17
Deep Learning for Natural Language Processing

17.1 Word Embeddings

In traditional NLP, each word is regarded as discrete symbols each with a single value of weight. For example, in Chapter 1, we learned how to use linear regression on sentiment prediction. But with this approach, it is hard for the computer to learn the *meaning* of the word; instead, each of the words remain as some abstract symbols with numeric weights.

But how do computers know the meaning of words? We can easily think of one solution: we can look up words in a dictionary. For example, WordNet is a project that codes the meaning of the words and the relationship between the words, so that the data can be used for computers to parse. [1] But resources like WordNet require human labor to create and adapt, and it is impossible to keep up-to-date (because new words are coined up and new meanings appear out of existing words).

[1] For more information, check `http://wordnetweb.princeton.edu`.

An alternative approach is to represent words as short (50 - 300 dimensions [2]), real-valued vectors. These vectors encode the meaning and other properties of words. In this representation, the distance between vectors represent the *similarity* between words. This vectorized form of the words are much easier to be used as inputs in modern ML systems (especially neural networks). This vector form of words is known as *word embedding*. In this section, we explore the process of how to learn a good word embedding.

[2] The dimension of word vectors is a hyperparameter that needs to be decided first.

17.1.1 Distributional Hypothesis

Word embedding is based on a concept called the *distributional hypothesis*, a theory developed by John Rupert Firth. The hypothesis, one of the most successful ideas of modern statistical NLP, says that words that occur in similar contexts tend to have similar meaning.

Definition 17.1.1 (Context). *When a word w appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).*

Example 17.1.2. *Assume that you first heard the word **tejuino** and have no idea what the word means. But you learn that the word may appear in the following four contexts.*

- *C1: A bottle of ____ is on the table.*
- *C2: Everybody likes ____.*
- *C3: Don't have ____ before you drive.*
- *C4: We make ____ out of corn.*

Based on these contexts, it is reasonable to conclude that the word "tejuino" refers to some form of alcoholic drink made from corn.

Problem 17.1.3. *To find words with similar meanings as "tejuino," we tried filling out the contexts from Example 17.1.2 with 5 other words. The results are given in Table 17.1, where 1 means that the word was appropriate to be used in that context, and 0 means that it was inappropriate.*

	C1	C2	C3	C4
tejuino	1	1	1	1
loud	0	0	0	0
motor-oil	1	0	0	0
tortillas	0	1	0	1
choices	0	1	0	0
wine	1	1	1	0

Table 17.1: Data showing if 6 words are appropriate for the four contexts in Example 17.1.2.

Which word is closest to "tejuino"?

17.1.2 Word-word Co-occurrence Matrix

Given a very large collection of documents with words from a dictionary V, we construct a $|V| \times |V|$ matrix X, where the entry at the i-th row, j-th column denotes the number of times (*i.e.*, frequency) that w_j appears in the context window of w_i. This matrix is called the *word-word co-occurrence matrix*.

Example 17.1.4. *Table 17.2 shows a portion of a word-word co-occurrence matrix. Each row corresponds to the* center word w_i, *and each column corresponds to the* context word w_j. *The value X_{ij} at the (i, j) entry means that the context word w_j appeared X_{ij} times in the context (of length 4) of w_i in total.*

Although the portion shown in Table 17.2 mostly has non-zero entries, in general, the entries of the matrix are mostly zero.

	···	computer	data	result	pie	sugar	···
cherry	···	2	8	9	442	25	···
strawberry	···	0	0	1	60	19	···
digital	···	1670	1683	85	5	4	···
information	···	3325	3982	378	5	13	···

Table 17.2: A portion of a word-word co-occurrence matrix for a corpus of Wikipedia articles. Source: `https://www.english-corpora.org/wiki/`.

17.1.3 *Factorization of Word-word Co-occurrence Matrix*

Recall the example of movie recommendation through matrix factorization in Chapter 9. In that example $m \times n$ matrix M was factorized into $M \approx AB$ where the i-th row of A was a d-dimensional vector that represented user i and the j-th column of B was a d-dimensional vector that represented movie j.

We can imagine a similar factorization on the word-word co-occurrence matrix. That is, we can represent each *center word* and each *context word* as a d-dimensional vector such that $X_{ij} \approx A_{i*} \cdot B_{*j}$. But this particular idea does not work on the word-word co-occurrence matrix. The key difference is that X is a complete matrix with no missing entries (although most entries are zero). Therefore we instead use other standard matrix factorization techniques (*e.g.*, Singular Value Decomposition).

One popular choice of factorization is running the Singular Value Decomposition (SVD) on a weighted co-occurrence matrix. [3] This idea originates from a concept called *Latent Semantic Anlysis*. [4] If the SVD returns the following decomposition,

[3] The particular weighting scheme is called *PPMI*. We will not get into the detail here.
[4] From *Indexing by Latent Semantic Analysis* by Deerwester et al., 1990.

$$\begin{bmatrix} & & \\ & X & \\ & & \end{bmatrix} = \begin{bmatrix} & & \\ & W & \\ & & \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d \end{bmatrix} \begin{bmatrix} & W^\intercal & \end{bmatrix}$$

where X is a $|V| \times |V|$ matrix and W is a $|V| \times d$ matrix, then the i-th row of matrix W can be regarded as the embedding for word w_i.

Other modern approaches tend to treat word vectors as parameters to be optimized for some objective function and apply the gradient descent algorithm. But the principle is the same: "words that occur in similar contexts tend to have similar meanings." Some of the popular algorithms with this approach include: *word2vec* (Mikolov et al., 2013), *GloVe* (Pennington et al., 2014), and *fastText* (Bojanowski et al., 2017).

Here we briefly explain the GloVe algorithm. Given the co-occurrence table X, we will construct a *center word vector* $\vec{\mathbf{u}}_i \in \mathbb{R}^d$ and a *context word vector* $\vec{\mathbf{v}}_j \in \mathbb{R}^d$ such that they optimize the following objective:

$$J(\theta) = \sum_{i,j \in V} f(X_{ij}) \left(\mathbf{u}_i \cdot \mathbf{v}_j + b_i + \widetilde{b_j} - \log X_{ij} \right)^2 \tag{17.1}$$

where f is some non-linear function and $b_i, \widetilde{b_j}$ are bias terms. This is within the same line of logic as optimizing

$$L(A, B) = \frac{1}{|\Omega|} \sum_{i,j \in \Omega} (M_{ij} - (AB)_{ij})^2 \qquad \text{((9.5) revisited)}$$

17.1.4 Properties of Word Embeddings

A good word embedding should represent the meaning of the words and their relationship with other words as accurately as possible. Therefore there are some properties that we would like a word embedding to preserve. We will discuss three such properties and see how the current algorithms for word embedding perform on preserving those properties.

1. Similar words should have similar word vectors: This is the most important property we can think of.

Example 17.1.5. *In a certain word embedding, the following is the list of* 9 *most nearest words to the word "sweden."*

Word	*Cosine distance*
norway	*0.760124*
denmark	*0.715460*
finland	*0.620022*
switzerland	*0.588132*
belgium	*0.585835*
netherlands	*0.574631*
iceland	*0.562368*
estonia	*0.547621*
slovenia	*0.531408*

Notice Scandanavian countries are the top 3 *entries on the list, and the rest are also European country names.*

2. Vector difference should encode the relationship between words: If there are two or more pairs of words where each pair of words are distinguishable by the same attribute, you can imagine that the vector difference within each pair is nearly the same.

Example 17.1.6. *In Figure 17.1, notice that $v_{man} - v_{woman} \approx v_{king} - v_{queen}$. The vector difference in common can be understood as representing the male-female relationship. Similarly, there seems to be a common vector difference for representing the difference in verb tense.*

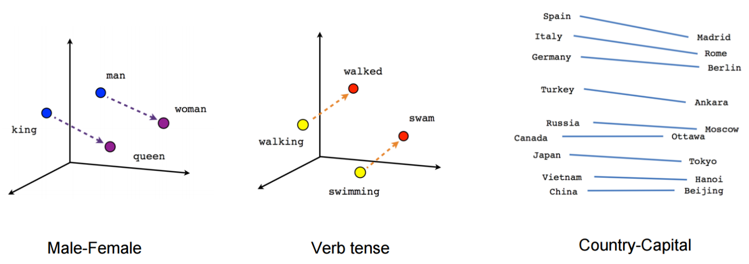

Figure 17.1: Two pairs of words that differ in the same attribute show a similar difference in their word embeddings.

3. *The embeddings should be translated between different languages:* When we independently find the word embedding in different languages, we can expect to have a bijective mapping that preserves the structure of the words in each language. [5]

[5] From *Exploiting Similarities among Languages for Machine Translation* by Mikolov et at., 2013.

Example 17.1.7. *In Figure 17.2, notice that if we let W to be the mapping from English to Spanish word embeddings,* $v_{cuatro} \approx W \circ v_{four}$

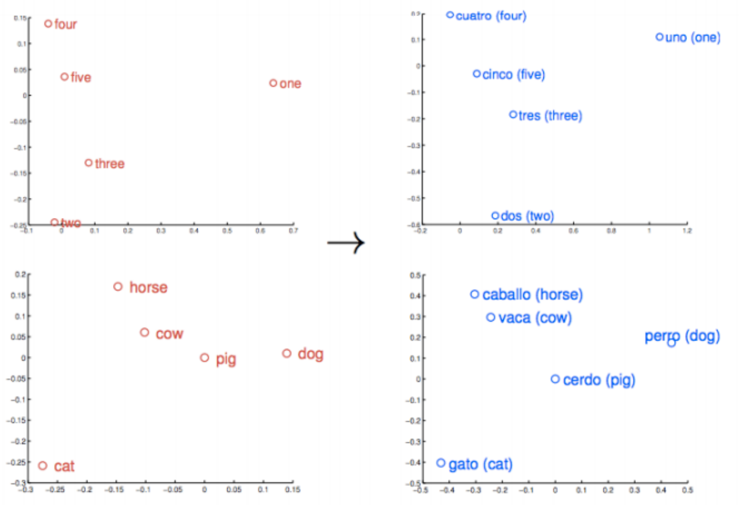

Figure 17.2: Word embeddings are translated into the embeddings of other languages.

17.2 N-gram Model Revisited

Recall the n-gram model from Chapter 8. It assigned a probability $\Pr[w_1 w_2 \ldots w_n]$ to every word sequence $w_1 w_2 \ldots w_n$. We discussed the concept of perplexity of the model to compare the performance of unigram, bigram, and trigram models. While the n-gram model is impressive, it has obvious limitations.

Problem 17.2.1. *"The students opened their ______." Can you guess the next word?*

Problem 17.2.2. *"As the proctor started the clock, the students opened their ______." Can you guess the next word?*

In a lot of cases, words in a sentence are closely related to other words and phrases that are far away. But the n-gram model cannot look beyond the specified frame.

Example 17.2.3. *The following is a text generated by a 4-gram model*

> *Today the price of gold per tan, while production of shoe lasts and shoe industry, the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks, sept 30 and primary 76 cts a share.*

The generated text is surprisingly grammatical, but incoherent.

Example 17.2.3 shows that we need to consider more than three words at a time if we want to model language well. But if we use a larger value of n for the n-gram model, the data will become too sparse to estimate the probabilities. But even when we restrict ourselves to words that appear in the dictionary, there are 10^{21} distinct sequences of 4 words.

17.2.1 Feedforward Neural Language Model

The idea of *feedforward neural language model* was proposed by Bengio et al. in 2003 in a paper called *A Neural Probabilistic Language Model*. The intuition is to use a neural network to learn the probabilistic distribution of language, instead of estimating raw probabilities. The key ingredient in this model is the word embeddings we discussed earlier.

Example 17.2.4. *Assume we are given two contexts "You like green ______" and "You like yellow ______" to fill the blanks in. A n-gram model will try to calculate the raw probabilities* $\Pr[w \mid \textit{You like green}]$ *and* $\Pr[w \mid \textit{You like yellow}]$*. However, if the word embeddings showed that* $v_{green} \approx v_{yellow}$*, then we can imagine that the two contexts are similar enough. Then we may be able to estimate the probabilities better.*

Now we show how to use feedforward neural language model on a n-gram model. Assume we want to estimate the probability $\Pr[w_{n+1} \mid w_1 \ldots w_n]$. Then the first step is to find a find a word embedding

$$v_1, v_2, \ldots, v_n \in \mathbb{R}^d$$

of each word $w_1, w_2, \ldots, w_n$. Then we concatenate the word embeddings into [6]

$$\vec{\mathbf{x}} = (v_1, \ldots, v_n) \in \mathbb{R}^{nd}$$

[6] the order of the input vectors cannot change

This will be the input layer. Then we define the fully connected hidden layer as

$$\vec{\mathbf{h}} = \tanh(\mathbf{W}\vec{\mathbf{x}} + \vec{\mathbf{b}}) \in \mathbb{R}^h$$

where $\mathbf{W} \in \mathbb{R}^{h \times nd}$ and $\vec{\mathbf{b}} \in \mathbb{R}^h$. Then we define the output layer as

$$\vec{\mathbf{z}} = \mathbf{U}\vec{\mathbf{h}} \in \mathbb{R}^{|V|}$$

where $\mathbf{U} \in \mathbb{R}^{|V| \times h}$. Then finally, the probability will be calculated with the softmax function:

$$\Pr[w = i \mid w_1 \dots w_n] = \text{softmax}_i(\vec{\mathbf{z}}) = \frac{e^{z_i}}{\sum\limits_{k \in V} e^{z_k}}$$

So the total number of parameters to train in this network is

$$d\,|V| + ndh + h + h\,|V|$$

where the terms are respectively for the input embeddings, $\mathbf{W}, \vec{\mathbf{h}}, \mathbf{U}$. When $d = h$, sometimes we tie the input and output embeddings. That is, we can consider $\mathbf{U}$ to be the parameters required for the output embeddings. At this point, the language model reduces to a $|V|$-way classification, and we can create lots of training example by sliding the input-output indices. That is, when given a huge text, we can create lots of input-output tuple as follows: $((w_1, \dots, w_n), w_{n+1}), ((w_2, \dots, w_{n+1}), w_{n+2}), \dots$.

17.2.2 Beyond Feedforward Neural Language Model

But the feedforward language model still has its limitations. The main reason is that $\mathbf{W} \in \mathbb{R}^{h \times nd}$ scales linearly with the window size. Of course, this is better than the traditional n-gram model which scales exponentially with n. Another limitation of the neural LM is that the model learns separate patterns for the same item. That is, a substring $w_k w_{k+1}$, for example, will correspond to different parameters in $\mathbf{W}$ when trained on $(w_k w_{k+1} \dots w_{k+n-1})$ or on $(w_{k-1} w_k \dots w_{k+n-2})$.

To mitigate these limitations, we can choose to use similar modeling ideas but use better and bigger neural network architectures like *recurrent neural networks (RNN)* or *transformers*.

Here we briefly explain the core ideas of a RNN. RNNs are a family of neural networks that handle variable length inputs. Whereas feedforward NNs map a fixed-length input to a fixed-length output, recurrent NNs map *a sequence* of inputs to *a sequence* of outputs. The sequence length can vary and the key is to *reuse the weight matrices* at different time steps. When the inputs are given as $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \dots \vec{\mathbf{x}}_T \in \mathbb{R}^d$ and we want to find outputs $\vec{\mathbf{h}}_1, \vec{\mathbf{h}}_2, \dots \vec{\mathbf{h}}_T \in \mathbb{R}^h$, we train the parameters

$$\mathbf{W} \in \mathbb{R}^{h \times h}, \mathbf{U} \in \mathbb{R}^{h \times d}, \vec{\mathbf{b}} \in \mathbb{R}^h$$

such that

$$\vec{\mathbf{h}}_t = g(\mathbf{W}\vec{\mathbf{h}}_{t-1} + \mathbf{U}\vec{\mathbf{x}}_t + \vec{\mathbf{b}}) \in \mathbb{R}$$

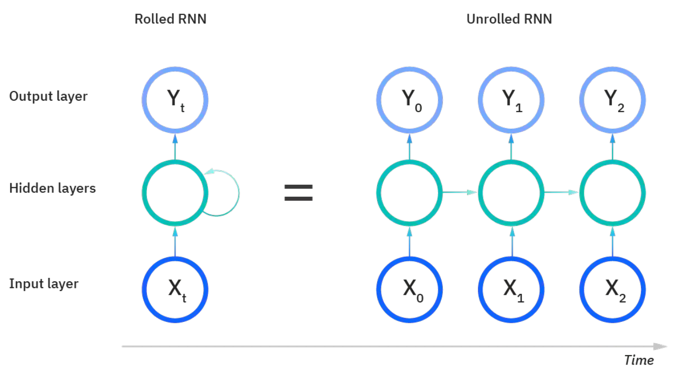

Figure 17.3: A visual representation of an RNN architecture.

where g is some non-linear function (*e.g.*, ReLU, tanh, sigmoid). We can also set $\vec{\mathbf{h}}_0 = \vec{\mathbf{0}}$ for simplicity.