Word Embedding Using Word2Vec

word2vec: CBOW, Skip-gram and Beyond

Ramaseshan Ramachandran

GOAL

- Learn a numeric (vector) representation for every word in the vocabulary
- The vectors should reflect semantic similarity, syntactic similarity, or both, between the words they represent
- Words appearing in similar contexts should map to nearby vectors

Maximize the dot product between words and the contexts they truly appear in, and minimize the dot product for wrong (word, context) pairs

CONTEXT WORDS AND CENTRAL WORD

Continuous Bag of Words (CBOW) Model – A central word is surrounded by context words. Given the context words identify the central word

- Wish you many | more | happy | returns | of | the | day

Skip Gram Model – Given the central word, identify the surrounding words

- Wish you many | more | happy | returns | of | the | day

CONTINUOUS BAG OF WORDS (CBOW)

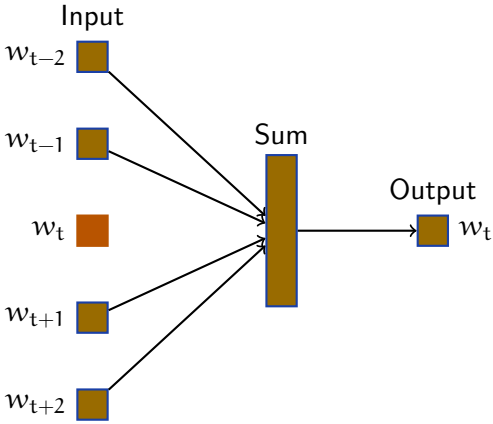


Figure: The CBOW architecture predicts the current word based on the context words of length n. Here the window size is 5

CBOW uses the sequence of words "Wish", "you", "a", "happy", "year" as a context and predicts or generates the central word "new"

- CBOW is used for learning the central word
- Maximize probability of word based on the word co-occurrences within a distance of n

SKIP GRAM MODEL

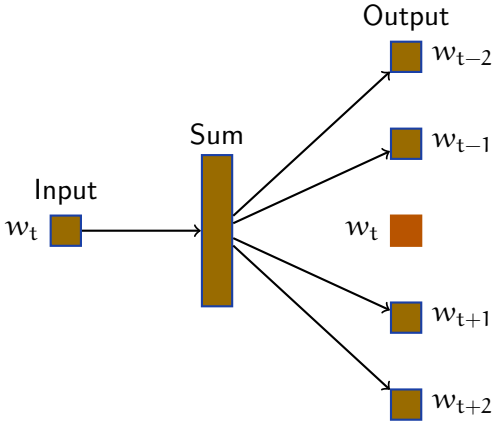


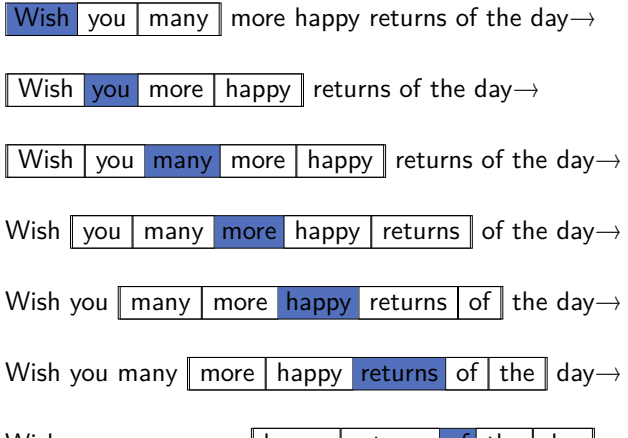
Figure: The SG architecture predicts the one context word at a time based on the center word. Here the window size is 5

SG uses the central word "new" and predicts the context words "Wish", "you", "a", "happy", "year"

- SG is used to learn the context words given the central word
- Maximize probability of word based on the word co-occurrences within a distance of $[-n, +n]$ from the center word

SOURCE PREPARATION FOR TRAINING

Source Text



Training Samples

$(wish, you)$
$(wish, many)$
$(you, Wish)$
$(you, more), (you, happy)$
$(many, Wish), (many, you)$
$(many, more), (many, happy)$
$(more, many), (more, you)$
$(more, happy), (more, returns)$
$(happy, many), (happy, more)$
$(happy, returns), (happy, of)$
$(returns, more), (returns, happy)$
$(returns, of), (returns, the)$
$(of, happy), (of, returns)$
$(of, the), (of, day)$

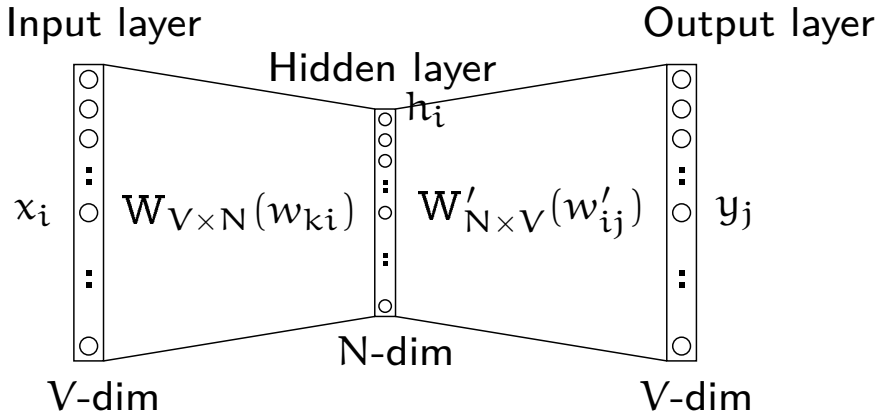


Figure: A CBOW model with only one word as input[1]. The layers are fully connected

Here the W and W' are learned

INPUT LAYER

$$t^{aback} = \begin{pmatrix} 0 \\ 1 \\ \dots \\ 0 \\ \dots \\ 0 \\ 0 \end{pmatrix} \dots t^{zoom} = \begin{pmatrix} 0 \\ 0 \\ \dots \\ 0 \\ \dots \\ 1 \\ 0 \end{pmatrix} t^{zucchini} = \begin{pmatrix} 0 \\ 0 \\ \dots \\ 0 \\ \dots \\ 0 \\ 1 \end{pmatrix}$$

$$x_k = 1 \text{ and } x'_k = 0, \forall k' \neq k$$

HIDDEN LAYER

This neural network is fully connected. Input to the network is a one-hot vector. v_w^T is the $|V|$-dimensional vector representation of the word presented as input[1]

$$h = W^T X = v_{w_I}^T \tag{1}$$

Now v_{w_I}, a row of the matrix W, is the vector representation of the input word w_I. Since the input is one-hot with $x_k = 1$, h is simply a copy of the k-th row of W. In the same way, we get a score u_j for each output word

$$u_j = v_{w_j}'^T h = v_{w_j}'^T v_{w_I} \tag{2}$$

where v_{w_I} is the vector representation of the input word w_I and v'_{w_j} is the j^{th} column of W'

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space, 2013.

OUTPUT LAYER

At the output layer, we apply the softmax to get the posterior distribution of the word(s). It is obtained by,

$$p(w_j|w_I) = y_j \tag{3}$$

where y_j is the output of the j^{th} unit in the output layer

$$y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})} \tag{4}$$

$$= \frac{\exp(v'^{T}_{w_j} v_{w_I})}{\sum_{j'=1}^{V} \exp((v'^{T}_{w_{j'}}) v_{w_I})} \tag{5}$$

where v_w, andv'_w are the input vector (word vector) and output vector (feature vector) representations of w, respectively.

UPDATE WEIGHTS - HIDDEN-OUTPUT LAYERS I

Through backpropagation,[2] the learning/training objective is to maximize the log-probability of the target word w_o given the input word w_I:

$$\max \log p(w_o \mid w_I) = \max \log y_{j^*} \quad \text{(maximize log-likelihood)} \tag{6}$$

where y_{j^*} is the softmax probability assigned to the correct output word $w_o(j^*$ is the index at the output layer), computed from the logits $u = [u_1, u_2, \ldots, u_V]$ as:

$$y_{j^*} = \frac{\exp(u_{j^*})}{\sum_{j=1}^{V} \exp(u_j)} \tag{7}$$

To train the model, we minimize the cross-entropy loss function[3] which measures the

difference between the true distribution (a one-hot vector with $p_{j^*} = 1$) and the predicted softmax distribution y:

$$E = -\log y_{j^*} \tag{8}$$

$$= -u_{j^*} + \log \sum_{j=1}^{V} \exp(u_j) \tag{9}$$

where u_j is the logit (pre-softmax activation) corresponding to output word j. This loss represents a special case of cross-entropy between the one-hot true distribution and the predicted softmax probabilities.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. Nature, 323(6088):533536, 1986.

[3] Cross-entropy: $H(p, q) = -\sum_j p_j \log q_j$

UPDATE WEIGHTS (HIDDEN-OUTPUT LAYER) - MINIMIZATION OF E

To minimize the loss E, take the partial derivative of E with respect to the logit u_j of output unit j:

$$\frac{\partial E}{\partial u_j} = y_j - t_j = e_j \tag{10}$$

where y_j is the predicted softmax probability, t_j is the target label (one-hot encoded), and e_j is the prediction error for unit j.

Next, taking the partial derivative of E with respect to the weight w'_{ij} connecting hidden unit i to output unit j:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w'_{ij}} = e_j \cdot h_i \tag{11}$$

where h_i is the activation of hidden unit i.

Using gradient descent, the weight update rule is:

$$w'^{(new)}_{ij} = w'^{(old)}_{ij} - \eta e_j h_i \tag{12}$$

or in vector form:

$$v'^{(new)}_{w_j} = v'^{(old)}_{w_j} - \eta e_j \mathbf{h} \tag{13}$$

where v'_{w_j} is the output embedding vector (weights connecting hidden units to output unit j), $\mathbf{h}$ is the hidden layer activation vector, and η is the learning rate.

UPDATE INPUT-TO-HIDDEN WEIGHTS

Taking the derivative of the loss E with respect to the hidden unit activation h_i:

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} = \sum_{j=1}^{V} e_j \cdot w'_{ij} = EH_i \tag{14}$$

where $e_j = \frac{\partial E}{\partial u_j} = y_j - t_j$ is the error at output unit j, and w'_{ij} is the weight from hidden unit i to output unit j.
Taking the derivative of E with respect to an input-to-hidden weight w_{ki}:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} = EH_i \cdot x_k \tag{15}$$

where x_k is the input feature corresponding to weight w_{ki}.
The input embedding vector corresponding to input word w_i, denoted v_{w_i}, is updated as:

$$v_{w_i}^{(new)} = v_{w_i}^{(old)} - \eta EH^T \tag{16}$$

where η is the learning rate, and EH is the vector of propagated errors for hidden units.

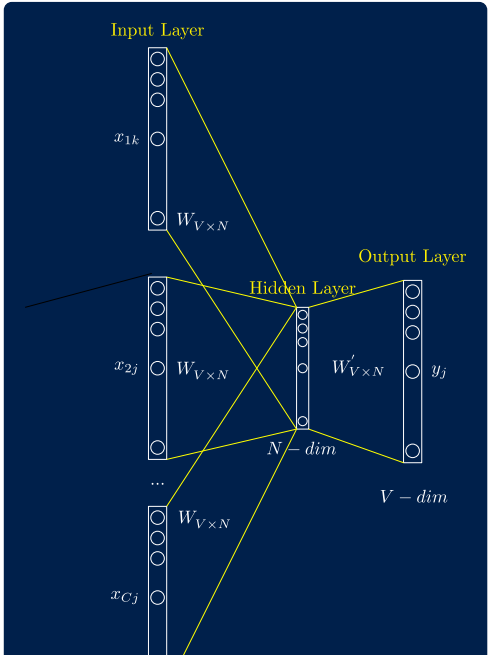
INSIGHTS ON WEIGHT UPDATES

- The prediction errors e_j at the output layer propagate back to the hidden layer as weighted sums $\mathbb{E}\mathrm{H}_i$, influencing updates to both output and input embeddings
- Changes in the input word vector v_{w_i} are driven by this propagated error signal through the hidden layer
- Model parameters are iteratively updated during training until the system converges to a stable state
- The input-to-hidden weight matrix (rows corresponding to v_{w_i}) effectively stores continuous feature representations (embeddings) of words in the vocabulary V

Word2Vec Demo

CBOW MODEL FOR MULTIPLE WORDS

- C is the number of context words
- V is the size of the vocabulary
- h_i receives average of the vectors of the input context words
- Output vector v'_{wj} is the column vector in the W' representing relationship between the context words and the target word
- Softmax is used for the output layer probability distribution for the target word



INPUT-HIDDEN WEIGHT VECTORS AND LOSS FUNCTION

The hidden layer representation $\mathbf{h}$ is computed as the average of the input context word vectors:

$$\mathbf{h} = \frac{1}{C}\sum_{c=1}^{C} \mathbf{v}_{w_{I,c}} \tag{17}$$

$$= \frac{1}{C}(\mathbf{v}_{w_1} + \mathbf{v}_{w_2} + \cdots + \mathbf{v}_{w_C}) \tag{18}$$

The logits u_j for output word w_j are computed by the dot product of the output embedding $\mathbf{v}'_{w_j}$ and $\mathbf{h}$:

$$u_j = \mathbf{v}'^{\top}_{w_j}\mathbf{h} \tag{19}$$

The cross-entropy loss to minimize for the target output word w_O is:

$$E = -\log p(w_O \mid w_{I,1}, \ldots, w_{I,C})$$

$$= -\mathbf{v}'^{\top}_{w_O}\mathbf{h} + \log \sum_{j=1}^{V} \exp\left(\mathbf{v}'^{\top}_{w_j}\mathbf{h}\right) \tag{20}$$

The output embedding vectors (hidden-to-output weights) update rule remains:

$$v'^{(new)}_{w_j} = v'^{(old)}_{w_j} - \eta e_j h, \quad j = 1, 2, \ldots, V$$

where $e_j = y_j - t_j$ is the prediction error.
The context input word vectors update as:

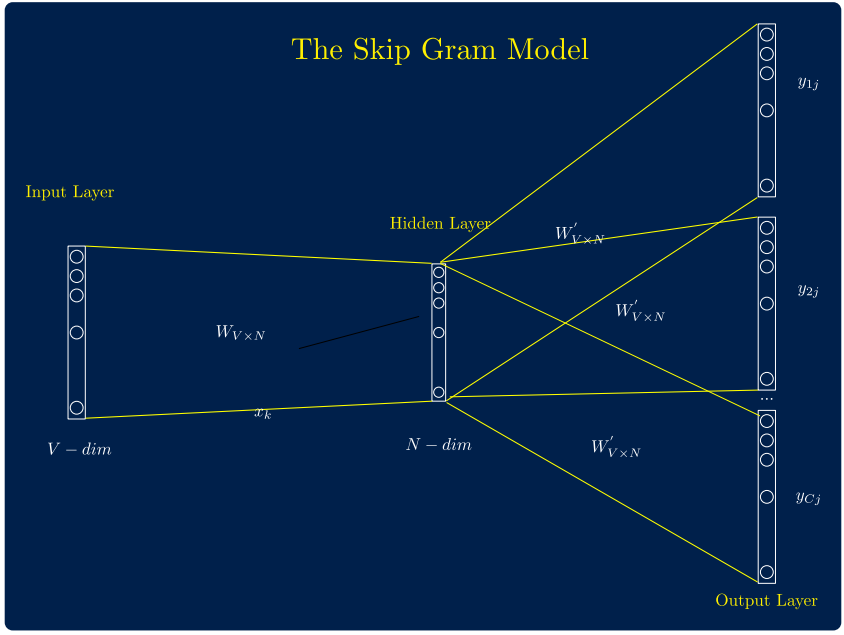
$$v^{(new)}_{w_{I,c}} = v^{(old)}_{w_{I,c}} - \frac{\eta}{C} EH^T, \quad c = 1, 2, \ldots, C \tag{21}$$

where EH is the backpropagated error vector at the hidden layer and η is the learning rate.

WHAT DOES IT LEARN?

- Learns distributed representations of words as continuous vectors in a semantic space.
- The learned vectors capture many linguistic regularities and semantic patterns
- Words that appear in similar contexts tend to have similar vector representations
- We measure similarity between word vectors using metrics like cosine similarity
- Does the model handle morphological variants? For example, are vectors for *run*, *running*, *ran* close?
  - He runs a half-marathon.
  - He ran a half-marathon.
  - He is running a half-marathon.
- What about semantically related words like *car*, *cars*, *automobile*?
- What about synonymous words like *awesome*, *fantastic*, *great*?

SKIP-GRAM MODEL



SKIP-GRAM MODEL - FORWARD AND BACKPROPAGATION UPDATE OF WEIGHTS

$$\text{Hidden layer } h = W^T x \tag{22}$$

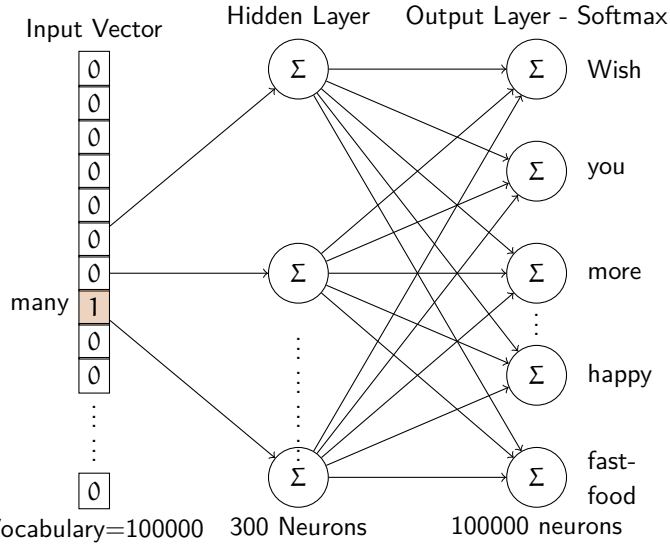
$$\text{Output neuron logits } u_{c,j} = v'^T_{w_j} h, \quad c = 1, 2, \ldots, C \tag{23}$$

$$\text{Cross-entropy loss } E = -\sum_{c=1}^{C} u^*_{c,j} + C \cdot \log \sum_{j'=1}^{|V|} \exp(u_{j'}) \tag{24}$$

$$\text{Output weights update } w'^{new}_{ij} = w'^{old}_{ij} - \eta \cdot e_j \cdot h_i \tag{25}$$

$$\text{Input weights update } w^{new}_{ij} = w^{old}_{ij} - \eta \cdot E_j \cdot x_i \tag{26}$$

NEURAL NETWORK ARCHITECTURE - A SAMPLE



FORWARD PASS

```
def forward_pass(target_word_idx, W1, W2, vocab_size):
    x = OHV(target_word_idx, vocab_size)
    h = np.dot(x, W1)
    u = np.dot(h, W2)
    y_pred = softmax(u)
    return x, h, y_pred
```

BACK PROPAGATION

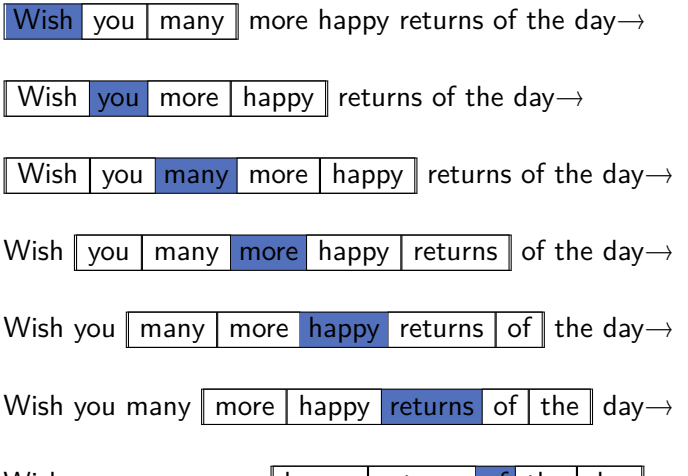
```
def back_propagation(x, h, y_pred, context_word_idx,
                     input_weights, context_weights,
                     learning_rate):
    y_true = OHV(context_word_idx, vocab_size)
    e = y_pred - y_true
    dW2 = np.outer(h, e)
    dW1 = np.outer(x, np.dot(W2, e))
    input_weights -= learning_rate * dW1
```

PROBLEM WITH THE SIZE OF THE NETWORK

- All weights $(\mathrm{output} \rightarrow \mathrm{hidden})$ and $(\mathrm{hidden} \rightarrow \mathrm{input})$ are adjusted by taking a training sample so that the prediction cycle minimizes the loss function
- This amounts to updating all the weights in the neural network - amounts to several million weights for a network which has input neurons of size $|V| = 1\mathrm{M}$, and hidden unit size as 300
- In addition, we should consider the several million training samples pairs

SOURCE PREPARATION FOR TRAINING

Source Text



Training Samples

(wish, you)
(wish, many)
(you, Wish)
(you, more), (you, happy)
(many, Wish), (many, you)
(many, more), (many, happy)
(more, many), (more, you)
(more, happy), (more, returns)
(happy, many), (happy, more)
(happy, returns), (happy, of)
(returns, more), (returns, happ
(returns, of), (returns, the)
(of, happy), (of, returns)
(of, the), (of, day)

SUB-SAMPLING

- The words $(\mathrm{of}, \mathrm{the})$ in the pairs $(\mathrm{of}, \mathrm{happy}), (\mathrm{returns}, \mathrm{the})$ do not give much information about the words happy and returns, respectively. Similarly, some pairs reappear with the order of the words switched.
- Some words can be randomly removed from the training text based on their frequencies
- Very rare words appearing as context words can also be discarded, as they provide little reliable contextual information about the central word
- Sub-sampling discards or down-weights frequent words (like “the”, “is”) to focus training on less frequent but more informative words

SUB-SAMPLING IN WORD2VEC.C-GOOGLE

To counter the imbalance between the rare and frequent words, a simple sub-sampling approach is used: each word w_i in the training set is discarded with probability computed by the formula [2]

$$P(w_i) = \max\left(0, 1 - \sqrt{\frac{t}{f(w_i)}}\right) \tag{27}$$

where $t = sample$ is the threshold parameter controlling subsampling aggressiveness.

Here is the code for sub-sampling used by word2vec.c that randomly removes a word from the sample using an algorithm similar to Linear Congruential generator (LCG) algorithm.

```
if (word == 0) break; //word=index of the current word in the vocabulary
// The subsampling randomly discards frequent
//words while keeping the ranking same
if (sample > 0) { //sample is set to 0.001 (default).
                  //If set to 0, then subsampling is disabled
  real ran = (sqrt(vocab[word].cn/(sample * train_words))+1)*
             (sample * train_words)/vocab[word].cn;
  next_random = next_random*(unsigned long long)25214903917+11;
  //7*443*739*11003=25214903917
  if (ran < (next_random & 0xFFFF) / (real)65536) continue;
}
```

$$\text{let} \quad f(w) = \frac{vocab[word].cn}{train\_words} \quad \text{and} \quad ran = \left(\sqrt{\frac{f(w)}{t}} + 1\right) \times \frac{t}{f(w)} \tag{28}$$

where $vocab[word].cn$ is the count of the word, $train\_words$ is the total number of training tokens, and t is the *sample* threshold. The keep/discard decision compares ran with a generated random value $random$: if $ran < random$, the word is **discarded**; otherwise it is kept

- This calculates a random probability ran that is used to decide whether a word should be subsampled (discarded) or not.
- *sample*: Usually 10^{-5} -keeping it very small makes the algorithm more aggressive in discarding frequent words.
- The term $vocab[word].cn/(sample * train\_words)$ measures the relative frequency of the word compared to the sampling threshold.
- The sqrt() function reduces the impact of very frequent words, allowing words that are just above the threshold to be less likely discarded.
- The entire expression generates a value ran that increases as the word becomes less frequent, meaning that less frequent words are more likely to be retained.

SUB-SAMPLING - EXAMPLE

Let the word **the** appears 7% of the time in the corpus ($f(the) = 0.07$) and the sub-sampling threshold $t = 0.001$. Then

$$P(\text{the}) = \left(\sqrt{\frac{0.07}{0.001}} + 1\right) \times \frac{0.001}{0.07} = 0.134$$

The chance of keeping the word is 13.4%. Let us assume for word **adiabatic** $f(x) = 0.005$. Then

$$P(\text{adiabatic}) = \left(\sqrt{\frac{0.005}{0.001}} + 1\right) \times \frac{0.001}{0.005} = .647$$

The chance of keeping the word is 65%.

- Frequent words tend to appear in many contexts, which dilutes useful contextual signals
- Subsampling reduces training examples involving those frequent words, thereby focusing training on more informative word relationships
- It accelerates training and often improves the quality of embeddings

For classification over multiple classes, softmax converts raw scores into a probability distribution: if there are K classes, it maps the outputs into $[0,1]$ so that they sum to one. It is the standard choice for the output layer and is a normalized exponential function

$$P(C_k|x) = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \quad k = 1, \ldots, K \tag{29}$$

HIERARCHICAL SOFTMAX

- Has a flat hierarchy with a probability value for every output node of depth $= 1$
- Normalized over the probabilities of all |V| words
- Error correction happens for every output$\rightarrow$hidden units
- Huge costs if the vocabulary size $|V|$ is of the order of several thousands
- Decompose the flat hierarchy into a binary tree
- Form a hierarchical description of a word as a sequence of $O(\log_2|V|)$ decisions and there by reducing the computing complexity of Softmax - $O(|V|) \rightarrow O(\log_2(|V|))$
- Lay the words in a tree-based hierarchy - words as leaves
- Binary tree with $|V|-1$ internal nodes for left (0) and right (1) traversal
- Every leaf represents the probability of one word

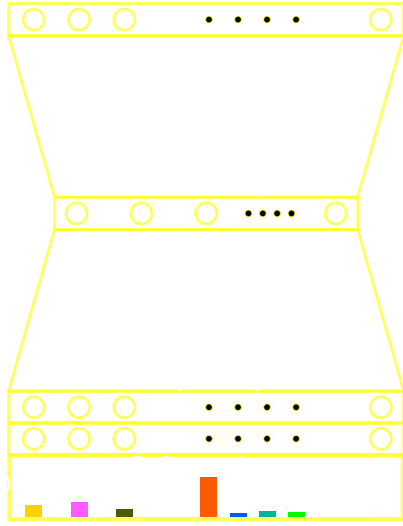
HIERARCHICAL SOFTMAX - 2

- Path length of a balanced Tree is $\log_2(|V|)$. If the $|V| = 1 \quad million \quad words$, then the path length $= 19.9$ bits/word
- Constructing a Huffman-encoded tree gives frequent words short unique binary codes
- Learn to take these probabilistic decisions instead of directly predicting each word's probability [3]
- Every intermediate node denotes the relative probabilities of its child nodes
- The path to reach every leaf (word) is unique
- H-Softmax in many cases increases the prediction speed by more than $50\times$

Softmax is a normalized exponential function

$$P(C_k|x) = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \quad k = 1, \ldots, K \tag{30}$$

- Has a flat hierarchy with a probability value for every o/p node - depth $= 1$
- Normalized over the probabilities of all $|V|$ words
- Error correction happens for every output$\rightarrow$hidden units
- Huge costs if the vocabulary size $|V|$ is of the order of several thousands



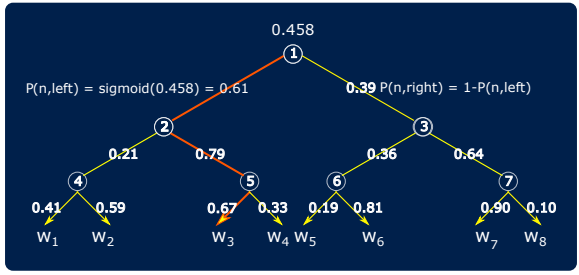
BALANCED BINARY TREE

- Move the words into a binary tree - depth depends on the vocabulary
- The path to any word in the vocabulary is known
- Traverse through the binary tree to reach any word
- At every step, make a binary decision to reach the word
- The length to reach any word in a balanced tree is $\log_2(|V|)$
- Words could be arranged using
  - random order
  - IS-A relationship
  - TF-IDF frequency

EFFECT OF WORD ARRANGEMENTS IN BALANCED BINARY TREE

Arrangement	**Performance Impact**	**Use Case**
Random Order	Uniform depth, no semantic or frequency bias	Baseline; when vocabulary distribution or semantics are unknown or uniform
IS-A Relationship	Semantic clustering enables faster semantic queries	Useful for semantic search, ontology-based NLP tasks
TF-IDF Frequency	Faster access to frequent or important words on average	Information retrieval, keyword extraction, ranking tasks

BALANCED BINARY TREE WITH 8 WORDS



$P(w_i) = \prod_{j \in N_L} P(n(w,j))$,where N_L is the list of nodes to reach the word, $n(w,j)$ indexes the nodes one-by-one along the root-to-leaf path for word w and $\sum_{i=1}^{V} P(w_i) = 1$

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
$P(w_i)$	0.05	0.08	0.32	0.16	0.04	0.11	0.22	0.02

Thus the hierarchical Softmax is a well defined multinomial distribution among all words

HIERARCHICAL SOFTMAX - ADVANTAGES

- Decomposes the flat hierarchy into a binary tree
- The path to reach every leaf (word) is unique
- Lays the words in a tree-based hierarchy - words as leaves
- Binary tree with $|V|-1$ nodes for left and right traversal
- Every intermediate node denotes the relative probabilities of its child nodes
- Every leaf represents the probability of the word

HIERARCHICAL SOFTMAX - ADVANTAGES

- Each node is indexed by a bit vector corresponding to the path from the root to the node
- The probability(ies) corresponding to the context word(s) is(are) computed. Rest are ignored - hence a huge saving on the computation of softmax at the output layer
- Normalized values for the words are calculated without finding the probability for every word
- The entire vocabulary is partitioned into classes
- ANN learns to take these probabilistic decisions instead of directly predicting each word's probability[5]

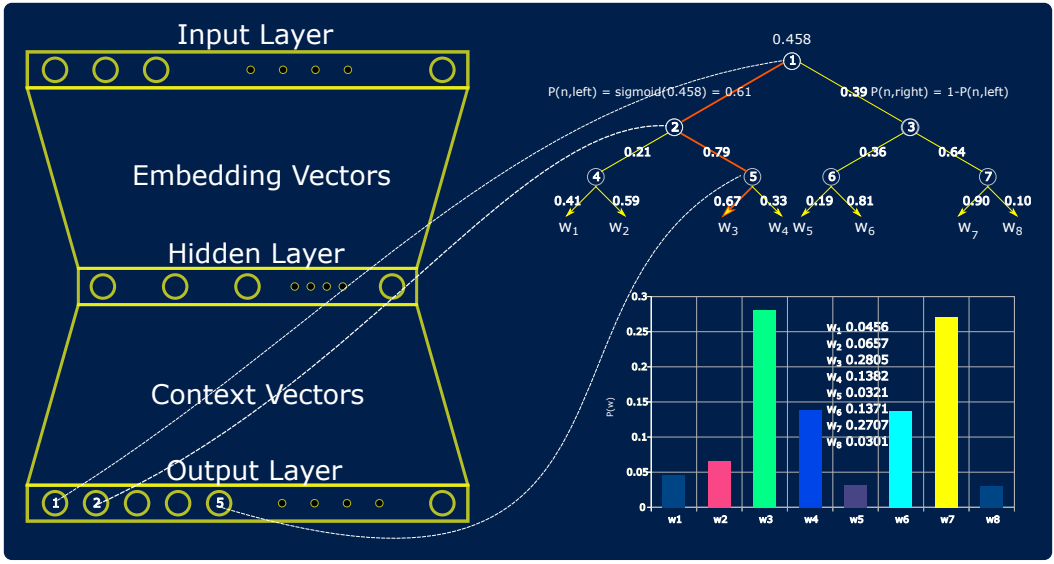
[5] Yoshua Bengio et al. "A Neural Probabilistic Model" In: J.Mach.Learn Res.3 (Mar.2003), pp.1137-1155. issn: 1532-4435

HIERARCHICAL SOFTMAX - ADVANTAGES

- Forms a hierarchical description of a word as a sequence of $O(\log_2|V|)$ decisions
- Reduces the computing complexity of Softmax - $O(|V|) \rightarrow O(\log_2(|V|))$
- Path length of a balanced Tree is $\log_2(|V|)$. If the $|V| = 1 \quad million \quad words$, then the path length $= 19.9$ bits/word
- A balanced binary tree should provide an exponential speed-up, on the order of $\frac{|V|}{\log_2(|V|)}$
- Constructing a Huffman-encoded tree gives frequent words short unique binary codes
- H-Softmax in many cases increases the prediction speed by more than $50\times$

Note: Hierarchical softmax significantly reduces the computational complexity and it doesn't completely eliminate the need to compute probabilities for all words

WORD2VEC WITH HIERARCHICAL SOFTMAX



UPDATING WEIGHTS - 1/3

Let $L(w)$ be the number of nodes to traverse to the word from the root and $n(w,i)$ is the i^{th} node on this path and the associated vector in the context matrix is $v_{n(w,i)}$. $ch(n)$ is the child node [4][3][2][5]. Then the probability of word is

$$P(w|w_i) = \prod_{j=1}^{L(w)-1} \sigma\left([n(w,j+1) = ch(n(w,j))] \cdot v'^{T}_{n(w,j)} h\right)$$

$$= \prod_{j=1}^{L(w)-1} \sigma\left([n(w,j+1) = ch(n(w,j))] \cdot v'^{T}_{n(w,j)} v_{w_i}\right) \quad (31)$$

$$\text{as } h = v_{w_i} \quad (32)$$

where the indicator function $[\![x]\!] = [n(w,j+1) = ch(n(w,j))]$ is

$$[\![x]\!] = \begin{cases} 1, & \text{if } x \text{ is true} \\ -1, & \text{otherwise} \end{cases}$$

and $\sigma(.)$ is the sigmoid function.

If the child node $ch(n(w,j))$ is the left child of the parent $n(w,j)$, the indicator is 1; if the path goes to the right child, the indicator is -1

UPDATING WEIGHTS - 2/3

Since the sigmoid function satisfies $\sigma(z)+\sigma(-z)=1$ because $1-\sigma(z)=\sigma(-z)$, the probabilities at each binary node sum to 1.

Example: The probability of a word w given context w_i is a product along nodes in the path:

$$P(w|w_i)=\prod_{j=1}^{L(w)-1}\sigma\left([\![x_j]\!]\cdot v'^{T}_{n(w,j)}v_{w_i}\right),$$

where

$$[\![x_j]\!]=\begin{cases}1, & \text{if } n(w,j+1) \text{ is the left child of } n(w,j)\\ -1, & \text{otherwise}\end{cases}$$

Training minimizes the negative log-likelihood:

$$E=-\sum_{j=1}^{L(w)-1}\log\sigma\left([\![x_j]\!]\cdot u_j\right),\quad \text{where } u_j=v'^{T}_{n(w,j)}h,$$

with $v'^{T}_{n(w,j)}$ as the context vector for node $n(w,j)$ and h as the input vector for w_i.

$$\frac{\partial E}{\partial u_j} = (\sigma([\![x_j]\!]u_j) - 1)[\![x_j]\!], \quad \text{where } [\![x_j]\!] = \begin{cases} 1, & \text{if } n(w, j+1) \text{ is left child} \\ -1, & \text{otherwise} \end{cases} \tag{33}$$

$$\frac{\partial E}{\partial v'_j} = \frac{\partial E}{\partial u_j}\frac{\partial u_j}{\partial v'_j} = (\sigma([\![x_j]\!]u_j) - 1)[\![x_j]\!] \cdot h = \delta v'_j \tag{34}$$

$$v'^{new}_j = v'^{old}_j - \eta \delta v'_j \tag{35}$$

$$\frac{\partial E}{\partial h} = \sum_{j=1}^{L(w)-1} (\sigma([\![x_j]\!]u_j) - 1)[\![x_j]\!] \cdot v'_j \tag{36}$$

$$v^{new}_{w_i} = v^{old}_{w_i} - \eta \frac{\partial E}{\partial h} \tag{37}$$

NEGATIVE SAMPLING

- Network size grows with vocabulary size V; every weight updated per training input, even in hierarchical models
- Updating all weights in fully connected networks is computationally expensive
- Can we update only a small fraction of weights?
- Select a small set of *negative* words
- Negative samples produce zero output; positive samples retain their values
- During backpropagation, only weights linked to negative and positive words are updated; others remain unchanged
- This drastically reduces computation cost ◡

CHOOSING NEGATIVE SAMPLES IN SKIP-GRAM - I

- Negative samples are drawn from a noise distribution $P_n(w)$, often the unigram distribution raised to the 3/4 power:

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{n} f(w_j)^{\frac{3}{4}}} \tag{38}$$

where $f(w_i)$ is the frequency of word w_i in the corpus. Using the frequency of words directly for negative sampling would bias the model towards frequent words. Negative sampling does favor more frequent words, but it uses a smoothed frequency distribution with the $\frac{3}{4}$ power to ensure a good balance between frequent and rare words.

CHOOSING NEGATIVE SAMPLES IN SKIP-GRAM - II

- This helps balance frequent and rare words and avoids domination by very frequent words
- **Hard negatives** are sampled by choosing words close in embedding space but not actual contexts, improving model discrimination.
- Avoid **false negatives** (synonyms or related words wrongly treated as negatives) to maintain embedding quality.
- Sophisticated methods adaptively select negatives during training based on model state or embedding similarity.

NEGATIVE SAMPLING - EXAMPLE AND LOSS FUNCTION

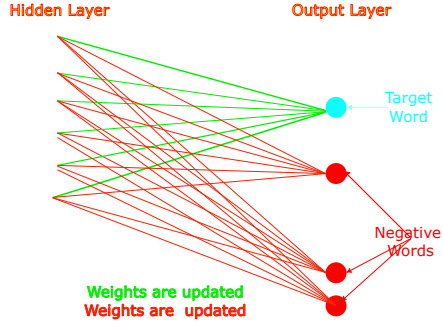
- Update only a small subset of weights instead of the full vocabulary
- Consider the word pair chalk and blackboard as a positive example
  - Goal: output layer activation for blackboard should be close to 1
  - Traditionally, all other words in the vocabulary have output 0
  - With negative sampling, randomly select a small set of negative samples, e.g., *coffee, car, elephant, tower*
  - Instead of updating all other words, only update these negatives pushing their outputs toward 0, while keeping the rest unchanged
- The logistic regression loss function used is:

$$E = -\left(\log\left(\sigma(z_{pos})\right) + \sum_{i=1}^{k} \log\left(1 - \sigma(z_{neg_i})\right)\right) \tag{39}$$

where z_{pos} is the input score for the positive sample, and z_{neg_i} for the i-th negative sample; $\sigma(\cdot)$ is the sigmoid function.

ADVANTAGES OF NEGATIVE SAMPLING

- Faster training time, especially on large vocabularies, by updating only a subset of weights per step, enabling scalability
- Improved embedding quality by focusing on distinguishing positive samples from selected negatives
- Reduced computational and memory overhead during training due to fewer gradient updates



NEGATIVE SAMPLING IN SKIP-GRAM MODEL

- **Computational Efficiency:** Updates are performed only on a small subset of negative samples plus positive samples, reducing computation drastically compared to full softmax over the entire vocabulary.
- **Training Simplification:** Converts the multi-class classification problem into multiple binary classification tasks using logistic regression, easing gradient computations.
- **Improved Model Quality:** Enables learning fine distinctions by focusing on distinguishing true context words from carefully chosen negative samples (e.g., "hard negatives").
- **Handling Class Imbalance:** Balances positive and negative samples during training, preventing bias towards frequent words.
- **Hyperparameter Sensitivity:** Number of negative samples K influences model quality and training speed.

RESULTS - SIMILAR WORDS FOR Virus

Vocab size	Words in the corpus
637722	222502540

Word	Similarity
virus,	0.889620
viral	0.785719
(herpesvirus)	0.764385
avirus	0.759567
fluav)	0.757418
polio-virus	0.724740
⋮	⋮
(vsv;	0.723436
(denv-2)	0.722825
(cowpox)	0.717185
⋮	⋮

ANALOGIES - *virus : covid :: ransomware : ?*

Word	Distance
scams	0.605626
cybercrime	0.592376
denial-of-service	0.590397
⋮	⋮
privacy	0.589996
cyberattack	0.586179
frauds	0.585077
wartime	0.584866
(ddos)	0.582940
⋮	⋮

BUILDING WORD2VEC FROM SCRATCH

1. **Data Preparation**
   - Preprocess the text corpus
   - Create a vocabulary of unique words.
   - Construct a reverse lookup to map words to their indices
2. **Generate Training Data**
   - Initialize a sliding window of size w
   - For each word in the corpus:
     - Extract the center word and context words
     - Add the pair to the training data
3. **Implement Skip-Gram Model**
   - Define the following layers:
     - *Input layer*: One-hot encoded center word
     - *Hidden layer*: Word embeddings
     - *Softmax layer:* predict context words
4. **Training**
   - Initialize the model parameters
   - For each epoch:
     - For every training sample, compute the loss
     - Update the model parameters
5. **Negative Sampling**
   - Use negative sampling to speed up the training
     - Optimize the model to distinguish context words from negative words
6. **Training Loop**
   - Implement a training loop to iterate through the following steps:
     - Compute the gradients of the model parameters
     - Update the model parameters based on the gradients
7. **Save Word Embeddings**
   - Save the word embeddings to a file for future use
8. **Evaluation**
   - Evaluate the word embeddings using tasks like word similarity, word analogy, or text classification.
9. **Optimization and Performance**
   - Optimize your code for performance, as training Word2Vec can be computationally intensive
     - Use multi-threading or distributed computing if needed

COMPLEXITIES OF THESE MODELS

Task	Complexity
Skip-gram (Hierarchical Softmax)	$O(T \cdot N \cdot C \cdot \log V)$
Skip-gram (Negative Sampling)	$O(T \cdot N \cdot C \cdot k)$
CBOW (Hierarchical Softmax)	$O(T \cdot N \cdot C \cdot \log V)$
CBOW (Negative Sampling)	$O(T \cdot N \cdot C \cdot k)$
Cosine Similarity	$O(N)$
Analogy Task	$O(V \cdot N)$

where

T is total number of words in the corpus

V is vocabulary size

N is embedding size

C is context window size

k is number of negative samples

LIMITATIONS

- Separate training is required for phrases
- Embeddings are learned from a small local window surrounding each word - antonyms like *good* and *bad* end up with almost the same embedding
- Does not address polysemy - one vector per word, regardless of sense
- Does not use global co-occurrence statistics (GloVe was designed to address this)

SOURCE CODE FOR WORD2VEC

GitHub link to W2V

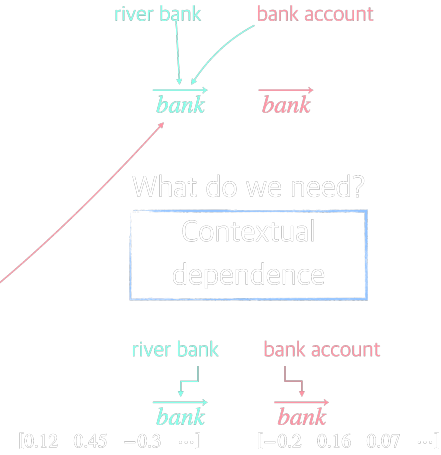
- **Word2Vec** is a neural network model that learns word embeddings.
- Words are mapped to high-dimensional vectors where semantically similar words have similar vector representations.
- The word embeddings are pre-trained and can be used as features in downstream tasks.
- Embeddings transform sparse word representations (one-hot vectors) into dense, lower-dimensional continuous vectors.

$$\text{Embedding: } \mathbf{w} \in \mathbb{R}^{\mathrm{d}}$$

- Handles contextual similarity

VECTOR REPRESENTATION OF WORDS

- Word embeddings are the most fundamental to NLP
- **Capturing meaning**: Represents meaning of a word numerically
- **Dimensionality reduction**: Projects words onto a much lower-dimensional space
- **Semantic relationships**: Captures semantic relationships like synonymy, antonymy and semantically similar words
- **Context free**:Captures the meaning based on co-occurrence statistics



IDENTIFIES SIMILAR WORDS

Thank you for your **help**

Thank you for your **assistance**
Thank you for your **aid**
Thank you for your **support**
Thank you for your **guidance**
Thank you for **backing**

Identifies Microsoft and IBM as similar words - they are similar in semantic sense

WORD VECTOR EXAMPLES

Similar words for apple

```
('apple',       0)
('iphone',      0.266)
('ipad',        0.287)
('apples',      0.356)
('blackberry',  0.361)
('ipod',        0.365)
('macbook',     0.383)
('mac',         0.391)
('android',     0.391)
('google',      0.395)
('microsoft',   0.418)
('ios',         0.433)
('iphones',     0.445)
('touch',       0.446)
('sony',        0.447)
```

Similar words for american

```
('american',    0)
('america',     0.255)
('americans',   0.312)
('u.s.',        0.320)
('british',     0.323)
('canadian',    0.329)
('history',     0.356)
('national',    0.364)
('african',     0.374)
('society',     0.375)
('states',      0.386)
('european',    0.387)
('world',       0.394)
('nation',      0.399)
('us',          0.399)
```

VECTOR DIFFERENCE BETWEEN TWO WORDS

$$\overrightarrow{\text{apple}} - \overrightarrow{\text{iphone}}$$

('raisin',	0.574)	('feta',	0.618)
('pecan',	0.576)	('apples',	0.623)
('cranberry',	0.584)	('avocado',	0.624)
('butternut',	0.588)	('fennel',	0.631)
('cider',	0.591)	('chutney',	0.631)
('apricot',	0.604)	('spiced',	0.633)
('tomato',	0.607)		
('rosemary',	0.615)		
('rhubarb',	0.616)		

VECTOR ARITHMETIC ON WORD VECTORS...

840B words and 300 elements word vectors used for this computation

$\overrightarrow{\text{apple}}$	$\overrightarrow{\text{apple}} - \overrightarrow{\text{iphone}}$	$\overrightarrow{\text{apple}} - \overrightarrow{\text{fruit}}$
('apple', 0)	('apples', 0.39)	('ipad', 0.412)
('apples', 0.25)	('fruit', 0.43)	('iphone', 0.433)
('blackberry', 0.31)	('grape', 0.44)	('macbook', 0.435)
('Apple', 0.35)	('tomato', 0.44)	('ipod', 0.445))
('iphone', 0.37)	('pecan', 0.45)	('imac', 0.465
('fruit', 0.37)	('rhubarb', 0.45)	('3gs', 0.473)
('blueberry', 0.38)	('pears', 0.45)	('Ipad', 0.490)
('strawberry', 0.38)	('cranberry', 0.452)	('itouch', 0.512)
('ipad', 0.39)	('raisin', 0.453)	('ipad2', 0.514)
('pineapple', 0.39)	('apricot', 0.459)	('Iphone', 0.514)
('pear', 0.39)	('carrot', 0.461)	('ios', 0.520)
('cider', 0.39)	('candied', 0.462)	('Macbook', 0.524)
('mango', 0.40)	('blueberry', 0.463)	('ibook', 0.534)
('ipod', 0.40)	('apricots', 0.466)	('IPhone', 0.541)
('raspberry', 0.40)	('tomatoes', 0.466)	('32gb', 0.545)

- Eigenvalues define principal semantic axes in embedding spaces [6][7]
- Hidden Layer Elements in models like Word2Vec correspond to coordinates on these axes [6].
- Top eigenvalues capture most semantic relationships. Optimal dimensionality is determined by the signal-to-noise trade-off [7]
- Word2Vecs hidden layer dimension parallels the number of significant eigenvalues [7]

EIGENVALUES IN WORD EMBEDDINGS

- Embedding algorithms such as SVD, GloVe, and Word2Vec can be interpreted as matrix factorizations of co-occurrence, PMI, or probability ratio matrices
- Top-k eigenvalues and eigenvectors capture the bulk of semantic information
- All words can be represented as linear combinations of these principal eigenvectors
- Selecting largest eigenvalues projects the high-dimensional space onto a lower-dimensional semantic basis, preserving key structural properties

DIMENSIONALITY REDUCTION

- Dimensionality selection is a bias-variance trade-off
  - Too few dimensions: high bias, semantic information lost
  - Too many: variance/noise increases, over-fitting risk
- The optimal number of eigenvalues balances information capture and robustness

RELATION TO WORD2VEC HIDDEN LAYER

- Word2Vecs hidden layer dimension (d) provides the semantic axes onto which words are projected
- Spectral analysis shows only a subset of these axes (top-k eigenvalues) capture meaningful word relations
- Empirical studies: increasing d beyond significant eigenvalue count yields diminishing returns - higher d may introduce noise
- In practice, the optimal d is often close to the number of top eigenvalues in the singular value spectrum of a co-occurrence/statistical matrix

INTERPRETING INDIVIDUAL EMBEDDING DIMENSIONS

- Each embedding dimension can be interpreted as a latent semantic feature - often grouping related words, topics, or syntactic classes
- Eigenvector analysis reveals that salient embedding elements correspond to interpretable word clusters or latent topics

Implication

The structure and interpretability of dimensions are enhanced when projections align with principal eigenvectors

SUMMARY TABLE

Aspect	Spectral Models/SVD	Word2Vec Hidden Layer
Semantic Axes	Eigenvectors	Learned Weights
Dimension Selection	Magnitude of Eigenvalues	Task Performance/Heuristics
Linear Combination	Vectors = Sum Over Axes	Coordinates in Hidden Layer
Optimal Dimension	Top-k Eigenvalues	Hidden Units $\approx$ k

REFERENCES I

[1] Xin Rong. *word2vec Parameter Learning Explained*. 2014. arXiv: 1411.2738. URL: https://sites.socsci.uci.edu/~lpearl/courses/readings/Rong2014_Word2Vec.pdf.

[2] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and Their Compositionality". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'13. Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 3111–3119. URL: http://dl.acm.org/citation.cfm?id=2999792.2999959.

[3] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* 3 (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=944919.944966.

[4] Tomas Mikolov et al. *Computing numeric representations of words in a high-dimensional space*. May 2015.

REFERENCES II

[5] Andriy Mnih and Geoffrey Hinton. "A Scalable Hierarchical Distributed Language Model". In: *Proceedings of the 21st International Conference on Neural Information Processing Systems*. NIPS'08. Vancouver, British Columbia, Canada: Curran Associates Inc., 2008, pp. 1081–1088. ISBN: 978-1-6056-0-949-2. URL: http://dl.acm.org/citation.cfm?id=2981780.2981915.

[6] Jamin Shin, Andrea Madotto, and Pascale Fung. "Interpreting word embeddings with eigenvector analysis". In: *32nd Conference on Neural Information Processing Systems (NIPS 2018), IRASL workshop*. 2018, pp. 73–81.

[7] Paramveer S Dhillon, Dean P Foster, and Lyle H Ungar. "Eigenwords: spectral word embeddings.". In: *J. Mach. Learn. Res.* 16 (2015), pp. 3035–3078.