Chapter 1
A Computational Network [1]

Abstract In previous chapters we have discussed various deep learning models for automatic speech recognition (ASR). In this chapter we introduce the computational network (CN), a unified framework for describing a wide range of arbitrary learning machines, such as deep neural networks (DNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs) including its long short term memory (LSTM) version, logistic regression, and maximum entropy models. All these learning machines can be formulated and 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 acting upon its children. We describe algorithms to carry out forward computation and gradient calculation in the CN and introduce most popular computation node types used in a typical CN.

1.1 Computational Network

There is a common property in the key models, such as deep neural networks (DNNs) [7, 24, 23, 31, 16, 13], convolutional neural networks (CNNs) [19, 5, 17, 18, 2, 6, **?**, 22, 1, 8, 21], and recurrent neural networks (RNNs) [14, 26, 27, 20, 25]. 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 [3], CNTK [12] and RASR/NN [29]..

[1] This chapter has been published as part of the CNTK document [30]

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:     $\mathbf{O} \leftarrow \mathrm{softmax}\left(\mathbf{P}^{(2)}\right)$        ▷ 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 computation node, represents a computation operation. 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 same one-hidden-layer sigmoid neural network 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, computation node is the basic element of operations. This makes the description of the simple model such as DNN more cumbersome which can be alleviated by grouping computation nodes together with macros. In return, CN provides us 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 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.

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

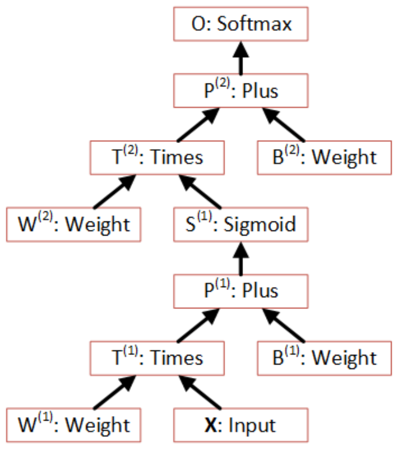

Fig. 1.1 Represent the one-hidden-layer sigmoid neural network with a computational network

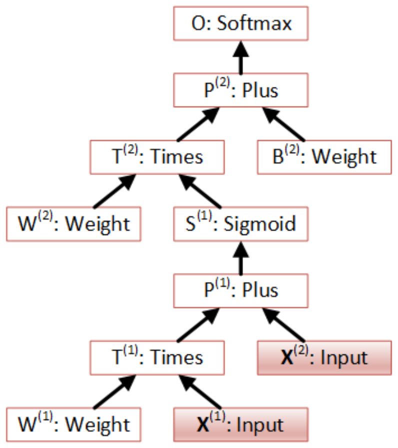

Fig. 1.2 A computational network with two input nodes (highlighted).

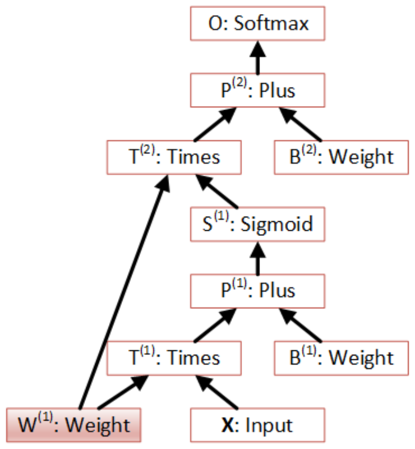

Fig. 1.3 A computational network with shared model parameters (highlighted). Here we assume the input $\mathbf{X}$, hidden layer $\mathbf{S}^{(1)}$, and output layer $\mathbf{O}$ have the same dimension.

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 keep the same for all the subsequent runs regardless of the computational environment. In other words, this algorithm only needs to be executed once for the node that we need to compute values by caching 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 $root \notin visited$ then          ▷ the same node may be a child of several nodes.
3:         $visited \leftarrow visited \cup root$
4:         for each $c \in root.children$ do
5:             call DecideForwardComputationOrder(c, visited, order)
6:             $order \leftarrow order + root$                ▷ 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 describes an example algorithm that carries out the forward computation of a CN asynchronously. In this algorithm, all the nodes whose children have not been computed stay in the waiting set and those whose children are computed stay 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 all its parent nodes that are still waiting. If all the children of a node have been computed, the node is moved from the waiting set to the ready set. 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 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 one node and later for another node given the same 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 as have been discussed in Chapter **??**. 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.

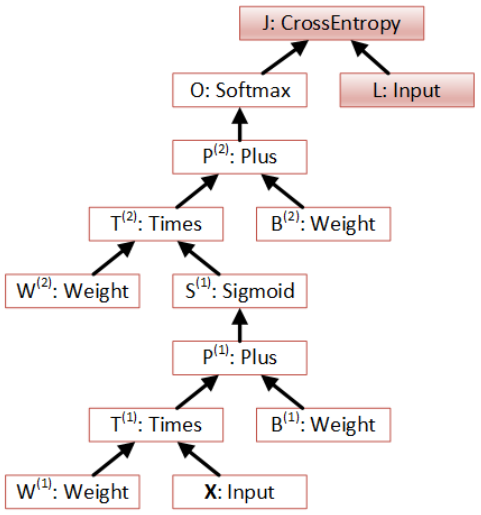

Fig. 1.4 The one-hidden-layer sigmoid neural network augmented with a cross entropy training criterion node J and a label node L.

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 similar to that described in 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

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

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

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 $\nabla^J_{\mathbf{W}^{(2)}}$.

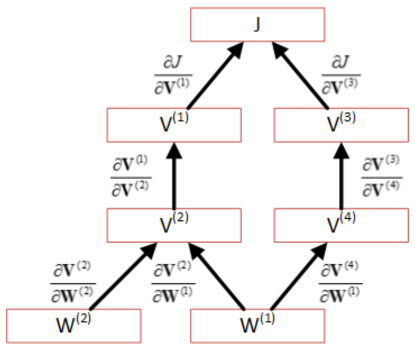
J
$\frac{\partial J}{\partial \mathbf{V}^{(1)}}$
$\frac{\partial J}{\partial \mathbf{V}^{(3)}}$
$V^{(1)}$
$V^{(3)}$
$\frac{\partial \mathbf{V}^{(1)}}{\partial \mathbf{V}^{(2)}}$
$\frac{\partial \mathbf{V}^{(3)}}{\partial \mathbf{V}^{(4)}}$
$V^{(2)}$
$V^{(4)}$
$\frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(2)}}$
$\frac{\partial \mathbf{V}^{(2)}}{\partial \mathbf{W}^{(1)}}$
$\frac{\partial \mathbf{V}^{(4)}}{\partial \mathbf{W}^{(1)}}$
$W^{(2)}$
$W^{(1)}$

Fig. 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.

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 are the parents of the nodes, rather than the children.

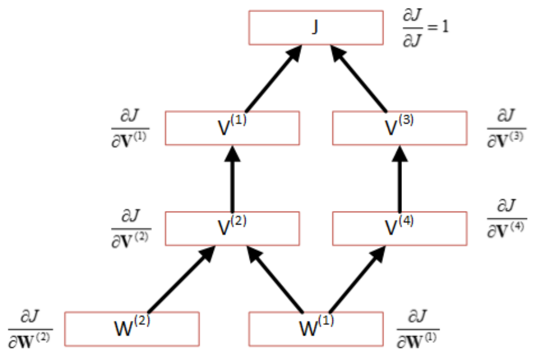
J
$\frac{\partial J}{\partial J}=1$
$V^{(1)}$
$V^{(3)}$
$\frac{\partial J}{\partial \mathbf{V}^{(1)}}$
$\frac{\partial J}{\partial \mathbf{V}^{(3)}}$
$V^{(2)}$
$V^{(4)}$
$\frac{\partial J}{\partial \mathbf{V}^{(2)}}$
$\frac{\partial J}{\partial \mathbf{V}^{(4)}}$
$W^{(2)}$
$W^{(1)}$
$\frac{\partial J}{\partial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

Fig. 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 $\boldsymbol{v}_n$ and the gradient ∇_n^J .

Automatic differentiation has been an active research area for decades and many techniques have been proposed (see [10] for a good summary). A simple recursive algorithm for deciding the gradient computation order is shown in Algorithm 1.4 to which a similar recursive algorithm for scalar functions has been provided in [4] and [11]. 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 descend (SGD) algorithm discussed in Chapter **??** 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.

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 1.4 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 $parentsLeft[node] == 0 \wedge node \notin order$ then    ▷ All parents have been computed.
5:             $order \leftarrow order + node$    ▷ Add $node$ to the end of $order$
6:             for each $c \in node.children$ do
7:                 call DecideGradientComputationOrder($c$, $parentsLeft$, $order$)
8:             end for
9:         end if
10:    end if
11: end procedure
```

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 $root \notin visited$ then    ▷ The same node may be a child of several nodes and revisited.
3:         $visited \leftarrow visited \cup root$
4:         for each $c \in root.children$ do
5:             call UpdateNeedGradientFlag($c$, $visited$, $order$)
6:             if $IsNotLeaf(node)$ then
7:                 if $node.AnyChildNeedGradient()$ then
8:                     $node.needGradient \leftarrow true$
9:                 else
10:                    $node.needGradient \leftarrow false$
11:                end if
12:            end if
13:        end for
14:    end if
15: end procedure
```

Since every instantiation of 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:

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

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

1.4 Typical Computation Nodes

For the above forward computation and gradient calculation algorithms to work we have assumed 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 $\mathbf{V}_n$ (for simplicity we will remove the subscript in the following discussion) and the gradient ∇_n^J of the node n and values of all its child nodes.

In this section we introduce the most widely used computation node types and the related *Evaluate* and *ComputePartialGradient*(*child*) functions. In the following discussion we assume λ is a scalar, $\mathbf{X}$ and $\mathbf{Y}$ are matrices of the first and second operand, respectively, $\mathbf{d}$ is a column vector that represents the diagonal of a square matrix, $\boldsymbol{V}$ is the value of current node, ∇_n^J (or $\nabla_{\mathbf{V}}^J$), $\nabla_{\mathbf{X}}^J$, and $\nabla_{\mathbf{Y}}^J$ are the gradient of the current node and child nodes (operands) $\mathbf{X}$ and $\mathbf{Y}$, respectively, $\bullet$ is the element-wise product, $\oslash$ is the element-wise division, $\circ$ is the inner product of vectors applied on matrices column-wise, $\circledcirc$ is inner product applied to each row, $\delta(.)$ is the Kronecker delta, $\mathbf{1}_{m,n}$ is an $m \times n$ matrix with all 1's, $\boldsymbol{X}^{\alpha}$ is element-wise power, and $\operatorname{vec}(\mathbf{X})$ is the vector formed by concatenating columns of $\mathbf{X}$. We treat each minibatch of input 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, $Evaluate = \boldsymbol{V}(\mathbf{X})$ and $ComputePartialGradient(\mathbf{X}) = \nabla_{\mathbf{X}}^{J}$

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

$$\boldsymbol{V}(\mathbf{X}) \leftarrow -\mathbf{X} \tag{1.7}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} - \nabla_{\mathbf{n}}^{J}. \tag{1.8}$$

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 $\mathbf{X}$.

$$\boldsymbol{V}(\mathbf{X}) \leftarrow \frac{1}{1 + e^{-\mathbf{X}}} \tag{1.11}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet [\boldsymbol{V} \bullet (1 - \boldsymbol{V})]. \tag{1.12}$$

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 $\mathbf{X}$.

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

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

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 **X**.

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

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

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 **X**.

$$\boldsymbol{V}(\mathbf{X}) \leftarrow \log(\mathbf{X}) \tag{1.23}$$

$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \bullet \frac{1}{\mathbf{X}}. \tag{1.24}$$

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 **X**.

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

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 **X**. Each column is treated as a separate sample.

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

The gradient can be derived by observing

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

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 **X**.

$$\boldsymbol{\upsilon}(\mathbf{X}) \leftarrow \sum_{i,j} x_{ij} \tag{1.38}$$
$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J}. \tag{1.39}$$

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

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

and

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

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

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

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

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

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

and

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

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

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

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

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

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

and

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

1.4.3 Computation Node Types with Two Operands

In these computation node types, *Evaluate* $= \boldsymbol{V}\left(a, \mathbf{Y}\right)$, 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.

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

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

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

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 $\mathbf{X}.cols = \mathbf{Y}.rows$.

$$\boldsymbol{V}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X}\mathbf{Y} \tag{1.57}$$

$$\nabla_{\mathbf{X}}^{J} \leftarrow \nabla_{\mathbf{X}}^{J} + \nabla_{\mathbf{n}}^{J} \mathbf{Y}^{T} \tag{1.58}$$

$$\nabla_{\mathbf{Y}}^{J} \leftarrow \nabla_{\mathbf{Y}}^{J} + \mathbf{X}^{T} \nabla_{\mathbf{n}}^{J}. \tag{1.59}$$

The gradient $\nabla_{\mathbf{X}}^{J}$ 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 $\nabla_{\mathbf{Y}}^{J}$, 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 $\mathbf{X}.cols = \mathbf{Y}.cols$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\nabla_{\mathbf{d}}^{J} \leftarrow \nabla_{\mathbf{d}}^{J} + \nabla_{\mathbf{n}}^{J} \circledcirc \mathbf{Y} \tag{1.80}$$

$$\nabla_{\mathbf{Y}}^{J} \leftarrow \nabla_{\mathbf{Y}}^{J} + \mathrm{DiagTimes}\left(\mathbf{d}, \nabla_{\mathbf{n}}^{J}\right). \tag{1.81}$$

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

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

and

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

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

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

and get

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

- *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:

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

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

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

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.

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

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

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

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

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

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 $\nabla_{\mathbf{y}}^{J}$ can be derived similarly.

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

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

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

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

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

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

and

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

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

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

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

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

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 $\mathbf{X}.cols = \mathbf{Y}.cols$.

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

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 $\mathbf{X}.cols = \mathbf{Y}.cols$.

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

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

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

and

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

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

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

and get

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

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

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

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

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

and get

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

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 as explained in Chapter **??**. 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) 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 as to avoid recomputation. When a minibatch of input $\mathbf{X}$ is fed in

$$k \leftarrow k + \mathbf{X}.cols \tag{1.123}$$
$$\boldsymbol{\upsilon}(\mathbf{X}) \leftarrow \frac{1}{k}\mathbf{X}\mathbf{1}_{\mathbf{X}.cols,1} + \frac{k - \mathbf{X}.cols}{k}\boldsymbol{\upsilon}(\mathbf{X}) \tag{1.124}$$

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

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

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

When the end of the training set is reached,

$$\boldsymbol{\upsilon} \leftarrow (\omega - (\upsilon \bullet \upsilon))^{1/2} \tag{1.128}$$
$$\boldsymbol{\upsilon} \leftarrow 1 \oslash \boldsymbol{\upsilon}. \tag{1.129}$$

- *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.

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

1.5 Convolutional Neural Network

Convolutional neural network (CNN) [19, 5, 17, 18, 2, 6, **?**, 22, 1, 8, 21] provides shift invariance 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 [2, 1, 8, 21, 22]. 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 convolution operations 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 the input with strides (or subsampling rate) S_r and S_c at the vertical (row) and horizontal (column) direction, respectively. For each output channel ℓ and input slice (i,j) (the i-th step along the vertical direction and j-th step along the horizontal direction)

$$v_{\ell ij}(\mathbf{K},\mathbf{Y}) = \sum_k \mathrm{vec}(\mathbf{K}_{k\ell}) \circ \mathrm{vec}(\mathbf{Y}_{kij}), \tag{1.131}$$

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. [5] 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. [5] 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

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

is the number of rows in the output image, and

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

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

$$\mathbf{X}(\mathbf{Y}) \leftarrow \text{Pack}(\mathbf{Y}) \tag{1.134}$$

$$\boldsymbol{V}(\mathbf{W},\mathbf{Y}) \leftarrow \mathbf{W}\mathbf{X} \tag{1.135}$$

$$\nabla_{\mathbf{W}}^{J} \leftarrow \nabla_{W}^{J} + \nabla_{\mathbf{n}}^{J}\mathbf{X}^{T} \tag{1.136}$$

$$\nabla_{\mathbf{X}}^{J} \leftarrow \mathbf{W}^{T}\nabla_{\mathbf{n}}^{J} \tag{1.137}$$

$$\nabla_{\mathbf{Y}}^{J} \leftarrow \nabla_{\mathbf{Y}}^{J} + \text{Unpack}\left(\nabla_{\mathbf{X}}^{J}\right). \tag{1.138}$$

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.

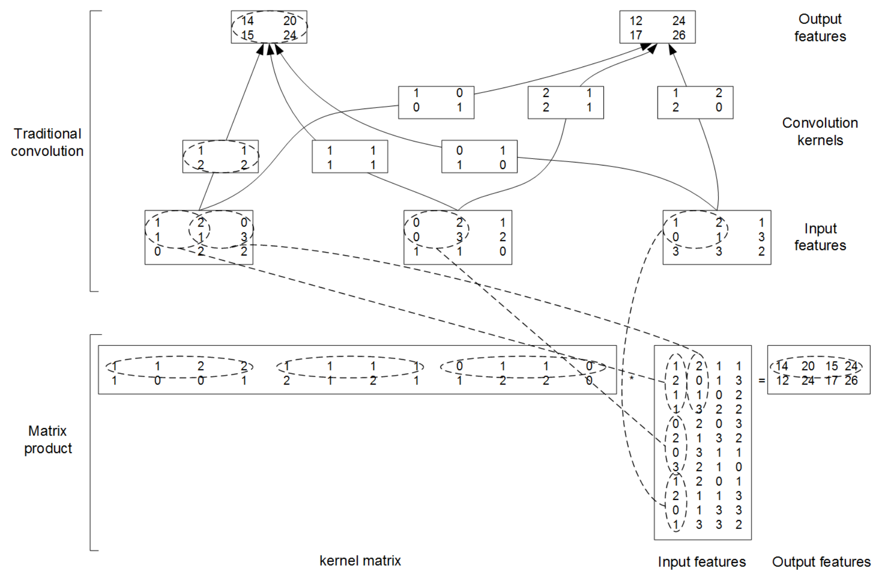

Fig. 1.7 Example convolution operations. Top: the original convolution operations. Bottom: the same operations represented as a large matrix product. Our matrix organization is different from what proposed by Chellapilla et al. [5] to allow for stacked convolution operations. (Modified from the figure in Chellapilla et al. [5], permitted to use by Simard)

- *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

$$\upsilon_{\ell ij}\left(\mathbf{X}\right) \leftarrow \max\left(\mathbf{X}_{\ell ij}\right) \tag{1.139}$$

$$\left[\nabla_{\mathbf{X}}^{J}\right]_{\ell,i_m,j_n} \leftarrow \begin{cases} \left[\nabla_{\mathbf{X}}^{J}\right]_{\ell,i_m,j_n} + \left[\nabla_{\mathbf{n}}^{J}\right]_{\ell,i_m,j_n} & (m,n) = arg\max_{m,n} x_{\ell,i_m,j_n} \\ \left[\nabla_{\mathbf{X}}^{J}\right]_{\ell,i_m,j_n} & else \end{cases} \tag{1.140}$$

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

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

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

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

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.

$$\boldsymbol{\upsilon}_{.j}\left(\lambda, \mathbf{Y}\right) \leftarrow \mathbf{Y}_{.(j-\lambda)} \tag{1.143}$$

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

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 in a recurrent network is to unroll all samples in the sequence over time. Once unrolled, the graph is expanded into a DAG and the forward computation and gradient calculation algorithms we just discussed can be directly used. This, however, means all computation nodes in the CN need to be computed sample by sample and would significantly reduce the potential of parallelization.

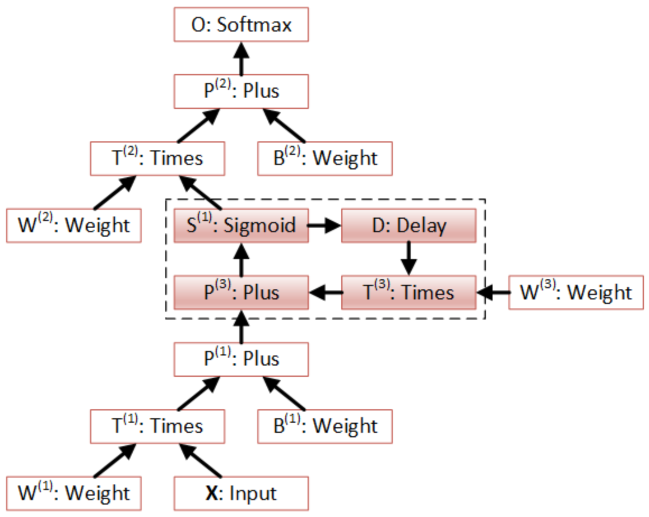

Fig. 1.8 An example CN with a delay node. The shaded nodes form a recurrent loop which can be treated as a composite node.

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 apply 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 which there is a path between each pair of vertices and adding any edges or vertices would violate this property, in the graph using algorithms such as Tarjan's strongly connected components algorithm [28][2]. Once the loops are identified, they can be treated as a composite node in the CN and the CN is reduced to a DAG. All the nodes inside each loop (or composite node) can be unrolled over time and also reduced to a DAG. For all these DAGs the forward computation and backpropagation algorithms we discussed in the previous sections can be applied. The detailed procedure in determining the forward computation order in the CN with arbitrary recurrent connections is described in Algorithm 1.6. Since the input to the delay nodes are computed in the past time 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 in batch. 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 multiple frames from different sequences in batch when inside a loop and compute all samples in all utterances in one batch when 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.

With the inclusion of delay nodes, we can easily construct complicated recurrent networks and dynamic systems. For example, the long-short-term-memory (LSTM) [14, 9] neural network that is widely used to recognize and generate hand-written characters involves the following operations:

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

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

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

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

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

[2] Tarjan's algorithm is favored over others such as Kosaraju's algorithm [15] since it only requires one depth-first traverse, has a complexity of $O\left(|\mathbb{V}| + |\mathbb{E}|\right)$, and does not require reversing arcs in the graph.

Algorithm 1.6 Forward computation of an arbitrary CN

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

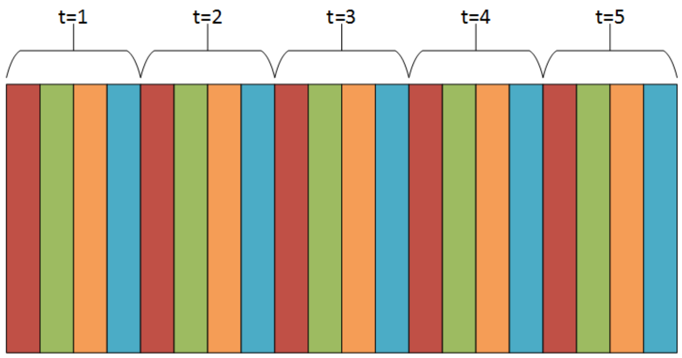

Fig. 1.9 Process multiple sequences in a batch. Shown in the figure is an example with 4 sequences. Each color represents one sequence. Frames with the same frame id from different sequences are grouped together and computed in batch.

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:

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

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

$$\begin{aligned}
\mathbf{S}^{(1)} &= Macro2W1b\left(\mathbf{W}^{(1)}, \mathbf{I}^{(1)}, \mathbf{W}^{(2)}, \mathbf{I}^{(2)}, \mathbf{b}\right) && (1.167)\\
Macro3W1b &= Plus\left(\mathbf{S}^{(1)}, Times\left(\mathbf{W}^{(3)}, \mathbf{I}^{(3)}\right)\right) && (1.168)
\end{aligned}$$

Using CN we can easily build arbitrary network architectures to solve different problems. For example, it will be easy to add a masking layer to reduce noises in the input features in an otherwise conventional DNN. CN provides the needed tool to explore new speech recognition architectures and we believe additional advancement in the field will attribute at least partially to CN.

References

1. Abdel-Hamid, O., Deng, L., Yu, D.: Exploring convolutional neural network structures and optimization techniques for speech recognition pp. 3366–3370 (2013)
2. Abdel-Hamid, O., Mohamed, A.r., Jiang, H., Penn, G.: Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In: Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 4277–4280. IEEE (2012)
3. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for scientific computing conference (SciPy), vol. 4 (2010)
4. Bischof, C., Roh, L., Mauer-Oats, A.: ADIC: an extensible automatic differentiation tool for ANSI-C. Urbana **51**, 61,802 (1997)
5. Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: Tenth International Workshop on Frontiers in Handwriting Recognition (2006)
6. Ciresan, D.C., Meier, U., Schmidhuber, J.: Transfer learning for Latin and Chinese characters with deep neural networks. In: Proc. International Conference on Neural Networks (IJCNN), pp. 1–6 (2012)
7. Dahl, G.E., Yu, D., Deng, L., Acero, A.: Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. IEEE Transactions on Audio, Speech and Language Processing **20**(1), 30–42 (2012)
8. Deng, L., Abdel-Hamid, O., Yu, D.: 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), pp. 6669–6673 (2013)
9. Graves, A.: Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850 (2013)
10. Griewank, A., Walther, A.: Evaluating derivatives: principles and techniques of algorithmic differentiation. Siam (2008)
11. Guenter, B.: Efficient symbolic differentiation for graphics applications. In: ACM Transactions on Graphics (TOG), vol. 26, p. 108 (2007)
12. Guenter, B., Yu, D., Eversole, A., Kuchaiev, O., Seltzer, M.L.: Stochastic gradient descent algorithm in the computational network toolkit
13. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Processing Magazine **29**(6), 82–97 (2012)

14. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
15. Hopcroft, J.E.: Data structures and algorithms. Pearson education (1983)
16. Jaitly, N., Nguyen, P., Senior, A.W., Vanhoucke, V.: Application of pretrained deep neural networks to large vocabulary speech recognition. In: Proc. Annual Conference of International Speech Communication Association (INTERSPEECH) (2012)
17. Kavukcuoglu, K., Sermanet, P., Boureau, Y.L., Gregor, K., Mathieu, M., LeCun, Y.: Learning convolutional feature hierarchies for visual recognition. In: NIPS, vol. 1, p. 5 (2010)
18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS, vol. 1, p. 4 (2012)
19. LeCun, Y., Bengio, Y.: Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks **3361** (1995)
20. Mikolov, T., Zweig, G.: Context dependent recurrent neural network language model. In: Proc. IEEE Spoken Language Technology Workshop (SLT), pp. 234–239 (2012)
21. Sainath, T.N., Kingsbury, B., Mohamed, A.r., Dahl, G.E., Saon, G., Soltau, H., Beran, T., Aravkin, A.Y., Ramabhadran, B.: Improvements to deep convolutional neural networks for lvcsr. In: Proc. IEEE Workshop on Automfatic Speech Recognition and Understanding (ASRU), pp. 315–320 (2013)
22. Sainath, T.N., Mohamed, A.r., Kingsbury, B., Ramabhadran, B.: Deep convolutional neural networks for LVCSR. In: Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 8614–8618 (2013)
23. Seide, F., Li, G., Chen, X., Yu, D.: Feature engineering in context-dependent deep neural networks for conversational speech transcription. In: Proc. IEEE Workshop on Automfatic Speech Recognition and Understanding (ASRU), pp. 24–29 (2011)
24. Seide, F., Li, G., Yu, D.: Conversational speech transcription using context-dependent deep neural networks. In: Proc. Annual Conference of International Speech Communication Association (INTERSPEECH), pp. 437–440 (2011)
25. Shi, Y., Wiggers, P., Jonker, C.M.: Towards recurrent neural networks language models with linguistic and contextual features. In: Proc. Annual Conference of International Speech Communication Association (INTERSPEECH) (2012)
26. Socher, R., Lin, C.C., Ng, A., Manning, C.: Parsing natural scenes and natural language with recursive neural networks. In: Proc. International Conference on Machine Learning (ICML), pp. 129–136 (2011)
27. Sutskever, I., Martens, J., Hinton, G.E.: Generating text with recurrent neural networks. In: Proc. International Conference on Machine Learning (ICML), pp. 1017–1024 (2011)
28. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM journal on computing **1**(2), 146–160 (1972)
29. Wiesler, S., Richard, A., Golik, P., Schluter, R., Ney, H.: RASR/NN: The RWTH neural network toolkit for speech recognition. In: Proc. International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 3305–3309 (2014)
30. Yu, D., Eversole, A., Seltzer, M., Yao, K., Guenter, B., Kuchaiev, O., Seide, F., Wang, H., Droppo, J., Huang, Z., Zweig, G., Rossbach, C., Currey, J.: An introduction to computational networks and the computational network toolkit. Microsoft Technical Report (2014)
31. Yu, D., Seltzer, M.L.: Improved bottleneck features using pretrained deep neural networks. In: Proc. Annual Conference of International Speech Communication Association (INTERSPEECH), pp. 237–240 (2011)