Author: Burak Himmetoglu

Notes on the Neural Probabilistic Language Model

Below is a schematic representation of the neural language model. The network can be summarized as follows:

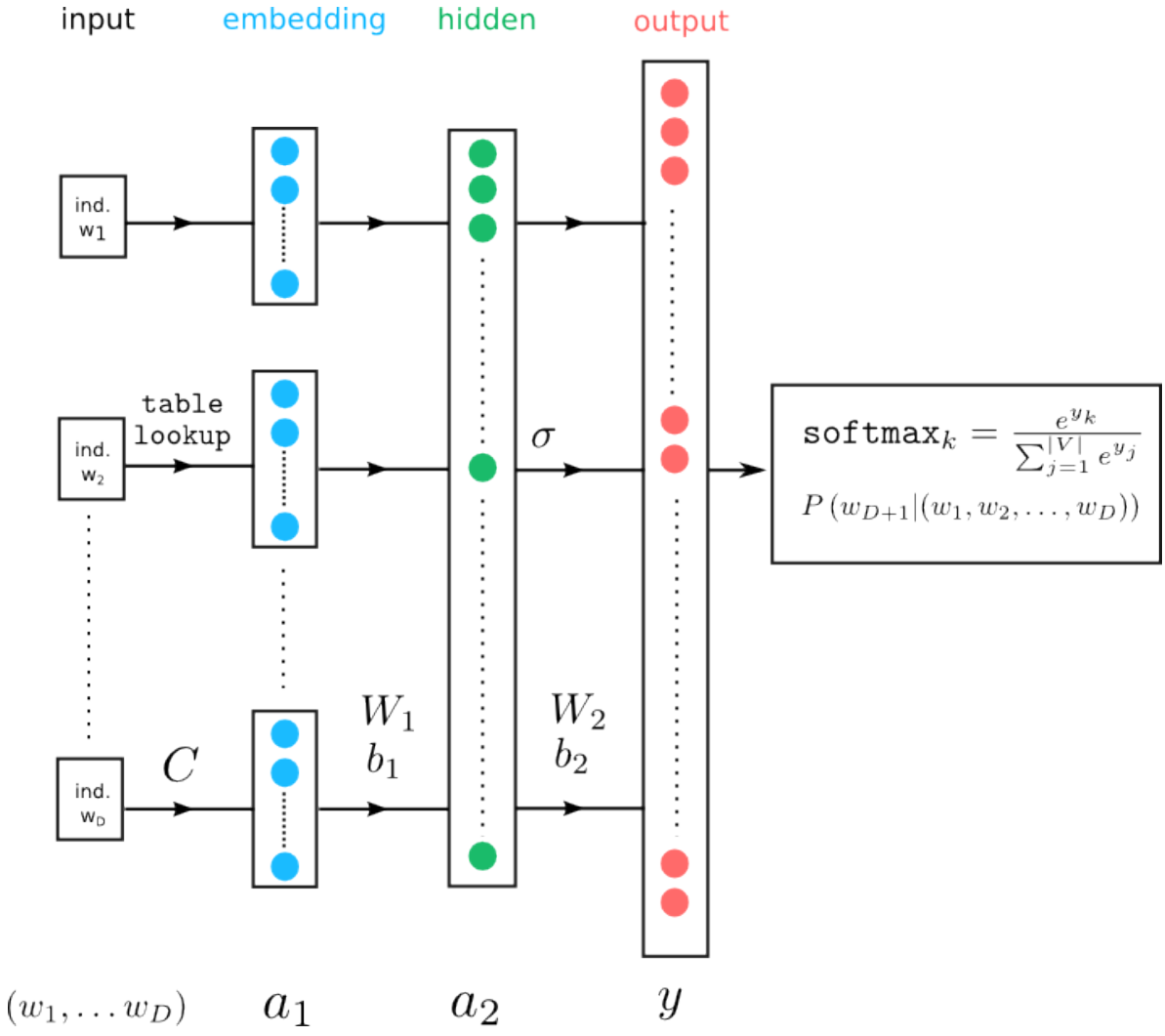

FIG. 1. The Neural Language Model

1. Size of vocabulary: $|V|$
2. Number of words in the sequence: D
3. Word vectors: $w_1, w_2, \ldots, w_D \in \mathbb{R}^{|V|}$
4. Dimension of each embedding layer : h_1

5. Word embedding weights: $C \in \mathbb{R}^{|V| \times h_1}$

6. Hidden layer size: h_2

7. Embedding to hidden layer weights and biases: $W_1 \in \mathbb{R}^{Dh_1 \times h_2}$, $b_1 \in \mathbb{R}^{h_2}$

8. Output layer size: $|V|$

9. Hidden to output layer weights and biases: $W_2 \in \mathbb{R}^{h_2 \times |V|}$, $b_2 \in \mathbb{R}^{|V|}$

Each word is encoded as a $|V|$ dimensional vector $\mathbf{w}$, where $|V|$ is the size of the vocabulary. Given a set of sequence of D words, the task is to compute the probability of the $(D+1)^{\text{th}}$ word, i.e.

$$P\left(w_{D+1}^{(i)} | (w_1^{(i)}, w_2^{(i)}, \ldots, w_D^{(i)})\right)$$

for a given example i. The first layer of the network (embedding) transforms the indices of words in the vocabulary into a h_1 dimensional vector representation using the word embedding weights $C \in \mathbb{R}^{|V| \times h_1}$. To understand better, let's look at an example. Consider the following sentence as a given training example:

`We all have it`

In this case there are 3 preceding words: `"We", "all", "have"` so $D = 3$ and the target word is `"it"` for which we need to determine the conditional probability $P(\texttt{it} | (\texttt{We}, \texttt{all}, \texttt{have}))$. Suppose that these words are indexed in the vocabulary (V) as follows:

$$\texttt{We} : 91 \quad \texttt{all} : 1 \quad \texttt{have} : 3 \quad \texttt{it} : 181$$

The vector $\mathbf{w}_1$ representing `"We"` in this training example is then $|V|$ dimensional with the 91^{th} element being 1 and the rest are 0's. The embedding weights C take each D-word example, and convert them into a $D \cdot h_1$ dimensional embedding vector a_1. More precisely,

$$a_1^{(i)} = \begin{bmatrix} C^T w_1^{(i)} \\ C^T w_2^{(i)} \\ \vdots \\ C^T w_D^{(i)} \end{bmatrix} \tag{1}$$

The embeddings C are shared among all the D words. In passing from the input layer with word indices to the embedding layer to get a_1, equation (1) represents a table lookup.

Namely, for each given word index, one needs to consider what row in C its embedding lies to obtain a_1. To make it clearer, consider our example above. Let's say that we have the following batch of training examples, stacked in a matrix containing the corresponding word indices:

$$\texttt{input\_batch} = \begin{bmatrix} 91 & 5 & \\ 1 & 112 & \dots \\ 3 & 82 & \end{bmatrix} \tag{2}$$

Each column represent a training example: the first column represent the words `We all have` given above. Given word embeddings C, the table lookup involves unfolding `input_batch` into a vector and subsetting the rows of C by this vector, giving the words' location in the vocabulary V. Namely,

$$\begin{aligned}
&\texttt{input_batch_vec} = [91\ 1\ 3\ 5\ 112\ 82\ \dots] \\
&a_1^{(1)} = C\left[[91\ 1\ 3], :\right] = \begin{bmatrix} C[91,:] \\ C[1,:] \\ C[3,:] \end{bmatrix} \\
&a_1^{(2)} = C\left[[5\ 112\ 82], :\right] = \begin{bmatrix} C[5,:] \\ C[112,:], \\ C[82,:] \end{bmatrix} \\
&\vdots
\end{aligned}$$

In Matlab or Octave, this is easily achieved by

```
embedding_layer_state = reshape(...
  word_embedding_weights(reshape(input_batch, 1, []),:)',...
  numhid1 * numwords, []);
```

where `embedding_layer_state` is vectors $a_1^{(i)}$'s stacked columnwise into a matrix $A_1 \in \mathbb{R}^{Dh_1 \times n_{\text{batch}}}$ to obtain

$$A_1 = \begin{bmatrix} | & | & & | \\ a_1^{(1)} & a_1^{(2)} & \dots & a_1^{(n_{\text{batch}})} \\ | & | & & | \end{bmatrix} \tag{3}$$

`word_embedding_weights` is C, `numwords` is $D = 3$ and `numhid1` is the size of the embedding layer h_1, and n_{batch} is the size of the input training batch.

Feed forward: Embedding to Hidden Layer

Now we have obtained the embedding layer states $a_1^{(i)}$ for each example in the training batch, we can move to the hidden layer by weights W_1 and biases b_1, using

$$z_2 = b_1 + W_1^T \cdot a_1 \tag{4}$$

where $z_2 \in \mathbb{R}^{h_2}$, $b_1 \in \mathbb{R}^{h_2}$ and $W_1 \in \mathbb{R}^{Dh_1 \times h_2}$. In Matlab or Octave, this is achieved by

```
% Compute inputs to hidden units.
inputs_to_hidden_units = embed_to_hid_weights'*embedding_layer_state + ...
  repmat(hid_bias, 1, batchsize);
```

Here, `inputs_to_hidden_units` represent the vectors z_2 stacked into a matrix $Z_2 \in \mathbb{R}^{h_2 \times n_{\text{batch}}}$ i.e.,

$$Z_2 = \begin{bmatrix} | & | & & | \\ z_2^{(1)} & z_1^{(2)} & \dots & z_2^{(n_{\text{batch}})} \\ | & | & & | \end{bmatrix} \tag{5}$$

and `batchsize` is n_{batch}. The final step in this layer is to compute the hidden layer state, obtained by the activation function σ, which we choose to be the sigmoid for this example:

$$a_2 = \sigma(z_2) \tag{6}$$

In Matlab or Octave:

```
% Apply logistic activation function
hidden_layer_state = 1 ./ (1 + exp(-inputs_to_hidden_units));
```

Similary, `hidden_layer_state` is the matrix $A_2 \in \mathbb{R}^{h_2 \times n_{\text{batch}}}$:

$$A_2 = \begin{bmatrix} | & | & & | \\ a_2^{(1)} & a_1^{(2)} & \dots & a_2^{(n_{\text{batch}})} \\ | & | & & | \end{bmatrix} \tag{7}$$

Feed forward: Hidden to Output Layer

Now we can move to the output layer using the weights W_2 and biases b_2:

$$y = b_2 + W_2^T \cdot a_2 \tag{8}$$

where $y \in \mathbb{R}^{|V|}$, $b_2 \in \mathbb{R}^{|V|}$ and $W_2 \in \mathbb{R}^{h_2 \times |V|}$. The elements of the vector y represent in the unnormalized log-probabilities for each $(D+1)^{\text{th}}$ word following $(w_1, \ldots, w_D)$ for the examples in the training batch. This vector is an input to the softmax function which we will use to compute the conditional probability that the $(D+1)^{\text{th}}$ word has index k in the vocabulary:

$$P\left(w_{D+1}^{(i)} = k | (w_1^{(i)}, \ldots, w_D^{(i)})\right) = \texttt{softmax}_k(y) = \frac{e^{y_k}}{\sum_{j=1}^{|V|} e^{y_j}} \tag{9}$$

In Matlab or Octave:

```
% Compute inputs to softmax.
inputs_to_softmax = hid_to_output_weights' * hidden_layer_state + ...
    repmat(output_bias, 1, batchsize);
```

Here, `inputs_to_softmax` are vectors y stacked columnwise into a matrix $Y \in \mathbb{R}^{|V| \times n_{\text{batch}}}$, i.e.

$$Y = \begin{bmatrix} | & | & & | \\ y^{(1)} & y^{(2)} & \ldots & y^{(n_{\text{batch}})} \\ | & | & & | \end{bmatrix} \tag{10}$$

`hid_to_output_weights` is W_2 and `output_bias` is b_2.

Computing Probabilities

Now, we can compute word probabilities using $y^{(i)}$ for each example with the softmax function in equation (9). For numerical stability, we subtract from the argument of softmax, the maximum value of each $y^{(i)}$, which does not affect the final probabilities. In Matlab or Octave:

```
% Subtract the max
inputs_to_softmax = inputs_to_softmax...
  - repmat(max(inputs_to_softmax), vocab_size, 1);
% Compute exp.
output_layer_state = exp(inputs_to_softmax);
```

```
% Normalize to get probability distribution.
output_layer_state = output_layer_state ./ repmat(...
  sum(output_layer_state, 1), vocab_size, 1);
```

where `output_layer_state` is the conditional probabilities stacked columnwise, i.e.

$$P = \begin{bmatrix} P(w_{D+1}^{(1)} = 1) & \dots & P(w_{D+1}^{(n_{\text{batch}})} = 1) \\ P(w_{D+1}^{(1)} = 2) & \dots & P(w_{D+1}^{(n_{\text{batch}})} = 2) \\ \vdots & \ddots & \vdots \\ P(w_{D+1}^{(1)} = |V|) & \dots & P(w_{D+1}^{(n_{\text{batch}})} = |V|) \end{bmatrix} \tag{11}$$

In summary, the whole neural network performs the following operation

$$\boxed{y = b_2 + W_2^T \cdot \sigma\left(b_1 + W_1^T \cdot a_1\right), \qquad P = \texttt{softmax}(y)} \tag{12}$$

Loss Function

The loss function that we minimize during training is the maximum log-likelihood, given the word probabilties. Namely,

$$L = \sum_{i=1}^{n_{\text{batch}}} \sum_{k=1}^{|V|} \log(P_k^{(i)})\, I(w_{D+1}^{(i)} = k) \tag{13}$$

where $P_k^{(i)} = P_{ki}$ from equation (11) which is the probability of obtaining word k in example (i) for the $(D+1)^{\text{th}}$ word. At the moment, we do not include any regularization term for brevity. We also define $I(w_{D+1}^{(i)} = k) \equiv \mathbb{I}_{ji}$. $\mathbb{I} \in \mathbb{R}^{|V| \times n_{\text{batch}}}$ is 1 whenever the index of the word in the training batch meets with its location in the vocabulary. For example,

$$w_{D+1} = \begin{bmatrix} 2 \\ 4 \\ 181 \\ \vdots \end{bmatrix}, \qquad \mathbb{I} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad \mathbb{I}_{2,1} = 1,\ \mathbb{I}_{4,2} = 1,\ \mathbb{I}_{181,3} = 1, \dots \tag{14}$$

In Matlab or Octave, the Loss function is computed as a double sum over the training examples and words, with $\mathbb{I}$ subsetting the correct indices for words:

```
% Expand the target to a sparse 1-of-K vector.
expansion_matrix = eye(vocab_size);
expanded_target_batch = expansion_matrix(:, target_batch);
% Cross-Entropy
CE = -sum(sum(...
  expanded_target_batch .* log(output_layer_state + tiny))) / batchsize;
```

Here, the `target_batch` is the indices of words w_{D+1} (a $|V|$ dimensional vector) in the training batch, `expanded_target_batch` is $\mathbb{I}$ defined above, `tiny` is an infinitesimally small constant added for numerical stability and `batchsize` is the number of examples in the training batch. The division by `batchsize` turns the loss function into the Cross Entropy.

Backpropagation: Probabilities to Output Layer

Now that we have obtained the forward propagation scheme for the network, we need to compute the gradients with respect to the weights and biases to optimize their values. First, we compute the gradient of the loss function with respect to the unnormalized log-probabilities y. Let's do this for the derivative for a single training example. Using the definition of softmax in equation (9),

$$
\begin{aligned}
L^{(i)} &= -\log P_k^{(i)}, \qquad (w_k^{(i)} = 1) \\
\frac{\partial L^{(i)}}{\partial y_j} &= -\frac{1}{P_k^{(i)}} \frac{\partial P_k^{(i)}}{\partial y_j} = -\frac{1}{P_k^{(i)}} \begin{cases} P_j^{(i)} \left(1 - P_j^{(i)}\right), & k = j \\ -P_k^{(i)} P_j^{(i)}, & k \neq j \end{cases} \\
&= P_j^{(i)} - I(w_{D+1}^{(i)} = j)
\end{aligned}
$$

where $L^{(i)}$ is the contribution to loss from a single example, and δ_{kj} is the Kronecker delta. This derivative above is defined as the error derivative:

$$
\mathbb{E}_{ji} \equiv P_j^{(i)} - \mathbb{I}_{ji} \tag{15}
$$

which will multiply all the weights in all the layers. In Matlab or Octave:

```
% Compute derivative of cross-entropy loss function.
error_deriv = output_layer_state - expanded_target_batch;
```

Backpropagation: Output to Hidden Layer

For this backpropagation step, we first notice that equation (12) implies

$$y^{(i)} = b_2 + W_2^T \cdot a_2^{(i)}$$

Then, the derivative of the loss function with respect to W_2 is obtained as follows: First, we compute the derivative of the loss for a single example as

$$\begin{aligned}\frac{\partial L^{(i)}}{\partial W_{2\,mn}} &= \sum_{j=1}^{|V|} \frac{\partial L^{(i)}}{\partial y_j^{(i)}} \frac{\partial y_j^{(i)}}{\partial W_{2\,mn}} \\ &= \sum_{j=1}^{|V|} \mathbb{E}_{ji}\, a_{2\,n}^{(i)}\, \delta_{jm} = a_{2\,m}^{(i)}\, \mathbb{E}_{ni}\end{aligned}$$

Summing over all the examples in the training batch, we obtain the derivative of the full loss function as

$$\begin{aligned}\frac{\partial L}{\partial W_{2\,nm}} &= \frac{\partial}{\partial W_{2\,nm}} \left(\sum_{i=1}^{n_{\text{batch}}} L^{(i)} \right) = \sum_{i=1}^{n_{\text{batch}}} a_{2\,n}^{(i)}\, \mathbb{E}_{mi} \\ &= \sum_{i=1}^{n_{\text{batch}}} A_{2\,mi}\, \mathbb{E}_{in}^T\end{aligned}$$

which in matrix notation:

$$\left(\frac{\partial L}{\partial W_2} \right) = A_2 \cdot \mathbb{E}^T \tag{16}$$

The derivative with respect to the bias is obtained similarly:

$$\frac{\partial L^{(i)}}{\partial b_{2\,n}} = \sum_{j=1}^{|V|} \frac{\partial L^{(i)}}{\partial y_j^{(i)}} \frac{\partial y_j^{(i)}}{\partial b_{2\,n}} = \mathbb{E}_{ni}, \qquad \frac{\partial L}{\partial b_{2\,n}} = \sum_{i=1}^{n_{\text{batch}}} \mathbb{E}_{ni} \tag{17}$$

In Matlab or Octave, the above equations (16) and (17) can be implemented as

```
% Gradient w.r.t W2
hid_to_output_weights_gradient =  hidden_layer_state * error_deriv';
% Gradient w.r.t b2
output_bias_gradient = sum(error_deriv, 2);
```

We also define the following quantity, the back propagated derivative of W_2, as follows:

$$\Delta_{1\,m}^{(i)} = \Delta_{1\,mi} \equiv (W_2 \cdot \mathbb{E})_{mi}\, A_{2\,mi}\, (1 - A_{2\,mi}) \tag{18}$$

which will be used in backpropagating from the hidden layer to embedding layer. Notice that $\Delta_1 \in \mathbb{R}^{h_2 \times n_{\text{batch}}}$. In Matlab or Octave, it can be calculated as

```
back_propagated_deriv_1 = (hid_to_output_weights * error_deriv) ...
   .* hidden_layer_state .* (1 - hidden_layer_state);
```

Backpropagation: Hidden to Embedding Layer

In this step we compute the derivatives with respect to W_1 and b_1. For this, we use equation (12). The derivative with respect to W_1 can be obtained as follows:

$$\begin{aligned}\frac{\partial L^{(i)}}{\partial W_{1\,nm}} &= \sum_{j=1}^{|V|} \frac{\partial L^{(i)}}{\partial y_j^{(i)}} \sum_{k=1}^{h_2} \frac{\partial y_j^{(i)}}{\partial a_{2k}^{(i)}} \sigma'(z_{2k}^{(i)}) \frac{\partial z_{2k}^{(i)}}{\partial W_{1\,nm}} \\ &= \sum_{j=1}^{|V|} \mathbb{E}_{ji}\, W_{2\,mj}\, A_{2\,mi}\,(1 - A_{2\,mi})\, A_{1\,ni}\end{aligned}$$

where we have used the derivative of the sigmoid function:

$$\sigma'(z_{2k}^{(i)}) = a_{2m}^{(i)}\,(1 - a_{2m}^{(i)}) = A_{2\,mi}\,(1 - A_{2\,mi})$$

and

$$\frac{\partial z_{2k}^{(i)}}{\partial W_{1\,nm}} = a_1^{(i)}\,\delta_{km} = A_{1\,ni}\,\delta_{km}, \qquad \frac{\partial y_j^{(i)}}{\partial a_{2k}^{(i)}} = W_{2\,kj} \tag{19}$$

Then, the derivative with respect to the total loss function is

$$\begin{aligned}\frac{\partial}{\partial W_{1\,nm}} \left(\sum_{i=1}^{n_{\text{batch}}} L^{(i)} \right) &= \sum_{i=1}^{n_{\text{batch}}} A_{1\,ni}\, \left[(W_2 \cdot \mathbb{E})_{mi}\, A_{2\,mi}\,(1 - A_{2\,mi}) \right] \\ &= \sum_{i=1}^{n_{\text{batch}}} A_{1\,ni}\, \Delta_{1\,mi}\end{aligned}$$

where Δ_1 is defined in equation (18). In matrix form,

$$\left(\frac{\partial L}{\partial W_1} \right) = A_1 \cdot \Delta_1^T \tag{20}$$

The derivative of the bias b_1 is obtained similarly

$$\frac{\partial L}{\partial b_{1n}} = \sum_{i=1}^{n_{\text{batch}}} \left[(W_2 \cdot \mathbb{E})_{mi}\, A_{2\,mi}\,(1 - A_{2\,mi}) \right] = \sum_{i=1}^{n_{\text{batch}}} \Delta_{1\,ni} \tag{21}$$

In Matlab or Octave, these two can be implemented as

```
% Gradient w.r.t W1
embed_to_hid_weights_gradient = embedding_layer_state ...
      * back_propagated_deriv_1';
% Gradient w.r.t b1
hid_bias_gradient = sum(back_propagated_deriv_1, 2);
```

where `back_propagated_deriv_1` is Δ_1. We also define the following quantity, the back propagated derivative of W_1, as follows:

$$\Delta_{2\,m}^{(i)} = \Delta_{2\,mi} \equiv W_1 \cdot \Delta_1 \tag{22}$$

which will be used in backpropagating from the embedding layer to input layer. In Matlab or Octave, it can be calculated as

```
back_propagated_deriv_2 = embed_to_hid_weights * back_propagated_deriv_1;
```

Backpropagation: Embedding to Input Layer

In this step, we compute the derivative of the loss function with respect to embedding weights C. Following similar steps above, the chain rule gives

$$\begin{aligned}
\frac{\partial L^{(i)}}{\partial C_{mn}} &= \sum_{j=1}^{|V|} \frac{\partial L^{(i)}}{\partial y_j^{(i)}} \sum_{s=1}^{h_1} \frac{\partial y_j^{(i)}}{\partial a_{1s}^{(i)}} \frac{\partial a_{1s}^{(i)}}{\partial C_{mn}} \\
&= \sum_{k=1}^{h_2} \sum_{s=1}^{h_1} \sum_{j=1}^{|V|} W_{2\,kj}\, \mathbb{E}_{ji}\, A_{2\,ki}\, (1 - A_{2\,ki})\, W_{1\,sk} \frac{\partial a_{1s}^{(i)}}{\partial C_{mn}} \\
&= \sum_{s=1}^{h_1} (W_1 \cdot \Delta_1)_{si} \frac{\partial a_{1s}^{(i)}}{\partial C_{mn}} \\
&= \sum_{s=1}^{h_1} \Delta_{2\,si} \frac{\partial a_{1s}^{(i)}}{\partial C_{mn}}
\end{aligned} \tag{23}$$

At this point, we need to compute the derivative of a_1 with respect to C to continue. This requires some care, since C multiplies D blocks of w's each with dimension h_1 inside a_1.

More precisely,

$$a_1^{(i)} = \begin{bmatrix} \left[C^T \cdot w_1^{(i)}\right] \\ \left[C^T \cdot w_2^{(i)}\right] \\ \vdots \\ \left[C^T \cdot w_D^{(i)}\right] \end{bmatrix}, \qquad C^T \cdot w_j^{(i)} \in \mathbb{R}^{h_1} \tag{24}$$

This means that in the above equation (23), the back propagated derivative Δ_2, which is of dimensions $Dh_1 \times n_{\text{batch}}$ has to multiply $a_{1s}^{(i)}$ block by block D times. Thus, we can recast the above equation (23) by a sum over these D blocks, while constraining the sum over index s to be confined to the right block's indices. Namely,

$$\frac{\partial L^{(i)}}{\partial C_{mn}} = \sum_{d=1}^{D} \sum_{s \in S_d} \Delta_{2\,si} \frac{\partial}{\partial C_{mn}} \left[C^T \cdot w_d^{(i)}\right]_s \tag{25}$$

where

$$S_d : \quad (d-1)h_1 \leq s \leq dh_1 \tag{26}$$

which does the constraining of Δ_2 to the right block. Then, we can define

$$\Delta_{2\,ij}^d \equiv \Delta_{2\,ij}, \qquad \text{provided :} \quad (d-1)h_1 \leq j \leq dh_1 \tag{27}$$

Moreover,

$$\frac{\partial}{\partial C_{mn}} \left[C^T \cdot w_d^{(i)}\right]_s = \delta_{sn} \, (w_d^{(i)})_m$$

which gives us

$$\frac{\partial L^{(i)}}{\partial C_{mn}} = \sum_{d=1}^{D} \left(w_d^{(i)}\right)_m \Delta_{2\,in}^d \tag{28}$$

This equation deserves some more explanation. First of all, notice that the term $(w_d^{(i)})_m$ is 1 if for the i^{th} example, the word at position d is m of the vocabulary; else it is zero. For instance, the d^{th} word in the `input_batch` for all the examples in the batch, looks like

$$\texttt{input\_batch[d, :]} = [2 \;\; 4 \;\; 123 \;\; \ldots]$$

Then,

$$(w_d^{(i)}) = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \qquad \mathbb{I}_{2,1} = 1, \ \mathbb{I}_{4,2} = 1, \ \mathbb{I}_{123,3} = 1, \dots \tag{29}$$

which is nothing but $\mathbb{I}$ defined above. The only difference is that, the locations of 1's will be different for each word position d, thus we define:

$$\mathbb{I}_{mi}^d \equiv (w_d^{(i)})_m \tag{30}$$

With this definition, we finally sum equation (28) over all the training examples in the batch, which gives us

$$\frac{\partial L}{\partial C_{mn}} = \sum_{d=1}^{D} \left(\mathbb{I}^d \cdot (\Delta_2^d)^T \right)_{mn} \tag{31}$$

In Matlab or Octave, this can be implemented as follows:

```
for w = 1:numwords
  word_embedding_weights_gradient = word_embedding_weights_gradient + ...
    expansion_matrix(:, input_batch(w, :)) * ...
    (back_propagated_deriv_2(1 + (w - 1) * numhid1 : w * numhid1, :)');
end
```

where `expansion_matrix(:, input_batch(w, :))` is $\mathbb{I}_{mi}^d = (w_d^{(i)})_m$.
`(back_propagated_deriv_2(1 + (w - 1) * numhid1 : w * numhid1, :)` is Δ_2^d which is Δ_2 subsetted using S_d.