

# Learning meanings for sentences

Charles Elkan

elkan@cs.ucsd.edu

February 25, 2014

This chapter explains how to learn the parameters of recursively structured functions, called neural networks, that can represent, to some degree, the semantic content of sentences.

## 1 Recursive definition of meaning

Consider an English sentence of length  $n$ , say “Cats like to chase mice” with  $n = 5$ . Suppose that we are given a binary tree representing the syntactic structure of the sentence. Each word is a leaf node of the tree, and there are  $n - 1$  internal nodes. Each internal node covers a phrase of two or more consecutive words. We will associate a column vector in  $\mathbb{R}^d$  with each node, to represent the meaning of the corresponding phrase. A typical value for the dimensionality  $d$  is 100.

The meaning of each word is initialized to be a random vector in  $\mathbb{R}^d$ . This means that we create a fixed lexicon containing a random vector for each word. Each random vector is generated independently from a Gaussian of dimension  $d$  with mean zero and diagonal covariance matrix  $\sigma^2 I$ . Each time the same word is used in any sentence, the same vector is used as its meaning.

The meaning of a phrase is a function of the meanings of its two components. This is called a compositional approach to semantics. Let the node  $k$  have children  $i$  and  $j$ , whose meanings are  $x_i$  and  $x_j$ . The meaning of node  $k$  is

$$x_k = h(W[x_i; x_j] + b)$$

where  $W$  and  $b$  are parameters to be learned.<sup>1</sup> The notation  $[x_i; x_j]$  means the vector  $x_i$  concatenated vertically with  $x_j$ , so  $W$  is a matrix in  $\mathbb{R}^{d \times 2d}$  and  $b$  is a

---

<sup>1</sup>The node  $k$  is the parent of  $i$  and  $j$  in the undirected binary tree that represents the grammatical

vector in  $\mathbb{R}^d$ . The function  $h()$  is a pointwise sigmoid-shaped function from  $\mathbb{R}^d$  to the interval  $[-1, +1]^d$ .

We want to learn the parameters  $W$  and  $b$ . To do so in a supervised way, we need target values for the meanings of training sentences. The predicted meaning of a whole sentence is  $x_r$  where  $r$  is the root node. (We say “predicted” because that is standard terminology for what is computed by a model, but the word “output” is more appropriate.) Suppose that a target value  $t$  is known for the meaning of the root node, that is of the whole sentence. Then we can define a loss function  $E$ , for example square loss  $E = (t - x_r)^2$ , and train the parameters  $W$  and  $b$  to minimize  $E$ . Specifically, we can compute the gradients  $\partial E/\partial W$  and  $\partial E/\partial b$ , and use any gradient descent method to minimize  $E$ .

The error  $E$  is for a single sentence. Obviously many sentences are needed in any useful training set. We can use stochastic gradient descent (SGD) and optimize  $W$  and  $b$  based on one sentence at a time. Or, we can define the error for a whole training set as the sum of the errors for each sentence in it. Then, the gradient with respect to the whole set is the sum of the gradients with respect to each sentence. We can use the LBFGS quasi-Newton method to optimize  $W$  and  $b$  based on the whole training set. Of course, we can add a regularization term to the error function regardless of what minimization method is used.

One difficulty with the approach above is that the same parameters  $W$  and  $b$  are used at every node inside a tree. This implies that the predicted meaning  $x_r$  of the root node is specified by an expression that depends on the parameters in multiple ways. This makes the derivatives  $\partial E/\partial W$  and  $\partial E/\partial b$  more complicated, but the concept behind gradient descent is unchanged. Section 5 below shows how to compute the derivatives numerically in an efficient and elegant way, without making symbolic expressions for the derivatives explicit, as was done in other chapters.

Another practical difficulty is that the gradient for a single example is not sparse. This means that a lot of computation is needed even to do one small update of the parameters.

## 2 Autoencoders

A major difficulty with the approach just suggested is that we do not know what the target meaning  $t$  should be for a sentence. In the autoencoder approach, the

---

structure of the sentence. Viewed as a neural network, there are directed edges from  $i$  to  $k$  and from  $j$  to  $k$ . From this perspective, leaf nodes are sources and  $k$  is a child, not parent, of  $i$  and  $j$ .

goal of supervised learning is different: it is to reconstruct the input. Remember the definition

$$x_k = h(W[x_i; x_j] + b).$$

Now, consider the additional definition

$$[z_i; z_j] = Ux_k + c$$

where  $U$  is a matrix in  $\mathbb{R}^{2d \times d}$  and  $c$  is a vector in  $\mathbb{R}^{2d}$ . We view  $z_i$  and  $z_j$  as approximate reconstructions of the inputs  $x_i$  and  $x_j$ , while  $U$  and  $v$  are additional parameters to be trained to maximize the accuracy of reconstructions. Specifically, the square loss at the node  $k$  is

$$E = \|x_i - z_i\|^2 + \|x_j - z_j\|^2 = \|[x_i; x_j] - Uh(W[x_i; x_j] + b) - c\|^2.$$

The error for a whole tree is defined to be the sum of the errors at all the non-leaf nodes of the tree. Note that at each such node, the definition of  $E$  implicitly assumes that the meanings of its children are correct. With fixed meaning vectors for the leaves of the tree, gradient methods can be used to learn  $W$ ,  $b$ ,  $U$ , and  $c$ , with no training labels provided from the outside.

To work in practice, the autoencoder approach needs two refinements. The purpose of the first refinement is to avoid ending up with all meanings equal to zero. That would give zero error at all nodes whose children are not leaf nodes, but not be useful. A simple way to achieve this is to enforce that all meaning vectors have unit length. This is done by defining

$$x_k = \frac{h(W[x_i; x_j] + b)}{\|h(W[x_i; x_j] + b)\|}.$$

This definition does make derivatives more complicated. It may be not needed when the tree structure is determined endogenously, as in the next section.

The second refinement is based on the insight that it is more important, or more difficult, to reconstruct accurately the meanings of longer phrases. Therefore, the definition of loss for node  $k$  is changed to be weighted:

$$E_1(k) = \frac{n_i}{n_i + n_j} \|x_i - z_i\|^2 + \frac{n_j}{n_i + n_j} \|x_j - z_j\|^2 \quad (1)$$

where  $z_i$  and  $z_j$  are the reconstructions defined above, and  $n_i$  and  $n_j$  are how many words are covered by nodes  $i$  and  $j$ .

### 3 Selecting a tree structure

The previous section eliminated the need for target values for the meanings of sentences. This section eliminates the need to know the tree structure of a sentence.

Equation 1 defines the reconstruction error of a single non-leaf node. For a given sentence, let  $T$  be the set of non-leaf nodes of its binary tree. The error of the whole tree is

$$\sum_{k \in T} E_1(k).$$

For a sentence of length  $n$ , there is an exponential number of possible trees.<sup>2</sup> Define the optimal tree to be the one that minimizes this total error. Although one could find the optimal tree by exhaustive enumeration, we will use a greedy algorithm to find a tree that is good but not necessarily optimal.

The greedy algorithm is simple. First, consider all  $n - 1$  pairs of consecutive words. Evaluate the reconstruction error for each pair. Select the pair with smallest error. Now, either one or two of the original pairs can no longer be combined. Consider the remaining feasible pairs, plus the one or two new possible pairs on top of the first selected pair. Select the pair with smallest error among these  $n - 2$  possible pairs. Continue until there is only one possible choice to create the root node.

The parameters of the recursive autoencoder model are  $W$ ,  $b$ ,  $U$ , and  $c$ . For a given sentence and current values for these parameters, consider the tree found by the greedy algorithm using these parameter values. A small change in the parameter values either causes no change in this tree, or a jump to a different tree. In principle, gradient descent could cause cycling between two or more different trees, without convergence. However, LBFSGS using a large training set converges smoothly in practice. At first, because the parameters are initialized randomly, the tree constructed for each sentence is arbitrary. However, as training progresses, the tree becomes more sensible.

---

<sup>2</sup>The precise number is  $C_{n-1}$ , where  $C_n$  is the  $n$ th Catalan number,

$$C_n = \binom{2n}{n} - \binom{2n}{n+1}.$$

For details, see Wikipedia.

## 4 Using meanings to predict labels

Suppose that there is a target value (not a target meaning) to be predicted for each sentence. For example, the target value might be a measure of the positivity or negativity of the sentence. The target value for “Selena Gomez is the raddest” might be  $+1$ , while the target value for “She makes Britney Spears sound good” might be  $-1$ .

Each node  $k$  of a tree has a certain meaning vector  $x_k$ . We can add a linear model on top of these vectors to predict the target values. If the values are binary, then the linear model is a standard logistic regression classifier. If there are three or more discrete values, the model is called multinomial or multiclass logistic regression, which is a special case of log-linear modeling as described in a previous chapter.

Let  $x_k$  be the meaning of node  $k$ , and suppose that there are  $r$  alternative discrete labels. Using multiclass logistic regression, the vector of predicted probabilities of the label values can be written as

$$\bar{p} = \text{softmax}(Vx_k)$$

where the parameter matrix  $V$  is in  $\mathbb{R}^{r \times d}$ . Note that  $\bar{p}$  is the value of an additional node in the neural network. Let  $\bar{t}$  be the binary vector of length  $r$  indicating the true label value of node  $k$ . The squared error of the predictions can be written  $\|\bar{t} - \bar{p}\|^2$ . Or, the log loss of the predictions can be written

$$E_2(k) = - \sum_{i=1}^r t_i \log p_i.$$

Suppose that a target value is known for a whole sentence. We could predict this value just as a function of the meaning  $x_r$  of the root node of the sentence. Or, we could predict it as a function of any subset of the nodes of the sentence. We choose to predict it for all the internal nodes, but not for the leaf nodes. Intuitively, we assume that the label for the sentence applies to all the phrases of the sentence, but not necessarily to each word of the sentence.

Specifically, the objective function to be minimized during learning, given a collection  $S$  of  $m$  labeled training sentences, is

$$J = \frac{1}{m} \sum_{(s,t) \in S} E(s, t, \theta) + \frac{\lambda}{2} \|\theta\|^2$$

where  $\theta = \langle W, b, U, c, V \rangle$  is all the parameters of the model,  $\lambda$  is the strength of  $L_2$  regularization, and  $E(s, t, \theta)$  is the total error for one sentence  $s$  with label  $t$ . This total error is

$$E(s, t, \theta) = \sum_{k \in T(s)} \alpha E_1(k) + (1 - \alpha) E_2(k)$$

where  $T(s)$  is the set of non-leaf nodes of the tree that is constructed greedily for  $s$ . Remember that this tree depends on  $\theta$ . The hyperparameter  $\alpha$  defines the relative importance of the reconstruction and label errors. Both  $E_1$  and  $E_2$  are per-node losses, while  $E$  is the per-sentence loss.

Until now, we have assumed that the meanings of individual words, that is the vectors assigned to leaf nodes, are fixed. In fact, these can be adjusted. Let  $x_n$  be the meaning of a word. We can compute the derivative of the loss with respect to  $x_n$  and use it to change  $x_n$ . In practice, the gradient of label errors  $E_2$  is used to modify word meanings, but not the gradient of reconstruction errors  $E_1$ . This makes changes in the meanings of words more focused on the ultimate task, which is to predict labels. Because the meaning of each word is the same across all sentences, the loss that drives changes in meanings must be a sum over all training sentences. Specifically, the derivative used is

$$\frac{\partial}{\partial x_n} \sum_{\langle s, t \rangle \in S} \sum_{k \in T(s)} E_2(k).$$

Note that the loss  $E_2$  is defined at non-leaf nodes, but the derivative is with respect to leaf node values. For each word, the derivative is a sum over all leaf nodes that involve that word, in all sentences.

## 5 Backpropagation

The critical question not answered above is how to evaluate all the derivatives that are needed for training. Backpropagation (backprop for short) is the name of a method to compute these derivatives efficiently.

This section uses notation that is slightly different from above. Here, each node is designated by an index such as  $j$ , and has a single real value. This value is the scalar computed at the node, which is designated  $z_j$ . Each node above corresponds to  $d$  nodes in this section. The explanations here apply to any feedforward network; recursive autoencoders are just a special case.

Let  $j$  be any non-leaf node, and let  $i$  be the nodes that feed into it. There is a directed edge  $i \rightarrow j$  from each  $i$  to  $j$ , and each  $i$  is a parent of  $j$ . The value computed at node  $j$  is

$$z_j = h\left(\sum_i w_{ij} z_i\right) = h(a_j)$$

where  $h$  is a scalar function  $\mathbb{R} \rightarrow \mathbb{R}$  and  $a_j$  is called the total activation coming into node  $j$ . Note that  $w_{ij}$  is the weight from node  $i$  to node  $j$ . Let  $J$  be the loss for one training example. We want to know the partial derivative of  $J$  with respect to  $w_{ij}$ . This can be written as

$$\frac{\partial J}{\partial w_{ij}} = \frac{\partial J}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}}$$

using the chain rule for derivatives.<sup>3</sup> This rewriting is useful because the second factor is easy to evaluate:

$$\frac{\partial a_j}{\partial w_{ij}} = z_i.$$

Now, make the definition

$$\frac{\partial J}{\partial a_j} = \delta_j.$$

For every node  $j$ ,  $\delta_j$  is the derivative of the overall loss with respect to the total activation  $a_j$  coming into  $j$ . This derivative must take into account any nonlinearity that is part of the operation performed inside node  $j$ . The delta notation  $\delta_j$  is traditional in explanations of backpropagation.

An output node is one that does not feed into any other node, on which part of the total loss is defined directly. From a graph perspective, an output node is a sink, while a leaf node is a source. For each output node  $k$ , a target value  $t_k$ , i.e. a training label, is known. The total loss  $J$  is the sum over all output nodes of the loss  $J_k$  at each one; different output nodes may have different loss functions. For example, some output nodes may use log loss while others use square loss.

---

<sup>3</sup>The chain rule is not as straightforward as it may appear. The rule looks quite different in different notations. Using the notation of function composition, it is  $(f \circ g)' = (f' \circ g)g'$ , which means  $[f(g(x))]' = f'(g(x)) \cdot g'(x)$ . Using the notation of Leibniz and writing  $y = f(g(x))$ , the chain rule is

$$\frac{dy}{dx} = \frac{dy}{dg} \frac{dg}{dx}.$$

The two forms of the rule are equivalent, of course. The second form seems more intuitive, because it looks like the  $dg$  expressions cancel each other, but the first form shows more explicitly what is really meant.

For an output node  $k$  with local loss function  $J_k = L(z_k, t_k)$ ,

$$\delta_k = \frac{\partial J}{\partial a_k} = \frac{\partial J_k}{\partial a_k} = \frac{\partial L(z_k, t_k)}{\partial z_k} \frac{\partial z_k}{\partial a_k} = \frac{\partial L(z_k, t_k)}{\partial z_k} h'(a_k).$$

If the node  $k$  is linear, then  $h'(a_k) = 1$ . If the nonlinear function is  $h(a) = \tanh(a) = (e^a - e^{-a})/(e^a + e^{-a})$ , then the derivative  $h'(a) = dh/da = 1 - h(a)^2$ .

As a special case, suppose that the output node is linear and has square loss  $J_k = \frac{1}{2}(z_k - t_k)^2$ . Then  $z_k = a_k$  and  $\delta_k = z_k - t_k$ . As a different special case, suppose the output node is linear and has log loss  $J_k = -t_k \log z_k - (1 - t_k) \log(1 - z_k)$ . Then  $\delta_k = -t_k/z_k + (1 - t_k)/(1 - z_k)$ .

In many applications, it is beneficial for output nodes to be linear, for the following reason. Consider the equation  $\delta_k = (\partial L/dz_k)h'(a_k)$  and suppose that the output value  $z_k = h(a_k)$  is large when the target value is small, or vice versa. In this case we want  $\delta_k$  to be large, in order to cause major changes in  $w_{jk}$  for nodes  $j$  feeding into  $k$ . However, when  $a_k$  is large positive or large negative, then  $h'(a_k)$  is near zero, for any sigmoid-shaped function  $h$ . So if an output node uses a sigmoid, then its delta value may be small even though its value is very wrong. Hence it is preferable to have linear output nodes.

Now consider a node  $j$  that is not an output node, that feeds into nodes indexed by  $k$ . The node  $j$  influences the overall loss through all of these, so

$$\delta_j = \frac{\partial J}{\partial a_j} = \sum_k \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j}.$$

Each node  $k$  can have many input nodes in addition to  $j$ . Let  $i$  index these nodes, including  $j$ , so the incoming activation of node  $k$  is

$$a_k = \sum_i w_{ik} z_i = \sum_i w_{ik} h(a_i).$$

The needed partial derivative is

$$\frac{\partial a_k}{\partial a_j} = w_{jk} h'(a_j)$$

and the delta value of node  $j$  is

$$\delta_j = h'(a_j) \sum_k \delta_k w_{jk}. \quad (2)$$

Usually, a leaf node  $j$  is considered to be an input value  $x_j$  that is fixed. However, it is possible to compute derivatives with respect to input values. Suppose that the input node  $j$  feeds into nodes  $k$ . Then

$$\frac{\partial J}{\partial x_j} = \sum_k \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial x_j} = \sum_k \delta_k w_{jk}.$$

This derivative can be interpreted as the importance of the  $j$ th input for the overall loss  $J$ . If inputs are adjustable, as at the end of the previous section, this derivative can be used for that purpose.

There is another way to work out the derivative with respect to  $x_j$ . Imagine that the node  $j$  has no nonlinearity and a single incoming node  $i$  with fixed value  $z_i = 1$ . Let  $w_{ij} = x_j$  be the weight on the edge  $i \rightarrow j$ . Then

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial w_{ij}} = \delta_j z_i = \delta_j = h'(a_j) \sum_k \delta_k w_{jk} = \sum_k \delta_k w_{jk}$$

using Equation 2.

Concretely, the backpropagation algorithm is as follows:

- Step 1: Compute  $\delta_k$  for all output nodes  $k$ .
- Step 2: Working backwards, for other nodes  $j$  compute  $\delta_j$  using Equation 2.
- Step 3: In any order, for each weight compute  $\partial J / \partial w_{ij} = \delta_j z_i$ .

Remember that  $w_{ij}$  is the weight of the edge  $i \rightarrow j$ . The partial derivative with respect to this weight is the feedforward value of node  $i$  times the delta value of node  $j$ .

If the same weight is shared by more than one edge  $i \rightarrow j$ , then the derivatives computed in Step 3 are added, because any change in the weight influences the total loss  $J$  along each of the edges.

The backpropagation algorithm as just stated is valid for any configuration of nodes that is a directed acyclic graph (DAG). There can be any number of output nodes (sinks), any number of internal nodes in any arrangement as long as cycles are never directed, and any number of input nodes (sources).

In the previous sections, the neural network structure for each sentence is found by a greedy algorithm. The same weights are used for many edges, so derivatives are summed over these edges. Another unusual aspect of the networks above is that they have numerous output nodes. Backpropagation as described here is still valid. Equation 2 essentially says that the deltas from output nodes  $k$  are simply added as necessary.

## 6 Numerical differentiation

One way to verify that derivatives calculated by backpropagation are correct is to compare them to those computed by finite differencing. In general,

$$\frac{\partial J}{\partial w_{ij}} = \frac{J(w_{ij} + \epsilon) - J(w_{ij})}{\epsilon} + O(\epsilon)$$

if  $\epsilon$  is small. Interestingly, numerical derivatives are more accurate if central differences are used:

$$\frac{\partial J}{\partial w_{ij}} = \frac{J(w_{ij} + \epsilon) - J(w_{ij} - \epsilon)}{2\epsilon} + O(\epsilon^2).$$

Suppose that the network contains  $m$  weights in total. Then the feedforward evaluation of  $J$  for a single training example (a single set of input values at the leaf nodes) requires  $O(m)$  time. Hence, computing the derivative for every weight numerically requires  $O(m^2)$  time. In contrast, computing every derivative using backpropagation needs only  $O(m)$  time.

## 7 Backpropagation for vector-valued nodes

Suppose that each node consists of  $d$  scalar subnodes, so the feedforward value of every node is a column vector in  $\mathbb{R}^d$ . Let  $r$  be the value of a node, and let  $q^1$  to  $q^m$  be the values of the nodes feeding into  $r$ . This situation is equivalent to  $md$  scalar nodes feeding into  $d$  scalar nodes. For the recursive autoencoders in previous sections, the number  $m$  of incoming nodes is at most two.<sup>4</sup>

Let the feedforward column vector value  $r$  of a node be given by the equation

$$r = h(a) = h(W[q^1; \dots; q^m] + b)$$

where  $W \in \mathbb{R}^{d \times md}$  and  $b \in \mathbb{R}^d$  is a bias vector. One element  $W_{ij}$  of the matrix  $W$  is the weight from one subnode of one incoming node to subnode  $i$  of the outgoing node. Hence, the meaning of the subscripts  $ij$  is the reverse of the meaning of the subscripts of a scalar weight  $w_{ij}$  in Section 5. The bias vector  $b$  can be

---

<sup>4</sup>This section mostly does not use names such as  $i$  for nodes, in order to make notation as lightweight as possible, so the value of a node is written just as  $r$ , not  $r_i$ . In the notation  $q^k$ , the superscript is an index that ranges from  $k = 1$  to  $k = m$ .

incorporated by writing  $r = h([W, b][q^1; \dots; q^m; 1])$  where  $[W, b]$  is an extended weight matrix.

Let  $J$  be the total loss for one training example. We need to compute the partial derivative  $\partial J / \partial W_{ij}$  for every element  $W_{ij}$  of every extended weight matrix used in the network. Note that these partial derivatives together form a matrix  $\partial J / \partial W$  that has the same shape as  $W$ .

Consider any node whose value is  $h(a) = h(W[q^1; \dots; q^m; 1])$  where  $W$  is an extended weight matrix. The scalar  $W_{ij}$  influences  $J$  only through the  $i$ th element  $a_i$  of  $a$ . Hence

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial a_i} \frac{\partial a_i}{\partial W_{ij}} = \delta_i \frac{\partial a_i}{\partial W_{ij}}. \quad (3)$$

The scalar  $\delta_i$  is the derivative of the overall loss with respect to the total activation  $a_i$  coming into the  $i$ th subnode. This derivative must take into account the nonlinearity in  $h$ . For the whole node,  $\delta$  is a column vector with dimension  $d$ .

For each  $j$ , the factor on the far right of Equation 3 is one of the elements of one of the incoming vector values. Specifically, if  $j \leq d$  then  $\partial a_i / \partial W_{ij} = q_j^1$  and if  $d < j \leq 2d$  then  $\partial a_i / \partial W_{ij} = q_{j-d}^2$  and so on. In vector notation,

$$\frac{\partial a_i}{\partial W_{i:}} = [q^1; \dots; q^m; 1]^T.$$

As a special case,  $\partial a_i / \partial b_i = 1$ . Remember that  $\partial a_i / \partial b_j = 0$  when  $i \neq j$ .

For each output node with vector value  $r$ , assume that the contribution to  $J$  from this node is a sum over the elements of  $r$ , where the contribution from  $r_i$  is  $L_i$  for  $i = 1$  to  $i = d$ . Then

$$\delta_i = \frac{\partial J}{\partial a_i} = \frac{\partial L_i}{\partial a_i} = \frac{\partial L_i}{\partial r_i} \frac{\partial r_i}{\partial a_i} = \frac{\partial L_i}{\partial r_i} h'(a_i).$$

As a special case, consider a linear output node with target vector value  $t$ . If the node uses log loss  $-\sum_{i=1}^d t_i \log r_i$ , then  $\delta_i = -t_i / r_i$ , while if it uses square loss, then  $\delta_i = r_i - t_i$ .

Now consider a non-output node with current vector value  $h(a)$  that feeds into one or more nodes indexed by  $k$ . This node influences the overall loss through all subnodes  $p$  of all of those, so for it

$$\delta_i = \frac{\partial J}{\partial a_i} = \sum_k \sum_{p=1}^d \frac{\partial J}{\partial a_p^k} \frac{\partial a_p^k}{\partial a_i} = \sum_k \sum_{p=1}^d \delta_p^k \frac{\partial a_p^k}{\partial a_i}$$

where  $\delta^k$  is the delta vector for node  $k$ . Each node  $k$  can have other input nodes also, so the incoming activation of node  $k$  is the vector

$$a^k = W[\dots; h(a); \dots; 1].$$

Let  $V^k$  be the portion of the weight matrix  $W$  that multiplies  $h(a)$  in the equation above, so  $a_p^k = V_p^k h(a) + c$  where  $c$  depends on the other incoming nodes of node  $k$ , but is constant with respect to  $a$ . Note that  $V^k$  is a square matrix that in general is different for different  $k$ , even if  $W$  is shared, because for different  $k$  the position of  $z$  in the list of incoming nodes of  $k$  varies. The needed partial derivative is

$$\frac{\partial a_p^k}{\partial a_i} = h'(a_i) V_{pi}^k.$$

Therefore

$$\delta_i = h'(a_i) \sum_k \sum_{p=1}^d \delta_p^k V_{pi}^k = h'(a_i) \sum_k \delta^k \cdot V_{:i}^k = h'(a_i) \sum_k (\delta^k)^T V_{:i}^k$$

where  $\cdot$  signifies dot product and  $V_{:i}^k$  means the  $i$ th column of  $V^k$ . In matrix notation, the whole delta vector is

$$\delta = h'(a) \circ \left[ \sum_k (\delta^k)^T V^k \right]^T \quad (4)$$

where  $\circ$  signifies pointwise multiplication, which is also called Hadamard multiplication.

When nodes are vector-valued, the backpropagation algorithm is as follows:

- Step 1: Compute the delta vector for each output node.
- Step 2: Working backwards, compute the delta vector of each non-output node using Equation 4.
- Step 3: For every node, let  $\delta$  be its delta vector and let  $q^1$  to  $q^m$  be the values of the nodes feeding into it, using the weight matrix  $W$ . If  $m = 0$  then the node is a leaf. Otherwise, the matrix of derivatives with respect to  $W$  is the outer product

$$\frac{\partial J}{\partial W} = \delta [q^1; \dots; q^m]^T.$$

Whether  $m = 0$  or not,  $\partial J / \partial b = \delta$ .

A leaf node, which is also called an input or source node, has no incoming nodes. Instead, it has an input value  $x \in \mathbb{R}^d$ . Usually,  $x$  is fixed, i.e. not adjustable. If  $x$  is adjustable, it is equivalent to a bias vector  $b$  for the case  $m = 0$ . Precisely, the input vector  $x = h(b)$ .

As mentioned before, weights may be shared, meaning that the same weights are used by more than one node. Like other weights, a bias vector may be shared. In particular, the meaning of a word that appears more than once, in one or more sentences, is one input vector that is tied to have the same value across all its appearances.

Whenever a weight is shared, its derivative is a sum over all nodes that share the weight. The bias vector  $b$  that is the meaning of a word may be adjusted using the derivative

$$\frac{\partial J}{\partial b} = \sum_k \delta^k$$

where  $k$  ranges over all input nodes that use this same word.

*Additional explanations needed above: Using the total derivative to justify adding derivatives from different nodes. The full Jacobian for  $\tanh(a)/\|\tanh(a)$ . Separate output nodes for reconstruction error. Extra derivative terms for leaf nodes because they contribute to reconstruction error.*