An Introduction to Word Embeddings

Patrick Wu

ICPSR Summer Program 2020

The meaning of "meaning"

- What does "meaning" even mean?
- Merriam-Webster: "the thing one intends to convey especially by language; the thing that is conveyed especially by language"
- The most intuitive way we think about "meaning"

signifier (symbol) ⇔ signified (idea or thing)

- Known as "denotational semantics"

How do we quantitatively represent meaning?

- One solution: consider words as discrete symbols
- Use one-hot vectors (aka a discrete *embedding*)

cat = [0,0,0,0,0,0,1,0]
mat = [0,0,0,0,0,1,0,0]

- The dimension of this vector is simply the number of words in the vocabulary (i.e. the number of unique words across the corpus)

Problems with discrete embeddings

cat = [0,0,0,0,0,0,1,0]
mat = [0,0,0,0,0,1,0,0]

- Notice that the one-hot vectors are orthogonal to each other---that is, $v(cat) \cdot v(mat) = 0$
- So no measure of similarity or dissimilarity between words
- Maybe we can upweight the word (e.g., change it from 1 to 2) and then mark the synonyms (using a thesaurus) using a 1 or even marking antonyms with -1
- But using synonyms is problematic! Is "proficient" the same as "good"? A thesaurus would say they are synonyms, but are they really synonyms?
- What if we learned to encode the similarity in the vectors themselves?

Why vectors?

- As we saw with SVM, vectors can be compared using a similarity function
- For example, the inner dot product is a measure of vector similarity
- Normalizing the dot product by the product of the length of the vectors produces a *cosine similarity*, which is a measure of similarity between -1 and 1
  - Defined as $\frac{A \cdot B}{||A||\,||B||}$
  - -1 means the vectors are perfectly facing away from each other
  - 1 means the vectors align perfectly
  - 0 means the vectors are orthogonal

Meaning through context

- Distributional semantics: a word's meaning is given by the other words that frequently appear in its contextual neighborhood
- "You shall know a word by the company it keeps" (J. R. Firth)
- So when a word *w* appears in the text, its context is the set of words that appear near that word
- We can use the many contexts that word *w* exists in to build up a representation of the word
- The key idea is that the word is best understood by investigating its context

The Jabberwocky by Lewis Carroll

Beware the Jabberwock, my son!

The jaws that bite, the claws that snatch!

Beware the Jubjub bird, and shun

The frumious Bandersnatch!

The Jabberwocky by Lewis Carroll

Beware the **Jabberwock**, my son!

The jaws that bite, the claws that snatch!

Beware the **Jubjub** bird, and shun

The **frumious Bandersnatch**!

Word vectors

- Notice that one-hot encodings are sparse vectors, because every element of the vector is o except for one
- In contrast, word vectors, or word *embeddings* or word *representations*, are dense vectors

$$\text{bank} = \begin{bmatrix} -0.123 \\ 0.111 \\ 0.120 \\ \dots \\ 0.441 \end{bmatrix}$$

Example of a word vector space

$$expect = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$

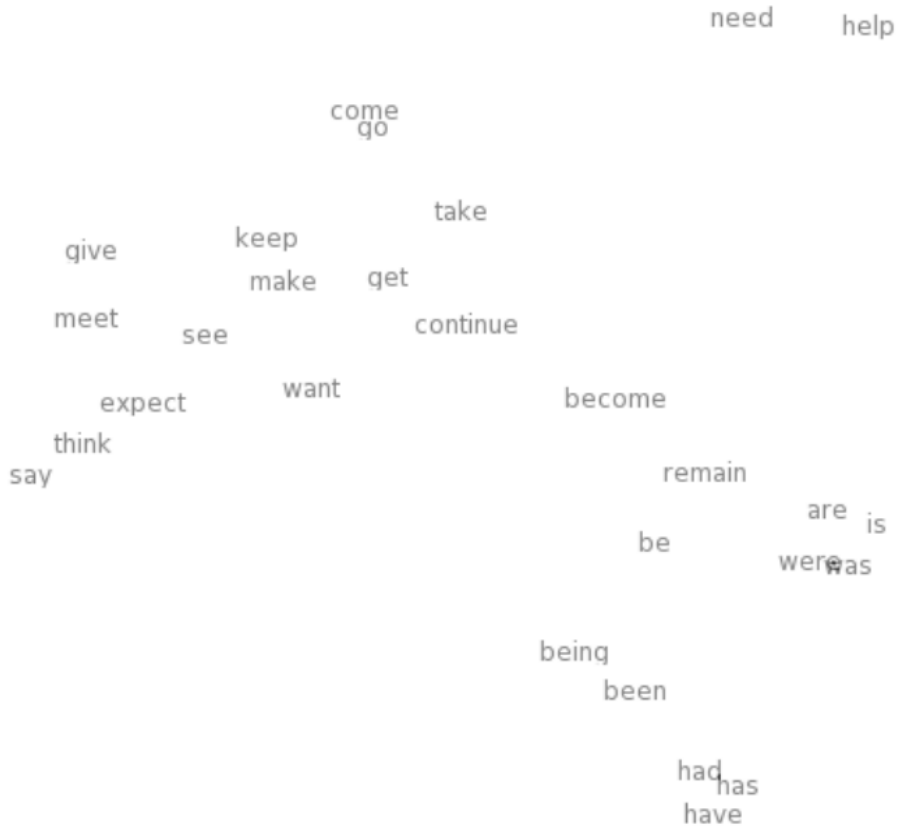


Diagram taken from lecture notes of CS224n at Stanford University

Word2vec

- The basis of word2vec is a shallow two-layer neural network (that is, one hidden layer) that is trained on a "fake" classification task or a "proxy" classification task
- The trick is that we won't use the actual results of the neural network; instead, we simply take the results of the weights between the input and hidden layer, which becomes our word embedding

Word2vec: the main idea

- The main idea is that we want to map every unique word in a corpus, or the *vocabulary*, to a vector
- Go through every position in the text. At each position, there is one center word *c* and context words *o*.
- We will use the similarity of the word vectors for *c* and *o* to calculate the probability of *c* given *o* (common bag-of-words, or CBOW) or the probability of *o* given *c* (skip-gram)
- Ultimately, we want to maximize this probability; to do this, we adjust the values of the word vectors

Word2vec: Some technical details of the skip-gram approach

For each position $t = 1,\ldots,T$, predict context words within a context of size m, given center word w_j. Note here that θ is all the parameters that we want to optimize.

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} P(w_{t+j} | w_t; \theta)$$

The objective function is the average of the negative log-likelihood

$$J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log P(w_{t+j} | w_t; \theta)$$

Thus, we want to minimize this objective function, because minimizing the objective function is equivalent to maximizing predictive accuracy. Ultimately, we are interested in using θ as our word vectors.

Word2vec: Some technical details of the skip-gram approach

- How do we calculate $P(w_{t+j}|w_t;\theta)$?
- We will use *two* vectors per word: v_w when *w* is the center word and u_w when *w* is the context word
- Then,

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

You might recognize this as a softmax function

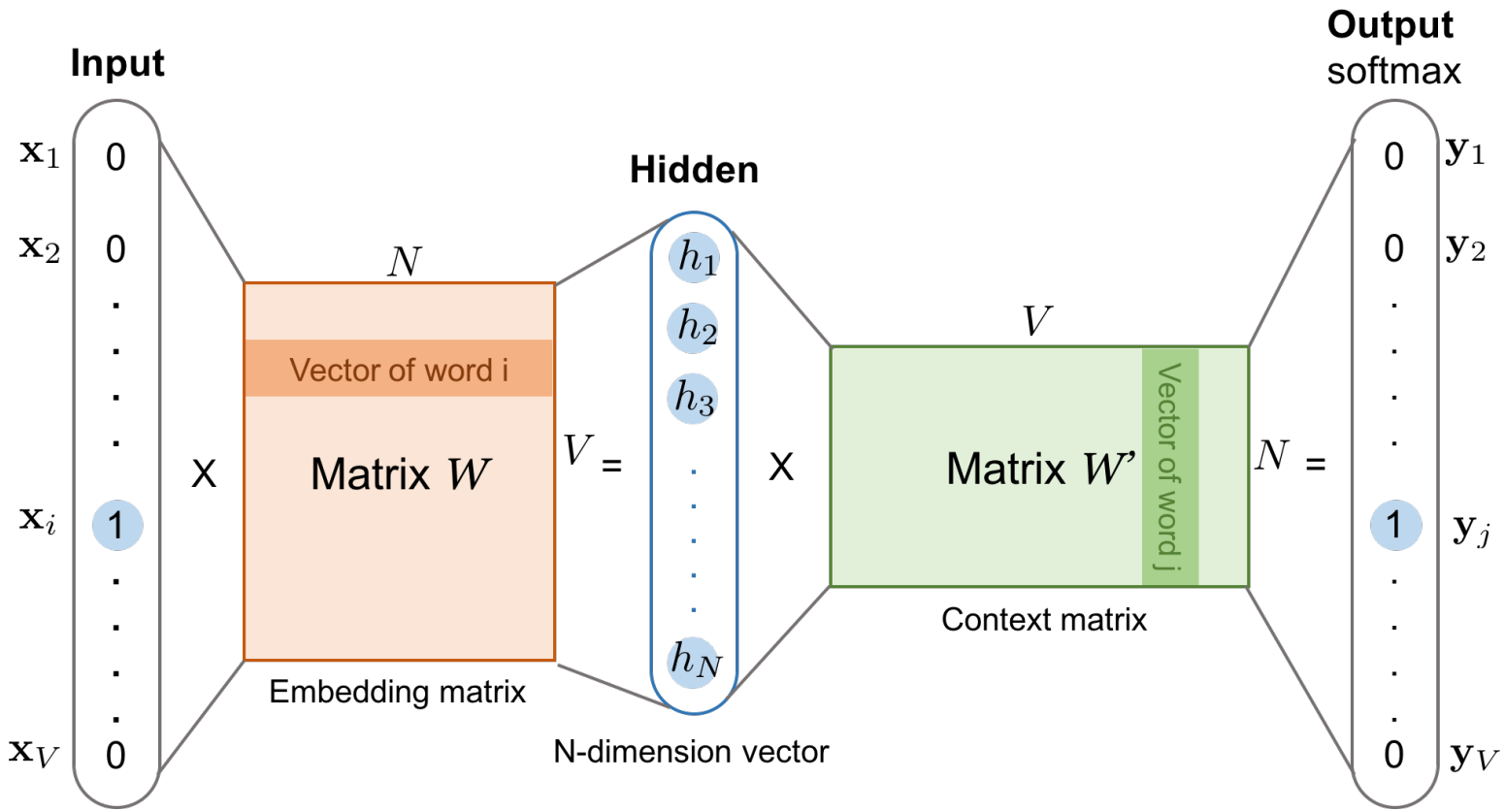
This is just a two-layer neural network with no activation function

- We recognize that this set-up is a two-layer neural network with no activation function
- The vectors v_w are weights from the first layer; the vectors u_w are weights from the second layer
- What do the features actually look like?

Creating features from "The cat sat on the mat." with a window size of two.

- **The** cat sat on the mat. → (the, cat), (the, sat)
- The **cat** sat on the mat. → (cat, the), (cat, sat), (cat, on)
- The cat **sat** on the mat. → (sat, the), (sat, cat), (sat, on), (sat, the)
- The cat sat **on** the mat. → (on, cat), (on, sat), (on, the), (on, mat)
- The cat sat on **the** mat. → (the, sat), (the, on), (the, mat)
- The cat sat on the **mat**. → (mat, on), (mat, the)

Skip-gram in action



Source of diagram: https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html

The intuition of why this all works

- Context words *o* that are near the current word *c* will have a high probability, $p(o|c)$, and non-context words $\tilde{o}$ that are not near the current word *c* will have low probability, $p(\tilde{o}|c)$
- So if two words have similar contexts (similar words appear around them), the neural network has to output very similar results for the two words
- The way to make the output of a NN similar across these two words is to make the weights associated with the two words similar
- In other words, if two words have similar contexts, our neural network will learn similar word vectors for these two words, which means they appear near each other in vector space
- Note that, much like with neural networks used in other contexts, we get to set the dimensionality of the hidden layer, which means we get to control the dimensions of these word vectors

Negative sampling

- As a reminder, our objective is to calculate

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

- But notice that the denominator is very costly to calculate, especially if the vocabulary of the corpus is very large
- We can use *negative sampling* to calculate the probability instead
- The idea is that, rather than calculate the softmax directly, we can sample a set of words that do not exist in the contextual neighborhood
- Our new objective function then becomes

$$J(\theta) = -\left[\log \sigma(u_o^T v_c) + \sum_{i=1}^{N} \log \sigma(-u_{\tilde{o}_i}^T v_c)\right]$$

$$\text{where } \tilde{o}_i \sim Q$$

Hyperparameters

- Dimensionality: dimensions of the word vectors
- Window Size: how big of a window you want around the center word
- Minimum Count: how many times a word has to appear in the corpus for it to be assigned a vector (if a word happens too few times, it is difficult to assign it a good vector)
- Model Type: skip-gram or continuous bag-of-words
- Number of Iterations: number of iterations (epochs) over the corpus

The linearity property of word vectors

- One of the more remarkable properties of word2vec embeddings is the property of linearity
- Mikolov et al. (2013) found that if you fit vectors over an appropriately large corpus, the word vector space can actually solve analogies using basic vector addition and subtraction
- They found that *vec*(king) – *vec*(man) + *vec*(woman) ≈ *vec*(queen)
- Rheault and Cochrane (2020) take advantage of this property and produce scaling estimates of ideological placement of political parties using parliamentary floor speeches
- Wu et al. (2020) use this property to calculate partisan associations of Twitter users using user bios

Other word embedding methods

- doc2vec (Le and Mikolov (2014))
- GloVe (Pennington, Socher, and Manning (2014))
- fastText (Bojanowski, Grave, Joulin, and Mikolov (2016))
- Latent semantic analysis (Deerwester, Dumais, Furnas, Harshman, Landauer, Lochbaum, and Streeter (1988))
- BERT (Devlin, Chang, Lee, and Toutanova (2018))

Code for Word Embeddings

https://colab.research.google.com/drive/1LiyOpSxkFN-XFTJiKF1OoKUwOPZnddsT?usp=sharing