An Introduction to Computational Networks and the Computational Network Toolkit

Dong Yu, Adam Eversole, Michael L. Seltzer, Kaisheng Yao, Zhiheng Huang,
Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang,
Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jon Currey,
Jie Gao, Avner May, Baolin Peng, Andreas Stolcke, Malcolm Slaney

MSR-TR-2014-112 (DRAFT v0.1: August 29, 2014)

Abstract

We introduce computational network (CN), a unified framework for describing arbitrary learning machines, such as deep neural networks (DNNs), computational neural networks (CNNs), recurrent neural networks (RNNs), long short term memory (LSTM), logistic regression, and matrixum 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

Chapter 1

Computational Network

1.1 Computational Network

There is a common property in key machine learning models, such as deep neural networks (DNNs) [1, 2, 3, 4, 5, 6], convolutional neural networks (CNNs) [7, 8, 9, 10, 11, 12, 13, 14, 15, 16], and recurrent neural networks (RNNs) [17, 18, 19, 20, 21]. 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 1.1. 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 [22], CNTK [23] and RASR/NN [24]..

Algorithm 1.1 Computational Steps Involved in an One-Hidden-Layer Sigmoid Neural Network

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

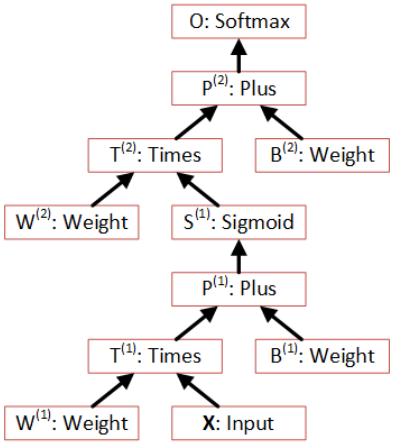

Figure 1.1: Represent the one-hidden-layer sigmoid neural network of Algorithm 1.1 with a computational network

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 $\{n : c_1, \cdots, c_{K_n}\}$, 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 1.1 is the one-hidden-layer sigmoid neural network of Algorithm 1.1 represented as a CN in which each node n is identified by a $\{nodename : operatortype\}$ 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 1.1 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 1.2 or a network with shared model parameters as shown in Figure 1.3.

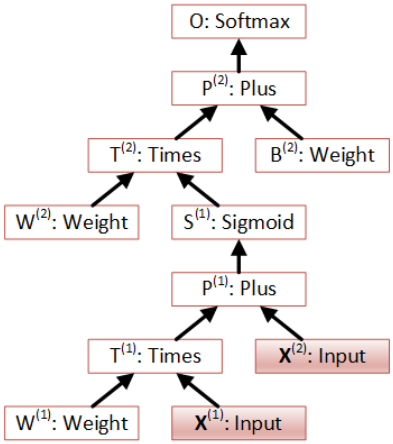

Figure 1.2: A computational network with two input nodes (highlighted).

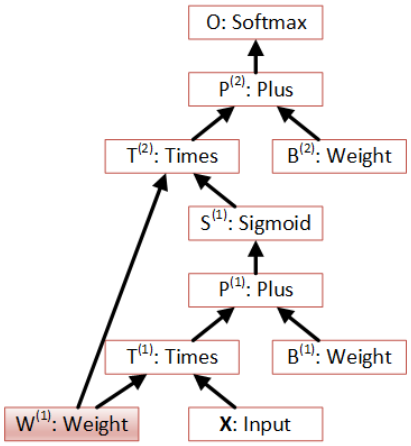

Figure 1.3: A computational network with shared model parameters (highlighted). Here we assume the input $\mathbf{X}$, hidden layer $\mathbf{S}^{(1)}$, and output layer $\mathbf{O}$ have the same dimension.

1.2 Forward Computation

When the model parameters (i.e., weight nodes in Figure 1.1) 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 1.3 is an example. This is because the same computation node may be a child of several other nodes. Algorithm 1.2. 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 1.2, 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 1.2 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 ∉ visited then      ▷ the same node may be a child of several nodes.
3:         visited ← visited ∪ root
4:         for each c ∈ root.children do          ▷ apply to children recursively
5:             call DECIDEFORWARDCOMPUTATIONORDER(c, visited, order)
6:             order ← order + root               ▷ Add root to the end of order
7:         end for
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 1.3 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 descendents of *root* are in the waiting set and all leaf descendents 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 1.3, 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 1.3 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 descendents of *root*.
▷ *ready* is initialized to include all leaf descendents of *root*.
2: **for** each $p \in node.parents \wedge p \in waiting$ **do**
3: $p.numFinishedChildren++$
4: **if** $p.numFinishedChildren == p.numChildren$ **then**
▷ all children have been computed
5: $waiting \leftarrow waiting - node$
6: $ready \leftarrow ready \cup node$
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 $node \in ready$ according to some policy
12: $ready \leftarrow ready - node$
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 1.2 and 1.3 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 1.4.

1.3 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 1.4 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

$$\mathbf{W}_{t+1} \leftarrow \mathbf{W}_{\mathrm{t}} - \varepsilon \triangle \mathbf{W}_t, \tag{1.1}$$

where

$$\triangle \mathbf{W}_t = \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla_{\mathbf{W}_t} J\left(\mathbf{W}; \mathbf{x}^m, \mathbf{y}^m\right), \tag{1.2}$$

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

A naive solution to compute $\nabla^J_{\mathbf{W}}$ is illustrated in Figure 1.5, in which $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters. In this solution, each edge is associated with a partial derivative, and

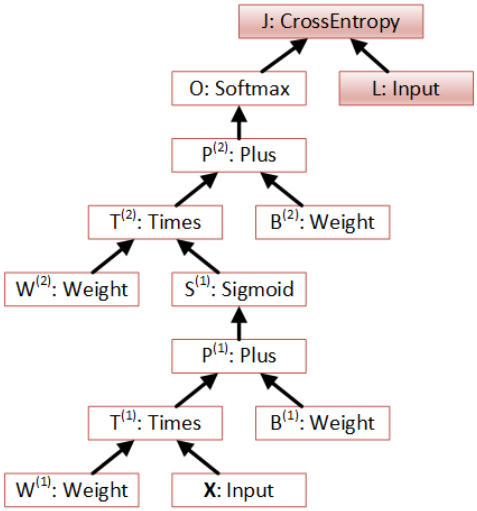

Figure 1.4: The one-hidden-layer sigmoid neural network augmented with a cross entropy training criterion node J and a label node L.

$$\nabla^J_{\mathbf{W}^{(1)}} = \frac{\partial J}{\partial \mathbf{V}^{(1)}}\frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}}\frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(1)}} + \frac{\partial J}{\partial \mathbf{V}^{(3)}}\frac{\partial \mathbf{V}^{(3)}}{\partial \mathbf{V}^{(4)}}\frac{\partial \mathbf{V}^{(4)}}{\partial \mathbf{W}^{(1)}} \tag{1.3}$$

$$\nabla^J_{\mathbf{W}^{(2)}} = \frac{\partial J}{\partial \mathbf{V}^{(1)}}\frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}}\frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(2)}}. \tag{1.4}$$

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 $\nabla^J_{\mathbf{W}^{(1)}}$ and once for $\nabla^J_{\mathbf{W}^{(2)}}$.

Fortunately, there is a much simpler and more efficient approach to compute the gradient as illustrated in Figure 1.6. In this approach, each node n keeps two values: the evaluation (forward computation) result $\mathbf{V}_n$ and the gradient ∇^J_n . Note that the training criterion J is always a scalar, If $\mathbf{V}_n \in \mathbb{R}^{N_1 \times N_2}$ then $\nabla^J_n \in \mathbb{R}^{N_1 \times N_2}$. This requires significantly less memory than that required in the naive solution illustrated in Figure 1.5. 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 common subexpression elimination in a conventional expression graph, only here the common subexpressions

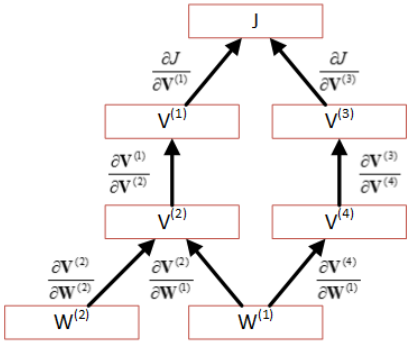

Figure 1.5: The naive gradient computation in CN. $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters and each edge is associated with a partial derivative.

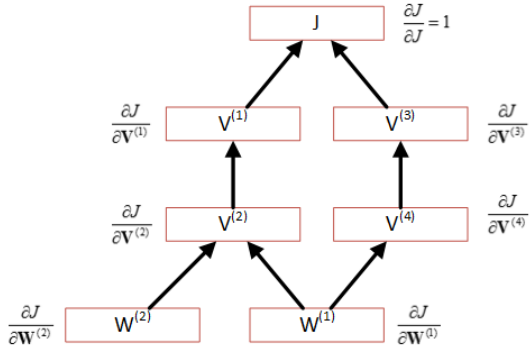

Figure 1.6: An efficient gradient computation algorithm in CN. $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are model parameters. Each node n stores both the value of the node υ_n and the gradient ∇_n^J .

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 ([25]). A simple recursive algorithm for deciding the gradient computation order so that the gradient computation can be efficiently carried out is shown in Algorithm 1.4 to which a similar recursive algorithm for scalar functions has been provided [26, 27]. 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 1.4 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 descendents 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 1.5, which is essentially a depth-first traversal over the DAG. Since both Algorithms 1.2 and 1.5 are essentially depth-first traversal over the DAG and both only need to be executed once they may be combined in one function.

Algorithm 1.5 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 ∉ visited then▷ The same node may be a child of several nodes and revisited.
3:         visited ← visited ∪ root
4:         for each c ∈ root.children do
5:             call UPDATENEEDGRADIENTFLAG(c, visited, order)
6:             if IsNotLeaf(node) then
7:                 if node.AnyChildNeedGradient() then
8:                     node.needGradient ← true
9:                 else
10:                    node.needGradient ← 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:

$$\frac{\partial J}{\partial w_{ij}} \approx \frac{J(w_{ij}+\varepsilon) - J(w_{ij}-\varepsilon)}{2\varepsilon}, \tag{1.5}$$

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 $J(w_{ij}+\varepsilon)$ and $J(w_{ij}-\varepsilon)$ are the objective function values evaluated with all other parameters fixed and w_{ij} changed to $w_{ij}+\varepsilon$ and $w_{ij}-\varepsilon$, 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.

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 $\nabla_{\mathbf{V}}^J$)	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
$\circledcirc$	inner product applied to each row
$\delta(.)$	Kronecker delta
$\mathbf{1}_{m,n}$	an $m \times n$ matrix with all 1's
X^{α}	element-wise power
vec$(\mathbf{X})$	vector formed by concatenating columns of $\mathbf{X}$

Table 1.1: Symbols used in describing the computation nodes

1.4 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 1.1 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

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}}, \tag{1.6}$$

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

1.4.1 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.

1.4.2 Computation Node Types with One Operand

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

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

$$\begin{aligned} V(\mathbf{X}) &\leftarrow -\mathbf{X} && (1.7)\\ \nabla^J_{\mathbf{X}} &\leftarrow \nabla^J_{\mathbf{X}} - \nabla^J_{\mathbf{n}}. && (1.8) \end{aligned}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} -1 & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.9}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = -\frac{\partial J}{\partial \upsilon_{ij}}. \tag{1.10}$$

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

$$\begin{aligned} V(\mathbf{X}) &\leftarrow \frac{1}{1+e^{-\mathbf{X}}} && (1.11)\\ \nabla^J_{\mathbf{X}} &\leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \bullet [V \bullet (1-V)]. && (1.12) \end{aligned}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} \upsilon_{ij}(1-\upsilon_{ij}) & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.13}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \upsilon_{ij} (1 - \upsilon_{ij}) . \tag{1.14}$$

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

$$V(\mathbf{X}) \leftarrow \frac{e^{\mathbf{X}} - e^{-\mathbf{X}}}{e^{\mathbf{X}} + e^{-\mathbf{X}}} \tag{1.15}$$

$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet (1 - V \bullet V) . \tag{1.16}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} 1 - \upsilon_{ij}^2 & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.17}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \left(1 - \upsilon_{ij}^2\right) \tag{1.18}$$

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

$$V(\mathbf{X}) \leftarrow \max(0, \mathbf{X}) \tag{1.19}$$

$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet \delta(\mathbf{X} > 0) . \tag{1.20}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} \delta(x_{ij} > 0) & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.21}$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \delta(x_{ij} > 0) . \tag{1.22}$$

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

$$V(\mathbf{X}) \leftarrow \log(\mathbf{X}) \tag{1.23}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet \frac{1}{\mathbf{X}}. \tag{1.24}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} \frac{1}{x_{ij}} & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.25}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \frac{1}{x_{ij}}. \tag{1.26}$$

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

$$V(\mathbf{X}) \leftarrow \exp(\mathbf{X}) \tag{1.27}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet V. \tag{1.28}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} \upsilon_{ij} & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.29}$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \upsilon_{ij}. \tag{1.30}$$

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

$$m_j(\mathbf{X}) \leftarrow \max_i x_{ij} \tag{1.31}$$
$$e_{ij}(\mathbf{X}) \leftarrow e^{x_{ij} - m_j(\mathbf{X})} \tag{1.32}$$
$$s_j(\mathbf{X}) \leftarrow \sum_i e_{ij}(\mathbf{X}) \tag{1.33}$$
$$\upsilon_{ij}(\mathbf{X}) \leftarrow \frac{e_{ij}(\mathbf{X})}{s_j(\mathbf{X})} \tag{1.34}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \left[\nabla_{\mathbf{n}}^{J} - \nabla_{\mathbf{n}}^{J} \circ V\right] \bullet V. \tag{1.35}$$

The gradient can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} \upsilon_{ij}\left(1-\upsilon_{ij}\right) & m = i \wedge n = j \\ -\upsilon_{mj}\upsilon_{ij} & n = j \\ 0 & else \end{cases} \tag{1.36}$$

we have

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \left(\frac{\partial J}{\partial \upsilon_{ij}} - \sum_{m} \frac{\partial J}{\partial \upsilon_{mj}} \upsilon_{mj} \right) \upsilon_{ij}. \tag{1.37}$$

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

$$\upsilon\left(\mathbf{X}\right) \leftarrow \sum_{i,j} x_{ij} \tag{1.38}$$

$$\nabla^J_{\mathbf{X}} \leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}}. \tag{1.39}$$

The gradient can be derived by noting that υ and $\nabla^J_{\mathbf{n}}$ are scalars,

$$\frac{\partial \upsilon}{\partial x_{ij}} = 1 \tag{1.40}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon} \frac{\partial \upsilon}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon}. \tag{1.41}$$

- *L1Norm*: take the matrix L_1 norm of the operand X.

$$\upsilon\left(\mathbf{X}\right) \leftarrow \sum_{i,j} \left|x_{ij}\right| \tag{1.42}$$

$$\nabla^J_{\mathbf{X}} \leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \mathrm{sgn}\left(\mathbf{X}\right). \tag{1.43}$$

The gradient can be derived by noting that υ and $\nabla^J_{\mathbf{n}}$ are scalars,

$$\frac{\partial \upsilon}{\partial x_{ij}} = \mathrm{sgn}\left(x_{ij}\right) \tag{1.44}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon} \frac{\partial \upsilon}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon} \mathrm{sgn}\left(x_{ij}\right). \tag{1.45}$$

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

$$\upsilon(\mathbf{X}) \leftarrow \sqrt{\sum_{i,j}(x_{ij})^2} \tag{1.46}$$

$$\nabla_{\mathbf{X}}^J \leftarrow \nabla_{\mathbf{X}}^J + \frac{1}{\upsilon}\nabla_{\mathbf{n}}^J \mathbf{X}. \tag{1.47}$$

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

$$\frac{\partial \upsilon}{\partial x_{ij}} = \frac{x_{ij}}{\upsilon} \tag{1.48}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon}\frac{\partial \upsilon}{\partial x_{ij}} = \frac{1}{\upsilon}\frac{\partial J}{\partial \upsilon}x_{ij}. \tag{1.49}$$

1.4.3 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.

$$V(\lambda, \mathbf{Y}) \leftarrow \lambda \mathbf{Y} \tag{1.50}$$

$$\nabla_{\lambda}^J \leftarrow \nabla_{\lambda}^J + \text{vec}\left(\nabla_{\mathbf{n}}^J\right) \circ \text{vec}(\mathbf{Y}) \tag{1.51}$$

$$\nabla_{\mathbf{Y}}^J \leftarrow \nabla_{\mathbf{Y}}^J + \lambda \nabla_{\mathbf{n}}^J. \tag{1.52}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial \lambda} = y_{mn} \tag{1.53}$$

and

$$\frac{\partial J}{\partial \lambda} = \sum_{m,n}\frac{\partial J}{\partial \upsilon_{mn}}\frac{\partial \upsilon_{mn}}{\partial \lambda} = \sum_{m,n}\frac{\partial J}{\partial \upsilon_{mn}} y_{mn}. \tag{1.54}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \begin{cases} \lambda & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.55}$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \lambda \frac{\partial J}{\partial \upsilon_{ij}} \tag{1.56}$$

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

$$\begin{aligned} V(\mathbf{X},\mathbf{Y}) &\leftarrow \mathbf{XY} && (1.57)\\ \nabla^J_{\mathbf{X}} &\leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \mathbf{Y}^T && (1.58)\\ \nabla^J_{\mathbf{Y}} &\leftarrow \nabla^J_{\mathbf{Y}} + \mathbf{X}^T \nabla^J_{\mathbf{n}}. && (1.59) \end{aligned}$$

The gradient $\nabla^J_{\mathbf{X}}$ can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} y_{jn} & m = i \\ 0 & else \end{cases} \tag{1.60}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \sum_{n} \frac{\partial J}{\partial \upsilon_{in}} y_{jn}. \tag{1.61}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \begin{cases} x_{mi} & n = j \\ 0 & else \end{cases} \tag{1.62}$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \sum_{m} \frac{\partial J}{\partial \upsilon_{mj}} x_{mi} \tag{1.63}$$

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

$$\begin{aligned} \upsilon_{ij}(\mathbf{X},\mathbf{Y}) &\leftarrow x_{ij} y_{ij} && (1.64)\\ \nabla^J_{\mathbf{X}} &\leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \bullet \mathbf{Y} && (1.65)\\ \nabla^J_{\mathbf{Y}} &\leftarrow \nabla^J_{\mathbf{Y}} + \nabla^J_{\mathbf{n}} \bullet \mathbf{X}. && (1.66) \end{aligned}$$

The gradient $\nabla^J_{\mathbf{X}}$ can be derived by observing

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} y_{ij} & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.67}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} y_{ij}. \tag{1.68}$$

The gradient $\nabla^J_{\mathbf{Y}}$ 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.

$$V(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} + \mathbf{Y} \tag{1.69}$$

$$\nabla^J_{\mathbf{X}} \leftarrow \begin{cases} \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{X}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = 1 \\ \nabla^J_{\mathbf{X}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} \mathbf{1}_{v.cols,1} & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = 1 \end{cases} \tag{1.70}$$

$$\nabla^J_{\mathbf{Y}} \leftarrow \begin{cases} \nabla^J_{\mathbf{Y}} + \nabla^J_{\mathbf{n}} & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{Y}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{Y}} + \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = 1 \\ \nabla^J_{\mathbf{Y}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = 1 \end{cases} \tag{1.71}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} 1 & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.72}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{ij}} \tag{1.73}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \begin{cases} 1 & n = j \\ 0 & else \end{cases} \tag{1.74}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial x_{ij}} = \sum_{m} \frac{\partial J}{\partial v_{mj}} \tag{1.75}$$

We can derive $\nabla^J_{\mathbf{X}}$ and $\nabla^J_{\mathbf{Y}}$ 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.

$$V(\mathbf{X},\mathbf{Y}) \leftarrow \mathbf{X} - \mathbf{Y} \tag{1.76}$$

$$\nabla^J_{\mathbf{X}} \leftarrow \begin{cases} \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{X}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{X}.rows = \mathbf{V}.rows \wedge \mathbf{X}.cols = 1 \\ \nabla^J_{\mathbf{X}} + \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} \mathbf{1}_{v.cols,1} & \mathbf{X}.rows = 1 \wedge \mathbf{X}.cols = 1 \end{cases} \tag{1.77}$$

$$\nabla^J_{\mathbf{Y}} \leftarrow \begin{cases} \nabla^J_{\mathbf{Y}} - \nabla^J_{\mathbf{n}} & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{Y}} - \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = \mathbf{V}.cols \\ \nabla^J_{\mathbf{Y}} - \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{Y}.rows = \mathbf{V}.rows \wedge \mathbf{Y}.cols = 1 \\ \nabla^J_{\mathbf{Y}} - \mathbf{1}_{1,\mathbf{V}.rows} \nabla^J_{\mathbf{n}} \mathbf{1}_{\mathbf{V}.cols,1} & \mathbf{Y}.rows = 1 \wedge \mathbf{Y}.cols = 1 \end{cases} \tag{1.78}$$

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 $\mathbf{d}.rows = \mathbf{Y}.rows$.

$$v_{ij}(\mathbf{d},\mathbf{Y}) \leftarrow d_i y_{ij} \tag{1.79}$$

$$\nabla^J_{\mathbf{d}} \leftarrow \nabla^J_{\mathbf{d}} + \nabla^J_{\mathbf{n}} \circledcirc \mathbf{Y} \tag{1.80}$$

$$\nabla^J_{\mathbf{Y}} \leftarrow \nabla^J_{\mathbf{Y}} + \text{DiagTimes}\left(\mathbf{d}, \nabla^J_{\mathbf{n}}\right). \tag{1.81}$$

The gradient $\nabla^J_{\mathbf{d}}$ can be derived by observing

$$\frac{\partial v_{mn}}{\partial d_i} = \begin{cases} y_{in} & m = i \\ 0 & else \end{cases} \tag{1.82}$$

and

$$\frac{\partial J}{\partial d_i} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial d_i} = \sum_{n} \frac{\partial J}{\partial v_{in}} y_{in} \tag{1.83}$$

Similarly to derive the gradient $\nabla^J_{\mathbf{Y}}$ we note that

$$\frac{\partial v_{mn}}{\partial y_{ij}} = \begin{cases} d_i & m = i \wedge n = j \\ 0 & else \end{cases} \tag{1.84}$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial v_{mn}} \frac{\partial v_{mn}}{\partial y_{ij}} = \frac{\partial J}{\partial v_{ij}} d_i \tag{1.85}$$

- *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:

$$m_{ij}(\lambda) \leftarrow \begin{cases} 0 & rand(0,1) \leq \lambda \\ \frac{1}{1-\lambda} & else \end{cases} \tag{1.86}$$

$$v_{ij}(\lambda, \mathbf{Y}) \leftarrow m_{ij} y_{ij} \tag{1.87}$$

$$\nabla^J_{\mathbf{Y}} \leftarrow \nabla^J_{\mathbf{Y}} + \begin{cases} \nabla^J_{\mathbf{n}} & \lambda = 0 \\ \nabla^J_{\mathbf{n}} \bullet \mathbf{M} & else \end{cases} \tag{1.88}$$

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.

$$v_{.j}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{x}_{.j} \otimes \mathbf{y}_{.j} \tag{1.89}$$

$$\left[\nabla^J_{\mathbf{X}}\right]_{.j} \leftarrow \left[\nabla^J_{\mathbf{X}}\right]_{.j} + \left[\left[\nabla^J_{\mathbf{n}}\right]_{.j}\right]_{\mathbf{X}.rows, \mathbf{Y}.rows} \mathbf{Y} \tag{1.90}$$

$$\left[\nabla^J_{\mathbf{Y}}\right]_{.j} \leftarrow \left[\nabla^J_{\mathbf{Y}}\right]_{.j} + \left[\left[\left[\nabla^J_{\mathbf{n}}\right]_{.j}\right]_{\mathbf{X}.rows, \mathbf{Y}.rows}\right]^T \mathbf{X}, \tag{1.91}$$

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

$$\frac{\partial v_{mn}}{\partial x_{ij}} = \begin{cases} y_{kj} & n = j \wedge i = m/\mathbf{Y}.rows \wedge k = modulus(m, \mathbf{Y}.rows) \\ 0 & else \end{cases} \tag{1.92}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial x_{ij}} = \sum_{i,k} \frac{\partial J}{\partial \upsilon_{i \times y.rows+k,j}} y_{kj}. \tag{1.93}$$

The gradient $\nabla^J_{\mathbf{y}}$ 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.

$$\upsilon_{.j}(\mathbf{X}, \mathbf{Y}) \leftarrow \frac{\mathbf{x}_{.j}^T \mathbf{y}_{.j}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} \tag{1.94}$$

$$\left[\nabla^J_{\mathbf{X}}\right]_{.j} \leftarrow \left[\nabla^J_{\mathbf{X}}\right]_{.j} + \left[\nabla^J_{\mathbf{n}}\right]_{.j} \bullet \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} \upsilon_{.,j}}{\|\mathbf{x}_{.j}\|^2}\right] \tag{1.95}$$

$$\left[\nabla^J_{\mathbf{Y}}\right]_{.j} \leftarrow \left[\nabla^J_{\mathbf{Y}}\right]_{.j} + \left[\nabla^J_{\mathbf{n}}\right]_{.j} \bullet \left[\frac{x_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{y_{ij} \upsilon_{.,j}}{\|\mathbf{y}_{.j}\|^2}\right]. \tag{1.96}$$

The gradient $\nabla^J_{\mathbf{X}}$ can be derived by observing

$$\frac{\partial \upsilon_{.n}}{\partial x_{ij}} = \begin{cases} \frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij}\left(\mathbf{x}_{.j}^T \mathbf{y}_{.j}\right)}{\|\mathbf{x}_{.j}\|^3 \|\mathbf{y}_{.j}\|} & n = j \\ 0 & else \end{cases} \tag{1.97}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \sum_n \frac{\partial J}{\partial \upsilon_{.n}} \frac{\partial \upsilon_{.n}}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon_{.,j}} \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij}\left(\mathbf{x}_{.j}^T \mathbf{y}_{.j}\right)}{\|\mathbf{x}_{.j}\|^3 \|\mathbf{y}_{.j}\|}\right]. \tag{1.98}$$

$$= \frac{\partial J}{\partial \upsilon_{.,j}} \left[\frac{y_{ij}}{\|\mathbf{x}_{.j}\| \|\mathbf{y}_{.j}\|} - \frac{x_{ij} \upsilon_{.,j}}{\|\mathbf{x}_{.j}\|^2}\right]. \tag{1.99}$$

The gradient $\nabla^J_{\mathbf{y}}$ 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 $\mathbf{X}.cols = \mathbf{Y}.cols$.

$$a_j(\mathbf{X}) \leftarrow \arg\max_i x_{ij} \tag{1.100}$$
$$b_j(\mathbf{Y}) \leftarrow \arg\max_i y_{ij} \tag{1.101}$$
$$v(\mathbf{X},\mathbf{Y}) \leftarrow \sum_j \delta\left(a_j(\mathbf{X}) \neq b_j(\mathbf{Y})\right) \tag{1.102}$$

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 $ComputePartialGradient(\mathbf{b})$ 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 $\mathbf{X}.cols = \mathbf{Y}.cols$.

$$v(\mathbf{X},\mathbf{Y}) \leftarrow \frac{1}{2}\mathrm{Tr}\left((\mathbf{X}-\mathbf{Y})(\mathbf{X}-\mathbf{Y})^T\right) \tag{1.103}$$
$$\nabla^J_{\mathbf{X}} \leftarrow \nabla^J_{\mathbf{X}} + \nabla^J_{\mathbf{n}}(\mathbf{X}-\mathbf{Y}) \tag{1.104}$$
$$\nabla^J_{\mathbf{Y}} \leftarrow \nabla^J_{\mathbf{Y}} - \nabla^J_{\mathbf{n}}(\mathbf{X}-\mathbf{Y}). \tag{1.105}$$

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

$$\frac{\partial v}{\partial \mathbf{X}} = \mathbf{X}-\mathbf{Y} \tag{1.106}$$
$$\frac{\partial v}{\partial \mathbf{Y}} = -(\mathbf{X}-\mathbf{Y}). \tag{1.107}$$

- *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 $\mathbf{X}.rows = \mathbf{Y}.rows$ and $\mathbf{X}.cols = \mathbf{Y}.cols$.

$$\mathbf{R}(\mathbf{Y}) \leftarrow \log(\mathbf{Y}) \tag{1.108}$$
$$v(\mathbf{X},\mathbf{Y}) \leftarrow -\mathrm{vec}(\mathbf{X}) \circ \mathrm{vec}(\mathbf{R}(\mathbf{Y})) \tag{1.109}$$
$$\nabla^J_{\mathbf{X}} \leftarrow \nabla^J_{\mathbf{X}} - \nabla^J_{\mathbf{n}}\mathbf{R}(\mathbf{Y}) \tag{1.110}$$
$$\nabla^J_{\mathbf{Y}} \leftarrow \nabla^J_{\mathbf{Y}} - \nabla^J_{\mathbf{n}}(\mathbf{X} \oslash \mathbf{Y}). \tag{1.111}$$

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

$$\frac{\partial \upsilon}{\partial x_{ij}} = -\log(y_{ij}) = -r_{ij}(\mathbf{Y}) \tag{1.112}$$

and

$$\frac{\partial J}{\partial x_{ij}} = \frac{\partial J}{\partial \upsilon}\frac{\partial \upsilon}{\partial x_{ij}} = -\frac{\partial J}{\partial \upsilon} r_{ij}(\mathbf{Y}) \tag{1.113}$$

Similarly to derive the gradient $\nabla^J_{\mathbf{Y}}$ we note that

$$\frac{\partial \upsilon}{\partial y_{ij}} = -\frac{x_{ij}}{y_{ij}} \tag{1.114}$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \frac{\partial J}{\partial \upsilon}\frac{\partial \upsilon}{\partial y_{ij}} = -\frac{\partial J}{\partial \upsilon}\frac{x_{ij}}{y_{ij}}. \tag{1.115}$$

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

$$\begin{aligned}
\mathbf{P}(\mathbf{Y}) &\leftarrow \text{Softmax}(\mathbf{Y}) && (1.116)\\
\mathbf{R}(\mathbf{Y}) &\leftarrow \log(\mathbf{P}(\mathbf{Y})) && (1.117)\\
v(\mathbf{X},\mathbf{Y}) &\leftarrow \text{vec}(\mathbf{X}) \circ \text{vec}(\mathbf{R}(\mathbf{Y})) && (1.118)\\
\nabla^J_{\mathbf{X}} &\leftarrow \nabla^J_{\mathbf{X}} - \nabla^J_{\mathbf{n}}\mathbf{R}(\mathbf{Y}) && (1.119)\\
\nabla^J_{\mathbf{Y}} &\leftarrow \nabla^J_{\mathbf{Y}} + \nabla^J_{\mathbf{n}}(\mathbf{P}(\mathbf{Y}) - \mathbf{X}) && (1.120)
\end{aligned}$$

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

$$\frac{\partial \upsilon}{\partial y_{ij}} = \mathbf{p}_{ij}(\mathbf{Y}) - x_{ij} \tag{1.121}$$

and get

$$\frac{\partial J}{\partial y_{ij}} = \frac{\partial J}{\partial \upsilon}\frac{\partial \upsilon}{\partial y_{ij}} = \frac{\partial J}{\partial \upsilon}(\mathbf{p}_{ij}(\mathbf{Y}) - x_{ij}). \tag{1.122}$$

1.4.4 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

$$\begin{aligned} k &\leftarrow k + \mathbf{X}.cols && (1.123)\\ \upsilon(\mathbf{X}) &\leftarrow \frac{1}{k}\mathbf{X}\mathbf{1}_{\mathbf{X}.cols,1} + \frac{k-\mathbf{X}.cols}{k}\upsilon(\mathbf{X}) && (1.124) \end{aligned}$$

  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

$$\begin{aligned} k &\leftarrow k + \mathbf{X}.cols && (1.125)\\ \upsilon(\mathbf{X}) &\leftarrow \frac{1}{k}\mathbf{X}\mathbf{1}_{\mathbf{X}.cols,1} + \frac{k-\mathbf{X}.cols}{k}\upsilon(\mathbf{X}) && (1.126)\\ \omega(\mathbf{X}) &\leftarrow \frac{1}{k}(\mathbf{X}\bullet\mathbf{X})\mathbf{1} + \frac{k-\mathbf{X}.cols}{k}\omega(\mathbf{X}) && (1.127) \end{aligned}$$

  When the end of the training set is reached,

$$\begin{aligned} \upsilon &\leftarrow (\omega - (\upsilon\bullet\upsilon))^{1/2} && (1.128)\\ \upsilon &\leftarrow 1 \oslash \upsilon. && (1.129) \end{aligned}$$

- *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.

$$V(\mathbf{X}) \leftarrow (\mathbf{X}-\mathbf{m})\bullet\mathbf{s}. \quad (1.130)$$

1.5 Convolutional Neural Network

A Convolutional neural network (CNN) [7, 8, 9, 10, 11, 12, 13, 14, 15, 16] 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 [11, 15, 16, 13, 14]. 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 1.7, 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 1.7 $C_x = 3$, $C_v = 2$ 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)

$$\upsilon_{\ell ij}(\mathbf{K}, \mathbf{Y}) = \sum_k \text{vec}(\mathbf{K}_{k\ell}) \circ \text{vec}\left(\mathbf{Y}_{kij}\right), \tag{1.131}$$

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. [8] proposed a technique to convert all these small matrix operations to a large matrix product as shown at the bottom of Figure 1.7. With this trick all the kernel parameters are combined into a big kernel matrix $\mathbf{W}$ as shown at left bottom of Figure 1.7. Note that to allow for the output of the convolution node to be used by another convolution node, in Figure 1.7 we have organized the conversion slightly differently from what proposed by Chellapilla et al. [8] 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

$$O_r = \begin{cases} \frac{I_r - K_r}{S_r} + 1 & \text{no padding} \\ \frac{(I_r - \text{mod}(K_r, 2))}{S_r} + 1 & \text{zero padding} \end{cases} \tag{1.132}$$

is the number of rows in the output image, and

$$O_c = \begin{cases} \frac{I_c - K_c}{S_c} + 1 & \text{no padding} \\ \frac{(I_c - \text{mod}(K_c, 2))}{S_c} + 1 & \text{zero padding} \end{cases} \tag{1.133}$$

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

$$\begin{aligned} \mathbf{X}(\mathbf{Y}) &\leftarrow \text{Pack}(\mathbf{Y}) && (1.134) \\ V(\mathbf{W}, \mathbf{Y}) &\leftarrow \mathbf{W}\mathbf{X} && (1.135) \\ \nabla^J_{\mathbf{W}} &\leftarrow \nabla^J_W + \nabla^J_{\mathbf{n}} \mathbf{X}^T && (1.136) \\ \nabla^J_{\mathbf{X}} &\leftarrow \mathbf{W}^T \nabla^J_{\mathbf{n}} && (1.137) \\ \nabla^J_{\mathbf{Y}} &\leftarrow \nabla^J_{\mathbf{Y}} + \text{Unpack}\left(\nabla^J_{\mathbf{X}}\right). && (1.138) \end{aligned}$$

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

$$\begin{aligned} \upsilon_{\ell ij}(\mathbf{X}) &\leftarrow \max\left(\mathbf{X}_{\ell ij}\right) && (1.139) \\ \left[\nabla^J_{\mathbf{X}}\right]_{\ell, i_m, j_n} &\leftarrow \begin{cases} \left[\nabla^J_{\mathbf{X}}\right]_{\ell, i_m, j_n} + \left[\nabla^J_{\mathbf{n}}\right]_{\ell, i_m, j_n} & (m,n) = arg\max_{m,n} x_{\ell, i_m, j_n} \\ \left[\nabla^J_{\mathbf{X}}\right]_{\ell, i_m, j_n} & else \end{cases} && (1.140) \end{aligned}$$

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

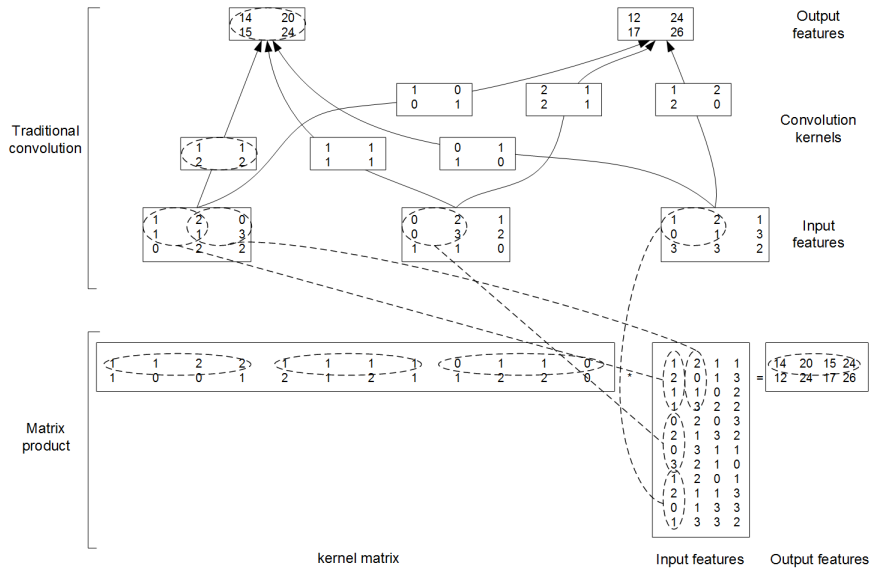

Figure 1.7: Example convolution operations. Top: the original convolution operations. Bottom: the same operations represented as a large matrix product. Our matrix organization is different from what proposed by Chellapilla et al. [8] to allow for stacked convolution operations. (Modified from the figure in Chellapilla et al. [8], permitted to use by Simard)

- *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

$$\upsilon_{\ell ij}(\mathbf{X}) \leftarrow \frac{1}{K_r \times K_c} \sum_{m,n} x_{\ell, i_m, j_n} \tag{1.141}$$

$$\left[\nabla^J_{\mathbf{X}}\right]_{\ell ij} \leftarrow \left[\nabla^J_{\mathbf{X}}\right]_{\ell ij} + \frac{1}{K_r \times K_c} \left[\nabla^J_{\mathbf{n}}\right]_{\ell ij}, \tag{1.142}$$

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

1.6 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.

$$\upsilon_{.j}(\lambda, \mathbf{Y}) \leftarrow \mathbf{Y}_{.(j-\lambda)} \tag{1.143}$$

$$\left[\nabla^J_{\mathbf{Y}}\right]_{.j} \leftarrow \left[\nabla^J_{\mathbf{Y}}\right]_{.j} + \left[\nabla^J_{\mathbf{n}}\right]_{.j+\lambda}. \tag{1.144}$$

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

$$\frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \begin{cases} 1 & m = i \wedge n = j + \lambda \\ 0 & else \end{cases} \tag{1.145}$$

and

$$\frac{\partial J}{\partial y_{ij}} = \sum_{m,n} \frac{\partial J}{\partial \upsilon_{mn}} \frac{\partial \upsilon_{mn}}{\partial y_{ij}} = \frac{\partial J}{\partial \upsilon_{ij+\lambda}}. \tag{1.146}$$

An example CN that contains a delay node is shown in Figure 1.8. 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 the previous samples. A simple way to do forward computation and backpropagation

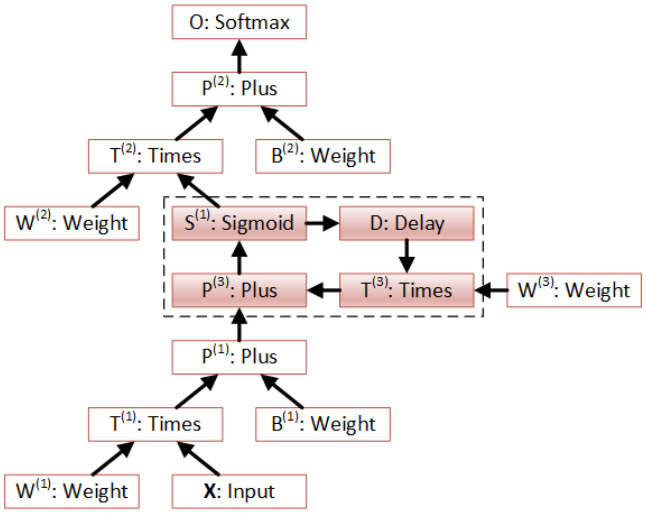

Figure 1.8: An example CN with a delay node. The shaded nodes form a recurrent loop which can be treated as a composite node.

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 calcalclation 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.

1.6.1 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 1.8 all the nodes included in the loop of $\mathbf{T}^{(3)} \rightarrow \mathbf{P}^{(3)} \rightarrow \mathbf{S}^{(1)} \rightarrow \mathbf{D} \rightarrow \mathbf{T}^{(3)}$ 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 [28][1]. Once

[1] Tarjan's algorithm is favored over others such as Kosaraju's algorithm [29] since it only requires one depth-first traverse, has a complexity of $O(|\mathbb{V}| + |\mathbb{E}|)$, and does not require reversing arcs in the

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 backprogation 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 1.6. Since the input to the delay nodes are computed in the past they can be considered as the leaf if we only consider one time slice. This makes the order decision inside loops much easier.

1.6.2 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 1.9. 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 1.9 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.

1.6.3 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) [17, 30] neural network that is widely used to recognize and generate hand-written characters involves the following operations:

$$\mathbf{i}_t = \sigma\left(\mathbf{W}^{(xi)}\mathbf{x}_t + \mathbf{W}^{(hi)}\mathbf{h}_{t-1} + \mathbf{W}^{(ci)}\mathbf{c}_{t-1} + \mathbf{b}^{(i)}\right) \quad (1.147)$$

$$\mathbf{f}_t = \sigma\left(\mathbf{W}^{(xf)}\mathbf{x}_t + \mathbf{W}^{(hf)}\mathbf{h}_{t-1} + \mathbf{W}^{(cf)}\mathbf{c}_{t-1} + \mathbf{b}^{(f)}\right) \quad (1.148)$$

$$\mathbf{c}_t = \mathbf{f}_t \bullet \mathbf{c}_{t-1} + \mathbf{i}_t \bullet \tanh\left(\mathbf{W}^{(xc)}\mathbf{x}_t + \mathbf{W}^{(hc)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right) \quad (1.149)$$

$$\mathbf{o}_t = \sigma\left(\mathbf{W}^{(xo)}\mathbf{x}_t + \mathbf{W}^{(ho)}\mathbf{h}_{t-1} + \mathbf{W}^{(co)}\mathbf{c}_t + \mathbf{b}^{(o)}\right) \quad (1.150)$$

$$\mathbf{h}_t = \mathbf{o}_t \bullet \tanh\left(\mathbf{c}_t\right), \quad (1.151)$$

graph.

Algorithm 1.6 Forward computation of an arbitrary CN

procedure DECIDEFORWARDCOMPUTATIONORDERWITHRECCURENT-LOOP($G = (V, E)$)
    StronglyConnectedComponentsDetection(G, G') ▷ G' is a DAG of strongly connected components (SCC)
    Call DecideForwardComputationOrder on $G' \rightarrow order$ for DAG
    for $v \in G, v \in V$ **do**
        Set the *order* of v equal the max order of the SCC V ▷ This guarantee the forward order of SCC is correct
    end for
    for each SCC V in G **do**
        Call GetLoopForwardOrder(*root* of V) $\rightarrow$ *order* for each SCC
    end for
    return *order* for DAG and *order* for each SCC (loop)
end procedure
procedure GETLOOPFORWARDORDER(*root*)
    Treat all the *delayNode* as leaf and Call DecideForwardComponentionOrder
end procedure
procedure STRONGLYCONNECTEDCOMPONENTSDETECTION($G = (V, E), DAG$)
    $index = 0, S = empty$
    for $v \in V$ **do**
        if $v.index$ is undefined **then** StrongConnectComponents(v, DAG)
        end if
    end for
end procedure
procedure STRONGCONNECTCOMPONENT(v, DAG)
    $v.index = index, v.lowlink = index, index = index + 1$
    $S.push(v)$
    for $(v, w) \in E$ **do**
        if $w.index$ is undefined **then**
            StrongConnectComponent(w)
            $v.lowlink = \min(v.lowlink, w.lowlink)$
        else if $w \in S$ **then**
            $v.lowlink = \min(v.lowlink, w.index)$
        end if
    end for
    if $v.lowlink = v.index$ **then** ▷ If v is a root node, pop the stack and generate an SCC
        start a new strongly connected component
        repeat
            $w = S.pop()$
            add w to current strongly connected component
        until $w == v$
        Save current strongly connected component to *DAG*
    end if
end procedure

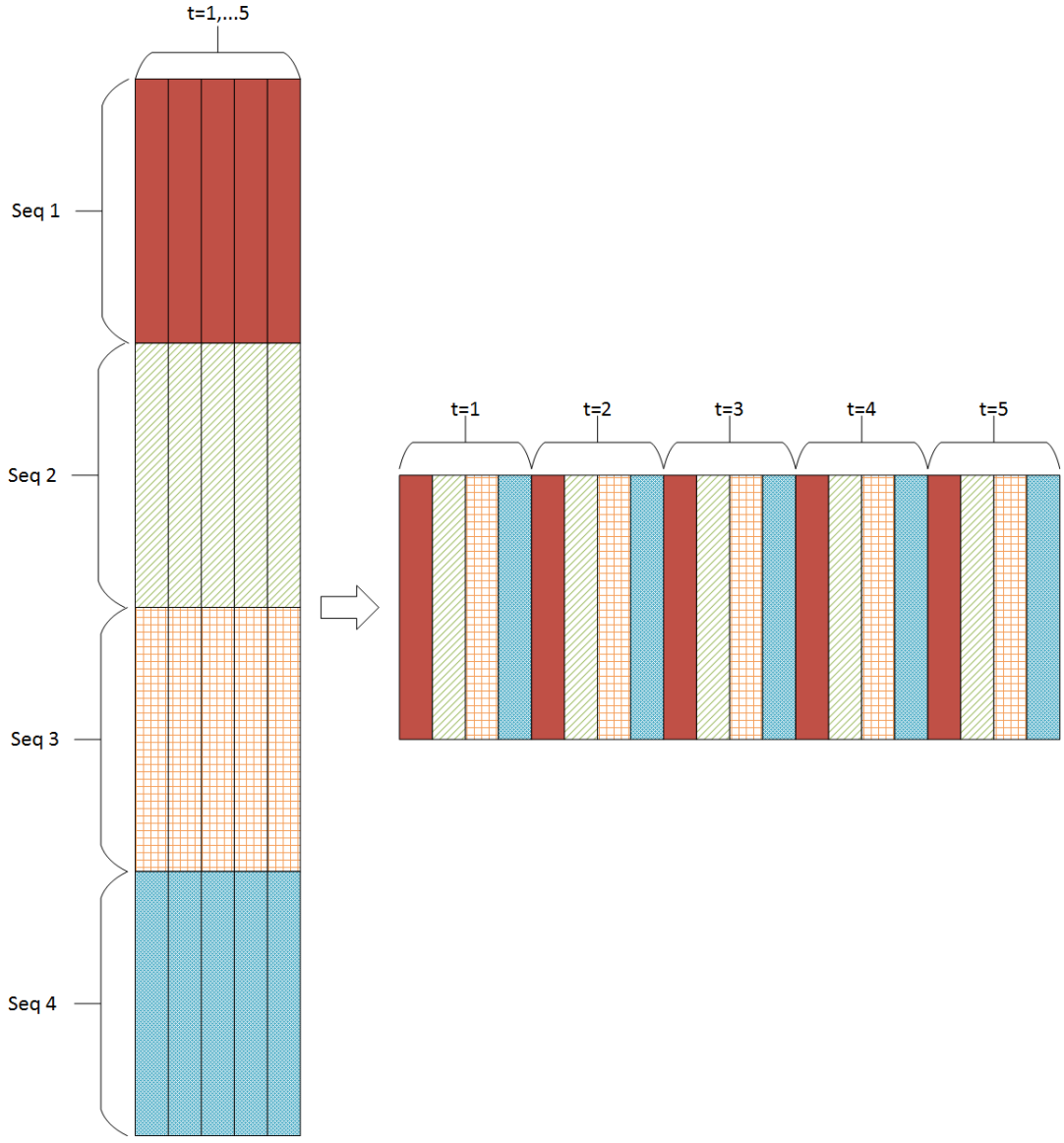

Figure 1.9: Process multiple sequences in a batch. Shown in the figure is an example with 4 sequences. Each color represents one sequence. Frames with the same frame id from different sequences are grouped together and computed in batch. The right matrix is a reshape of the left matrix.

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

$$\begin{aligned}
\mathbf{H}^{(d)} &= Delay(\mathbf{H}) && (1.152)\\
\mathbf{C}^{(d)} &= Delay(\mathbf{C}) && (1.153)\\
\mathbf{T}^{(1)} &= Macro2W1b(\mathbf{W}^{(xi)}, \mathbf{X}, \mathbf{W}^{(hi)}, \mathbf{H}^{(d)}, \mathbf{b}^{(i)}) && (1.154)\\
\mathbf{I} &= Sigmoid\left(Plus\left(\mathbf{T}^{(1)}, DiagTimes\left(\mathbf{d}^{(ci)}, \mathbf{C}^{(d)}\right)\right)\right) && (1.155)\\
\mathbf{T}^{(2)} &= Macro3W1b\left(\mathbf{W}^{(xf)}, \mathbf{X}, \mathbf{W}^{(hf)}, \mathbf{H}^{(d)}, \mathbf{W}^{(cf)}\mathbf{C}^{(d)}, \mathbf{b}^{(f)}\right) && (1.156)\\
\mathbf{F} &= Sigmoid\left(\mathbf{T}^{(2)}\right) && (1.157)\\
\mathbf{T}^{(3)} &= Macro2W1b\left(\mathbf{W}^{(xc)}, \mathbf{X}, \mathbf{W}^{(hc)}, \mathbf{H}^{(d)}, \mathbf{b}^{(c)}\right) && (1.158)\\
\mathbf{T}^{(4)} &= ElementTime\left(\mathbf{F}, \mathbf{C}^{(d)}\right) && (1.159)\\
\mathbf{T}^{(5)} &= ElementTimes\left(\mathbf{I}, Tanh\left(\mathbf{T}^{(3)}\right)\right) && (1.160)\\
\mathbf{C} &= Plus\left(\mathbf{T}^{(4)}, \mathbf{T}^{(5)}\right) && (1.161)\\
\mathbf{T}^{(6)} &= Macro3W1b\left(\mathbf{W}^{(xo)}, \mathbf{X}, \mathbf{W}^{(ho)}, \mathbf{H}^{(d)}, \mathbf{W}^{(co)}\mathbf{C}, \mathbf{b}^{(o)}\right) && (1.162)\\
\mathbf{O} &= Sigmoid\left(\mathbf{T}^{(6)}\right) && (1.163)\\
\mathbf{H} &= ElementTime\left(\mathbf{O}, Tanh\left(\mathbf{C}\right)\right), && (1.164)
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{S}^{(1)} &= Plus\left(Times\left(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}\right), Times\left(\mathbf{W}^{(2)}, \mathbf{I}^{(2)}\right)\right) && (1.165)\\
Macro2W1b &= Plus\left(\mathbf{S}^{(1)}, \mathbf{b}\right) && (1.166)
\end{aligned}$$

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

$$\mathbf{S}^{(1)} = Macro2W1b\left(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}, \mathbf{W}^{(2)}, \mathbf{I}^{(2)}, \mathbf{b}\right) \tag{1.167}$$

$$Macro3W1b = Plus\left(\mathbf{S}^{(1)}, Times\left(\mathbf{W}^{(3)}, \mathbf{I}^{(3)}\right)\right) \tag{1.168}$$

Chapter 2

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.

2.1 Run CNTK

To run CNTK you use a command line similar to

```
cn.exe configFile=yourExp.config
```

where yourExp.config 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

  NDLNetworkBuilder=[
```

```
    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
  - SimpletNetworkBulder: 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.
  - SequenceReader: 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.

2.2 Network Builders

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

2.2.1 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 $[-0.05 \times initValueScale, 0.05 \times initValueScale]$. If the model parameters are initialized using the Gaussian distribution, the standard deviation will be adjusted to $0.2 \times initValueScale/\sqrt{fanout}$.

- *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.

- *layerSize*s: 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
  - SIMPLENET: 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 uesd for language modeling tasks.

2.2.2 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 3.

- *networkDescription*: the file path of the NDL script, which will be described in Chapter 3, 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.

- *ru*n: 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 3, 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 3, 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.

2.3 Learners

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

2.3.1 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.

2.3.1.1 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).

2.3.1.2 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.

- *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*: weather to load the best model if the current model decreases the performance. Valid values are true (default) and false.

  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.

2.3.1.3 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 thread for each sample. Default value is 1#INF which means infinity (i.e., clipping is turned off).
- *gradUpdateType*: gradient update type. Valid values are None (default, no special treatment to the gradient), AdaGrad, and RmsProp.
- *gaussianNoiseInjectStd*: the standard deviation of the Gaussian noise added when using the AdaGrad approach. Default is 0.

2.3.1.4 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 3.

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

2.3.1.5 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.

2.3.1.6 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.

2.4 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 4. 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.

2.4.1 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.

2.4.2 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 rollowing window over the data is this file and randomizing the data within the window. This method produces more thorough randomization of the data but requires a large temprorary 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.

- *randomize:* Auto or None, whether the reader should randomize the data
- *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 Descrition 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 confguration 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 thee 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.

2.4.3 SequenceReader

SequenceReader 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=SequenceReader
  randomize=false
  nbruttineachrecurrentiter=10
  unk="<unk>"
  wordclass=$DataDir$\wordclass.txt
  file=$DataDir$\penntreebank.train.txt

  labelIn=[
    labelDim=10000

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

]
```

In this example, the SequenceReader 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 begining 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 occurances, 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.

2.4.4 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

  nbruttineachrecurrentiter=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 begining 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 – [Ture, 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.
- *label*s: 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.

2.5 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.

2.5.1 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.
- NDLNetworkBuilder – the NDL network builder configuration block.
- simpleNetworkBuilde] – 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 Edit Command

This command executes a model editing language (MEL) script. We will describe MEL in detail in Chapter 3. 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.

2.5.7 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.

2.5.8 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.

2.6 Additional Top-Level Configurations

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

2.6.1 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.

2.6.2 DeviceId

CNTK supports CPU and GPU computation. Users can determine what device to use by setting the deviceId papameter. 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)

2.6.3 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.

2.6.4 TraceLevel

The traceLevel parameter is uniformly 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 ouput) and 2 (verbose output) are the only values supported.

2.7 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

```
cn.exe configFile=yourExp.config
```

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

2.7.1 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

]
```

2.7.2 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:

```
cn.exe configFile=yourExp.config 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

```
cn.exe configFile=yourExp.config mnistTrain=[minibatchSize
    =256]
```

or modify the data file used for an experiment as

```
cn.exe configFile=yourExp.config mnistTrain=[reader=[file=
    mynewfile.txt]]
```

2.7.3 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 "configFile=" tag multiple times. The following are equivalent.

```
cn.exe configFile=yourExp1.config+yourExp2.config
cn.exe configFile=yourExp1.config configFile=yourExp2.
    config
```

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

```
cn.exe configFile=yourExp1.config 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,

```
cn.exe configFile=yourExp1.config+yourExp2.config 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.config was

```
include=yourExp1.config
```

then simply running

```
cn.exe configFile=yourExp2.config
```

would be equivalent to running

```
cn.exe configFile=yourExp1.config+yourExp2.config
```

where in this latter case, yourExp2.config 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.config includes yourExpB.config, and yourExpB.config includes yourExpC.config, then the full chain will be resolved, and yourExpC.config will effectively be included in yourExpA.config. 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.

2.7.4 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.

2.7.5 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 3

Advanced Setups in Computational Network Toolkit

3.1 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.

3.1.1 Basic Concepts

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

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

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

#Parameters
W0=Parameter(HDim, SDim)
B0=Parameter(HDim)
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
CrossEntropy=CrossEntropyWithSoftmax(labels, Plus2)

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

#Special Nodes
FeatureNodes=(features)
LabelNodes=(labels)
CriteriaNodes=(CrossEntropy)
EvalNodes=(ErrPredict)
OutputNodes=(Plus2)
```

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

3.1.1.1 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, CriteriaNodes, 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.

3.1.1.2 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.

3.1.1.3 Parameters

```
B0=Parameter(HDim)
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 3.1.4).

3.1.1.4 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 3.1.4.

The hidden layer in this example involves three computation steps: It first gets the product of the weight matrix W0 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.

3.1.1.5 Training and Testing Criteria

```
#Training Criterion
CrossEntropy=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 CrossEntropy and ErrPredict, respectively.

3.1.1.6 Special Nodes

```
FeatureNodes=(features)
LabelNodes=(labels)
CriteriaNodes=(CrossEntropy)
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 which are 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.

3.1.1.7 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
```

3.1.2 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.

3.1.2.1 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.

3.1.2.2 Using Macros

The following example uses the macros defined above. It describes the same one-hidden-layer network discussed in Section 3.1.1 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)
```

3.1.3 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.

3.1.3.1 Parameter initialization

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

```
B0=Parameter(HDim, 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 3.1.4 to find which functions accept optional parameters

3.1.3.2 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)
CriteriaNodes=(CrossEntropy)
```

```
EvalNodes=(ErrPredict)
```

we can tag these nodes as they are defined.

```
features=Input(SDim, tag=feature)
labels=Input(LDim, tag=label)
CE = SMBFF(L3, LDim, HDim, labels, tag=Criteria)
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
- *criteria*: training criteria
- *eval*: evaluation node
- *output*: output node

3.1.4 NDL Functions

This section describe the currently implemented NDL functions (operators). These operations are implemented as computation nodes internally and are discussed in Chapter 1. As CNTK is expanded, additional functions become available. In the following discussion, *ordered optional* parameters are operational parameters that are identified by the position of the parameter in the argument list, and named optional parameters are optional parameters that are specified by an augment name. For example, if cols is an ordered optional parameter in

```
Function(rows, [cols=1])
```

we can use

```
Function(rowsValue, colsValue)
```

to specify the cols value. If cols is a named optional parameter, we need to call the function as

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

instead.

3.1.4.1 Input, InputValue

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

```
Input(rows, [cols=1], {tag=feature|label})
InputValue(rows, [cols=1])
```

- rows - row dimension of the data.
- cols - [ordered optional] 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.

3.1.4.2 ImageInput, 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=1],
           {tag=feature|label})
Image(width, height, channels, [numImages=1],
           {tag=feature|label})
```

- 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 - [ordered optional] 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].

3.1.4.3 Parameter, LearnableParameter

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

```
Parameter(row, [cols=1], {needGradient=true|false,
          init=fixedValue|Uniform|Gaussian|fromFile,
          value=0, initValueScale=number})
LearnableParameter(row, [cols=1],
          {computeGradient=true|false,
          init=fixedValue|Uniform|Gaussian|fromFile,
          value=0, initValueScale=number})
```

- rows - number of rows in the parameter.
- cols - [ordered optional] 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 $[-0.05 \times initValueScale, 0.05 \times initValueScale]$
  - Gaussian - Initializes the parameter matrix with Gaussian random numbers with zero mean and standard deviation of $0.2 \times initValueScale/\sqrt{cols}$
- 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.

3.1.4.4 Constant, 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=1, cols=1])
Const(value, [rows=1, cols=1])
```

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

3.1.4.5 RectifiedLinear, 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)
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).

3.1.4.6 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.

3.1.4.7 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

3.1.4.8 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.

3.1.4.9 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

```
RowSlicete(m, startRow, numRows)
```

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

3.1.4.10 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

3.1.4.11 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.

3.1.4.12 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.

3.1.4.13 Plus, Minus, ElementTimes

Calculate the sum (Plus), difference (Minus), or element-wise product (ElementTimes) of two matrices. The resulting matrices have the same dimension as that of the input matrices. The syntax is

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

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

3.1.4.14 KhatriRaoProduct, ColumnwiseCrossProduct

Compute the cross product of each column of two input matrices. These two functions mean the same thing but ColumnwiseCrossProduct is easier to understand for most people. The resulting matrix is a (m1.rows times m2.rows) by m1.cols matrix. The syntax is

```
KhatriRaoProduct(m1, m2)
ColumnwiseCrossProduct(m1, m2)
```

- m1, m2 - input matrices. The matrices should have same columns.

3.1.4.15 SquareError, 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)
```

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

3.1.4.16 CrossEntropyWithSoftmax, 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)
```

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

3.1.4.17 ErrorPrediction, 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.

3.1.4.18 CosDistance, 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)
```

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

3.1.4.19 MatrixL1Reg, L1Reg, MatrixL2Reg, 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)
MatrixL2Reg(m)
L2Reg(m)
```

m - input matrix

3.1.4.20 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

3.1.4.21 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

3.1.4.22 PerDimMeanVarNormalization, 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)
```

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.

3.1.4.23 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

3.1.4.24 Convolution, 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])
Convolve(w, image, kernelWidth, kernelHeight,
            outputChannels, horizontalSubsample,
            verticalSubsample, [zeroPadding=false,
            maxTempMemSizeInSamples=0])
```

- 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.

3.1.4.25 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)
AveragePooling(m, windowWidth, windowHeight, stepW, stepH)
```

- 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

3.1.4.26 Delay

Used to apply a value in the past to the current time. 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

```
Delay(m, [delayTime=1, defaultPastValue=0.1])
```

- m - input matrix to be delayed. Each column is a sample. The samples may be from different utterances as explained in Chapter .1
- delayTime - [named optional] the amount of delay. Default is 1.
- defaultPastValue - [named optional] the default value to use if the past value is not available. Default is 0.1.

3.2 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 [2, 3] on DNNs by building shallow networks first and then inserting new layers one on top of another.

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.

3.2.1 Basic Features

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

```
model1 = LoadModel("c:\models\mymodel.dnn", format=cntk)
SetDefaultModel(model1)

DumpModel(model1, "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=Criteria)
Err=ErrorPrediction(labels, CE.F, tag=Eval)

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

3.2.1.1 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.

```
model1 = LoadModel("c:\models\mymodel.cn", format=cntk)
SetDefaultModel(model1)
```

Here *model1* 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.

3.2.1.2 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(model1, "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
...
```

3.2.1.3 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.

3.2.1.4 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 checkes 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).

3.2.1.5 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.

3.2.1.6 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.

3.2.1.7 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]}.[nameNode]
```

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 nameNode 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

3.2.2 MEL Command Reference

This section contains the currently implemented MEL Command functions.

3.2.2.1 CreateModel, CreateModelWithName

Creates a new empty model. The syntax is

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

- m - the name of newly created model

3.2.2.2 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.

3.2.2.3 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.

3.2.2.4 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.

3.2.2.5 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.

3.2.2.6 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.

3.2.2.7 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.

3.2.2.8 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.

3.2.2.9 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.

3.2.2.10 SetInput, 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.

3.2.2.11 SetInputs, 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.

3.2.2.12 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=true|false: A flag that determine if a node participates in gradient calculations. Applies to Parameter nodes
- Feature=true|false: Sets the node as a feature input. Applies to input nodes.
- Label=true|false: Set the node as a label input. Applies to input nodes.
- FinalCriterionCriteria=true|false: Sets the node as one of the criteria nodes of the network.
- EvaluationEval=true|false: Set the node as one of the evaluation nodes.
- Output=true|false: Set the node as one of the output nodes.

3.2.2.13 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

3.2.2.14 Remove, RemoveNode, Delete, DeleteNode

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

```
Remove(node, [node2, node3])
Delete(node, [node2, node3])
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.

3.2.2.15 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)
```

- 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 4

Extending the Computational Network Toolkit

CNTK is designed for extension as illustrated by Figure 4.1, which indicates that the building blocks in CNTK are decoupled by interfaces. It separates the core computational network operations, training algorithms, network builders, and data readers. Adding new computation nodes and data readers to fit your needs is as simple as plug and play as we will introduce in this chapter.

At the center of the CNTK is the ComputationNetwork class, which manages the life span of computation nodes comprising the network and all the functions operating at the network level such as forward computations and gradient calculations. To build a computational network you need to use one of the ComputationNetBuilder classes that implement the IComputationNetBuilder interface. These classes include SimpleNetworkBuilder that supports building simple layer-by-layer fully connected networks, recurrent neural networks (RNNs) such as simple RNN and long short-term memory (LSTM) neural networks. It also includes NDLNetworkBuilder that can build neural network, using any computation node we have described in Section 1.4, based on the network definition language described in Chapter 3.

IDataReader is an interface for loading data and its transcriptions. Different data file format requires different data readers. CNTK already implements the UCIFastReader and the BinaryReader that reads in UCI data in either text or binary format, the HTKMLFReader that reads in HTK/MLF speech data, the SequenceReader that is designed for language model data files, and the LUSequenceReader designed for reading language understanding data files. Users need to either convert their data files into one of the data file formats already supported or implement their own data reader.

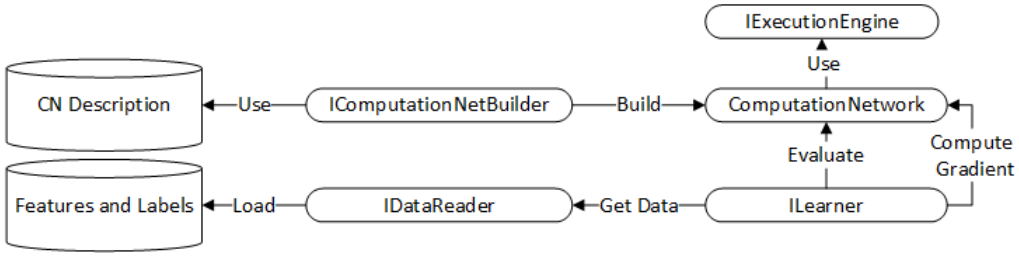

Figure 4.1: CNTK Architecture

To train a model, a learner, such as the stochastic gradient descent (SGD) learner, reads in features and labels through the IDataReader interface, calls the Evaluate and ComputeGradient methods of the ComputationNetwork object, and updates the model based on some training criterion and algorithm. The SGD learner available in CNTK implements common SGD-based parameter update methods such as momentum and AdaGrad. In addition, the SGD learner also implements a gradient checker so that users can validate gradient computations of their networks.

In most cases you will find that the existing functionality in the CNTK is sufficient to support your research. However, occasionally you may need to modify CNTK to support, for example, your special data format or computation. In this chapter we introduce how to extend the CNTK to support your special requirements. We foresee that the most frequent needs are adding a special data reader and writer, a special computation node, and a special training algorithm. This chapter is organized to cover these topics in order.

4.1 Adding a Data Reader and Writer

CNTK was designed with the idea that data input and output would need to transpire in many different formats. For this reason we have designed the data reader (IDataReader) and writer (IDataWriter) interfaces to cover various data needs. The reader/writer code is housed in separate DLLs which are dynamically loaded to provide data services. Each reader DLL exports the functions

```
extern "C" DATAREADER_API void GetReaderF(IDataReader<float
    >** preader);
extern "C" DATAREADER_API void GetReaderD(IDataReader<
    double>** preader);
```

to return the IDataReader interfaces for the floating and double precision, respectively, and each writer DLL exports the functions

```
extern "C" DATAWRITER_API void GetWriterF(IDataWriter<float
    >** pwriter);
extern "C" DATAWRITER_API void GetWriterD(IDataWriter<
    double>** pwriter);
```

to return the IDataWriter interfaces for the floating and double precision, respectively. This allows the user to simply specify, for example, a reader block, in the configuration setting to use a different reader.

4.1.1 IDataReader

To add a new reader, you need to implement the IDataReader interface

```
// implemented by DataReader and underlying classes
template<class ElemType>
class DATAREADER_API IDataReader
{
public:
    typedef std::string LabelType;
    typedef unsigned LabelIdType;

    virtual void Init(const ConfigParameters& config) = 0;
    virtual void Destroy() = 0;

 virtual void StartMinibatchLoop(size_t mbSize, size_t
    epoch, size_t requestedEpochSamples=requestDataSize) =
    0;
    virtual bool GetMinibatch(std::map<std::wstring, Matrix
       <ElemType>*>& matrices) = 0;

  virtual const std::map<typename LabelIdType, typename
     LabelType>& GetLabelMapping(const std::wstring&
     sectionName) = 0;
  virtual void SetLabelMapping(const std::wstring&
     sectionName, const std::map<typename LabelIdType,
     typename LabelType>& labelMapping) = 0;

   virtual bool GetData(const std::wstring& sectionName,
      size_t numRecords, void* data, size_t& dataBufferSize
      , size_t recordStart) = 0;
    virtual bool DataEnd(EndDataType endDataType) = 0;
```

```
    // Recursive network specific methods
    virtual size_t NumberSlicesInEachRecurrentIter() = 0;
    virtual void SetNbrSlicesEachRecurrentIter(const size_t
        ) = 0;
    virtual void SetSentenceEndInBatch(vector<size_t> &
        sentenceEnd)=0;
};
```

and the GetReaderF and GetReaderD methods, where

- *Init* – initialize the reader from a set of ConfigurationParameters. You can use configuration setups similar to that implemented in the existing readers or add new setups specific to your own reader.
- *Destroy* – release the resources used by the reader.
- *StartMinibatchLoop* – Starts the minibatch loop with the parameters
  - *mbSize* – minibatch size, can be number of frames for frame based training or number of series for sequence level training.
  - *epoch* – epoch number we are currently processing
  - *requestedEpochSize* – the number of records in an epoch. It is used to determine when an epoch ends. The epoch size can be different (larger or smaller) from the dataset size. When a user passes the constant requestDataSize as the epoch size, the actual epoch size equals the dataset size.
- *GetMinibatch* – Get the values of the next minibatch. To support multiple inputs/outputs for the CNs, *matrices*, a dictionary that maps from the computation node names to the actual matrices are passed into the function. This function returns true if the next minibatch is fetched or false if end of epoch is reached.
- *GetLabelMapping* – Get the label map from the reader, where *sectionName* specifies the section which contains the label map, if applicable. Some readers do not need a section name if only one label map is supported and some readers may not need a label mapping at all. This function returns the map from *labelId* (integer) to *label* (std::string).
- *SetLabelMapping* – Set the label map for the reader, where *sectionName* specifies the section which is assigned to the label map, if applicable, and *labelMapping* is the label map that is being set. Some readers do not need a section name or even a label map.

- *GetData* – Get data from a predefined section with parameters
  - *sectionName* – the section which contains the data,
  - *numRecords* – the number of records to read,
  - *data* – pointer to the data buffer. Needs to be released by the caller,
  - *dataBufferSize* – size of the buffer. If *dataBufferSize* equals zero or *data* equals nullptr, enough memory will be allocated by the function and the number of bytes allocated will be returned through this variable,
  - *recordStart* – the record to start reading from.
- *DataEnd* – Returns whether it is the end of a dataset, an epoch or a sentence as specified by *endDataType.*
- *NumberSlicesInEachRecurrentIter* – Get the number of slices for each truncated BPTT computation. It is used in recurrent networks.
- *SetNbrSlicesEachRecurrentIter* – Set the number of slices for each truncated BPTT computation. It is used in recurrent networks.
- *SetSentenceEndInBatch* – Set the end of sentences in the sentence minibatch. It is used in recurrent networks.

In some cases you don't need to write the data reader from scratch. For example, you can build your reader upon an existing reader by deriving from it or using it as a cache.

CNTK was designed to support multiple input and output streams. This is implemented by passing pairs of computation node names and matrices to the GetMinibatch function. In most cases you only need one named pair to get one feature stream during testing and two named pairs to get both the feature and label during training. However, you may pass any number of pairs to get as many streams as needed if it is supported by your data reader.

Randomization is important for many stochastic training algorithms. It is thus suggested that you either require users to pre-randomize the data or implement a random shuffling algorithm inside your reader. Sample and sequence should be the unit for randomization, respectively, when sample and sequence level model (e.g., RNN) or training criteria are used.

To support recurrent networks, it's suggested to implement the data reader so that multiple sequences can be used as a batch as discussed in Chapter 1. You can find example implementations in the HTKMLFReader.

4.1.2 IDataWriter

To add a new writer, you need to implement the IDataWriter interface

```
// implemented by some DataWriters
template<class ElemType>
class DATAWRITER_API IDataWriter
{
public:
    typedef std::string LabelType;
    typedef unsigned LabelIdType;

    virtual void Init(const ConfigParameters& config) = 0;
    virtual void Destroy() = 0;

    virtual void GetSections(std::map<std::wstring,
        SectionType, nocase_compare>& sections) = 0;
    virtual bool SaveData(size_t recordStart, const std::
        map<std::wstring, void*, nocase_compare>& matrices,
        size_t numRecords, size_t datasetSize, size_t
        byteVariableSized) = 0;
   virtual void SaveMapping(std::wstring saveId, const std
      ::map<typename LabelIdType, typename LabelType>&
      labelMapping) = 0;
};
```

and the GetWriterF and GetWriterD methods, where

- *Init* – Initialize the writer from a set of ConfigurationParameters.
- *Destroy* – Release the resources used by the writer.
- *GetSections* – Gets the *sections* that are available in the file to write to.
- *SaveData* – Save data to the file with parameters
  - *recordStart* – the record to start writing to
  - *matrices* – a dictionary that maps from the section names to the data pointers. The names of the sections in the dictionary should be equal to the sections returned by GetSections().
  - *numRecords* – number of records to write out
  - *datasetSize* – size of the dataset
  - *byteVariableSized* – the number of bytes used for variable sized data.

Table 4.1: The Data Formats the Configuration Classes Support. # means any number. $ means a character used as a separator. [] means optional.

Config Type	C++ Type	Data Format
integer	int, long, short, size_t	[-]#
floating-point	float, double	[-]#.#[e{+-}#]
string	std::wstring, std::string	Any valid character
boolean	bool	T, True, 1 F, False, 0
array	ConfigArray	value:value:value value:value*#:value {|value|value|value} {|value|value*#|value} { value value value*# }
dictionary	ConfigParameters	param1=value1;param2=value2;boolparam [$param1=value1$param=value2$boolparam] [param1=value1 param=value2 boolparam]

- *SaveMapping* – Save the label mapping table, where *saveId* is the section name where the mapping will be saved and *labelMapping* is the label map from *labelId* (integer) to *label* (std::string).

4.1.3 Configuration

For users to use new data readers and writers they need to specify parameters. CNTK provides a set of easy to use configuration parsing functions. It is recommended that these functions are used so that the configuration parsing can be consistent with other components.

The programmer interface to the configuration files is contained in a few C++ classes and focuses on "just-in-time" evaluation of the parameter values. The idea is simple, leave the configuration values in string format until they actually need to be parsed into some other form. Table 4.1 summarizes the different data formats the configuration classes support.

The three most frequently used configuration classes are *ConfigValue, ConfigParameters* and *ConfigArray.*

4.1.3.1 *ConfigValue*

The *ConfigValue* class allows the just-in-time (JIT) evaluation of configuration strings. It inherits from std::string, and stores an optional configuration path string,

which is mainly used for error messages. It contains many cast operators that parse the string value into the target type on demand.

4.1.3.2 *ConfigParameters*

The *ConfigParameters* class represents dictionaries of *ConfigValue* and is used to describe the hierarchy of configuration sets. It accesses the configuration values and automatically searches up the hierarchy of ConfigParameter classes if a value is not found on the current level. The hierarchy is maintained by the order of class instantiations on the stack. *ConfigParameters* should only be created on the stack.

In configuration files the 'name=value' named pair are usually separated by newlines. However, they also can be separated by other characters and placed on the same line. The default separator for ConfigParmeters is a ';' (semicolon). This can be overridden by placing the alternate separator character immediately following the opening brace. For example '[|' causes '|' to be the separator for that ConfigParameter instance:

```
name=[|parameter1=value1|parameter2=value2|parameter3=
    value3]
```

There are several ways to access the values stored inside the ConfigParameters object *config*:

- *value = config("name")* – returns the named parameter cast to the type of the value parameter. If the named configuration parameter does not exist an exception will be thrown.
- *value = config("name", "defaultValue")* – returns the named parameter, if it doesn't exist returns defaultValue.
- *config.Exists("name")* – returns whether the named value exists in the *ConfigParameters* object *config*.

To insert elements into the *ConfigParameters* object *config* the following methods can be used:

- *config.Insert("name", value)* – inserts a new value into *config*. If the value already exists, it will be replaced, unless the value is itself another *ConfigParameters* object, or string representation surrounded by square braces '[]', in which case the parameters are "merged".
- *config.Insert("name=value")* – inserts the named pair in the format of 'name=value' into the dictionary.

4.1.3.3 *ConfigArray*

The *ConfigArray* class holds an array of *ConfigValue*s. Since *ConfigValue* is evaluated JIT. The values in the array need not be homogeneous, as long as the code knows how to interpret the value of each element.

In a *ConfigArray* the values are normally separated by the default separator character ':' (colon). However, they also can be separated by the newline character or other characters. The default separator can be overridden by placing the alternate separator character immediately following the opening brace. For example '{|' causes '|' to be the separator for a ConfigArray object as in

```
array={|c:\temp\new.txt|12*3|1e-12}
```

A value may be repeated multiple times with the '*' character followed by an integer. In the above example, there are 5 elements in the array, with three '12' values occupying the center 3 positions.

The values in a *ConfigArray* can be accessed just like values in a normal std::vector type. If the index exceeds the length of the vector the last value in the vector is returned.

4.1.3.4 Other Useful Configuration Methods

Another convenient method that exists for both *ConfigParameters* and *ConfigArray* classes is to load a config file into an existing object as in

```
config.LoadConfigFile(path.c_str());
```

It is implemented in the *ConfigParser* class, from which both of these two classes inherit.

To use this method with a *ConfigArray*, the file can simply contain a list of values each on its own line. Both simple and complex types such as *ConfigParameters* and *ConfigArray* can be contained in the array using the data format summarized in Table 4.1.

ConfigArray objects can also be converted to *argvector<T>* objects simply by assigning them as

```
ConfigArray configLearnRatesPerMB = config("
    learningRatesPerMB");
argvector<float> learnRatesPerMB = configLearnRatesPerMB;
```

ConfigParameters and *ConfigArray* objects are very flexible. However parsing is required every time a value is accessed. Accessing *argvector<T>*, on the other

hand, is very efficient. Parsing happens only when *ConfigParameters* or *ConfigArray* objects are converted to *argvector<T>* objects. Care should be taken when the value is assigned to a local variable, due to lifetime issues.

4.1.3.5 Configuration Parsing Example

The following is a code snippet showing various ways of parsing configuration files:

```
#include "commandArgUtil.h"

// process the command
void DoCommand(const ConfigParameters& config)
{
    ConfigArray command = config("command");
    for (int i=0; i < command.size(); i++)
        {
        //get the configuration parameters that match the
            command
        ConfigParameters commandParams=config(command[i]);
        ConfigArray action = commandParams("action","train
            ");

        // determine the action to perform, and do it
        for (int j=0; j < action.size(); j++)
        {
            if (action[j] == "train")
                DoTrain(commandParams);
            else if (action[j] == "test" || action[j] == "
                eval")
                DoEval(commandParams);
            else
             throw runtime_error("unknown action: " +
                 action[j] + " in command set: " + command[i
                 ]);
        }
    }
}

void DoTrain(const ConfigParameters& config)
{
    ConfigParameters configSGD=config("SGD");
    ConfigParameters readerConfig = config("reader");
```

```
    ConfigParameters configNDL = config("NDLNetworkBuilder
        ");
    IComputationNetBuilder* netBuilder = (
        IComputationNetBuilder*)new NDLBuilder(configNDL);

    DataReader* dataReader = new DataReader(readerConfig);

    ConfigArray learningRatesPerMBStr = configSGD("
        learningRatesPerMB", "");
    floatargvector learningRatesPerMB =
        learningRatesPerMBStr;
    ConfigArray minibatchSize = configSGD("minibatchSize",
        "256");
    size_t epochSize = configSGD("epochSize", "0");
    if (epochSize == 0)
    {
        epochSize = requestDataSize;
    }
    size_t maxEpochs = configSGD("maxEpochs");
    wstring modelPath = configSGD("modelPath");
    int traceLevel = configSGD("traceLevel", "0");

    SGD = sgd(learningRatesPerMB, minibatchSize, epochSize,
        maxEpochs, modelPath, traceLevel);
    sgd.Train(netBuilder, dataReader);

    delete netBuilder;
    delete dataReader;
}
```

As shown in this example parsing configurations files is very simple: you simply declare a variable on the stack and assign something from a ConfigParameters class to that variable.

The configuration classes are meant to be used on the stack as shown in this example. Storing them in member variables or allocating them using ‘new’ or other methods is not supported. This is because an internal pointer is used to link to parent objects of configuration classes. This allows us to trace up the stack and look for configuration values that exist at a higher level. Since our search traverses up the stack, we need to ensure that all the parent configuration classes still exist, which is guaranteed if all configuration parameters are stack allocated and have lifetimes that extend past any children.

4.2 Adding a New Computation Node

The set of computation nodes implemented in the CNTK are described in Chapters 1 and 3. These computation node types are sufficient for most applications. However, sometimes you may need to add new computation node types due to the special requirement you have.

Adding a new computation node involves several steps: Implementing the new computation node type, adding the new node type to the computational network, and adding it to the network builder.

4.2.1 Implementing a New Computation Node Type

In the current file structure, computation nodes are implemented in four files:

- ComputationNode.h: the base class ComputationNode and most computation nodes are implemented in this file.
- EvaluationCriterionNode.h: computation nodes used mainly as evaluation criterion are implemented in this file.
- TrainingCriterionNode.h: computation nodes used mainly as training criterion are implemented in this file.
- CompositeComputationNode.h: complicated computation nodes such as those used in the convolutional neural networks are implemented in this file.

All computation node classes should be inherited from the ComputationNode class. The simplest way to create a new computation node type is to find a node type that is already implemented in the CNTK and use it as a template. Here let's use the ScaleNode as the example.

4.2.1.1 Inherits from ComputationNode<ElemType>

```
template<class ElemType>
class ScaleNode : public ComputationNode<ElemType>
```

4.2.1.2 Create Constructors

There are three constructors that need to be implemented as shown below.

```
ScaleNode(const short deviceId=AUTOPLACEMATRIX, const std::
    wstring name = L"")
: ComputationNode(deviceId)
{
    m_nodeName = (name == L""? CreateUniqNodeName() : name)
        ;
    m_deviceId = deviceId;
    MoveMatricesToDevice(deviceId);
    InitRecurrentNode();
}

ScaleNode(File& fstream, const size_t modelVersion, const
    short deviceId=AUTOPLACEMATRIX, const std::wstring name
    = L"")
        : ComputationNode(deviceId)
{
    m_nodeName = (name == L""? CreateUniqNodeName() : name)
        ;
    LoadFromFile(fstream, modelVersion, deviceId);
}

ScaleNode(const ScaleNode<ElemType>* node, const std::
    wstring& newName, const CopyNodeFlags flags)
        : ComputationNode(node->m_deviceId)
{
    node->CopyTo(this, newName, flags);
}
```

The first constructor creates the node based on a deviceId and a node name. If node name passed in is empty a new node name will be created automatically. The MoveMatricesToDevice(deviceId) call is important here. It will set the preferred computation device to deviceId and move the matrices to that device. The InitRecurrentNode() call will initialize all the members needed to handle recurrent loops in the network.

The second constructor creates a node from a file. It passes in a file stream to read data from and a modelVersion value to control how to load the file, in addition to the deviceId and node name. In this example, the actual code to load the node is in the LoadFromFile(fstream, modelVersion, deviceId) function implemented in the base class. For some complicated nodes with additional node states, you need to implement your own LoadFromFile function for your newly added node.

The third constructor creates a node by copying information from another node. It passes in a node to copy from, a name for the new node, and a copy flag. The

actual code in this example is in the node->CopyTo(this, newName, flags) function in the base class. For some complicated nodes with additional node states, you need to implement your own CopyTo function for your newly added node.

4.2.1.3 Duplicate a Node

The Duplicate function creates a new node based on the current node. Internally, it just calls the copy constructor.

```
virtual ComputationNodePtr Duplicate(const std::wstring&
    newName, const CopyNodeFlags flags) const
{
    const std::wstring& name = (newName == L"")?NodeName():
        newName;
    ComputationNodePtr node = new ScaleNode<ElemType>(this,
        name, flags);
    return node;
}
```

4.2.1.4 Give the Computation Node a Type Name

It is important to give the new computation node type a unique name that is easy to understand. This is implemented through a static function TypeName() and a member function OperationName().

```
virtual const std::wstring OperationName() const {return
    TypeName();}
static const std::wstring TypeName() {return L"Scale";}
```

To check whether a node is of a special type you can use the pattern

```
if (node->OperationName() == ScaleNode<ElemType>::TypeName
    ())
```

4.2.1.5 Attach Input Nodes

The AttachInputs function specifies the input nodes of the current node. There are three AttachInputs function defined in the base class. You only need to overwrite the one with the same number of inputs as what expected from your node.

```
virtual void AttachInputs(const ComputationNodePtr
    scalarValue, const ComputationNodePtr Value)
{
    m_children.resize(2);
```

```
    m_children[0] = scalarValue;
    m_children[1] = Value;
}
```

4.2.1.6 Propagate Image Size Information

If your input is an image, each column of the input is treated as an three dimensional image with channel, row, and column. The CopyImageSizeFromInputs propagate the image size information up through the network so that the users don't need to compute and specify them when convolutional network or pooling methods are used. In this specific example, it called the CopyImageSizeFromInput function implemented in the base class.

```
virtual void CopyImageSizeFromInputs()
{
    CopyImageSizeFromInput(1);
}
```

4.2.1.7 Validate the Node

The Validate function is used to validate the inputs and outputs of the node. It also sets the function value matrix's size and copy the image size information from it's inputs. Note that here FunctionValues() function returns the matrix that stores the value of the current computation node.

```
virtual void Validate()
{
    PrintSelfBeforeValidation();

    if (m_children.size() != 2)
        throw std::logic_error("Scale operation requires
            two inputs.");

    if (Inputs(0)->FunctionValues().GetNumElements() == 0
        || Inputs(1)->FunctionValues().GetNumElements() ==
        0)
        throw std::logic_error("Scale operation: one of the
            operants has 0 element.");

    if (Inputs(0)->FunctionValues().GetNumRows() != 1 ||
        Inputs(0)->FunctionValues().GetNumCols() != 1)
        throw std::logic_error("The left value of ScaleNode
            must be a scarlar value.");
```

```
    FunctionValues().Resize(Inputs(1)->FunctionValues().
        GetNumRows(), Inputs(1)->FunctionValues().GetNumCols
        ());

    //left Node must be a scalar
    CopyImageSizeFromInputs();
}
```

4.2.1.8 Forward Evaluation

For each node type you need to implement two forward computation functions EvaluateThisNode(), which evaluate the whole minibatch, and EvaluateThisNode(const size_t timeIdxInSeq), which is used in the recurrent networks to evaluate the timeIdxInSeq-th sample for all the sequences in the minibatch.

```
virtual void EvaluateThisNode()
{
    EvaluateThisNodeS(FunctionValues(), Inputs(0)->
        FunctionValues(), Inputs(1)->FunctionValues());
}

virtual void EvaluateThisNode(const size_t timeIdxInSeq)
{
    Matrix<ElemType> sliceInput1Value = Inputs(1)->
        FunctionValues().ColumnSlice(timeIdxInSeq *
        m_samplesInRecurrentStep, m_samplesInRecurrentStep);
    Matrix<ElemType> sliceOutputValue = m_functionValues.
        ColumnSlice(timeIdxInSeq * m_samplesInRecurrentStep,
         m_samplesInRecurrentStep);
    EvaluateThisNodeS(sliceOutputValue, Inputs(0)->
        FunctionValues(), sliceInput1Value);
}

static void WINAPI EvaluateThisNodeS(Matrix<ElemType>&
    functionValues, const Matrix<ElemType>& input0, const
    Matrix<ElemType>& input1)
{
    functionValues.AssignProductOf(input0, false, input1,
        false);
}
```

Note that both these functions call the static function

```
EvaluateThisNodeS(Matrix<ElemType>& functionValues, const
    Matrix<ElemType>& input0, const Matrix<ElemType>& input1
    )
```

which contains the actual evaluation code. In the EvaluateThisNode(const size_t timeIdxInSeq) function you will notice the calls to the ColumnSlice function. As the name suggests, this function returns a column slice of a matrix.

4.2.1.9 Gradient Computation

Similar to the forward computation, for each node type you need to implement two gradient computation functions ComputeInputPartial(const size_t inputIndex), which computes the gradient for the whole minibatch with regard to the inputIndex-th input, and ComputeInputPartial(const size_t inputIndex, const size_t timeIdxInSeq), which is used in the recurrent networks to compute the gradient of the timeIdxInSeq-th sample for all the sequences in the minibatch.

```
virtual void ComputeInputPartial(const size_t inputIndex)
{
    if (inputIndex > 1)
        throw std::invalid_argument("ScaleNode operation
            only takes two inputs.");

    //left Node must be a scalar
    if (inputIndex == 0)  //left derivative
    {
        ComputeInputPartialLeft(Inputs(1)->FunctionValues()
            , Inputs(0)->GradientValues(), GradientValues())
            ;
    }
    else
    {
    ComputeInputPartialRight(Inputs(0)->FunctionValues(),
        Inputs(1)->GradientValues(), GradientValues());
    }
}

virtual void ComputeInputPartial(const size_t inputIndex,
    const size_t timeIdxInSeq)
{
    if (inputIndex > 1)
        throw std::invalid_argument("ScaleNode operation
            only takes two inputs.");
```

```
    //left Node must be a scalar
    if (inputIndex == 0)  //left derivative
    {
        Matrix<ElemType> sliceOutputGrad = GradientValues()
            .ColumnSlice(timeIdxInSeq *
            m_samplesInRecurrentStep,
            m_samplesInRecurrentStep);
        Matrix<ElemType> sliceInput1Value = Inputs(1)->
            FunctionValues().ColumnSlice(timeIdxInSeq *
            m_samplesInRecurrentStep,
            m_samplesInRecurrentStep);
        ComputeInputPartialLeft(sliceInput1Value, Inputs(0)
            ->GradientValues(), sliceOutputGrad);
    }
    else
    {
        Matrix<ElemType> sliceInput1Grad = Inputs(1)->
            GradientValues().ColumnSlice(timeIdxInSeq *
            m_samplesInRecurrentStep,
            m_samplesInRecurrentStep);
        Matrix<ElemType> sliceOutputGrad = GradientValues()
            .ColumnSlice(timeIdxInSeq *
            m_samplesInRecurrentStep,
            m_samplesInRecurrentStep);
        ComputeInputPartialRight(Inputs(0)->FunctionValues
            (), sliceInput1Grad, sliceOutputGrad);
    }
}

static void WINAPI ComputeInputPartialLeft(const Matrix<
    ElemType>& inputFunctionValues, Matrix<ElemType>&
    inputGradientValues, const Matrix<ElemType>&
    gradientValues)
{
    inputGradientValues += Matrix<ElemType>::
        InnerProductOfMatrices(gradientValues,
        inputFunctionValues);
}

static void WINAPI ComputeInputPartialRight(const Matrix<
    ElemType>& inputFunctionValues, Matrix<ElemType>&
    inputGradientValues, const Matrix<ElemType>&
    gradientValues)
{
    Matrix<ElemType>::ScaleAndAdd(inputFunctionValues.
```

```
        Get00Element(), gradientValues, inputGradientValues)
        ;
}
```

Note that both these functions call the static functions

```
ComputeInputPartialLeft(const Matrix<ElemType>&
    inputFunctionValues, Matrix<ElemType>&
    inputGradientValues, const Matrix<ElemType>&
    gradientValues)
```

and

```
ComputeInputPartialRight(const Matrix<ElemType>&
    inputFunctionValues, Matrix<ElemType>&
    inputGradientValues, const Matrix<ElemType>&
    gradientValues)
```

which contains the actual gradient computation code.

4.2.1.10 The CNTKMath Library

In both the forward evaluation and backward gradient computation functions we need to use matrix operations to complete the computation. In the CNTK all the math operations are implemented in a separate DLL CNTKMath.dll. The library supports CPU and GPU computation with sparse and dense matrix formats.

The math library contains a wrapper class called Matrix<ElemType>, where ElemType is either float or double. This Matrix<ElemType> class hides the differences between the multiple matrix implementations and takes care of data transfers between GPU and CPU. GPUs and CPUs have different memory spaces, and copying data between them is necessary to access or modify the data from either device. The library attempts to keep data on the GPU as much as possible if a GPU is being used.

When data are accessed or modified from the CPU, if the data is currently on the GPU the matrix will automatically be relocated to the CPU, and relocated back when the GPU attempts to access or modify the data. Currently the entire matrix object is transferred, so care should be taken when accessing matrix data from the CPU, e.g., when using the element value access operators. Each such memory transfer will cause significant slow down during training and testing.

If the operations needed by your node are already implemented in the Matrix<ElemType> class you can just use them. If, however, they are not implemented yet, you will need to implement the related functions in the math library, for both CPU and GPU. It is to be advised that you should not use operations that

returns a value instance of the matrix class except the ColumnSlice method. This is because those objects will be created and released after each minibatch, causing inefficiency, and will not update values passed through the ColumnSlice function.

4.2.2 Adding the New Node Type to the Computational Network

Once the new computation node type is implemented, we need to make them available in the computational network. This is accomplished by adding them in three functions in the ComputationNetwork.h file.

The first function is CreateComputationNode as shown below. This function allows a programmer to create a new node of a specific type.

```
ComputationNodePtr CreateComputationNode(const std::wstring
    nodeType, const std::wstring nodeName)
{
    ComputationNode<ElemType>* newNode;

    if (nodeType == NegateNode<ElemType>::TypeName())
        newNode = new NegateNode<ElemType>(m_deviceId,
            nodeName);
//other node types
    else if (nodeType == ScaleNode<ElemType>::TypeName())
        newNode = new ScaleNode<ElemType>(m_deviceId,
            nodeName);
    //other node types
}
```

The second function is CreateNodeFromFile as shown below. This function allows a programmer to create a new node of a specified type from a file stream.

```
ComputationNode<ElemType>* CreateNodeFromFile(const std::
    wstring nodeType, const std::wstring nodeName, File &
    fstream, size_t modelVersion)
{
    ComputationNode<ElemType>* newNode = nullptr;
    if (nodeType == LearnableParameter<ElemType>::TypeName
        ())
        newNode = new LearnableParameter<ElemType>(fstream,
            modelVersion, m_deviceId, nodeName);
    //other node types
    else if (nodeType == ScaleNode<ElemType>::TypeName())
        newNode = new ScaleNode<ElemType>(fstream,
            modelVersion, m_deviceId, nodeName);
    //other node types
}
```

The third function is type specific. For the ScaleNode class we defined the Scale operation as shown below.

```
ComputationNodePtr Scale (const ComputationNodePtr scalar,
    const ComputationNodePtr matrix, const std::wstring
    nodeName = L"")
{
    ComputationNodePtr newNode(new ScaleNode<ElemType>(
        m_deviceId, nodeName));
    newNode->AttachInputs(scalar, matrix);
    AddNodeToNet(newNode);
    return newNode;
}
```

This allows programmers to create nodes directly through operations. For example, we can use

```
sNode = Scale(s, m, L"scale1")
```

to create a ScaleNode named scale1 in the network and referenced as sNode in the program. This node scales the matrix m with a scalar s.

4.2.3 Adding the New Node Type to the Network Definition Language

End users create computation nodes through network builders. For this reason, we also need to add the node type to the network description language (NDL). This is done by modifying the CheckFunction function to make Scale a valid function name.

```
bool CheckFunction(std::string& p_nodeType, bool*
    allowUndeterminedVariable)
{
    std::wstring nodeType = msra::strfun::utf16(p_nodeType)
        ;
    bool ret = false;

    if (allowUndeterminedVariable)
        *allowUndeterminedVariable = true;
    //other node types
    else if (EqualInsensitive(nodeType, ScaleNode<ElemType
        >::TypeName()))
        ret = true;
    //other node types and codes
}
```

Sometimes you may also need to modify the

```
virtual void Evaluate(NDLNode<ElemType>* node, const
    wstring& baseName, const NDLPass pass)
```

function in the SynchronousExecutionEngine.h file to handle special parameters.

4.2.3.1 NDL Processing Phases

The ability to describe a network architecture in NDL (Network Description Language) is one of the major features of CNTK. However, it is not immediately obvious to the developer looking at the code how it all works. Here we briefly describe the inner workings of NDL processing in CNTK.

NDL is based on the same configuration parser that is used for config files and MEL (Model Editing Language). While this is convenient to share code, it also makes things a little less clear when viewing the code. The configuration file classes, MEL class, and NDL classes all inherit from ConfigParser, which provides the basic parsing, bracket matching, and other common features (quote handling, etc.). The parsing engine implemented in ConfigParser calls back to a virtual method called ParseValue() when it has a token that needs to be interpreted. So ParseValue() in the NDLScript class is the main location where interpretation of tokens takes place.

NDL supports Macros, which makes things much more convenient, but a bit messier for the developer to deal with. All the macros are parsed and stored in a global script so they can be accessed by any NDL script. It also means that you don't want to load or define a set of macros more than once, or you will get "function already exists" errors.

The processing of NDL proceeds through the following phases:

1. Parsing the script
2. Evaluation (initial pass) – create ComputationNodes for all NDLNodes that require it
3. Evaluation (second pass) – connect the inputs of the ComputationNodes
4. Validate the network – This also allocates all the matrix classes to their correct dimensions, and computes dimensions derived from input nodes.
5. Evaluation (final pass) – All operations that must have the matrix values present occur here. For example, matrix initialization happens here.

There is a helper class in the NDLUtil class, which will take care of executing through all phases. It also tracks how far along in the current processing phase a

script has progressed. Processing can continue statement by statement as needed. This is how in-line NDL is processed.

4.2.3.2 Parsing

The script in question is first parsed, and as each macro definition, macro call, parameter, variable, or function call is encountered an NDLNode is created. This NDLNode describes the entity and a reference is stored in the NDLScript class which owns it so it can be freed at some later point in time. If the NDLNode is an executable statement, it will be added in order to the list of statements to execute. All variable names used in a script will be added to a symbol table in the NDLScript class that owns the NDLNode.

If the NDLNode is a macro or function call its parameters will be parsed and added to a parameter list in the NDLNode. Note that parameters may actually be other function and macro calls. The actual parameter names used in the call and the names used in the macro that will be called are recorded.

If the NDLNode is a macro, it will have its own NDLScript, and contain its own list of executable statements. It will also be stored in the global script repository, which is just a global NDLScript class.

4.2.3.3 Evaluation (Initial Pass)

Each pass evaluates the entire script, but only certain actions are performed based on what pass is being executed. The main purpose of this pass is to create a computation node for every NDL node that requires one. Effectively every "Function call" in NDL maps to a computation node. The full "dot path" will be the name of the node in the computational network. Although all the parameters are evaluated in NDL, only function calls will create computation nodes.

4.2.3.4 Evaluation (Second Pass)

This pass goes through the entire evaluation process again, but this time all computation nodes should already exist. The main purpose of this pass is to hook up all the inputs between nodes. At the end of this pass the computational network is fully connected and complete.

Doing this in a separate pass allows nodes to be referenced before they are actually defined in the NDL Script. This is a necessary feature for recursive neural networks with a DelayNode.

4.2.3.5 Validation

Validation ensures that the network is fully connected and that all necessary network nodes exist. It also ensures that the dimensions of the matrices passed to nodes are compatible with the nodes and enough memory is allocated for the matrices. In addition, the existence of special nodes such as CriteriaNode, Features, Labels, and Output are checked.

4.2.3.6 Evaluation (Final Pass)

This pass does special processing, such as matrix initialization, that requires the matrices to exist. As an example there is an optional parameter for the Parameter() function that allows a parameter to be initialized in various ways (zero, random values, from a file). Since matrix initialization requires the matrix to be there, it is done in the final pass.

4.3 Adding a New Training Algorithm

To add a new training algorithm to the CNTK you can use the stochastic gradient algorithm (SGD) as a reference. Note that the computational network only provides the first-order gradient information, so only algorithms that depend on the first-order gradient (e.g., quasi-second-order algorithms such as L-BFGS) can be easily implemented in the CNTK. In the following we list the main steps in a typical training algorithm.

4.3.1 Prepare the Mapping from Node Names to Matrices for Features and Labels

Since features and labels are loaded from data files and in the IDataReader interface we need to pass in the map from node names to matrices to read features and labels, in a typical trainer the first step is to create this map as shown in the following sample, where net is a Computational Network object.

```
std::vector<ComputationNodePtr> & FeatureNodes = net.
    FeatureNodes();
std::vector<ComputationNodePtr> & labelNodes = net.
    LabelNodes();

std::map<std::wstring, Matrix<ElemType>*> inputMatrices;
for (size_t i=0; i<FeatureNodes.size(); i++)
{
```

```
    inputMatrices[FeatureNodes[i]->NodeName()] = &
        FeatureNodes[i]->FunctionValues();
}

for (size_t i=0; i<labelNodes.size(); i++)
{
    inputMatrices[labelNodes[i]->NodeName()] = &labelNodes[
        i]->FunctionValues();
}
```

4.3.2 Evaluate Precomputed Nodes

Before training happens, we need to evaluate the precomputed nodes. The precomputed nodes are typically used to compute the mean and standard deviations of input features so that we may normalize the features or get the frequency information of labels, which can be used as the prior probability of the output nodes. It is advised that you save a model after each major model update so that you may re-start from a checkpoint file in case you cannot finish the whole training at once.

```
if (PreCompute(net, trainSetDataReader, FeatureNodes,
    labelNodes, inputMatrices) || startEpoch == 0)
{
    net.SaveToFile(GetModelNameForEpoch(int(startEpoch)-1))
        ;
}
```

4.3.3 Main Loop

Most of the training happens inside the loop through epochs. In the following example we also show that you can specify some special properties for nodes such as convolutional node and dropout node. Depends on whether these properties should be set to different values for different epoch they can be handled either inside or outside of the epoch loop.

```
std::vector<ComputationNodePtr> & criterionNodes =
    GetTrainCriterionNodes(net);
std::vector<ComputationNodePtr> & evaluationNodes =
    GetEvalCriterionNodes(net);

SetMaxTempMemSizeForCNN(net, criterionNodes[0],
    m_maxTempMemSizeInSamplesForCNN);

for (int i=int(startEpoch); i<int(m_maxEpochs); i++)
```

```
{
    SetDropoutRate(net, criterionNodes[0], m_dropoutRates[i
        ], prevDropoutRate, dropOutSeed);
    DecideLearningRate();
        TrainOneEpoch();
}
```

4.3.4 Train One Epoch

The TrainOneEpoch function contains your core training algorithm. It typically looks like following:

```
//start the minibatch loop in the data reader.
trainSetDataReader->StartMinibatchLoop(mbSize, epochNumber,
    epochSize);

while (trainSetDataReader->GetMinibatch(inputMatrices))
{
    //update the time stamp of the input nodes for re-
        evaluation
        UpdateEvalTimeStamps(FeatureNodes);
        UpdateEvalTimeStamps(labelNodes);

        //set actual minibatch size, needed since delay
            nodes cannot infer it automatically
    size_t actualMBSize = net.GetActualMBSize();
        net.SetActualMiniBatchSize(actualMBSize);

        //set the number of samples for each gradient
            computation in the truncated BPTT.
        net.SetActualNbrSlicesInEachRecIter(
            trainSetDataReader->
            NumberSlicesInEachRecurrentIter());

        //set the sentence end information for multi-
            sequence RNN training
        trainSetDataReader->SetSentenceEndInBatch(net.
            m_sentenceEnd);

        //compute the gradients, which will be stored in
            the GradientValues() of each node
        net.ComputeGradient(criterionNodes[0]);
```

```
        //you can do statistics computation here for
            evaluation error and training error

        //Update the model based on the gradient computed
        UpdateModel();
}
```

The StartMinibatchLoop call will inform the reader how to prepare for the minibatches. The code then get minibatches until all minibatches are fetched for the epoch. Once a minibatch is fetched from the reader, you need to call the UpdateEvalTimeStamps function to inform the computational network that the values of these nodes have been changed so that all nodes that depend on these nodes will be recomputed when the forward computation is carried out. You need to do the same thing for all the model parameters when they are updated by your training algorithm.

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 $H \circ C \circ \mathscr{L}$ 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 $H \circ C \circ \mathscr{L}$ 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 $H^{\circ}C^{\circ}\mathscr{L}$. 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 $H^{\circ}C^{\circ}\mathscr{L}$ 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 $H^{\circ}C^{\circ}\mathscr{L}$ 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 $H^{\circ}C^{\circ}\mathscr{L}$.

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:)]: gzipp'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-indepenent 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 probabilites 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, -infiles, 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 differes from the best hypothesis by more than this amount will be 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 languag emodel.

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-indepenent 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 addtion, if needPrior is set to true, the prior probablities of the labels will be computed and a ScaledLogLikelihood node in the network will be automatically created. This is important if this netwok 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 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=$MlfDir$\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=$NdlDir$\default_macros.ndl
  networkDescription=$NdlDir$\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 peform 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=criteria)
    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 scpFiles 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 aproaches 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
  NDLNetworkBuilder=[
    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
  NDLNetworkBuilder=[
    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"

```
ml=LoadModel($CurrModel$, format=cntk)
SetDefaultModel(ml)
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(ml,$NewModel$, 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 optmize 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=criteria)

# 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 ErrorPredition 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 contruct 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 adn output data have to the same in both the NDL model description and the reader configuration.

6.1.8 Evaluating networks and cross validation

One 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 support 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=trainRNN. It indicates the section is for training.
- minibatchSize=10. Mini batch size is 10. That is, 10 words are processed at the same time during forward (computing evaluation value) and backward (computing gradients) processes.
- deviceId=-1. -1 is CPU device. One can change to GPU devices in non-negative intergers (0, 1, etc.).
- 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.
- rnnType=CLASSLM. The RNN network structure. It consists of an input layer, a recurrent hidden layer and an output layer (including classes and vocabularies).
- SimpleNetworkBuilder section
  - trainingCriterion=classcrossentropywithsoftmax. Training criterion used in model training.

  - nodeType=Sigmoid. Non-linearity function used in hidden layer.
  - layerSizes=10000:200:10050. Sizes of input, hidden and ouput 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 sum of vocabulary size and class size.
  - uniformInit=true. Whether to use uniformly randomizied values for initial paramter weights.

- 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 truncate gradient values if their abosolute values are greater than clippingThresholdPerSample.
  - clippingThresholdPerSample=15.0. Used when gradientClippingWithTruncation=true.
  - 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. One can also choose adagrad or rmsprop.
  - modelPath=C:\CNTKExp\RNN\log\modelRnnCNTK. The RNN model file location.
  - loadBestModel=true. Wether 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 first reduced.

  - learnRateDecreaseFactor=0.5. Learning rate decrese 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 interger, 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 [31]or more sophisticated way[32].
  - 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=1. Mini batch size is 1.
- deviceId=-1.-1 is CPU device. One can change to GPU devices in non-negative intergers (0, 1, etc.).
- 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.

6.2.3 Results

With the above configuration, we achieved the training speech of 2600 words/sec on CPU (Intel Xeon 2 processors with 2.60GHz). This setup resulted in perplexity of 130 on Penn Treebank test data after 12 epochs. We have GPU version running slightly slower (2400 words/sec) for this release with slightly different results (due to different randomization and slightly different implementation of prediction normalization). Currently the slowness of GPU is due to the data reader part which takes around 40% of total training time. Future release aims for training acceleration with a more efficient data reader.

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=trainRNN : this informs CNTK to call RNN function.
- 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=10000:200:10050 : this specifies input, hidden and output layer sizes.
- 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=SequenceReader : specifies using 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=SequenceReader : specifies using 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=SequenceReader : 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

6.3.3 Results

LSTM converges very fast with this setup. Perplexities are 161.96 and 137.37 at the 1st and the 10-th iteration. Perplexies are not reduced after 10-th itreation. For comparison, simple RNN by Tomas Mikolov obtained 136 perplexity with 50 classes and LSTM by Alex Graves obtained 138 perplexity.

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.config. 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=trainRNN : this informs CNTK to call RNN function.
- 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=crossentropywithsoftmax specifies that training set to use cross entropy
  - evalCriterion=crossentropywithsoftmax 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 coresponds 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 begining.
    * 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
    * beginingSequence="O" : this specifies the symbol of output sequence begining.
    * endSequence="O" : this specifies the symbol of output sequence end.
    * 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-spefied in CNTK's simple network builder. The node "outputs" is the node that output activies 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 writting decode results from LUSequenceWriter.
    - token=c:\exp\atis\data\output.txt : this specifies the semantic labels.

6.4.3 Results

With this setup, F1 scores are 94.13%. It is possible to get improved F1 score with larger hidden layer size, and with more than one layer of LSTM. Using more than one layer with LSTMs with different dimensions can be easily done by setting recurrentLayer=2:3 and also setting layerSizes=2832:50:300:200:127 for instance. In this case, two layers of LSTMs are used. The lower layer LSTM has 300 dimension and the upper LSTM has 200 dimension.

Bibliography

[1] 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.

[2] 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.

[3] 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 Automfatic Speech Recognition and Understanding (ASRU)*, 2011, pp. 24–29.

[4] 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.

[5] N. Jaitly, P. Nguyen, A. W. Senior, and V. Vanhoucke, "Application of pretrained deep neural networks to large vocabulary speech recognition," in *Proc. Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2012.

[6] 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.

[7] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, 1995.

[8] 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.

[9] 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.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks." in *NIPS*, vol. 1, no. 2, 2012, p. 4.

[11] 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.

[12] 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.

[13] 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 Automfatic Speech Recognition and Understanding (ASRU)*, 2013, pp. 315–320.

[14] 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.

[15] O. Abdel-Hamid, L. Deng, and D. Yu, "Exploring convolutional neural network structures and optimization techniques for speech recognition," pp. 3366–3370, 2013.

[16] 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.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] 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.

[19] 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.

[20] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model." in *Proc. IEEE Spoken Language Technology Workshop (SLT)*, 2012, pp. 234–239.

[21] 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 (INTERSPEECH)*, 2012.

[22] 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.

[23] B. Guenter, D. Yu, A. Eversole, O. Kuchaiev, and M. L. Seltzer, "Stochastic gradient descent algorithm in the computational network toolkit."

[24] 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.

[25] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Siam, 2008.

[26] C. Bischof, L. Roh, and A. Mauer-Oats, "ADIC: an extensible automatic differentiation tool for ANSI-C," *Urbana*, vol. 51, p. 61802, 1997.

[27] B. Guenter, "Efficient symbolic differentiation for graphics applications," in *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, 2007, p. 108.

[28] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[29] J. E. Hopcroft, *Data structures and algorithms*. Pearson education, 1983.

[30] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[31] 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.

[32] G. Zweig and K. Makarychev, "Speed regularization and optimality in word classing," in *Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.