Notes on Hierarchical Softmax

G.A. Jarrad

April 12, 2019

1 Introduction

In the context of natural language processing, it is common to maintain a list $\mathcal{V}$ of vocabulary words, which is usually extracted from one or more training corpora of text documents that have been suitably tokenised. Typically, the frequency of each token is observed, and the top $V = |\mathcal{V}|$ most common tokens form the vocabulary. The vocabulary $\mathcal{V}$ is typically ordered, either lexicographically or by descending frequency.

When it comes to representing a given word token, $w_i \in \mathcal{V}$, it can be uniquely indexed by its position i in the ordered list. Thus, for the example vocabulary $\mathcal{V} = \{\texttt{cat}, \texttt{dog}, \texttt{frog}, \texttt{mouse}\}$, we might choose $w_2 = \texttt{dog}$ and $w_4 = \texttt{mouse}$, etc. Alternatively, one can form a *1-of-V* or *one-hot* vector containing $V - 1$ zeros and a single one in the i-th element. For example, $\texttt{repr}(\texttt{cat}) = (1, 0, 0, 0)$ and $\texttt{repr}(\texttt{frog}) = (0, 0, 1, 0)$, etc. This latter representation is of fixed length V — the question is, can we form a shorter, but still unique, vector representation?

The answer is, yes we can. To do so, we first need to hierarchically partition the vocabulary $\mathcal{V}$ into a binary tree representation, which we shall call $\mathcal{T}_\mathcal{V}$. One way to do this is to simply partition $\mathcal{V}$ into two roughly equally sized lists, and then recursively partition each list until only single tokens remain in each innermost list. Alternatively, we can utilise the word frequency f_i of each token w_i to produce the probability, $p_i = f_i / \sum_{i'=1}^{V} f_{i'}$, that a word randomly sampled from the training corpora matches w_i. Using these probabilities, one can then partition $\mathcal{V}$ into nested groups, e.g. according to either Huffman or Fano coding (the latter method being simpler but producing less optimal code lengths). Figure 1.1 shows a (rather unbalanced) binary tree representation of the example vocabulary.

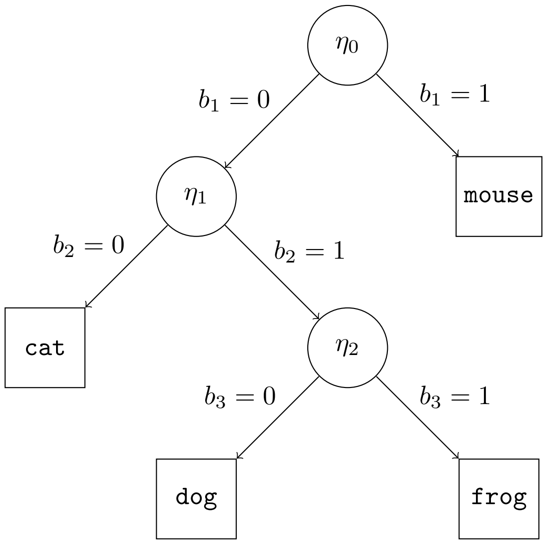

Figure 1.1: *An example of an unbalanced binary tree representation of a simple vocabulary.*

Note that the binary tree $\mathcal{T}_\mathcal{V}$ will contain V leaf nodes, corresponding to the vocabulary words, and $V - 1$ internal nodes, corresponding to binary decision points. Additionally, the tree will have a maximum of $L_\mathcal{V}$ levels, where $L_\mathcal{V} \geq 1 + \lceil \log_2 V \rceil$ (with equality holding if the tree is balanced). Thus, any path $(\eta_{i_1}, \eta_{i_2}, \ldots, \eta_{i_n})$ from the root node $\eta_{i_1} \equiv \eta_0$ to a leaf node η_{i_n} will pass through $n \leq L_\mathcal{V}$ nodes. This fact gives use scope for reducing the size of the unique word representation vectors.

It can be seen from Figure 1.1 that each path through $\mathcal{T}_\mathcal{V}$ from the root node towards any leaf node can be uniquely determined by specifying the sequence of binary branching decisions (0 for left, 1 for right) made along the path. Thus, in this example we now have $\texttt{repr}(\texttt{cat}) = (0,0)$, $\texttt{repr}(\texttt{dog}) = (0,1,0)$, $\texttt{repr}(\texttt{frog}) = (0,1,1)$ and $\texttt{repr}(\texttt{mouse}) = (1)$. These binary sequences collectively form the hierarchical representation of this vocabulary. Note that, for our simple example, the maximum size of the representation vector is three, whereas it would be two for a balanced tree, in contrast to the fixed size of four for the 1-of-V representation. In general, the size is reduced from V to about $\log_2 V$ by the binary partitioning.

2 Probability Model

A typical natural language problem is to determine which word $w \in \mathcal{V}$ is appropriate in some context $\mathbf{x}$. In probabilistic terms, this amounts to computing $p(w \mid \mathbf{x}) \doteq p(\texttt{repr}(w) \mid \mathbf{x})$. Here we assume that we have some binary tree representation $\mathcal{T}_\mathcal{V}$ of the vocabulary $\mathcal{V}$, such that $\texttt{repr}(w) = (b_1, b_2, \ldots, b_n)$. In order to decompose this probabilistic model, first reconsider Figure 1.1. Recalling that $\texttt{repr}(\texttt{frog}) = (0,1,1)$, we see that

$$\begin{aligned} p(\texttt{frog} \mid \mathbf{x}) &= p(b_1 = 0, b_2 = 1, b_3 = 1 \mid \mathbf{x}) \\ &= p(b_1 = 0 \mid \mathbf{x})\, p(b_2 = 1 \mid \mathbf{x}, b_1 = 0)\, p(b_3 = 1 \mid \mathbf{x}, b_1 = 0, b_2 = 1)\,. \end{aligned}$$

To make this more general, let $\vec{\rho}_{1:j} = (b_1, b_2, \cdots, b_j)$ for $j = 1, 2, \ldots, n$, where each binary decision b_j takes its corresponding value from the j-th element of $\texttt{repr}(w)$. Then we see that

$$p(w \mid \mathbf{x}) = \prod_{j=1}^{n} p(b_j \mid \mathbf{x}, \vec{\rho}_{1:j-1})\,, \tag{2.1}$$

where $\vec{\rho}_{1:n} = \texttt{repr}(w)$, and $\vec{\rho}_{1:0} = ()$ by definition.

Notionally, each subpath, specified by $\vec{\rho}_{1:j}$ for $j < n$, terminates at some unique, internal node η_k in $\mathcal{T}_\mathcal{V}$, indexed by $k = \iota(\vec{\rho}_{1:j})$. By definition, $\iota(()) = 0$, so that we always start from the root node η_0. Consequently, we allow each internal node η_k to have parameters $\boldsymbol{\theta}_k$. Furthermore, we suppose that composing the context $\mathbf{x}$ with node η_k results in local features $\phi_k(\mathbf{x}) = \phi(\mathbf{x}, \boldsymbol{\theta}_k)$. Hence, each factor of equation (2.1) becomes

$$p(b_j \mid \mathbf{x}, \vec{\rho}_{1:j-1}) \doteq p(b_j \mid \phi_k(\mathbf{x}), k = \iota(\vec{\rho}_{1:j-1}))\,, \tag{2.2}$$

for $j = 1, 2, \ldots, n$ with $n = |\texttt{repr}(w)|$.

2.1 Logistic Model

In a neural network framework, the probability terms comprising equation (2.1) would represent an output layer consisting of, for each internal node of the tree $\mathcal{T}_\mathcal{V}$, an activation function of an affine transformation of the input $\mathbf{x}$. Here we choose the logistic sigma activation function $\sigma(x) = 1/(1 + \exp(-x))$, and thus suppose that the local features of node η_k are given by $\phi_k(\mathbf{x}) = \alpha_k + \beta_k^T \mathbf{x}$. Hence, we take the probability of the j-th binary decision b_j along a specified path $\vec{\rho}_{1:n}$ to be

$$p(b_j \mid \mathbf{x}, \vec{\rho}_{1:j-1}) \doteq \sigma((2b_j - 1)(\alpha_k + \beta_k^T \mathbf{x}))\,, \tag{2.3}$$

from equation (2.2). We have made use of the fact that $1 - \sigma(x) = \sigma(-x)$.

Note that the scalar output of the neural network for node η_k must represent one of either $p(b_j = 0 \mid \cdots)$ or $p(b_j = 1 \mid \cdots)$. It is common to choose the former; however, we choose the latter here, since during training of the neural network one would optimise the parameters to minimise the 'distance' between $p(b_j \mid \cdots)$ and the known value of b_j. In other words, we desire $p(b_j = 1 \mid \cdots) \approx 1$ when $b_j = 1$, and $p(b_j = 1 \mid \cdots) \approx 0$ when $b_j = 0$. Hence, each hierarchical vector representation exactly specifies the desired output probabilities.

2.2 Optimal Path

In order to find the (or a) word w^* in the vocabulary that best matches the context $\mathbf{x}$, it would appear on the surface that we need to evaluate $p(w \mid \mathbf{x})$ for every $w \in \mathcal{V}$. However, we can do better. Note that:

- The probability factors $p(b_j \mid \mathbf{x}, \vec{\rho}_{1:j-1})$ are available at every internal node of the binary tree $\mathcal{T}_\mathcal{V}$, e.g. via one computational pass of the context $\mathbf{x}$ through a neural network;

- The tree is in fact a directed acyclic graph, which allows us to apply a graph search algorithm.

Consequently, we can apply the A^* search algorithm (albeit without a heuristic function).

To do so, we start with an empty *chart* (or *closed set*) $\mathcal{C}$ that will contain evaluated subpaths, and an empty *priority queue* (or *open set*) $\mathcal{Q}$ that will contain subpaths to be evaluated. We then insert the empty subpath () onto $\mathcal{Q}$ with its default score. Then repeatedly, while $\mathcal{Q}$ is not empty, we remove the best scoring subpath and: (i) if it terminates at a leaf node, return the word w; or else (ii) insert the subpath into $\mathcal{C}$, evaluate the two binary decisions ($b = 0$ and $b = 1$), and insert each new subpath with its corresponding score onto $\mathcal{Q}$. Pseudo-code for this method is given by Algorithm 1. Note that, based on the implementation, the chart $\mathcal{C}$ might or might not be required (depending upon whether or not it is used to reconstruct the optimal path via backtracking); here we show a chartless version.

Algorithm 1: Find word with highest probability

```
Input  : Binary vocabulary tree $\mathcal{T}_\mathcal{V}$; word context $\mathbf{x}$
Output: Word $w$; score $s$; path $\rho$
$\mathcal{Q} \leftarrow [\,]$;                                                  // Initialise queue
$\eta \leftarrow$ get_root($\mathcal{T}_\mathcal{V}$);                          // Obtain root node
$\nu \leftarrow$ make_node(node = $\eta$, path = (), score = 1);                // Construct score node
$Q \leftarrow$ insert_node($\mathcal{Q}, \nu$);                                 // Insert score node
while $|\mathcal{Q}| > 0$ do
    $(\mathcal{Q}, \nu) \leftarrow$ pop_node($\mathcal{Q}$);                    // Obtain highest-scoring node
    $s \leftarrow$ get_value($\nu$, score);                                     // Obtain score
    $\rho \leftarrow$ get_value($\nu$, path);                                   // Obtain path
    $\eta \leftarrow$ get_value($\nu$, node);                                   // Obtain tree node
    if is_leaf($\eta$) then
        $w \leftarrow$ get_word($\eta$);                                        // Obtain word
        return $(w, s, \rho)$;
    else
        $p \leftarrow$ calc_prob($\eta, \mathbf{x}$);                           // Compute probability of $b = 1$
        $s' \leftarrow [s(1-p), sp]$;                                           // Construct scores
        for $b \in \{0, 1\}$ do
            $\eta' \leftarrow$ get_child($\eta, b$);                            // Obtain child node
            $\rho' \leftarrow \rho \oplus (b)$;                                 // Construct child path
            $\nu' \leftarrow$ make_node(node = $\eta'$, path = $\rho'$, score = $s'[b]$);   // Construct score node
            $Q \leftarrow$ insert_node($\mathcal{Q}, \nu'$);                    // Insert score node
        end
    end
end
```

Note that this algorithm can easily be adapted to produce the top N most probable words. Also note that, in practice, it is better to sum (negative) log-probability scores rather than directly multiply probabilities, otherwise for large vocabulary trees we run into the problem of underflow of vanishingly small probabilities. There are in fact many more subtleties to implementing the A^* algorithm, including: how to build the priority queue; how to insert a node ranked by score; where to insert a node into a batch of equally-scored nodes; etc. These issues will not be addressed here.

References