to be written to a file.

3.5.6 Edit Command

This command executes a model editing language (MEL) script. We will describe MEL in detail in Chapter 4. Here we list the associated parameter blocks.

- `editPath` – the path to the MEL script to be executed.
- `ndlMacros` – the path to the NDL macros file that will be loaded and used in the MEL script.

3.5.7 SVD Command

This command conducts SVD decomposition for learnable parameters. For example:

```
svd=[
  action="SVD"
  modelPath="train\lstm.model.67"
  outputModelPath="train\lstm.model.svd"
  SVDConfig="nodeConfigs"
]
```

Here we list the associated parameter blocks.

- `modelPath` – specifies where to load the initial model
- `outputModelPath` – specifies where to save the revised model
- `SVDConfig` – a config file which specifies how the SVD is performed for different groups of nodes. This config is in a two-column format:

```
<NodeNameRegex> <float>
```

The first column is a regular expression which will match the node name in one group. The second column is a float indicating what percentage of SVD-energy will be kept after SVD, where SVD-energy is defined as the sum of singular values. For example, the config

```
LSTMOutput[1-3].Wx[ifco] 0.4
LSTMOutput[1-3].Wh[ifco] 0.6
```

will cause a more aggressive SVD (0.4) decomposition for the non-recurrent connections (from x to i,f,c,o gates) in LSTM and less aggressive SVD decomposition for recurrent connections (from h to i,f,c,o gates), where the parameter names are defined in the NDL.

3.5.8 Dumpnode Command

This command dumps the information of node(s) to an output file, which can also be accomplished in MEL with greater control. The related parameters are:

- `modelPath` – the path to the model file containing the nodes to dump.
- `nodeName` – the name of the node to be written to a file. If not specified all nodes will be dumped.
- `outputFile` – the path to the output file. If not specified a file name will be automatically generated based on the `modelPath`.
- `printValues` – determines whether to print the values associated with a node if the node's values are persisted in the model. Default is true.
- `printMetadata` – determines whether to print the metadata (node name, dimensions, etc.) associated with a node. Default is true.

3.5.9 WriteWordAndClass Command

This command reads the text training data, counts the number of occurrences of each each word in the training set, sorts the words in the descending order of the counts, gives each word a unique id, assign each word to a class, and generates a four column vocabulary file and a word-to-id mapping file to be used by the `LMSequenceReader` to train class-based language models. The related parameters are:

- `inputFile` – the path to the text training file.
- `outputVocabFile` – the name of the generated four-column vocabulary file. The first column is the word id, the second column is the count of the word, the third column is the word, and the fourth column is the class id.
- `outputWord2Cls` – the path to the generated word-to-class mapping file.
- `outputCls2Index` – the path to the generated class-to-wordId mapping file.
- `vocabSize` – the desired vocabulary size.
- `nbrClass` – the desired number of classes .
- `cutoff` – the cutoff count. When a word's count is below or equal to this value the word will be treated as `<unk>`. Default is 2. Note that this parameter is used only if the desired vocabulary size is larger than the actual number of words exist in the training set.

3.5.10 CreateLabelMap Command

Often it is easy to manually craft the label mapping file. Sometimes, however, you would prefer to have the label mapping file generated automatically which is the purpose of the CreateLabelMap command. Currently UCIFastReader is the only reader that supports this action. The related parameters are

- `section` – the parameter block name (usually a train block) which has the reader sub-block that will be used to generate the label mapping file. The generated label mapping file will be saved to the `labelMappingFile` specified in the reader sub-block of this parameter block.
- `minibatchSize` – the minibatch size to use when creating the label mapping file.

3.6 Plot Command

This command loads a given computation network and describes the network topology (usually a DAG) using the DOT (<http://www.graphviz.org/doc/info/lang.html>) language. It can also optionally call a third-part tool (<http://www.graphviz.org/Documentation/dotguide.pdf>) to render the network topology. Note that many DOT rendering tools are optimized for DAG, and the rendered graph becomes quite messy if the network structure is non-DAG. The non-DAG structure is usually caused by the use of delay node. In this case, this command will treat all the delay nodes as leaf nodes.

The related parameters of this command are:

- `modelPath` – the path to the computation network.
- `outputDOTFile` – the path to the output DOT file. If user does not specify it, `${modelPath}.dot` will be assumed.

Optionally, if users want to convert the dot description to a figure, he or she needs to specify the following parameters:

- `outputFile` – the path to the output figure.
- `renderCmd` – A string indicates how the third party tool is used to convert from a DOT file to a figure. For example, if graphviz is used, the `RenderCmd` can be written as

```
RenderCmd="d:\Tools\graphviz\bin\dot.exe -Tjpg <IN> -o<
OUT>"
```

where the plot command will substitute “<IN> “ with the outputDOTFile command parameters and “<OUT>” with the outputFile command parameters; -Tjpg indicates a jpg file format is selected.

Here is a complete example:

```
command=topoplot

topoplot=[
  action="plot"
  modelPath="train\lstm.model.0"
  outputdotFile="train\lstm.model.dot" # this specifies the
    dot file to output
  # if user does not specify this, it will be ${modelPath}.
    dot
  outputFile="train\lstm.model.jpg" # this specifies the
    rendered image
  renderCmd="d:\Tools\graphviz\bin\dot.exe -Tjpg <IN> -o<OUT>
    >"
  # if RenderCmd is specified, CNTK will call the plot
    command after replacing
  # <IN> with ${outputdotFile} and <OUT> with ${outputfile}
]
```

3.7 ConvertDBN Command

This command is used to convert a model generated by Microsoft’s dbn.exe tool to a CNTK model. This command is useful when you want to compare the performance of these two tools (dbn.exe only supports simple fully connected deep neural networks), port existing models trained with dbn.exe to CNTK, or if you want to use the RBM pre-training which is available in dbn.exe but not in CNTK right now. The related parameters are

- modelPath – the full path of the generated CNTK model.
- dbnModelPath – the full path of the model to be converted.

3.8 Additional Top-Level Configurations

Besides commands, there are several top-level configurations that you can set.

3.8.1 Stderr

All logging information in CNTK is sent to standard error, and will appear on the console screen. You can redirect the logging information to a file with the `stderr` parameter. The value of the `stderr` parameter defines the folder and the prefix of the log file. The suffix is determined by the command block executed. For example, if you set

```
stderr="c:\cntk\log\cntk"
```

and the command block `mnistTrain` is to be executed then the full log file name will be `c:\cntk\log\cntk_mnistTrain.log`. It is important to note that this file is overwritten on subsequent executions if the `stderr` parameter and the command being run are identical.

3.8.2 DeviceId

CNTK supports CPU and GPU computation. Users can determine what device to use by setting the `deviceId` parameter. The possible values are

- `auto`: choose the best device available. If multiple GPUs are available, it will choose the fastest, least busy GPU. If no usable GPUs can be found CPU will be used.
- `cpu` or `-1`: use the CPU for all computation
- a non-negative integer: use the GPU identified by this number to carry out all computation.
- `all` : use all the available GPU devices (currently this flag is not supported)

3.8.3 Precision

The precision of the floating point values determines the tradeoff between the numerical accuracy and the training and testing speed. In CNTK the precision is specified at the top-level as

```
precision="float"
```

The valid values are `float` (default) and `double`. For most experiments you should use `float` since it's significantly faster, esp. when running under GPUs. To run gradient check you should set precision to `double`.

3.8.4 NumCPUThreads

The NumCPUThreads parameter determines the maximum CPU threads used for CPU math operations. In CNTK it is specified at the top-level as

```
NumCPUThreads=4
```

By default the value is 0, which means using what ever number of threads determined by MKL, ACML, or OpenMP. If a positive number is specified, the number of threads will not be larger than the value specified and the number of hardware concurrency. If a negative number is specified, the actual number of thread will be the sum of the number of hardware concurrency and this value floored at 1.

3.8.5 TraceLevel

The traceLevel parameter is usually used by the code in CNTK to specify how much extra output (verbosity) is desired as in

```
traceLevel=0 # larger values mean more output
```

The default value is 0 and specifies minimal output. The higher the number the more output can be expected. Currently 0 (limited output), 1 (medium output) and 2 (verbose output) are the only values supported.

3.8.6 ShareNodeValueMatrices

The ShareNodeValueMatrices parameter is used to indicate whether to turn on the sharing of forward computation result matrices to further reduce memory usage. To turn it on, use

```
ShareNodeValueMatrices=true
```

The default value is false, which means the memory sharing only happens during the backward phase. You should decide whether to turn it on by running the same setup with this flag on and off. If the results are the same, you are safe to turn it on. In the future, we will make it on by default after more extensive test.

3.9 Advanced Command Line Parsing Rules

Here we describe CNTK command line parsing rules. As described above, CNTK consists of a number of components to complete a task. Most of these components need some configuration information available in order to function, and these configuration parameters are provided through configuration files in CNTK.

Configuration files are collections of name-value pairs. The configuration data can be one of the following types:

- Simple: a single value is assigned to the configuration parameter. For example, `deviceId=auto`.
- Array: a configuration parameter is assigned an array of values which need not be of a uniform type. ':' is the default separator for arrays. The separator may be changed by enclosing the array values in parenthesis and placing the new separator character immediately following the open parenthesis. The '*' character allows a particular value to be repeated multiple times in the array. For example, `minibatchSize=256:512:512:512:1024` equals to `minibatchSize=256:512*3:1024`.
- Set: parameter sets contain sets of configuration parameters of any type. Parameter sets can be nested. The default separator for parameter sets is ';' if multiple items are included on one line. Line separators also serve as separators for items. For example,

```
block1=[id=1;size=256]

block2=[
  subblock=[string="hi";num=5]
  value=1e-10
  array=10:"this is a test":1.25
]
```

In CNTK, configuration files are organized in a hierarchical fashion. The actual data values are not evaluated until a CNTK component requests the value. When a value is requested by a component, CNTK will search inside the component block first. If the value is not found, it will continue looking in the parent and grandparent parameter set until the parameter is found, or the top level of the configuration hierarchy is reached without a match. This allows the share of the same parameter values easier across different blocks.

As we mentioned earlier, to run CNTK you need to specify the configuration file in the command line as

```
cntk configFile=yourExp.cntk
```

This will load the requested configuration file, and execute any command block listed in the command parameters in the configuration file.

3.9.1 Commands and Actions

There must be a top-level command parameter, which defines the commands (separated with ':') that will be executed in the configuration file. Each command references a command block in the file, which must contain an action parameter defining the operation that block will perform. For example the following command will execute the `mnistTrain` block, which executes the `train` action, followed by the `mnistTest` block, which evaluates the model.

```
command=mnistTrain:mnistTest
```

```
mnistTrain=[
  action="train"
  ...
]
```

```
mnistTest=[
  action="eval"
  ...
]
```

3.9.2 Configuration Overloads In the Command Line

It is common to have a configuration that can be used as a base configuration, and modify only a few parameters for each experimental run. This can be done in a few different ways, one of which is to override settings on the command line. For example, to override the model file path, one can simply modify the command line as follows:

```
cntk configFile=yourExp.cntk stderr="c:\temp\newpath"
```

This will override the current setting for `stderr`, which is defined at the root level of the configuration file, with the new value. If a parameter inside a command block needs to be modified, the block also needs to be specified. For example, I can change the `minibatchSize` for an experiment in the command line as

```
cntk configFile=yourExp.cntk mnistTrain=[minibatchSize=256]
```

or modify the data file used for an experiment as

```
cntk configFile=yourExp.cntk mnistTrain=[reader=[file="
  mynewfile.txt"]]
```

3.9.3 Layered Configuration Files

Instead of overriding some parts of a configuration file using command line parameters, one can also specify multiple configuration files, where the latter files override the earlier ones. This allows a user to have a *master* configuration file, and then specify, in a separate configuration file, which parameters of the master they would like to override for a given run of CNTK. This can be accomplished by either specifying a '+' separated list of configuration files, or by using the "config-File=" tag multiple times. The following are equivalent.

```
cntk configFile=yourExp1.cntk+yourExp2.cntk
cntk configFile=yourExp1.cntk configFile=yourExp2.cntk
```

If yourExp2.cntk only contains the string "mnistTrain=[reader=[file=mynewfile.txt]]", then both of these commands would be equivalent to:

```
cntk configFile=yourExp1.cntk mnistTrain=[reader=[file="
  mynewfile.txt"]]
```

Note that the value of a variable is always determined by the last time it is assigned. It is also possible to mix command-line parameters, and layered configuration files, in arbitrary combinations. For example,

```
cntk configFile=yourExp1.cntk+yourExp2.cntk var1=value
  configFile=yourExp3.config
```

would process these configuration parameters in the order they appear on the command line and whatever value assigned last is the value used.

In addition being able to specify multiple configuration files at the command line, a user can *include* one configuration file within another. For example, if the first line of yourExp2.cntk was

```
include=yourExp1.cntk
```

then simply running

```
cntk configFile=yourExp2.cntk
```

would be equivalent to running

```
cntk configFile=yourExp1.cntk+yourExp2.cntk
```

where in this latter case, yourExp2.cntk doesn't contain the *include* statement. Note that these *include* statements can appear anywhere inside a configuration file; wherever the include statement appears, that is where the specified configuration file will be *included*. Including a configuration file is equivalent to pasting the

contents of that file at the location of the include statement. Include statements are resolved recursively (using a depth-first search), meaning that if `yourExpA.cntk` includes `yourExpB.cntk`, and `yourExpB.cntk` includes `yourExpC.cntk`, then the full chain will be resolved, and `yourExpC.config` will effectively be included in `yourExpA.cntk`. If a configuration file is included multiple times (eg, 'A' includes 'B' and 'C', and 'B' also includes 'C'), then it will effectively only be included the first time it is encountered.

3.9.4 Stringize Variables

While layered configuration files allow users to reuse configuration files across experiments, this can still be a cumbersome process. For each experiment, a user might have to override several parameters, some of which might be long file paths (eg, 'stderr', 'modelPath', 'file', etc). The "stringize" functionality can make this process much easier. It allows a user to specify configuration like the following:

```
command=SpeechTrain

stderr="$Root$\$RunName$.log"

speechTrain=[
  modelPath="$Root$\$RunName$.cn"

  SGD=[
    reader=[
      features=[
        type="real"
        dim="$DataSet1_Dim$"
        file="$DataSet1_Features$"
      ]
    ]
  ]
]
```

Here, "Root", "RunName", "DataSet1_Dim", and "DataSet1_Features" are variables specified elsewhere in the configuration (at a scope visible from the point at which they are used). When interpreting this configuration file, the parser would replace every string of the form "\$VarName\$" with the string "VarValue", where "VarValue" represents the value of the variable called "VarName". The variable resolution process is recursive; for example, if `A=B`, `B=C`, and `C=HelloWorld.txt`, then `A` would be resolved as "HelloWorld.txt". Please make sure there is no reference loop in your configuration file. Otherwise the parser will go into infinite loop at this moment.

Notice that because it is equivalent for a user to specify the value of a variable in a configuration file vs. at the command line, the values for these variables can be specified in either location. Recall that the value of a variable is determined by the last time it is assigned, whether that happens to be in a configuration file, or on the command line. Thus, if "Root" is defined in config1.txt, but overridden at the command-line, then the value specified at the command-line would be the one used to resolve instances of \$Root\$ in configFile1.txt. One useful feature is that if 'stderr' or 'modelPath' point to directories which do not exist, these directories will be created by CNTK; this allows you to specify something like: "stderr=\$Root\$\\$RunName\$\\$RunName\$.log", even if the directory "\$Root\$\\$RunName\$" doesn't exist.

3.9.5 Default, Repeated Values, and Comments

Most parameters in configuration files have a default value which will be used if no configuration value is specified. If there is no default value and the value cannot be found in a search, an exception will be displayed and the program will exit.

If a parameter name is specified more than once, the last value set to that value is the one that will be maintained. The only exception to this is in parameter sets, which are surrounded by '[' square braces ']', in these cases the values inside the braces are considered to be a parameter set, and will be added to the currently existing parameter set. For example:

```
params=[a=1;b=2;c=3]
params=[c=5;d=6;e=7]
```

is effectively equal to:

```
params=[a=1;b=2;c=5;d=6;e=7]
```

Note that this *append* processing is NOT used for array elements, and the entire array will be replaced if it is set multiple times.

The '#' character signifies the beginning of a comment, everything that occurs after the '#' is ignored. The '#' must be preceded by whitespace or be at the beginning of the line to be interpreted as a comment. The following is valid comment

```
stderr="c:\cntk\log\cntk" # "_mnistTrain_mnistTest.log"
```

The following is an example of a value that will not be interpreted as a comment. It sets a parameter "var" to infinity as the '#' in '1#INF' is not a comment marker

```
var=1#INF
```

Chapter 4

Advanced Setups in Computational Network Toolkit

4.1 Network Definition Language

The network description language (NDL) provides a simple yet powerful way to define a network in a code-like fashion. It contains variables, macros, and other well understood concepts and allows users of CNTK to define any computational network architecture they want.

4.1.1 Basic Concepts

The example NDL script below describes a one-hidden-layer DNN.

```
#Variables
SDim=784
HDim=256
LDim=10

#Inputs
features=Input(SDim, 1)
labels=Input(LDim, 1)

#Parameters
W0=Parameter(HDim, SDim)
B0=Parameter(HDim, 1)
W1=Parameter(LDim, HDim)
B1=Parameter(LDim, 1)

#Computation
```

```

Times1=Times(W0, features)
Plus1=Plus(Times1, B0)
RL1=RectifiedLinear(Plus1)

Times2=Times(W1, RL1)
Plus2=Plus(Times2, B1)

#Training Criterion
CE=CrossEntropyWithSoftmax(labels, Plus2)

#Test Objective Function
ErrPredict=ErrorPrediction(labels, Plus2)

#Special Nodes
FeatureNodes=(features)
LabelNodes=(labels)
CriterionNodes=(CE)
EvalNodes=(ErrPredict)
OutputNodes=(Plus2)

```

We will introduce the basic concepts such as variables, parameters, inputs, and training criteria using this example.

4.1.1.1 Variables

```

SDim=784
HDim=256
LDim=10

```

The first thing you will notice are the variables: `SDim`, `HDim` and `LDim`. Variables are defined in NDL when they appear on the left of an equal sign. From that point on that variable name will be associated with the value it was assigned. A variable can contain a matrix or scalar value. Variables are immutable, and assigning new values to an existing variable is not supported.

Variable names may be any alphanumeric sequence that starts with a letter and are case-insensitive. Any name that is also a function (operator) name is a reserved word and cannot be used for a variable. The special node names *FeatureNodes*, *LabelNodes*, *CriterionNodes*, *EvalNodes*, *OutputNodes* are also reserved and may not be used as variable names.

The variables `SDim`, `HDim` and `LDim` are set to scalar numeric values in this example and are used as parameters in the NDL functions. More specifically, `SDim`, `HDim` and `LDim` are the sizes of input, hidden, and output layers in this example and assigned the values 784, 256, and 10, respectively. The input and

output layer sizes are determined by the task while the hidden layer size can be chosen by the users depending on their needs.

4.1.1.2 Inputs

```
features=Input (SDim)
labels=Input (LDim)
```

Inputs are used to represent input data and labels associated with the samples. Input is a special function. They are represented as InputNodes internally and are not saved as part of the CN model. In this example, the *features* input will have the dimensions of the input data (SDim), and the *labels* input will have the dimensions of the labels (LDim). The variables chosen here are for convenience and could be any valid variable name.

4.1.1.3 Parameters

```
B0=Parameter (HDim, 1)
W0=Parameter (HDim, SDim)
W1=Parameter (LDim, HDim)
B1=Parameter (LDim, 1)
```

Parameters are matrices that constitute the learned model upon completion of training. They are represented as LearnableParameterNodes internally and are saved as part of the CN model. The model parameters transform the input data into the desired output data and are updated as part of the learning process.

In this example. CNTK will train W0 and W1 the weight matrices, and B0 and B1 the bias matrices. Parameter matrices are always represented as two-dimensional matrices internally. If only one dimension is given the other dimension is assumed to be 1. By default parameters are initialized with uniform random numbers between -0.5 and 0.5, but other options exist (see Section 4.1.4).

4.1.1.4 Functions

```
Times1=Times (W0, features)
Plus1=Plus (Times1, B0)
RL1=RectifiedLinear (Plus1)
```

Functions describe computation steps. Functions are called using a syntax similar to most programming languages. The function name is followed by parenthesis which contains the comma separated parameter list. Each function returns a single

value, which is identified by a variable. The complete list of functions can be found in Section 4.1.4.

The hidden layer in this example involves three computation steps: It first gets the product of the weight matrix W_0 and the features matrix; then gets the excitation by adding the product to the bias; and finally applies the activation function, in this case `RectifiedLinear()`, to the excitation. Internally the operators `Times()`, `Plus()` and `RectifiedLinear()` are represented as computation nodes `TimesNode`, `PlusNode`, and `RectifiedLinearNode`, respectively. These Computation nodes are saved as part of the CN model but their values are not.

4.1.1.5 Training and Testing Criteria

```
#Training Criterion
CE=CrossEntropyWithSoftmax(labels, Plus2)

#Test Objective Function
ErrPredict=ErrorPrediction(labels, Plus2)
```

Each CN can have multiple root nodes used for different purposes. In this example, the training and testing criteria are different. We use the operator `CrossEntropyWithSoftmax()` to compute the training criterion and the operator `ErrorPrediction()` to compute the testing criterion. These operators are internally represented as computation nodes `CrossEntropyWithSoftmaxNode` and `ErrorPredictionNode` with names `CE` and `ErrPredict`, respectively.

4.1.1.6 Special Nodes

```
FeatureNodes=(features1, features2)
LabelNodes=(labels)
CriterionNodes=(CE)
EvalNodes=(ErrPredict)
OutputNodes=(Plus2)
```

After defining the network, it's important to let CNTK know what the special nodes are in the network. For example, CNTK needs to know which input nodes are features and labels. It also needs to know the default output nodes, evaluation nodes and training criteria nodes. CNTK supports multiple inputs and outputs which can be represented by comma separated variable names surrounded by parentheses.

These special nodes can be specified in two different ways, the node arrays, or by use of special tags we will discuss later. If both methods are used the values are combined.

4.1.1.7 Comments

to be interpreted as a comment. The following are valid comments:

```
#Test Objective Function
ErrPredict=ErrorPrediction(labels, Plus2) # classification
      error

# the following variable is set to infinity and
# the # in 1#INF is not interpreted as a comment marker
var = 1#INF
```

4.1.2 Macros

While creating a network using the syntax shown above is not all that difficult, it can get wordy when creating deep neural networks with many layers. To alleviate this problem, common definitions can be combined into macros.

4.1.2.1 Defining Macros

Macros can be defined as a one line function, or as a block of code. For example, the single-line macro

```
RFF(x1, w1, b1)=RectifiedLinear(Plus(Times(w1,x1),b1))
```

defines the three steps involved in the hidden layer computation in the previous section, all in one line. When the functions calls are nested they will be evaluated from the innermost nested function call to the outermost.

A macro can also be defined as a block of code, for example

```
FF(X1, W1, B1)
{
  T=Times(W1,X1)
  FF=Plus(T, B1)
}
```

defines the feed forward computation without the activation function. It shows an alternate format of a macro. Semicolons are not required, but can be used if desired. The variables and parameters used inside the macros are local to the macro but are accessible externally with dot operators we will discuss later. The return value of a macro is defined by a local macro variable that has the same name as the macro. In this case the FF() macro's return value is the FF local variable. If no variable matches, the last variable in the macro will be returned.

The macro

```

BFF(in, rows, cols)
{
    B=Parameter(rows)
    W=Parameter(rows, cols)
    BFF = FF(in, w, b)
}

```

shows how parameters are declared within a macro. As in this example, a macro may call another macro. However recursion (i.e., calling itself) is not supported.

The macro

```

RBFF(input, rowCount, colCount)
{
    F = BFF(input, rowCount, colCount);
    RBFF = RectifiedLinear(F);
}

```

calls the previous macro and adds the RectifiedLinear() activation function. The macro

```

SMBFF(x, r, c, labels)
{
    F = BFF(x, r, c);
    SM = CrossEntropyWithSoftmax(labels, F)
}

```

computes the output layer value and the cross entropy training criterion. Since no variable matches the name of the macro, the last variable defined in the macro, SM in this case, is used as a return value.

4.1.2.2 Using Macros

The following example uses the macros defined above. It describes the same one-hidden-layer network discussed in Section 4.1.1 but is much simpler and easier to understand because of the use of macros. One new feature shown in this network definition is the access to macro-region variables. ErrorPrediction() needs to access an intermediate result from SMBFF before the CrossEntropyWithSoftmax() is applied. Although the needed variable is local to the macro, it can be accessed via the *dot* syntax. The return value of the macro is CE, so CE.F can be used to access the local variable F defined in the macro. This does requires the user to know the names used in the macro, so having all macro definitions available is important. In the single line version of macros, there are no user defined variable names, so this feature cannot be used. Since macros can be nested, dot names can be several layers deep if necessary.

```
# Sample, Hidden, and Label dimensions
SDim=784
HDim=256
LDim=10

#Inputs
features=Input (SDim)
labels=Input (LDim)

# Layer operations
L1 = RBFF(features, HDim, SDim)
CE = SMBFF(L1, LDim, HDim, labels)
Err=ErrorPrediction(labels, CE.F)
```

4.1.3 Optional Parameters

Optional Parameters are a feature that allows additional parameters to be specified to a function. While optional parameters can be specified for any function or macro, they are limited to constant values. In addition, the underlying function must support the passed optional parameters, or there is no effect on the network. When optional parameters are used on a macro, the macro must define local variables that match the optional parameter name and value.

4.1.3.1 Parameter initialization

One common use of these optional parameters is to define how parameters will be initialized. In the example

```
B0=Parameter(HDim, 1, init=zero)
W0=Parameter(HDim, SDim, init="uniform")
```

the Bias matrix will be zero initialized, and the weight matrix will be initialized with uniform random numbers. Please consult Section 4.1.4 to find which functions accept optional parameters

4.1.3.2 Tagging special values

As an alternate to providing an array of special nodes that are used as features, labels, criteria, etc, optional parameters can be used. So instead of using

```
FeatureNodes=(features)
LabelNodes=(labels)
CriterionNodes=(CE)
```

```
EvalNodes=(ErrPredict)
```

we can tag these nodes as they are defined.

```
features=Input(SDim, 1, tag="feature")
labels=Input(LDim, 1, tag="label")
CE = SMBFF(L3, LDim, HDim, labels, tag="criterion")
Err=ErrorPrediction(labels, CE.F, tag="eval")
```

The acceptable tag names correspond to the special node types and are as follows:

- *feature*: feature input.
- *label*: label input
- *criterion*: training criteria
- *eval*: evaluation node
- *output*: output node

4.1.4 NDL Functions

This section describe the currently implemented NDL functions (operators). These operations are implemented as computation nodes internally and are discussed in Chapter 2. As CNTK is expanded, additional functions become available. In the following discussion named optional parameters are optional parameters that are specified by an augment name. For example, if cols is a named optional parameter, we need to call the function as

```
Function(rowsValue, cols=colsValue)
```

4.1.4.1 InputValue or Input

Defines input data for the network. Inputs are read from a datasource. The data-source information is specified in the configuration file separately, allowing the same network to be used with multiple datasets easily. The syntax is

```
InputValue(rows, cols, {tag="feature"|"label"})
Input(rows, cols, {tag="feature"|"label"}) #deprecated
```

- rows - row dimension of the data.

- cols - column dimension of the data, default to 1. Each column is considered as a sample in CNTK. Default value is often used since the actual value may be determined by the minibatch size.
- tag - [named optional] tag for the inputs to indicate the intended usage. It can be either feature or label.

4.1.4.2 ImageInput or Image

Defines the three-dimensional (channel_id, img_row, img_col) image input data stored in column-major for the network. ImageInputs are read from a datasource which is specified in the configuration file separately, allowing the same network to be used with multiple datasets easily. The syntax is

```
ImageInput(width, height, channels, numImages,
           {tag="feature"|"label"})
Image(width, height, channels, numImages,
      {tag="feature"|"label"}) #deprecated
```

- width - width of the image data.
- height - height of the image data.
- channels - number of channels in the image data (i.e. RGB would have 3 channels)
- numImages - number of images, defaults to 1. Each image is stored as a column vector with size $width \times height \times channels$.
- tag - [named optional] tag for the inputs to indicate the intended usage. It can be either feature or label.

CNTK uses column-major (similar to matlab) to store the matrices. Each image is represented as a column vector internally and should be stored as [channel_id, img_row, img_col].

4.1.4.3 LearnableParameter or Parameter

Defines a parameter in the network that will be trained and stored as part of the model. Normally used for weight and bias matrices/vectors. These two function names mean the same thing but Parameter is a shorter name. The syntax is

```

LearnableParameter(row, cols,
    {computeGradient=true|false,
    init="fixedValue"|"uniform"|"gaussian"|"fromFile",
    value=0, initValueScale=number})
Parameter(row, cols, {needGradient=true|false,
    init="fixedValue"|"uniform"|"gaussian"|"fromFile",
    value=0, initValueScale=number}) #deprecated

```

- rows - number of rows in the parameter.
- cols - number of columns in the parameter, defaults to 1.
- needGradient- [named optional] determines whether the parameter should be updated by the training algorithm. Defaults is true.
- init - [named optional] parameter initialization method.
 - fixedValue- initialize all the values in the parameter with a fixed value determined by the optional value argument.
 - fromFile - No initialization is required, should only be used if the network will be initializing it in some other way
 - uniform - Initializes the parameter matrix with uniform random numbers in the range of $[-0.05 \times \text{initValueScale}, 0.05 \times \text{initValueScale}]$
 - gaussian - Initializes the parameter matrix with Gaussian random numbers with zero mean and standard deviation of $0.2 \times \text{initValueScale} / \sqrt{\text{cols}}$
- initValueScale - [named optional] scale the range of the random numbers used for initialization. Only meaningful when Uniform or Gaussian is used as a init method. Default is 1.
- value - [named optional] the initial value of all the parameters when the initialization method is set to fixedValue. Default is 0.

4.1.4.4 Constant or Const

Defines a constant parameter (i.e., will not change during the training process) that will be saved as part of the model. The syntax is

```

Constant(value, rows, cols)
Const(value, rows, cols) #deprecated

```

- value - the value of the constant.
- rows - row dimension of the data, default to 1.
- cols - column dimension of the data, default to 1.

4.1.4.5 RectifiedLinear or ReLU, Sigmoid, Tanh, Log

Apply the rectified linear function (RectifiedLinear or ReLU), Sigmoid function (Sigmoid), hyperbolic tangent (Tanh), or natural logarithm (Log) to each element of the input matrix. The resulting matrix has the same dimension as that of the input matrix. The syntax is

```
RectifiedLinear(m)
ReLU(m) #deprecated
Sigmoid(m)
Tanh(m)
Log(m)
```

- m - input matrix. Can be any matrix except when it's input to the log function under which condition each element must be positive (otherwise exception will be thrown).

4.1.4.6 Softmax

Compute the softmax of the input matrix for each column. The resulting matrix has the same dimensions as that of the input matrix. The syntax is

```
Softmax(m)
```

- m - input matrix.

4.1.4.7 LogSoftmax

Compute the log of softmax of the input matrix for each column. The resulting matrix has the same dimensions as that of the input matrix. The syntax is

```
LogSoftmax(m)
```

- m - input matrix.

4.1.4.8 SumElements

Calculate the sum of all elements in the input matrix. The result is a scalar (or one by one matrix). The syntax is

```
SumElements (m)
```

- m - input matrix

4.1.4.9 SumColumnElements

Calculate the sum of all elements in each column of the input matrix. The result is a row vector. The syntax is

```
SumColumnElements (m)
```

- m - input matrix

4.1.4.10 Negate

Negate each element of the matrix. The resulting matrix has the same dimension as that of the input matrix. The syntax is

```
Negate (m)
```

- m - input matrix.

4.1.4.11 Reshape

Reshape the input matrix. The resulting matrix has the same number of elements as that of the input matrix but is interpreted differently. The syntax is

```
Reshape(m, numRows, [imageWidth=], [imageHeight=], [imageChannels=]
```

- m - input matrix.
- numRows - reshape to a matrix with numRows rows.
- imageWidth, imageHeight, imageChannels - are named optional parameters that allow you to interpret each column as images with different dimensions.

4.1.4.12 RowSlice

Select a row slice from the input matrix for all columns (samples). The resulting matrix is a numRows by m.cols matrix. This function is often used to extract a portion of the input. The syntax is

```
RowSlice(startRow, numRows, m)
```

- m - input matrix.
- startRow - the start row to get a slice
- numRows - the number of rows to get

4.1.4.13 RowStack

Concatenate rows of input matrices to form a bigger matrix. The resulting matrix is a sumof(rows) by m1.cols matrix. It supports variable-length input. The syntax is

```
RowStack(m1, m2, ...)
```

- m1, m2 - input matrices. can be any number of input matrices.

4.1.4.14 Scale

Scale each element in the input matrix by a scalar value. The resulting matrix has the same dimension as that of the input matrix. The syntax is

```
Scale(scaleFactor, m)
```

- scaleFactor - floating point scalar scale factor
- m - input matrix

4.1.4.15 Times

Calculate the product of two matrices. The resulting matrix has a size of m1.rows by m2.cols. The syntax is

```
Times(m1, m2)
```

- m1, m2 - input matrices. The m1.cols must equal m2.rows.

4.1.4.16 TransposeTimes

Calculate the product $m1^T m2$. The resulting matrix has a size of m1.cols by m2.cols. The syntax is

```
Times(m1, m2)
```

- m1, m2 - input matrices. The m1.rows must equal m2.rows.

4.1.4.17 DiagTimes

Calculate the product of two matrices in which the first matrix is a diagonal matrix whose diagonal values are represented as a vector. The resulting matrix is m1.rows by m2.cols. The syntax is

```
DiagTimes(m1, m2)
```

- m1 - the diagonal matrix whose diagonal values are represented as a vector of size m2.rows by one.
- m2 - a normal input matrix. m1.rows must equal m2.rows.

4.1.4.18 Plus, Minus

Calculate the sum (Plus) and difference (Minus) of two matrices. The resulting matrices have the same dimension as that of the input matrix with larger dimension. The syntax is

```
Plus(m1, m2)
Minus(m1, m2)
```

- m1, m2 - input matrices.

4.1.4.19 ElementTimes, RowElementTimes, ColumnElementTimes,

Calculate the element-wise product of two matrices. The resulting matrices have the same dimension as that of the input matrix with larger dimension. The syntax is

```
ElementTimes(m1, m2)
RowElementTimes(m, vrow)
ColumnElementTimes(m, vcol)
```

- m1, m2, m - input matrices.
- vrow, vcol - row and column vectors

m1 and m2 must have same dimension in ElementTimes. m and vrow must have same number of columns in RowElementTimes, and m and vcol must have same number of rows in ColumnElementTimes.

4.1.4.20 SquareError or SE

Compute the sum of the squared difference between elements in the two input matrices. The result is a scalar (i.e., one by one matrix). This is often used as a training criterion node. The syntax is

```
SquareError(m1, m2)
SE(m1, m2) #deprecated
```

- m1, m2 - input matrices. Must have the same dimensions.

4.1.4.21 CrossEntropyWithSoftmax or CEWithSM

Compute the softmax for each column of the input matrix, compare the result against the ground truth labels and compute sum of the cross entropy value for all the columns (i.e., samples). The result is a scalar (i.e., one by one matrix). This is often used as a training criterion node. The syntax is

```
CrossEntropyWithSoftmax(labels, matrix)
CEWithSM(labels, matrix) #deprecated
```

- labels - the ground truth labels
- matrix - input matrix.

4.1.4.22 ClassBasedCrossEntropyWithSoftmax or CBCEWithSM

Compute the sum of the cross entropy value for all the columns (i.e., samples) using the class-based evaluation trick to reduce the computation cost. The result is a scalar (i.e., one by one matrix). This is often used as a training criterion node when the number of classes is huge (e.g., in language model). The syntax is

```
ClassBasedCrossEntropyWithSoftmax(labels, matrix)
CBCEWithSM(labels, mainInputInfo, mainWeight,
            classProbBeforeSoftmax) #deprecated
```

- **labels** - the ground truth labels. This is a 4 by T matrix, where T is the number of samples. The first row is the ground truth output id. The second row is the ground truth class id. The third and fourth rows are the start (inclusive) and end (exclusive) output ids corresponding to the ground truth class id.
- **mainInputInfo** - contains the main information to make the classification decision. It's an inputDim by T matrix. In language model, inputDim is often the hidden layer size.
- **mainWeight** - the inputDim by outputDim weight matrix, where inputDim is the number of rows in mainInputInfo and outputDim is the number of output nodes (e.g., number of words in language model). During the computation, only the column slice corresponding to the start and end output ids of the class id is used to reduce the computation.
- **classProbBeforeSoftmax** - applying softmax on this matrix will result in class probabilities. This is a numOfClass by T matrix, where numOfClass is the number of classes.

4.1.4.23 ErrorPrediction or ClassificationError

Evaluate the classification error of the predictions made by the model. It finds the index of the highest value for each column in the input matrix and compares it to the actual ground truth label. The result is a scalar (i.e., one by one matrix). This is often used as an evaluation criterion. It cannot be used as a training criterion though since the gradient is not defined for this operation. The syntax is

```
ErrorPrediction(labels, m)
ClassificationError(labels, m)
```

- **labels** - the ground truth labels
- **m** - input matrix.

4.1.4.24 CosDistance or CosDist

Evaluate the cosine distance between each column of two input matrices. The resulting matrix is a one by m1.cols row vector. The syntax is

```
CosDistance(m1, m2)
CosDist(m1, m2) #deprecated
```

- **m1, m2** - input matrices. The matrices should have same dimensions.

4.1.4.25 CosDistanceWithNegativeSamples or CosWithNegSamples

Evaluate the cosine distance between each column of two input matrices as the distance of positive samples, and the cosine distance between each column of the left matrix (m1) and *numNegSamples* of samples in the right matrix (m2) selected by shifting the column *numShifts* times as the negative samples. The resulting matrix is *numNegSamples+1* by m1.cols matrix. Often used in the deep structured semantic model (DSSM). The syntax is

```
CosDistanceWithNegativeSamples(m1, m2, numShifts,
                               numNegSamples)
CosWithNegSamples(m1, m2, numShifts, numNegSamples) #
  deprecated
```

- m1, m2 - input matrices. The matrices should have same dimensions.
- numShifts - number of shifts in m2 for choosing negative samples, a 1 by 1 matrix
- numNegSamples- number of negative samples to use in m2, a 1 by 1 matrix

4.1.4.26 MatrixL1Reg or L1Reg, MatrixL2Reg or L2Reg

Compute the L1 (MatrixL1Reg, L1Reg) or Frobenius (MatrixL2Reg, L2Reg) norm of the input matrix. The result is a scalar (i.e., one by one matrix). This is often used as regularization terms. The syntax is

```
MatrixL1Reg(m)
L1Reg(m) #deprecated
MatrixL2Reg(m)
L2Reg(m) #deprecated
```

- m - input matrix

4.1.4.27 Mean

Compute the mean vector of the input matrix by sweeping the whole dataset. The resulting matrix is a m.rows by one matrix. This operation is precomputed before the first training pass. The syntax is

```
Mean(m)
```

- m - input matrix

4.1.4.28 InvStdDev

Compute the per-dimensional (i.e., assume each dimension is independent with each other and the covariance matrix is a diagonal) inversed standard deviation vector of the input matrix by sweeping the whole dataset. The resulting matrix is a $m.rows$ by one matrix. This operation is precomputed before the first training pass. The syntax is

```
InvStdDev(m)
```

- m - input matrix

4.1.4.29 PerDimMeanVarNormalization or PerDimMVNorm

Compute the mean-variance normalized matrix for each column (i.e., sample) of the input matrix. The resulting matrix has the same dimensions as the input matrix.

```
PerDimMeanVarNormalization(m, mean, invStdDev)
PerDimMVNorm(m, mean, invStdDev) #deprecated
```

- m - input matrix than needs to be normalized
- $mean$ - the mean vector. It's a $m.rows$ by one matrix.
- $invStdDev$ - the per-dimensional inversed standard deviation vector. It's a $m.rows$ by one matrix in which all values are non-negative.

4.1.4.30 PerDimMeanVarDeNormalization or PerDimMVDeNorm

De-normalize from the mean-variance normalized input matrix for each column (i.e., sample). It is often used in the generative model where during the training the network's output node is normalized which needs to be de-normalized at the actual generation process. The resulting matrix has the same dimensions as the input matrix.

```
PerDimMeanVarDeNormalization(m, mean, invStdDev)
PerDimMVDeNorm(m, mean, invStdDev) #deprecated
```

- m - input matrix than needs to be de-normalized
- $mean$ - the mean vector. It's a $m.rows$ by one matrix.
- $invStdDev$ - the per-dimensional inversed standard deviation vector. It's a $m.rows$ by one matrix in which all values are non-negative.

4.1.4.31 Dropout

Compute a new matrix with some percentage of random elements in the input matrix set to zero. The percentage (called dropout rate) is often set as part of the training configuration (e.g., the stochastic gradient descent block we will discuss in Section ??). During evaluation the dropout rate is set to zero and this operation has no effect (i.e., just pass the input matrix through). It is commonly used to prevent overfitting during the training process or to pretrain a model. The syntax is

```
Dropout (m)
```

- m - input matrix

4.1.4.32 Convolution or Convolve

Compute the convolution of a weight matrix with an image. The resulting matrix may have different dimension depends on the parameters passed in. The syntax is

```
Convolution(w, image, kernelWidth, kernelHeight,
            outputChannels, horizontalSubsample,
            verticalSubsample, [zeroPadding=false,
                               maxTempMemSizeInSamples=0, imageLayout="HWC"|cudnn
                               "])
Convolve(w, image, kernelWidth, kernelHeight,
         outputChannels, horizontalSubsample,
         verticalSubsample, [zeroPadding=false,
                             maxTempMemSizeInSamples=0, imageLayout="HWC"|cudnn
                             "]) #deprecated
```

- w - convolution weight matrix, it has the dimensions of [outputChannels, kernelWidth * kernelHeight * inputChannels]. If w's dimensions are not specified (i.e., all are zero's) they will be automatically set by CNTK using a depth-first traversing pass.
- image - the input image.
- kernelWidth - width of the kernel
- kernelHeight - height of the kernel
- outputChannels - number of output channels
- horizontalSubsample - subsamples (or stride) in the horizontal direction. In most cases this should be set to 1.

- `verticalSubsample` - subsamples (or stride) in the vertical direction. In most cases this should be set to 1.
- `zeroPadding` - [named optional] specify whether the sides of the image should be padded with zeros. Default is false. When it's true, the convolution window can move out of the image.
- `maxTempMemSizeInSamples` - [named optional] maximum amount of memory (in samples) that should be reserved to do matrix packing. Default is 0 which means the same as the input samples.
- `imageLayout` - [named optional] the storage format of each image. By default it's "HWC", which means each image is stored as [channel, width, height] in column major. If you use cuDNN to speed up training, you should set it to cudnn, which means each image is stored as [width, height, channel].

4.1.4.33 MaxPooling, AveragePooling

Computes a new matrix by selecting the maximum (MaxPooling) or computing the average (AveragePooling) value in the pooling window. This is used to aggregate information from the input and will reduce the dimensions of a matrix. The syntax is

```
MaxPooling(m, windowWidth, windowHeight, stepW, stepH,
           imageLayout="HWC" | cudnn")
AveragePooling(m, windowWidth, windowHeight, stepW, stepH,
              imageLayout="HWC" | cudnn")
```

- `m` - input matrix
- `windowWidth` - width of the pooling window
- `windowHeight` - height of the pooling window
- `stepW` - step (or stride) used in the width direction
- `stepH` - step (or stride) used in the height direction
- `imageLayout` - [named optional] the storage format of each image. By default it's "HWC", which means each image is stored as [channel, width, height] in column major. If you use cuDNN to speed up training, you should set it to cudnn, which means each image is stored as [width, height, channel].

4.1.4.34 PastValue (or Delay) and FutureValue

Used to get the past or future value of a node. It is most often used to create recurrent networks. The resulting matrix has the same dimension as that of the input matrix. The syntax is

```
PastValue(rows, cols, m, [timeStep=1, defaultHiddenActivity
    =0.1])
Delay(rows, cols, m, [delayTime=1, defaultPastValue=0.1]) #
    deprecated
FutureValue(rows, cols, m, [timeStep=1,
    defaultHiddenActivity=0.1])
```

- rows - the number of rows in the input matrix. This parameter is needed because under some loopy conditions the dimensions cannot be automatically inferred from the input matrix.
- cols - the number of columns in the input matrix.
- m - input matrix from which the past or future value is obtained. Each column is a sample. The samples may be from different utterances as explained in Chapter .2
- timeStep, delayTime - [named optional] the amount of time steps look into past or future. Default is 1.
- defaultHiddenActivity, defaultPastValue - [named optional] the default value to use if the past or future values are not available. Default is 0.1.

4.2 Model Editing Language

The model editing language (MEL) of the CNTK provides a means to modify both the structure and the model parameters of an existing trained network using a set of provided commands. It provides a number of functions to modify the network and can use network description language (NDL) to define new elements. MEL is very important since it allows users, for example, to train a network with one configuration and later use it as part of another network designed for another purpose. It also allows users to do discriminative pretraining [3, 4] on DNNs by building shallow networks first and then inserting new layers one on top of another. To use MEL, you need to use the “edit” command in the config file.

It looks similar to a scripting language in syntax, but give a simple way to modify an existing network. This network must have been defined in a format that

CNTK can read, currently only the CNTK computational network disk format is supported.

4.2.1 Basic Features

In this section we cover the basic features of the MEL by the following example.

```

modell1 = LoadModel("c:\models\mymodel.dnn", format=cntk)
SetDefaultModel(modell1)

DumpModel(modell1, "c:\temp\originalModel.dmp", includeData =
    true)

#create another hidden layer
Copy(L3.*, L4.*, copy="all")

#Hook up the layer
SetInput(L4.*.T, 1, L3.RL) # Layer 3 output to Layer 4 input
SetInput(CE.*.T, 1, L4.RL) # Layer 4 output to Top layer
    input

#Add mean variance normalization using in-line NDL
meanVal = Mean(features)
invstdVal = InvStdDev(features)
inputVal = PerDimMeanVarNormalization(features,meanVal,
    invstdVal)

#make the features input now take the normalized input
SetInput(L1.BFF.FF.T, 1, inputVal)

#save model
SaveModel("c:\models\mymodel4HiddenWithMeanVarNorm.cn")

```

This MEL script is using a network that was defined originally by the NDL script

```

# constants defined
# Sample, Hidden, and Label dimensions
SDim=784
HDim=256
LDim=10

features=Input(SDim, tag="feature")
labels=Input(LDim, tag="label")

```

```
# Layer operations
L1 = RBFF(features, HDim, SDim)
L2 = RBFF(L1, HDim, HDim)
L3 = RBFF(L2, HDim, HDim)
CE = SMBFF(L3, LDim, HDim, labels, tag="criterion")
Err=ErrorPrediction(labels, CE.F, tag="eval")

# rootNodes defined here
OutputNodes=(CE.F)
```

4.2.1.1 Loading and Setting Default Models

The first command of a MEL script is usually a `LoadModel()` command. This function takes the name of a model file on disk, and an optional parameter specifying the format of the model file. Currently only CNTK format model files are accepted, and CNTK format is the default value. Programmers can write file converters to support more model formats.

```
modell = LoadModel("c:\models\mymodel.cn", format="cntk")
SetDefaultModel(modell)
```

Here *modell* is the name this model is within the MEL script. This identifier is used in the next line to set this model as the default model. The default model defines the model that will be assumed in all name references within the script, where a model name is required but not specified, and the model to which any NDL commands will apply. If no model has been explicitly set to be the default model, the last loaded or created model is used as a default. However, It is recommended that the `SetDefaultModel()` command be used to make it explicit.

4.2.1.2 Viewing a Model File

It is often necessary to view a model file to determine the names used in the model file. MEL uses the node names in most commands, to specify which node(s) should be modified. The `Dump()` command dumps the node names and optionally values to a readable file. The parameters are the model name, the file name, and if the dump should include data. The `includeData` optional parameter defaults to false.

```
DumpModel(modell, "c:\temp\originalModel.dmp", includeData =
    true)
```

The dump looks something like this:

...

```

features=InputValue [784,32]
L1.BFF.B=LearnableParameter [256,1] NeedGradient=true
0.0127850091
-0.00473949127
0.0156492535
...
0.00529919751
#####

L1.BFF.FF.P=Plus ( L1.BFF.FF.T , L1.BFF.B )
L1.BFF.FF.T=Times ( L1.BFF.W , normInput )
L1.BFF.W=LearnableParameter [256,784] NeedGradient=true
0.0174789988 0.0226208009 -0.00648776069 0.0346485041
-0.0449098013 -0.0233792514
0.0154407881 0.000157605857 0.0206625946 0.0491085015
0.00128563121
...

```

4.2.1.3 Copy Nodes

The copy command will copy a node, or a group of nodes from one location to another location. This can be done within the same model, or between different models:

```

#create another hidden layer
Copy(L3.*, L4.*, copy=all)

```

The first parameter is the source of the copy and must exist, the second is the target and may or may not exist. If it does exist, those matching nodes will be overwritten by the copy. The optional parameter **copy** can be used to change this behavior, the options are: **all** - the default, which copies all node data and links, or **value** - which copies the node values only, leaving the connections between nodes (if any) unchanged.

Since the L3 used in this copy command was originally defined in NDL as

```
L3 = RBFF(L2, HDim, HDim)
```

The new L4 layer will contain all the nodes L3 contains (RectifiedLinear, Plus, Times, W and B Parameters) all connected just as they were in the original L3 layer.

4.2.1.4 SetInput

To integrate this new layer into the model, the inputs and outputs of the nodes must reset. After the copy any node whose connected nodes were not copied will have those connections set to an invalid value. These need to be fixed in order to have a valid model. Before a model can be saved CNTK first checks to see if all nodes are correctly connected.

You can change connections between nodes with the `SetInput()` command. This command takes a node to modify, the input number (zero-based) to modify, and the new value for that input. The following commands hook up the inputs and outputs for our copied nodes:

```
#hook up the layer
SetInput(L4.*.T, 1, L3.RL) # Layer 3 output to Layer 4 input
SetInput(CE.*.T, 1, L4.RL) # Layer 4 output to Top layer
input
```

To connect our new L4 layer, we need to set the second input of the Times node (`L4.BFF.FF.T`) to `L3.RL`, which is the output of the L3 layer. The input number is zero-based, so the first input is zero and the second input would be '1'. Likewise we need to hook the output of the L4 layer nodes to the input of the top layer. Once again this ends up being a Times node (`CE.BFF.FF.T`).

4.2.1.5 Adding New Nodes: In-line NDL and NDL Snippets

Adding new nodes to an existing model can be done just as a model was originally defined in NDL. There are two ways to do this, the simplest is to just type the NDL definitions into the MEL script, as if it was NDL, like so:

```
#Add mean variance normalization using in-line NDL
meanVal = Mean(features)
invstdVal = InvStdDev(features)
inputVal = PerDimMeanVarNormalization(features,meanVal,
invstdVal)
```

This is called in-line NDL and can be used for most tasks. The new nodes will be placed in the current default model in the MEL script. Note that the variable `features` used in the NDL is actually a node from the default model. In-line NDL may use node names from the default model as parameters, and MEL commands may use NDL symbols as parameters. There are a number of restrictions in using in-line NDL:

- Only fully quantified node names are accepted.

- NDL symbols only apply to the default model at the time they were created when used in MEL commands.
- Macros may not be defined in in-line NDL (though they can in an NDL snippet)
- Only macros defined in the default macro file referenced in the config file, or macros defined in an NDL snippet in the MEL Script may be used.
- NDL will be processed when the next MEL command that requires it to be processed is encountered. It is only at this time that the new nodes are fully created. If forward references are used to variables, they must be resolved before the next MEL command that requires the variables to be resolved.

Using NDL Snippet is another way. NDL snippets are sections of NDL definitions that generate new nodes. Any NDL construct that is legal in an NDL script can be used. This includes defining macros and other advanced NDL features. Inside the snippets wildcard naming and use of symbols from another model are not allowed. The syntax for defining an NDL snippet are as follows:

```
[modelName]=[
#ndl commands go here
]
```

Upon the completion of the snippet, the modelName will be the name of the newly defined model. This model need not be fully defined. For example, the special nodes (i.e. criteria nodes) do not need to be defined in the model. However, all referenced variables must be defined in the snippet. It is often easier to use in-line NDL to define new nodes in MEL, and NDL Snippets to define any macros. Macros are defined in a global namespace and can be defined in any model and used from any other model. One possible use of an NDL snippet is to define an entirely new model, and then use MEL to populate the new model with values.

4.2.1.6 SaveModel

After the model edits are complete, it's time to save the model:

```
#save model
SaveModel("c:\models\mymodel4HiddenWithMeanVarNorm.cn")
```

This command saves the default model to the path name specified. Alternatively, you can specify the model name as the first parameter with the path as the second to make the model name explicit. Before the save happens the model is validated to ensure it is a valid model. Should there be an error in the model, an error message will be displayed on the console and the model edit will terminate.

4.2.1.7 Name Matching

You may have noticed the use of the ‘*’ wildcard character in the commands presented to this point. Those are name matching wildcards, and are useful in matching a group of related nodes. Because of the hierarchical dot-naming scheme used by NDL, it is easy to select all the nodes that a particular macro generated because they will all start with the same prefix. Nodes generated by NDL macros have the following structure:

```
[name] { . [macroName] } . [nodeName]
```

Where *name* is the name assigned in NDL, *macroName* is the name given to a macro called by the initial macro, and can be several layers deep, and *nodeName* is the name given to a single node in the final macro. For example, this macro in NDL

```
L3 = RBFF (L2, HDim, HDim)
```

generates the following nodes:

- L3.RL: RectifiedLinear node
- L3.BFF.B: Parameter node - used for bias
- L3.BFF.W: Parameter node - used for weight
- L3.BFF.FF.T: Times node
- L3.BFF.FF.P: Plus node

These wildcard patterns can be used to access these nodes:

- L3.*: Select all the L3 nodes
- L3.*.P: Select the L3.BFF.FF.P node
- L3.*: All the L3 nodes in the model

4.2.2 MEL Command Reference

This section contains the currently implemented MEL Command functions.

4.2.2.1 CreateModel, CreateModelWithName

Creates a new empty model. The syntax is

```
m=CreateModel()
CreateModelWithName(m)
```

- m - the name of newly created model

4.2.2.2 LoadModel, LoadModelWithName

Load a model from a disk file and assign it a name. The syntax is

```
m=LoadModel(modelFileName, [format="cntk"])
LoadModelWithName(m, modelFileName, [format="cntk"])
```

- m - the name of loaded model
- modelFileName - name of the model file, can be a full path name. If it contains spaces, it must be enclosed in double quotes.
- Currently only the native CNTK format of model file is accepted. Other formats may be supported in the future.

4.2.2.3 SaveDefaultModel, SaveModel

Save a model to disk in the specified model format.

```
SaveDefaultModel(modelFileName, [format="cntk"])
SaveModel(m, modelFileName, [format="cntk"])
```

- m - the name of the model to save
- modelFileName - name of the model file, can be a full path name. If it contains spaces, it must be enclosed in double quotes.
- Currently only the native CNTK format of model file is accepted. Other formats may be supported in the future.

4.2.2.4 UnloadModel

Unload the specified model from memory. The syntax is

```
UnloadModel (m)
```

- m- the name of the model to unload.

In general it is unnecessary to unload a model explicitly since it will happen automatically at the end of the MEL script. It is also not recommended that you reuse a model identifier after unloading a model.

4.2.2.5 LoadNDLSnippet

Load an NDL Snippet from a file, and process it, assigning the results to a name. The syntax is

```
LoadNDLSnippet (m, nsdSnippetFileName, [section])
```

- m- the name of the model that the snippet will be applied to.
- ndlSnippetFileName - name of the file that contains the snippet we want to load.
- section - [named optional] name of the section that contains the snippet we want to load. If the entire file is the snippet no section name should be specified. Default is the first section appear in the file.

4.2.2.6 Dump, DumpModel

Dump the contents and structure of a model to a file. These two functions mean the same thing. The syntax is

```
Dump (m, dumpFileName, [includeData=true|false])  
DumpModel (m, dumpFileName, [includeData=true|false])
```

- m- name of the model to dump.
- dumpFileName- name of the file that we want to save the output.
- includeData - [named optional] if set to true the contents of the nodes that contain matrix values will also be dumped. Default is false.

4.2.2.7 DumpNode

Dump the contents and structure of a node to a file. The syntax is

```
DumpNode(node, dumpFileName, [includeData=true|false])
```

- node - node name, a wildcard name may be used to output multiple nodes in one call
- dumpFileName- name of the file that we want to save the output.
- includeData - [named optional] if set to true the contents of the nodes that contain matrix values will also be dumped. Default is false.

4.2.2.8 Copy, CopyNode

Copy a node, or a group of nodes from one location to another location. This can be done within the same model, or between different models. The copy can create new nodes or overwrite/update existing nodes. The network structure can be copied with multiple nodes, or just the values in the nodes. The syntax is

```
Copy(fromNode, toNode, [copy="all"|"value"])
CopyNode(fromNode, toNode, [copy="all"|"value"])
```

- fromNode - node identifier we are copying from. This can also be a wildcard pattern.
- toNode - node identifier we are copying to. This can also be a wildcard pattern, but must match the fromNode pattern. A copy from a single node to multiple nodes is also permitted.
- copy - [named optional] specifies how the copy will be performed. Default is all.

	if destination node exists	if destination node does not exist
all	Copies over the values of the nodes and any links between them overwriting the existing node values. Any node inputs that are not included in the copy set will remain unchanged.	Copies over the values of the nodes and any links between them creating new nodes. All nodes that include inputs in the copy set will still be connected. All other nodes will have no inputs and will need to be set using SetInput()
value	Copies over the node contents, the node inputs will remain unchanged	Not a valid option, the nodes must exist to copy only values.

4.2.2.9 CopySubTree

Copy all nodes in a subtree of a computational network from one location to another location. This can be done within the same model, or between different models. The syntax is

```
CopySubTree(fromRootNode, toRootNode, [copy="all"|"value"])
```

- fromRootNode - node identifier we are copying from. This can also be a wildcard pattern.
- toRootNode - node identifier we are copying to. This can also be a wildcard pattern, but must match the fromRootNode pattern.
- copy - [named optional] specifies how the copy will be performed. See the Copy and CopyNode command for details. Default is all.

If the fromRootNode is a wildcard pattern then the toRootNode must also be a similar wildcard pattern. The CopySubTree() command will execute separately for each root node.

4.2.2.10 SetInput or SetNodeInput

Set an input (child) of a node (i.e., operand of an operator) to a value. The syntax is

```
SetInput(node, inputNumber, inputNode)
```

- node - node whose input will be set. This can also be a wildcard pattern.
- inputNumber - a zero-based index to the input that will be set.
- inputNode - node identifier for input node. This must be a single node.

4.2.2.11 SetInputs or SetNodeInputs

Set all the inputs (children) of a node (i.e., operands of an operator). If only one input needs to be set use the SetInput() command instead. The syntax is

```
SetInputs(node, inputNode1[, inputNode2, inputNode3])
```

- node - node whose input we are modifying .
- inputNode1, inputNode2, inputNode3 - node identifier for input node. The number of input parameters must match the number of inputs the referenced node requires.

4.2.2.12 SetProperty

Set the property of a node to a specific value. The syntax is

```
SetProperty(node, propertyName, propertyValue)
```

- node - the node whose properties will be set
- propertyName - property name to modify.
- propertyValue - the new property value.

The acceptable property names and property values are as follows:

- computeGradient or needsGradient=truelfalse: A flag that determine if a node participates in gradient calculations. Applies to Parameter nodes
- feature=truelfalse: Sets the node as a feature input. Applies to input nodes.
- label=truelfalse: Set the node as a label input. Applies to input nodes.
- finalCriterion or criterion=truelfalse: Sets the node as one of the criteria nodes of the network.
- evaluation or eval=truelfalse: Set the node as one of the evaluation nodes.
- output=truelfalse: Set the node as one of the output nodes.

4.2.2.13 SetPropertyForSubTree

Set the property of a node to a specific value. The syntax is

```
SetProperty(rootNode, propertyName, propertyValue)
```

- `rootNode` - the node at the root of the subtree
- `propertyName` - property name to modify.
- `propertyValue` - the new property value.

The acceptable property names and property values for this command are as follows:

- `ComputeGradient` or `NeedsGradient=true/false`: A flag that determine if a node participates in gradient calculations. Applies to Parameter nodes

4.2.2.14 Remove or RemoveNode, Delete or DeleteNode

Delete or Remove node(s) from a model. All alternate commands perform the same operation. The syntax is

```
Remove(node, [node2, node3]) #deprecated
Delete(node, [node2, node3]) #deprecated
RemoveNode(node, [node2, node3])
DeleteNode(node, [node2, node3])
```

- `node` - the node to be removed. This can be a wildcard name.
- `node2, node3` - additional optional nodes that will also be removed, These can be wildcards

This command can leave unconnected nodes in a model that would need to be reconnected using the `SetInput()` or `SetInputs()` commands.

4.2.2.15 Rename

Rename a node. Note that this only changes the name of a node (e.g., making it easier to understand and to reference) but not how it connects with other nodes. The syntax is

```
Rename(oldNodeName, newNodeName)
```

*CHAPTER 4. ADVANCED SETUPS IN COMPUTATIONAL NETWORK TOOLKIT*118

- oldNodeName- the old node name. Wildcard naming may be used.
- newNodeName- the new node name. Matching wildcard naming may be used if oldNodeName contains wildcards.

Chapter 5

Speech Recognition Decoder Integration

5.1 Argon Speech Recognition Decoder

Argon is a dynamic decoder modeled after (Soltau & Saon, “Dynamic Network Decoding Revisited,” ASRU, 2009). To use Argon, one must supply a representation of the acoustic search space in the form of a weighted finite state acceptor that represents all possible word sequences and their associated context-dependent HMM state sequences. This is $H^{\circ}C^{\circ}\mathcal{L}$ of Mohri et al, (“Weighted Finite State Transducers in Speech Recognition,” Computer Speech & Language, 2002) . As a pre-processing step, Argon will convert the AT&T format acceptor into a binary image for its own internal use. Additionally, one supplies an acoustic model as trained by CNTK, a source of acoustic features, and a language model. Optionally, a set of language models and their interpolation weights may be specified.

As output, Argon produces one-best word hypotheses and lattices. Lattices are produced with state-level alignments, and the one-best hypotheses can optionally be produced with state-level alignments as well.

Argon is a token-passing decoder. It works by maintaining a set of tokens, each of which represents a partial path. Each token maintains a language model history state, a pointer to its location in the decoding graph, and a path score. At each time frame, all the tokens from the previous frame are advanced over emitting arcs in the decoding graph, and then across word arcs. When a token finishes a word (i.e. advances across a word arc), the language model is consulted, and the path score is adjusted with the language model probability.

Argon uses a simple form of language model lookahead. By default, the search space $H^{\circ}C^{\circ}\mathcal{L}$ includes the *most optimistic* log-probability that a word can have,

given the language model. When a word-end is encountered, the remainder of the language model log-probability is added. To prevent over-pruning at these jumps, we employ the notion of *language-model slack* in pruning. Each token maintains a slack variable, and when a language model score is encountered, the slack variable is incremented by this amount. It is also multiplied by a number slightly less than one at each frame so that it decreases exponentially. Tokens that fall outside the beam by less than the slack amount are not pruned. This effectively smooths the effect of adding language model scores at word ends.

5.1.1 Representing the Acoustic Search Space

Argon represents the acoustic search space as a determinized, minimized finite state transducer, encapsulating $H^\circ C^\circ \mathcal{L}$. Input arcs are labeled with context dependent HMM states, and output arcs with words. Every path from the start state to one of the final states represents a sequence of words beginning with “begin-of-sentence” or <s> and ending with “end-of-sentence” or </s>. The sequence of input labels encountered on such a path represents the sequence of context-dependent HMM states of the words, as determined by the lexicon and acoustic context model. A fragment of a search graph is shown in Figure \ref{fig:dcdgraph}.

Minimization of a transducer repositions the output labels in the process of minimizing the transducer size. This necessitates re-aligning the output to synchronize the word labels with the state sequence. Rather than allowing arbitrary label shifting, for all words that have two or more phonemes, Argon outputs word labels immediately before the last phone. This allows for a high degree of compression, good pruning properties, and predictable label positions.

Since $H^\circ C^\circ \mathcal{L}$ does not represent the language model, we have found it unnecessary to accommodate <epsilon> arcs in the decoding graph, thus somewhat simplifying the decoder implementation. Thus, it is a requirement that $H^\circ C^\circ \mathcal{L}$ contain only HMM states and word labels. Our reference scripts achieve this with the OpenFST remove-epsilon operation.

5.1.1.1 From HTK Outputs to Acoustic Search Space

CNTK and Argon use acoustic models based on HTK decision trees. In the first part of the acoustic model building process, context-independent alignments are made with HTK, and a decision tree is built to cluster the triphone states. After a couple of passes of re-alignment, a model file and state level alignments are written out. CNTK uses the alignments to train a DNN acoustic model.

The simplest way to build an acoustic model is to provide an HTK decision tree file, a lexicon, a file with pronunciation probabilities, and a model MMF file. The

MMF file is only used to extract transition probabilities, and as an option, these can be ignored, in which case the MMF is unnecessary. We provide sample scripts which process the lexicon and decision tree to produce $H^\circ C^\circ \mathcal{L}$.

Our sample script imposes two constraints:

- 1) 3-state left-to-right models except for silence
- 2) The standard HTK silence model, which defines a “short-pause” state and ties it to the middle state of silence

In the output, short-pause and silence are optional between words. All hypotheses start with <s> and end with </s>, both of which are realized as silence.

5.1.2 Representing the Language Model

The language model is dynamically applied at word-ends. Language model lookahead scores are embedded in the graph, and compensated for at word ends, when multiple language models may be consulted and interpolated. In the case that multiple language models are used, they are listed in a file, along with their interpolation weights; this file is given to Argon as a command-line parameter.

5.1.3 Command-Line Parameters

5.1.3.1 WFST Compilation Parameters

These parameters are used in the graph pre-compilation stage, when a fsm produced by a tool such as OpenFST is converted into a binary image that is convenient to read at runtime.

`[-infsm (string:)]:` gzip'd ascii fsm to precompile

`[-lookahead-file (string:)]:` file with the lookahead scores for each word, if lookahead was used

`[-precompile (bool: false)]:` precompile the graph? true indicates that precompilation should be done

`[-transition-file (string:)]:` transition probability file for use in precompilation

`[-graph (string)]:` decoding graph to read/write

5.1.3.2 Parameters specifying inputs

These parameters are related to the inputs used in decoding. Their use is exemplified in the sample scripts.

`[-infiles (string:)]:` file with a list of mfcc files to process

`[-graph (string)]:` decoding graph to read/write

`[-lm (string:)]:` language model to use in decoding; use when there is one LM only

[-lm-collection (string:)]: file specifying a set of language models to use for decoding
 [-sil-cost (float: 1.0)]: cost charged to silence
 [-am (string:)]: binary acoustic model to use in decoding
 [-cntkmap (string:)]: mapping from output names in decoding graph to dnn output nodes
 [-dnn_window_size (int: 11)]: number of consecutive acoustic vectors to stack into one context-window input, depends on acoustic model

5.1.3.3 Parameters specifying outputs

These parameters specify the outputs that Argon creates at runtime.

[-outmlf (string:)]: store recognition results in HTK-style master label format file
 [-detailed-alignment (bool: false)]: enables state-level recognition output
 [-lattice-prefix (string:)]: path and filename prefix for output lattices; will be post-pended with a number

5.1.3.4 Search Parameters

These parameters control Argon's search

[-acweight (float: 0.0667)]: acoustic model weight. language model weight is always 1
 [-beam (float: 15)]: absolute beam threshold
 [-alignmlf (string:)]: source transcription for forced alignment
 [-max-tokens (int: 20000)]: number of tokens to keep at each time frame
 [-word_end_beam_frac (float: 0.33333)]: beam at word ends is this times the normal beam

5.2 Constructing a Decoding Graph

As described above, the decoding graph is a representation of how sequences of acoustic model states form sequences of words, with associated path scores that are a combination of the acoustic model state transition probabilities and the language model lookahead scores. This section describes two different decoding graphs: a simple, context-independent phonetic recognition graph suitable for TIMIT recognition, and a more complex, cross-word triphone large vocabulary graph suitable for Wall Street Journal or Aurora-4 recognition tasks.

The decoding graph includes scores that represent the most optimistic score that can be achieved by a word. Scripts extract that score from an ARPA language

model, and put it in a file that lists the most optimistic score for each word. The optimistic scores are a simple context-independent form of language model lookahead. A smoothing mechanism is used internally in the decoder to “smear” the residual LM score across multiple frames after a word end.

5.2.1 Context-Independent Phonetic Recognition

To build the TIMIT graph, only three input files are needed: the model state map, the state transitions, and a language model.

The scripts assume each context-independent phone is represented by a three state, left to right, hidden markov model. The names of these states should be in a “model state map” file that has one line for every model. The first column is the name of the model, and subsequent columns are the names of the states, in left to right order. The transition probabilities between these states are stored in a separate “state transitions” file. Every state of the acoustic model should have one line, whose first column is the name of the state, and subsequent columns are the natural logarithm of the self and forward transition probabilities.

The language model should be an ARPA-style language model over the phone model names. Additionally, it should include the unknown word <UNK>, and the beginning and ending sentence markers <s> and </s>. For TIMIT data, they should represent the h# at the beginning and the h# at the end of every utterance, respectively, and h# should not exist in the language model.

The DecodingGraph.mak file demonstrates the process of turning these inputs into a proper Argon decoding graph. When the makefile is successfully run, the current working directory should contain a decoding graph file i_lp.final.dg that is suitable for phonetic decoding.

5.2.2 Cross-Word Context-Dependent Large Vocabulary Decoding

Building this style of graph is somewhat more involved, and divided into two stages. In the first stage, information about the graph structure is extracted from a donor HTK format acoustic model, and put into a distilled representation. In the second stage, this representation is used to construct the FST and compile it into Argon decoding graph format. We imagine that specialized first stages can be constructed for different styles of input, other than HTK format donor models.

5.2.2.1 Stage One: Extract structure from HTK donor model

The prerequisites for this process are as follows:

A pronunciation lexicon, an ARPA-format ngram language model, a senone map file, and an ASCII format HTK donor model, with associated list of all possible logical model names and their associated physical model names.

5.2.2.2 Stage Two: Build and Compile Graph

The L transducer is built from the normalized lexicon. The H transducer is built from the triphone to senone mapping file. The C FST is made to cover all possible triphone contexts that might occur on the input side of L.

H, C, and L are composed, determinized and minimized. Then, the FST is made into an FSA by moving every non-epsilon output token onto its own arc on the same path. The silence models between words are made optional, the language model lookahead scores are applied, and weight pushing is used to take these scores as early as possible on each path through the FSA.

The end result is a compiled graph, where every path represents a sequence of acoustic model states interspaced with the word sequence they represent. The score of the path is equal to the individual HMM transition scores combined with the language model lookahead scores.

5.2.3 Running Argon

To decode, the following parameters to Argon should be specified: `-graph`, `-lm`, `-am`, `-cntkmap`, `-infile`, and `-outmlf`. See above for descriptions of these parameters.

The decoder uses a Viterbi beam search algorithm, in which unlikely hypotheses are pruned at each frame. The `-beam` parameter prevents unlikely hypotheses from being pursued. Any hypothesis that differs from the best hypothesis by more than this amount will be discarded. The `-max-tokens` parameter controls the number of active hypotheses. If the `-beam` parameter causes more than `max-tokens` hypotheses to be generated, then only this many of the best hypotheses will be retained. Decreasing either of these parameters will speed up the search, but at the cost of a higher likelihood of search errors. Increasing these parameters has the opposite effect.

The `-graph` parameter tells Argon which compiled decoding graph should be used. The `-lm` should indicate an ARPA format ngram language model.

The `-am` and `-cntkmap` should point to the final CNTK compiled acoustic model, and to its associated mapping file.

The `-acweight` parameter controls the relative weight of the acoustic model scores to the language and HMM transition scores. Increasing this value will cause more weight to be placed on the acoustic model scores. The useful range for this parameter is between 0.1 and 2.0.

The `-infiles` parameter lists the files to be decoded. It is expected that these files are in HTK format and are ready to be fed into the input layer of the CNTK acoustic model.

To get a more detailed state-level alignment, specify `-detailed-alignment true`.

Chapter 6

Example Setups

6.1 Acoustic Model

In this section we describe how CNTK can be used to build several standard models that can be used for acoustic modeling for speech recognition applications. All examples are based on the TIMIT corpus for phonetic recognition but can easily be modified for use for large vocabulary continuous speech recognition. The only significant change is that context-independent phonetic states used in the TIMIT example would be replaced by context-dependent senone targets for large vocabulary tasks. We note that these examples are not meant to be representative of state of the art performance, but rather to demonstrate how CNTK can be used in a variety of speech recognition applications. All examples are located in the ExampleSetups folder (ExampleSetups\ASR\TIMIT)

6.1.1 Training a DNN with SimpleNetworkBuilder

The simplest way to create an acoustic model with CNTK is to use the SimpleNetworkBuilder. This network builder constructs a fully-connected feed-forward network of user-specified depth and size. The configuration file is shown below. As you can see there are several key blocks of data specified

- *SimpleNetworkBuilder*: the overall network topology is specified here using the `layerSizes` parameter. In this example, the network has 792 inputs (an 11-frame context window of a 72-dimensional feature vector), 3 hidden layers of 512 sigmoidal neurons and 183 outputs, corresponding to the TIMIT phoneme states (3 x 61 phonemes). The cross entropy is the criterion for training and the frame error rate will also be monitored during training using the `evalCriterion` parameter. The input data will be mean and variance

normalized since `applyMeanVarNorm` has been set to `true`. In addition, if `needPrior` is set to `true`, the prior probabilities of the labels will be computed and a `ScaledLogLikelihood` node in the network will be automatically created. This is important if this network will be used to generate acoustic scores in a speech recognition decoder.

- *SGD*: this block specifies the parameters for stochastic gradient descent optimization. In this example, a total of 25 epochs will be run using a fixed learning schedule, with a learning rate of 0.8 for the first epoch, 3.2 for the next 14 epochs, and then 0.08 for all remaining epochs. A minibatch size of 256 will be used for the first epoch, and all remaining epochs will use a minibatch size of 1024. A momentum value of 0.9 will also be used.
- *reader*: this example uses the `HTKMLFReader`. For `SimpleNetworkBuilder`, the inputs must be called “features” and the output labels must be called “labels”. The `scpFile` contains the list of files to be processed and the `mlfFile` contains the labels for these files. More details about the `HTKMLFReader` can be found in Section 1.4.2.

```
TIMIT_TrainSimple=[
  action="train"
  modelPath="$ExpDir$\TrainSimpleNetwork\model\cntkSpeech.
    dnn"
  deviceId=$DeviceNumber$
  traceLevel=1
  SimpleNetworkBuilder=[
    layerSizes=792:512*3:183
    trainingCriterion="CrossEntropyWithSoftmax"
    evalCriterion="ErrorPrediction"
    layerTypes="Sigmoid"
    initValueScale=1.0
    applyMeanVarNorm=true
    uniformInit=true
    needPrior=true
  ]
  SGD=[
    epochSize=0
    minibatchSize=256:1024
    learningRatesPerMB=0.8:3.2*14:0.08
    momentumPerMB=0.9
    maxEpochs=25
  ]
  reader=[
```



```

readerType="HTKMLFReader"
readMethod="rollingWindow"
miniBatchMode="partial"
randomize="auto"
verbosity=1

features=[
  dim=792
  scpFile="$ScpDir$\TIMIT.train.scp.fbank.fullpath"
]

labels=[
  mlfFile="$MlfDir$\TIMIT.train.align_cistate.mlf.cntk"
  labelDim=183
  labelMappingFile="$MlfDir$\TIMIT.statelist"
]
]
]

```

6.1.2 Adapting the learning rate based on development data

In addition to using a fixed learning schedule, CNTK can adjust the learning rate based on the performance on a held out development set. To do this, the development set needs to be specified using a second data reader known as a “cvReader” and the appropriate learning rate adjustment parameters need to be specified using an “AutoAdjust” configuration block within the SGD configuration block. For example, the following cvReader could be added to the configuration file in the previous example as follows:

```

cvReader=[
  readerType="HTKMLFReader"
  readMethod="rollingWindow"
  miniBatchMode="partial"
  randomize="auto"
  verbosity=1

  features=[
    dim=792
    scpFile="$ScpDir$\TIMIT.dev.scp.fbank.fullpath"
  ]

  labels=[
    mlfFile="$MlfDir$\TIMIT.dev.align_cistate.mlf.cntk"
    labelDim=183
  ]
]

```

```

    labelMappingFile="$MlDir$\TIMIT.statelist"
  ]
]

```

The learning rate adjustment is specified within the SGD block as follows.

```

SGD=[
  epochSize=0
  minibatchSize=256:1024
  learningRatesPerMB=0.8:3.2*14:0.08
  momentumPerMB=0.9
  maxEpochs=25

  AutoAdjust=[
    autoAdjustLR="adjustAfterEpoch"
    reduceLearnRateIfImproveLessThan=0
    loadBestModel=true
    increaseLearnRateIfImproveMoreThan=1000000000
    learnRateDecreaseFactor=0.5
    learnRateIncreaseFactor=1.382
  ]
]

```

In this example, the learning rate will be reduced by a factor of 0.5 if the error on the held out data gets worse. While the learning rate can also be increased based on the performance on a held-out set, this is effectively turned off by setting the criterion for increasing the learning rate to a high value.

6.1.3 Training a DNN with NDLNetworkBuilder

While standard feedforward architectures are simple to specify with SimpleNetworkBuilder, NDLNetworkBuilder can be used when alternative structures are desired. In this case, the SimpleNetworkBuilder configuration block in the previous example is replaced with an NDLNetworkBuilder block.

```

NDLNetworkBuilder=[
  ndlMacros="$NdDir$\default_macros.ndl"
  networkDescription="$NdDir$\classify.ndl"
]

```

In this example, a file contained several NDL macros will be loaded and then the NDL file containing the actual network description will be loaded. Note that macros can be defined in both files. For example, commonly used macros used across of variety of experiments can be specified by ndlMacros while macros specific to a particular network description can be specified within the networkDe-

scription file. This example creates the exact same network as the first case, but demonstrates how it can be done using NDL.

One key thing to note is that when NDL is used, there are no longer restrictions on the names for the network’s inputs and outputs. In this case, the inputs are associated with a node called “myFeatures” and the labels are associated with a node called “myLabels”. Note that these new node names need to be used in the *reader* block in the main configuration file.

6.1.4 Training an autoencoder

Training an autoencoder is straightforward using either SimpleNetworkBuilder or NDLNetworkBuilder by using the SquareError criterion node rather than the CrossEntropyWithSoftmax criterion node. In this example, the network is constructed with NDL. Below is a snippet from the NDL file for this example. This autoencoder has three hidden layers including a middle bottleneck layer of 64 neurons. A macro is defined to perform mean and variance normalization and it is applied to both the input and target features. Also,

```
load=ndlMacroDefine
run=ndlCreateNetwork

ndlMacroDefine=[
  MeanVarNorm(x)
  {
    xMean = Mean(x)
    xStdDev = InvStdDev(x)
    xNorm=PerDimMeanVarNormalization(x,xMean,xStdDev)
  }

  MSEBFF(x,r,c,labels)
  {
    BFF=BFF(x,r,c)
    MSE=SquareError(labels,BFF)
  }
]
ndlCreateNetwork=[
  featInDim=792
  featOutDim=792
  hiddenDim=512
  bottleneckDim=64
  featIn=Input(featInDim,tag="feature")
  featOut=Input(featOutDim,tag="feature")
```

```

featNormIn = MeanVarNorm(featIn)
featNormOut = MeanVarNorm(featOut)
L1 = SBFF(featNormIn,hiddenDim,featInDim)
L2 = SBFF(L1,bottleneckDim,hiddenDim)
L3 = SBFF(L2,hiddenDim,bottleneckDim)
MeanSqErr = MSEBFF(L3, featOutDim, hiddenDim, featNormOut,
    tag="criterion")
OutputNodes=(MeanSqErr.BFF.FF.P)
EvalNodes=(MeanSqErr)
]

```

For an autoencoder the reader will process features for both the input and desired output (targets) for the network. Thus, the reader configuration block in the main configuration file has two entries “featIn” and “featOut” that both have scp-Files specified. In this case, they point to the same file, but this is not required. For example, if a network was trained for feature enhancement, then featIn could be reading the noisy features while featOut would be reading the desired clean feature targets.

```

featIn=[
    scpFile=$ScpDir$\TIMIT.train.scp.fbank.fullpath
    dim=792
]

featOut=[
    scpFile=$ScpDir$\TIMIT.train.scp.fbank.fullpath
    dim=792
]

```

6.1.5 Using layer-by-layer discriminative pre-training

It is well known that deep networks can be difficult to optimize, especially when a limited amount of training data is available. As a result, a number of approaches to initializing the parameters of these networks have been proposed. One of these methods is known as discriminative pre-training. In this approach, a network with a single hidden layer is trained starting from random initialization. Then the network is grown one layer at a time, from one hidden layer to two hidden layers and so on. Each model is initialized using the parameters learned from the previous model and random initialization for of the topmost output layer.

This process can be performed in CNTK using alternating steps of model training and model editing using the Model Editing Language (MEL). The config file example shows how to do this using a series of execution commands. The top level

command contains five configuration blocks to execute in sequence,

```
command="TIMIT_DiscrimPreTrain1:TIMIT_AddLayer2:
        TIMIT_DiscrimPreTrain2:TIMIT_AddLayer3:TIMIT_Train3"
```

This main configuration file is a good example of how certain config blocks can be placed at the top level of the config file. In this example, the *SGD* and *reader* blocks are shared across all stages of processing. The first three configuration blocks to be processed are shown below:

```
TIMIT_DiscrimPreTrain1=[
  action="train"
  modelPath="$ExpDir$\TrainWithPreTrain\dptmodel1\cntkSpeech
    .dnn"
  NDLLNetworkBuilder=[
    NetworkDescription="$Ndldir$\create_1layer.ndl"
  ]
]

TIMIT_AddLayer2=[
  action="edit"
  CurrLayer=1
  NewLayer=2
  CurrModel="$ExpDir$\TrainWithPreTrain\dptmodel1\cntkSpeech
    .dnn"
  NewModel="$ExpDir$\TrainWithPreTrain\dptmodel2\cntkSpeech.
    dnn.0"
  editPath="$MelDir$\add_layer.mel"
]

TIMIT_DiscrimPreTrain2=[
  action="train"
  modelPath="$ExpDir$\TrainWithPreTrain\dptmodel2\cntkSpeech
    .dnn"
  NDLLNetworkBuilder=[
    NetworkDescription="$Ndldir$\create_1layer.ndl"
  ]
]
```

In the first block, `TIMIT_DiscrimPreTrain1`, the initial model is trained according to the network description language file “create_1layer.ndl” and stored in a folder called `dptmodel1`. Next, in `TIMIT_AddLayer2`, the previous model is edited according to the MEL script “add_layer.mel” which will add a layer to the existing model and write the new model out to a new model file. This script will process the variables set in this configuration block that refer to the current layer,

new layer, current model, and new model. Note that the new model has an extension “0” and has been placed in a folder called “dptlayer2”. Now, the third configuration block “TIMIT_DiscrimPreTrain2” will train a model to be located in the “dptmodel2” folder. Because, the previous step created a model in that location with the extension “0”, the training tool will use that model as the initial model rather than creating a model from scratch. This process repeats itself until the network contains the total number of layers desired.

To see how a new layer is added, we can look at the MEL script “add_layer.mel”

```
m1=LoadModel("$currModel$", format="cntk")
SetDefaultModel(m1)
HDim=512
L$newLayer$=SBFF(L$currLayer$.S, HDim, HDim)
SetInput(CE.*.T, 1, L$newLayer$.S)
SetInput(L$newLayer$.*.T, 1, L$currLayer$.S)
SaveModel(m1, "$newModel$", format="cntk")
```

In this script, the initial model is loaded and set as the default model. A new layer is created using the macro SBFF (the default macros have been loaded in the top-level configuration file). Then the new connections are made, with the input to the cross entropy layer of the existing model connected to the output of the new layer, and the input of the new layer connected to the output of the previous model’s top layer. The new model is then saved to the specified file. Note that through the use of configuration variables, this same script can be reused anytime a new layer can be added. If different layer sizes were desired, the HDim variable could be set by the higher level configuration file in the the appropriate edit block, rather than from within the MEL script.

6.1.6 Training a network with multi-task learning

One interesting approach to network training is multi-task learning, where the network is trained to optimize two objective functions simultaneously. This can be done in CNTK through the appropriate use of NDL. Let’s assume that we have a network specified in NDL that has three hidden layers and output of the third hidden layer is defined as L3. Furthermore, let’s assume we want to create a network that optimizes a weighted combination of phoneme classification and the dialect region of the speaker (dr1, dr2, etc in the TIMIT corpus). To do this, we first define a macro that can compute the weighted sum of two nodes as follows:

```
WtObjFcn(o1,w1,o2,w2)
{
  A1=Constant(w1)
  A2=Constant(w2)
```

```

T1=Times(A1,o1)
T2=Times(A2,o2)
O=Plus(T1,T2)
}

```

This macro can then be used to create a network for multi-task learning.

```

#objective function 1
BFF1=BFF(L3,LabelDim1,HiddenDim)
CE1=CrossEntropyWithSoftmax(labels,BFF1.FF.P,tag="eval")
FER1 = ErrorPrediction(labels,BFF1.FF.P,tag="eval")

# objective function 2
BFF2=BFF(L3,LabelDim2,HiddenDim)
CE2=CrossEntropyWithSoftmax(regions,BFF2.FF.P,tag="eval")
FER2 = ErrorPrediction(regions,BFF2.FF.P,tag="eval")

# weighted final objective function
Alpha1=0.8
Alpha2=0.2
ObjFcn = WtObjFcn(CE1,Alpha1,CE2,Alpha2,tag="criterion")

# for decoding
ScaledLogLikelihood=Minus(BFF1.FF.P, LogPrior)

# root Nodes
OutputNodes=(ScaledLogLikelihood)

```

The output of the hidden layer L3 is connected to two cross entropy nodes, one that predicts “labels” which correspond to phonetic labels in this example, and one that predict “region”, the dialect region of the speaker. The final criterion used for training is the weighted combination of these two criteria. By tagging the individual CrossEntropyWithSoftmax and ErrorPrediction nodes with the “eval” tag, the values of these nodes can also be monitored and logged during training. As before, the ScaledLogLikelihood is also computed for use in a decoder.

6.1.7 Training a network with multiple inputs

There are instances where it’s desirable to input multiple features into a network, such as MFCC and FBANK coefficients. In this case, multiple feature inputs (two, in this example) need to be specified in the *reader*, and the network needs to be constructed appropriately using NDL. The following is a snippet from reader configuration block showing how two different feature types, defined as *features1* and *features2* can be read, along with the phonetic labels. In this case, the log mel

filterbank features which are 72-dimensional will use an 11-frame context window, while the MFCC features which are 39-dimensional will use a single frame of input.

```
features1=[
  dim=792
  scpFile="$ScpDir$\TIMIT.train.scp.fbank.fullpath"
]
features2=[
  dim=39
  scpFile="$ScpDir$\TIMIT.train.scp.mfcc.fullpath"
]
labels=[
  mlfFile="$MlfDir$\TIMIT.train.align_cistate.mlf.cntk"
  labelMappingFile="$MlfDir$\TIMIT.statelist"
  labelDim=183
]
```

The NDL for constructing a network with these inputs and outputs can be done in a number of ways. One way is to construct a macro that constructs a layer that takes two inputs, as follows:

```
SBFF2(input1,rowCount,colCount1,input2,colCount2)
{
  B=Parameter(rowCount,init="fixedValue",value=0)
  W1=Parameter(rowCount, colCount1)
  W2=Parameter(rowCount, colCount2)
  T1=Times(W1,input1) T2=Times(W2,input2)
  P1=Plus(T1,T2)
  P2=Plus(P1,B)
  S=Sigmoid(P2)
}
```

This macro can then be used to create the network. For example, inputs and the first layer would be declared using the NDL code below.

```
FeatDim1 = 792
FeatDim2 = 39

features1=Input(FeatDim1,tag="feature")
features2=Input(FeatDim2,tag="feature")
labels=Input(LabelDim1,tag="label")

featInput1=MeanVarNorm(features1)
featInput2=MeanVarNorm(features2)
```

```
L1 = SBFF2(featInput1, HiddenDim, FeatDim1, featInput2,
           FeatDim2)
```

The rest of the hidden layers and the output layer with a cross entropy objective function would be the same as previous examples. Notice that the names and dimensionality of the input and output data have to be the same in both the NDL model description and the reader configuration.

6.1.8 Evaluating networks and cross validation

Once a network has been trained, we would like to test its performance. To do so, we use a configuration very similar to training, except the “train” action is replaced with the action “eval”, and the data reader needs to be updated to process the development or evaluation data rather than the training data. Of course, the SGD optimization block is not necessary and can be omitted.

```
action="eval"
```

One version of cross validation is early stopping, where a fixed learning schedule is used during training but all intermediate models are evaluated and the one with the best performance on a development set is selected for evaluation. If you perform network training for a large number of epochs, you can efficiently evaluate the performance of a series of models using the “cv” action.

```
action="cv"
```

In addition, you can choose the models you wish to evaluate if you do not want the output of every single epoch. This is done using the “crossValidationInterval” configuration parameters, which takes 3 colon-separated terms which are interpreted as a Matlab-style specification of an array. In the following example, the models from epoch 0, 2, 4, etc. up to the final epoch will be evaluated

```
crossValidationInterval=0:2:25
```

6.1.9 Writing network outputs to files

There are many examples where it is necessary to write either the output or an internal representation of the network to a file. In speech recognition, some common examples of this are writing the output of the network to a file for decoding or construction of a tandem style acoustic model, or writing the internal representation to create bottleneck features. CNTK supports this by the use of the “write” action, the specification of a desired output node, and the use of a data *writer*. To write

data from a particular node, the node is specified using the configuration parameter “outputNodeNames”. For example, for decoding, this would typically be the ScaledLogLikelihood node. A data reader is used to specify the names of the files that will be input to the network and a data writer is used to specify the names of the files to store the output. All nodes specified in the reader and writer must have SCP files that have a line-by-line correspondence to each other. Data is read from the first feature file listed in the SCP file specified in the reader and processed by the network. The values at the desired output node are then passed to the writer and written to the first file in the SCP file that corresponds to that node in the writer configuration. The rest of the files are processed in a similar manner, going line-by-line down the SCP files in the reader and writer. The data is written as an HTK format feature file with the “USER” parameter kind.

In the following example, data from the autoencoder described previously will be written to files. In particular, the activations in the second hidden layer prior to the sigmoid non-linearity will be captured. This node is specified as “L2.BFF.FF.P”. node will be written to files. The output file names are specified using the SCP file in the writer, which is line-by-line parallel with the SCP file used by the reader.

```
TIMIT_WriteBottleneck=[
  action="write"
  modelPath="$ExpDir$\TrainAutoEncoder\model\cntkSpeech.dnn"
  outputNodeNames="L2.BFF.FF.P"
  reader=[
    readerType="HTKMLFReader"
    features=[
      dim=792
      scpFile="$ScpDir$\TIMIT.core.scp.fbank.fullpath"
    ]
  ]

  writer=[
    writerType="HTKMLFWriter"
    L2.BFF.FF.P = [
      dim=64
      scpFile="$ScpDir$\TIMIT.core.scp.bottleneck.fullpath"
    ]
  ]
]
```

For writing scaled log likelihoods, the configuration file would look very similar, except a different node would be specified, e.g. ScaledLogLikelihood, and the writer should be changed appropriately.

If you are unsure of the exact node name you need to specify, you can use the “dumpnode” action with printValues set to false. This will generate a text file

listing of all valid nodes in the network.

```
TIMIT_DumpNodes=[
  action=dumpnode
  modelPath="$ExpDir$\TrainAutoEncoder\model\cntkSpeech.dnn"
  printValues=false
]
```

6.2 RNN Language Model

Recurrent neural network (RNN) language models have been proven to obtain state-of-the-art performance in language modeling. We use Penn Treebank data set to demonstrate how to build a RNN language model with CNTK. In particular, we will build a class based RNN language model. The setup file is `CNTK\MachineLearning\cn\rnnlmConfig.txt`, which consists of two components: train and test.

6.2.1 Train

Training parameters are specified in this section, with the most important ones listed as follows.

- `action="train"`. It indicates the section is for training.
- `minibatchSize=10`. In RNN training, the minibatch size is the truncation size of the truncated BPTT. That is, model parameters are updated after 10 words.
- `deviceId=-1`. -1 means CPU. One can change to GPU by specifying a non-negative GPU id.
- `epochSize=4430000`. The max number of words used to train RNN model in each epoch. This provides a way to use a proportion of entire training data for model training.
- SimpleNetworkBuilder section
 - `rnnType="CLASSLM"`. The RNN network structure. It consists of an input layer, a recurrent hidden layer and an output layer (including classes and vocabularies).
 - `trainingCriterion="ClassCrossEntropyWithSoftmax"`. Training criterion used in model training.
 - `nodeType="Sigmoid"`. Non-linearity function used in hidden layer.

- layerSizes=10000:200:10000. Sizes of input, hidden and output layers. Input layer size is equal to vocabulary size, hidden layer is normally in the range of 50 to 500, output layer size is the vocabulary size.
 - uniformInit=true. Whether to use uniformly randomized values for initial parameter weights.
 - vocabSize=10000. The vocabulary size
 - nbrClass=50 . number of word classes.
- SGD section
 - learningRatesPerSample=0.1. Learning rate in stochastic gradient descent.
 - momentumPerMB=0. Momentum used in updating parameter weights. The updating equation is $g_{new} = (1 - m) * g + m * g_{old}$, where m is momentum, g is the gradient computed in backward pass, g_{old} is the gradient value in previous mini-batch, and g_{new} is the new gradient for the current min-batch.
 - gradientClippingWithTruncation=true. Whether to control the upper bound of the gradient values using simple truncation or norm based approach (higher cost).
 - clippingThresholdPerSample=15.0. Indicates that the absolute value of the gradient should never be larger than 15.
 - maxEpochs=40. Maximum number of training epochs.
 - numMBsToShowResult=2000. The frequency of showing training/validation loss results (in terms of how many mini-batches are processed).
 - gradUpdateType="none". How the gradients are computed. None stands for standard gradient update (no special treatment). One can also choose adagrad or rmsprop.
 - modelPath="C:\CNTKExp\RNN\log\modelRnnCNTK". The resulting RNN model file location.
 - loadBestModel=true. Choose to use the best of previous models to start training in each epoch.
 - reduceLearnRateIfImproveLessThan=0.001. The learning parameter is reduced if the difference between previous criterion and current criterion is smaller than previous criterion multiplied by reduceLearnRateIfImproveLessThan.

- `continueReduce=true`. If true, the learning rate is always reduced per epoch once it is reduced.
- `learnRateDecreaseFactor=0.5`. Learning rate decrease factor.
- reader section
 - `wordclass="C:\CNTKExp\RNN\data\PennTreeBank\vocab.txt"`. Word class file which contains words, their ids and their class ids, in the following format
`word_id \t frequency \t word_string \t word_class`
`word_id` is a unique non-negative integer, frequency is the frequency of word (optional), `word_string` is the word string (low frequent words may be mapped to `<unk>`), and `word_class` is the class id of word. Word class can be derived using frequency based heuristics [33] or more sophisticated way [34].
 - `file="C:\CNTKExp\RNN\data\PennTreeBank\ptb.train.cntk.txt"`. The location of training data file which has the following format
`</s> word1 word2 ... </s>`
- cvReader section
 - `wordclass="C:\CNTKExp\RNN\data\PennTreeBank\vocab.txt"`. Word class file which contains words, their ids and their class ids.
 - `file="C:\CNTKExp\RNN\data\PennTreeBank\ptb.valid.cntk.txt"`. Validation data file location. It has the same format as training data.

6.2.2 Test

Test parameters are specified in this section, with the most important ones specified as follows.

- `action="eval"`. It indicates the section is for test.
- `minibatchSize=100`. Evaluation truncation size is 100.
- `deviceId=-1`. Use CPU device.
- reader section
 - `wordclass="C:\CNTKExp\RNN\data\PennTreeBank\vocab.txt"`. Word class file which contains words, their ids and their class ids.
 - `file="C:\CNTKExp\RNN\data\PennTreeBank\ptb.test.cntk.txt"`. Test data file location. It has the same format as training data.

6.3 LSTM Language Model

We apply long-short-term memory (LSTM) recurrent neural network for language modeling task. The example setup is at `ExampleSetups\LM\LSTMLM\lstmlmconfig.txt`. In this setup, the following need to specified for training

6.3.1 Training

- `action="train"` : this informs CNTK to train a model.
- `deviceId=-1` : this specifies using CPU.
- `SimpleNetworkBuilder`:
 - `recurrentLayer=1` : this specifies that layer 1 is recurrent layer.
 - `rnnType="CLASSLM"` : this informs CNTK to call class-based LSTM function in `simplenetworkbuilder`.
 - `trainingCriterion="ClassCrossEntropyWithSoftmax"` specifies that training set to use class-based cross entropy
 - `evalCriterion="ClassCrossEntropyWithSoftmax"` specifies validation set also uses class-based cross entropy for evaluation.
 - `layerSizes=10000:200:10000` : this specifies input, hidden and output layer sizes.
 - `vocabSize=10000`. The vocabulary size
 - `nbrClass=50` . number of word classes.
- `SGD`
 - `useAdagrad=true` : this specifies using AdaGrad for weight update.
 - `modelPath="c:\temp\penntreebank\cntkdebug.dnn"` : this is the trained model file name.
- `Reader`: specifies training reader
 - `readerType="LMSequenceReader"` : specifies using language model sequence reader.
 - `wordclass="c:\exp\penntreebank\data\wordclass.txt"` : specifies word class info.
 - `file="c:\exp\penntreebank\data\ptb.train.cntk.txt"` : specifies training file

- cvReader: specifies cross-validation reader
 - readerType="LMSequenceReader" : specifies using language model sequence reader.
 - wordclass="c:\exp\penntreebank\data\wordclass.txt" : specifies word class info.
 - file="c:\exp\penntreebank\data\ptb.valid.cntk.txt" : specifies validation file

6.3.2 Test

In this setup, the following need to specified for testing

- action="eval" : this informs CNTK to call simplenetwork evaluation.
- deviceId=-1 : this specifies using CPU.
- modelPath="c:\temp\penntreebank\cntkdebug.dnn" : this is the trained model file name.
- Reader: specifies test set reader
 - readerType="LMSequenceReader" : specifies using sequence reader.
 - wordclass="c:\exp\penntreebank\data\wordclass.txt" : specifies word class info.
 - file="c:\exp\penntreebank\data\ptb.test.cntk.txt" : specifies testing file

6.4 Spoken Language Understanding

One of the important tasks in spoken language understanding is labeling input sequence with semantic tags. In this example, we show how CNTK can be used to train a LSTM recurrent network for the labeling task. The setup file is under ExampleSetups\SLU\rnnlu.cntk. The data is ATIS, which consists of 944 unique words, including <unk>, in the training/dev set. Output has 127 dimension, each corresponding to a semantic tag in ATIS. Unseen words in test will be mapped to <unk>. A file provides such mapping from one word to the other, which is useful to map low-frequency input or unseen input to a common input. In this case, the common input is <unk>.

6.4.1 Training

In this setup, the following need to specified for training

- `action="train"` : this informs CNTK to train a model.
- `deviceId=-1` : this specifies using CPU.
- `minibatchSize = 10` : this specifies the maximum number of words per minibatch.
- `SimpleNetworkBuilder`:
 - `recurrentLayer=2` : this specifies that layer 2 is recurrent layer.
 - `rnnType=LSTM`: this informs CNTK to call LSTM function in `simplenetworkbuilder`.
 - `lookupTableOrder=3`: this specifies forming a context-dependent input with a context window size of 3.
 - `trainingCriterion="ClassCrossEntropyWithSoftmax"` specifies that training set to use cross entropy
 - `evalCriterion="ClassCrossEntropyWithSoftmax"` specifies validation set also uses cross entropy for evaluation.
 - `layerSizes=2832:50:300:127`: this specifies input, hidden and output layer sizes. Notice that input layer has dimension of 2832, which is 3 times 944. This number is obtained in consideration of context window size and the number of unique words, which 944. If `lookupTableOrder` is set to 1, the input layer size should be set to 944.
- `SGD`
 - `learningRatePerSample=0.1` : this specifies learning rate per sample.
 - `modelPath="c:\temp\exp\ATIS\temp\cntkdebug.dnn"` : this is the trained model file name.
 - `gradUpdateType=AdaGrad` : this specifies using AdaGrad for updating weights.
- `Reader`: specifies training reader
 - `readerType=LUSequenceReader` : specifies using `LUSequence` reader.
 - `nbruttsineachrecurrentiter=10` : this specifies using maximum of 10 sentences for each minibatch.

- wordcontext=0:1:2 : this specifies the time indices for forming a context window. In this example, this setup corresponds to using the current input, the next input, and the input after the next input for a context window of size 3. User can also use other cases such as wordcontext=0:-1:1 to form a context window of 3 but using the current input, the previous input, and the next input.
- wordmap=c:\exp\atis\data\inputmap.txt : specifies a file that lists a mapping from word to the other word. This mapping file should be constructed only from training/dev sets.
- file=c:\exp\ATIS\data\atis.train.apos.pred.pos.head.IOB.simple: specifies training file
- labelIn : this specifies information of inputs
 - * beginingSequence="BOS" : this specifies the symbol of sequence beginning.
 - * endSequence="EOS" : this specifies the symbol of sequence ending.
 - * token="c:\exp\atis\data\input.txt" : this specifies a list of word as input.
- labels : this specifies information of labels
 - * token="c:\exp\atis\data\output.txt" : this specifies output semantic labels.
- cvReader: specifies cross-validation reader. Most of setups should be same as those in Reader section.
 - readerType="LUSequenceReader" : specifies using sequence reader.
 - wordcontext=0:1:2 : this specifies the time indices for forming a context window. This should be the same as specified in Reader section.
 - file="c:\exp\ATIS\data\atis.dev.apos.pred.pos.head.IOB.simple": specifies validation file

6.4.2 Test

In this setup, the following need to specified for writing/decoding test set. It uses LUSequenceWriter to decode test set word sequence to their semantic tags.

- action="write": this informs CNTK to call simplenetwork evaluation and write outputs, which will be specified below.

- `deviceId=-1` : this specifies using CPU.
- `modelPath="c:\temp\exp\ATIS\temp\cntkdebug.dnn"` : this is the trained model file name.
- `outputNodeNames="outputs:labels"` : this specifies which nodes to output results. These node names are pre-specified in CNTK's simple network builder. The node "outputs" is the node that output activates before softmax. The node "labels" is the node that has reference labels for comparison.
- `reader`: specifies test set reader
 - `readerType="LUSequenceReader"` : specifies using LUSequence reader.
 - `wordmap="c:\exp\atis\data\inputmap.txt"`: specifies word map file, which should be the same as those used in training/validation readers.
 - `file="c:\exp\atis\data\atis.test.apos.pred.pos.head.IOB.simple"`: specifies testing file
- `writer` : this specifies where to write
 - `writerType="LUSequenceWriter"` : this specifies using LUSequenceWriter.
 - `outputs` : specifies where to write for the outputs node.
 - * `file="c:\temp\exp\atis\output\output.rec.txt"` : the file name for writing decode results from LUSequenceWriter.
 - * `token="c:\exp\atis\data\output.txt"` : this specifies the semantic labels.

Bibliography

- [1] R. Zhao, J. Li, and Y. Gong, “Variable-component deep neural network for robust speech recognition,” in *Interspeech*, September 2014. [Online]. Available: <http://research.microsoft.com/apps/pubs/default.aspx?id=230079>
- [2] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Transactions on Audio, Speech and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012.
- [3] F. Seide, G. Li, and D. Yu, “Conversational speech transcription using context-dependent deep neural networks,” in *Proc. Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2011, pp. 437–440.
- [4] F. Seide, G. Li, X. Chen, and D. Yu, “Feature engineering in context-dependent deep neural networks for conversational speech transcription,” in *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2011, pp. 24–29.
- [5] D. Yu and M. L. Seltzer, “Improved bottleneck features using pretrained deep neural networks,” in *Proc. Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2011, pp. 237–240.
- [6] N. Jaitly, P. Nguyen, A. W. Senior, and V. Vanhoucke, “Application of pre-trained deep neural networks to large vocabulary speech recognition,” in *Proc. Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2012.
- [7] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

- [8] Y. LeCun and Y. Bengio, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, 1995.
- [9] K. Chellapilla, S. Puri, and P. Simard, “High performance convolutional neural networks for document processing,” in *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.
- [10] K. Kavukcuoglu, P. Sermanet, Y.-L. Boureau, K. Gregor, M. Mathieu, and Y. LeCun, “Learning convolutional feature hierarchies for visual recognition,” in *NIPS*, vol. 1, no. 2, 2010, p. 5.
- [11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, vol. 1, no. 2, 2012, p. 4.
- [12] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn, “Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012, pp. 4277–4280.
- [13] D. C. Ciresan, U. Meier, and J. Schmidhuber, “Transfer learning for Latin and Chinese characters with deep neural networks,” in *Proc. International Conference on Neural Networks (IJCNN)*, 2012, pp. 1–6.
- [14] T. N. Sainath, B. Kingsbury, A.-r. Mohamed, G. E. Dahl, G. Saon, H. Soltau, T. Beran, A. Y. Aravkin, and B. Ramabhadran, “Improvements to deep convolutional neural networks for lvcsr,” in *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2013, pp. 315–320.
- [15] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, “Deep convolutional neural networks for LVCSR,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 8614–8618.
- [16] O. Abdel-Hamid, L. Deng, and D. Yu, “Exploring convolutional neural network structures and optimization techniques for speech recognition,” pp. 3366–3370, 2013.
- [17] L. Deng, O. Abdel-Hamid, and D. Yu, “A deep convolutional neural network using heterogeneous pooling for trading acoustic invariance with phonetic confusion,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 6669–6673.

- [18] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [19] R. Socher, C. C. Lin, A. Ng, and C. Manning, “Parsing natural scenes and natural language with recursive neural networks,” in *Proc. International Conference on Machine Learning (ICML)*, 2011, pp. 129–136.
- [20] I. Sutskever, J. Martens, and G. E. Hinton, “Generating text with recurrent neural networks,” in *Proc. International Conference on Machine Learning (ICML)*, 2011, pp. 1017–1024.
- [21] T. Mikolov and G. Zweig, “Context dependent recurrent neural network language model,” in *Proc. IEEE Spoken Language Technology Workshop (SLT)*, 2012, pp. 234–239.
- [22] Y. Shi, P. Wiggers, and C. M. Jonker, “Towards recurrent neural networks language models with linguistic and contextual features,” in *Proc. Annual Conference of International Speech Communication Association (INTER-SPEECH)*, 2012.
- [23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a CPU and GPU math expression compiler,” in *Proceedings of the Python for scientific computing conference (SciPy)*, vol. 4, 2010.
- [24] B. Guenter, D. Yu, A. Eversole, O. Kuchaiev, and M. L. Seltzer, “Stochastic gradient descent algorithm in the computational network toolkit.”
- [25] S. Wiesler, A. Richard, P. Golik, R. Schluter, and H. Ney, “RASR/NN: The RWTH neural network toolkit for speech recognition,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 3305–3309.
- [26] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.
- [27] C. Bischof, L. Roh, and A. Mauer-Oats, “ADIC: an extensible automatic differentiation tool for ANSI-C,” *Urbana*, vol. 51, p. 61802, 1997.
- [28] B. Guenter, “Efficient symbolic differentiation for graphics applications,” in *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, 2007, p. 108.
- [29] L. E. Atlas, T. Homma, and R. J. Marks II, “An artificial neural network for spatio-temporal bipolar patterns: Application to phoneme classification,” *Proc. Neural Information Processing Systems (NIPS)*, p. 31, 1988.

- [30] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [31] J. E. Hopcroft, *Data structures and algorithms*. Pearson education, 1983.
- [32] A. Graves, “Generating sequences with recurrent neural networks,” *arXiv preprint arXiv:1308.0850*, 2013.
- [33] T. Mikolov, S. Kombrink, L. Burget, J. H. Cernocky, and S. Khudanpur, “Extensions of recurrent neural network language model,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011.
- [34] G. Zweig and K. Makarychev, “Speed regularization and optimality in word classing,” in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.