Neural Language Models

9

9.1 Word Embeddings and Word Analogies

Neural language models are distributed representations of words and sentences. They are learned representations, meaning that they are numerical vectors. A *word embedding* is any method which converts words in numbers, and therefore, any learned neural language model is a way of obtaining word embeddings. We use the term 'word embedding' to denote a very concrete numerical representation of a certain word or words, represent 'Nowhere fast' as (1, 0, 0, 5.678, −1.6, 1). In this chapter, we focus on the most famous of the neural language models, the Word2vec model, which learns vectors which represent words with a simple neural network.

This is similar to the predict-next setting for recurrent neural networks, but it gives an added bonus: we can calculate word distances and have similar words only a short distance away. Traditionally, we can measure the distances of two words as strings with the *Hamming distance* [1]. For measuring the Hamming distance, two strings have to be of the same length and the distance is simply the number of characters that are different. The Hamming distance between the words 'topos' and 'topoi' is 1, while the distance between 'friends' and 'fellows' is 5. Note that the distance between 'friends' and '0r$8MMs' is also 5. It can easily be normalized to a percentage by dividing it by the words' length. You can probably see already how this would be a useful but very limited technique for processing language.

The Hamming distance is the simplest method from a wide variety of string similarity measures collectively known as *string edit distance metrics*. More evolved forms such as Levenshtein distance [2] or Jaro–Winkler [3,4] distance can compare strings of different lengths and penalize differently various errors, such as insertion, deletion or edit. All of these are measures of a word by the form of the word. They would be useless in comparing 'professor' and 'teacher', since they would never recognize the similarity in meaning. This is why, we want to embed a word in a

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_9

vector in a way which will convey information about the meaning of the word (i.e. its use in our language).

If we represent words as vectors, we need to have a distance measure between vectors. We have touched upon this idea a number of times before, but we can now introduce the notion of the *cosine similarity* of vectors. A good overview of cosine similarity is given in [5]. Cosine similarity of two n-dimensional vectors $\mathbf{v}$ and $\mathbf{u}$ is given by:

$$\mathbb{CS}(\mathbf{v}, \mathbf{u}) := \frac{\mathbf{v} \cdot \mathbf{u}}{||\mathbf{v}|| \cdot ||\mathbf{u}||} = \frac{\sum_{i=1}^{n} v_i u_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \sqrt{\sum_{i=1}^{n} u_i^2}} \tag{9.1}$$

Where v_i and u_i are components of $\mathbf{v}$ and $\mathbf{u}$, and $||\mathbf{v}||$ and $||\mathbf{u}||$ denote the norms of the vectors $\mathbf{v}$ and $\mathbf{u}$ respectively. The cosine similarity ranges from 1 (equal) to -1 (opposite), and 0 means that there is no correlation. When using the bag of words, one-hot encoding s or similar word embeddings the cosine similarity ranges from 0 to 1, since the vectors representing fragments do not contain negative components. This means that 0 takes the meaning of 'opposite' in such contexts.

We will now continue to show the Word2vec neural language model [6]. In particular, we will address the questions of what input does it need, what will it give as an output, does it have parameters to tune it and how can we use it in a complete system, i.e. how does it interact with other components of a bigger system.

9.2 CBOW and Word2vec

The Word2vec model can be built with two different architectures, the skip-gram and the Word2vec. Both of these are actually shallow neural networks with a twist. To see the difference, we will use the sentence 'Who are you, that you do not know your history?'. First, we clean the sentence from uppercase and interpunction. Both architectures use the context of the word (the words around it) as well as the word itself. We must define in advance how large will the context be. For the sake of simplicity, we will be using a context of size 1. This means that the context of a word consists of one word before and one word after. Let us break or sentence into word and context pairs:

We have already noted that both versions of the Word2vec are learned models, and this means they must learn something. The skip-gram model learns to predict a word from the context given the middle word. This means that if we give the model 'are' it should predict 'who', if we give it 'know' it should predict 'not' or 'your'. The CBOW version does the opposite, assuming the context to be 1, it takes two words[1] from the context (we will call them $c1$ and $c2$) and uses it to predict the middle or main word (which we will denote by m).

[1] If the context were 2, it would take 4 words, two before the main word and two after.

Context	Word
'are'	'who'
'who', 'you'	'are'
'are', 'that'	'you'
'you', 'you'	'that'
'that', 'do'	'you'
'you', 'not'	'do'
'do', 'know'	'not'
'not', 'your'	'know'
'know', 'history'	'your'
'your'	'history'

The production of the word embeddings is structurally quite similar to autoencoders. To make the network which produces the embeddings, we use a shallow feedforward network. The input layer will receive word index vectors, so we will need as many input neurons as there are unique words in the vocabulary. The number of hidden neurons is called *embedding size* (suggested values range between 100 and 1000, which is considerably less than the vocabulary size even for modest datasets), and the number of output neurons is the same as input neurons. The input to hidden connections are linear, i.e. they have no activation function, and the hidden to output have softmax activations. The weights of the input to hidden are the deliverables of the model (similar to the autoencoder deliverables), and this matrix contains as rows the individual word vectors for a particular word. One of the easiest methods of extracting the proper word vector is to multiply this matrix by the word index vector for a given word. Note that these weights are trained with backpropagation in the usual way. Figure 9.1 offers an illustration of the whole process. If something is unclear, we ask the reader to fill out the details for herself by using what we have previously covered in this book—there should be no problem in doing this.

Before continuing to the code for the CBOW Word2vec, we must correct a historical mistake. The idea behind Word2vec is that the meaning of a given word is determined by a context, which is usually defined as the way the word is used in a language. Most deep learning textbooks (including the official TensorFlow documentation on Word2vec) attribute this idea to a paper from 1954 by Harris [7], and note that he idea came to be known in linguistics as the distributional hypothesis in 1957 [8]. This is actually wrong. The first time this idea was proposed was in Wittgenstein's *Philosophical Investigations* in 1953 [9], and since ordinary language philosophy and philosophical logic (the area of logic dealing mainly with language formalization) played a major role in the history of natural language processing, the historical merit must be acknowledged and attributed correctly.

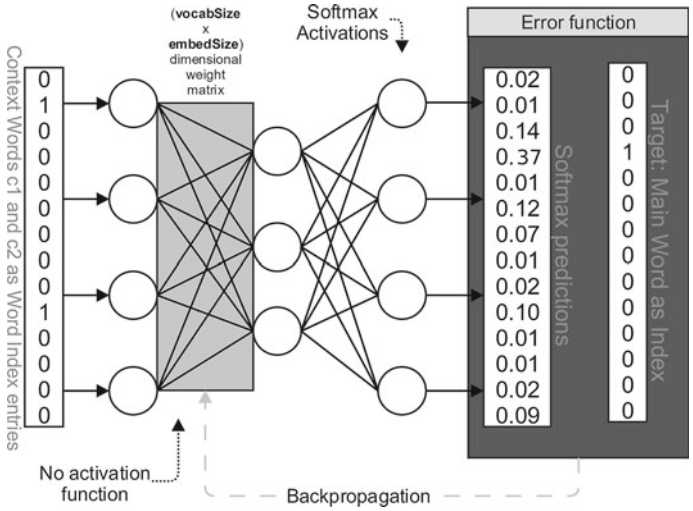

Fig. 9.1 CBOW Word2vec architecture

9.3 Word2vec in Code

In this and the next section, we give an example of a CBOW Word2vec implementation. All the code in these two sections should be placed in one Python file, since it is connected. We start with the usual imports and hyperparameters:

```
from keras.models import Sequential
from keras.layers.core import Dense
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
text_as_list=["who","are","you","that","you","do","not","know","your","history"]
embedding_size = 300
context = 2
```

The `text_as_list` can hold any text, so you can put here your text, or use the parts of the code from the recurrent neural network which parse a text file into a list of words. The embedding size is the size of the hidden layer (and, consequently, that the word vectors will have). The context is the number of words before and after the given word which will be used this. If the context is 2, this means we will use two words before the main word and two words after the main word to create the inputs (the main word will be the target). We continue to the next block of code which is exactly the same as the same part of code for recurrent neural networks:

```
distinct_words = set(text_as_list)
number_of_words = len(distinct_words)
```

```
word2index = dict((w, i) for i, w in enumerate(distinct_words))
index2word = dict((i, w) for i, w in enumerate(distinct_words))
```

This code creates word and index dictionaries in both ways, one where the word is the key and the index is the value and another one where the index is the key and the word is the value. The next part of the code is a bit tricky. It creates a function that produces two lists, one is a list of main words, and the other is a list of context words for a given word (it is a list of lists):

```
def create_word_context_and_main_words_lists(text_as_list):
    input_words = []
    label_word = []
    for i in range(0,len(text_as_list)):
        label_word.append((text_as_list[i]))
        context_list = []
        if i >= context and i<(len(text_as_list)-context):
            context_list.append(text_as_list[i-context:i])
            context_list.append(text_as_list[i+1:i+1+context])
            context_list = [x for subl in context_list for x in subl]
        elif i<context:
            context_list.append(text_as_list[:i])
            context_list.append(text_as_list[i+1:i+1+context])
            context_list = [x for subl in context_list for x in subl]
        elif i>=(len(text_as_list)-context):
            context_list.append(text_as_list[i-context:i])
            context_list.append(text_as_list[i+1:])
            context_list = [x for subl in context_list for x in subl]
        input_words.append((context_list))
    return input_words, label_word
input_words,label_word = create_word_context_and_main_words_lists(text_as_list)
input_vectors = np.zeros((len(text_as_list), number_of_words), dtype=np.int16)
vectorized_labels = np.zeros((len(text_as_list), number_of_words), dtype=np.int16)
for i, input_w in enumerate(input_words):
    for j, w in enumerate(input_w):
        input_vectors[i, word2index[w]] = 1
        vectorized_labels[i, word2index[label_word[i]]] = 1
```

Let us see what this block of code does. The first part is the definition of a function that takes in a list of words and returns two lists. One is a copy of that list of words (named `label_word` in the code), and the second is `input_words`, which is a list of lists. Each list in the list carries the words from the context of the corresponding word in `label_word`. After the whole function is defined, it is called on the variable `text_as_list`. After that two matrices to hold the word vectors corresponding to the two lists are created with zeros, and the final part of the code updates the corresponding parts of the matrices with 1, to make a final model of the context for inputs and of the main word for the target. The next part of the code initializes and trains the Keras model:

```
word2vec = Sequential()
word2vec.add(Dense(embedding_size, input_shape=(number_of_words,), activation=
"linear", use_bias=False))
word2vec.add(Dense(number_of_words, activation="softmax", use_bias=False))
word2vec.compile(loss="mean_squared_error", optimizer="sgd", metrics=['accuracy'])
word2vec.fit(input_vectors, vectorized_labels, epochs=1500, batch_size=10, verbose=1)
metrics = word2vec.evaluate(input_vectors, vectorized_labels, verbose=1)
print("%s: %.2f%%" % (word2vec.metrics_names[1], metrics[1]*100))
```

The model follows closely the architecture we presented in the last section. It does not use biases since we will be taking out the weights and we do not want any information to be anywhere else. The model is trained for 1500 epochs and you may want to experiment with these. If one wants to make a skip-gram model instead, one should just interchange these matrices, so the part that says `word2vec.fit(input_vectors, vectorized_labels, epochs =1500, batch_size=10, verbose=1)` should be changed to `word2vec.fit(vectorized_labels, input_vectors, epochs=1500, batch_size=10, verbose=1)` and you will have a skip-gram. Once we have this, we just take out the weights with the following code:

```
word2vec.save_weights("all_weights.h5")
embedding_weight_matrix = word2vec.get_weights()[0]
```

And we are done. The first line of this code returns the word vectors for all the words, in the form of a `number_of_words`×`embedding_size` dimensional array, and we can pick the appropriate row to get the vector for that word. The first line saves all the weights in the network to a H5 file. You can do several things with word2vec and for all of them we need these weights. First, we may just learn weights from scratch, as we did with our code. Second, we might want to fine-tune a previously learned word embedding (suppose it was learned from Wikipedia data), and in that case, we want to load previously saved weights in a copy of the original model and train it on new texts that are perhaps more specific and more closely connected with e.g. legal texts. The third way we may use word vectors is to simply use them instead of one-hot encoded words (or a Bag of Words), and feed them in another neural network which has the task of e.g. predicting sentiment.

Note that the H5 file contains all the weights of the network, and we want to use just the weight matrix from the first layer,[2] and this matrix is fetched by the last line of code and named `embedding_weight_matrix`. We will be using `embedding_weight_matrix` in the code in the next section (which should be in the same file as the code of this section).

[2] If we were to save and load from a H5 file, we would be saving ans loading all the weights in a new network of the same configuration, possibly fine-tuning them and then taking out just the weight matrix with the same code we used here.

9.4 Walking Through the Word-Space: An Idea That Has Eluded Symbolic AI

Word vectors are a very interesting type of word embeddings, since they allow much more than meets the eye. Traditionally, reasoning is viewed as a symbolic concept which ties together various relations of an object or even various relations of various objects. Objects, and symbols denoting them, have been seen as logically primitive. This means that they were defined, and as such void of any content other than that which we explicitly placed in them. This has been a dogma of the logical approach to artificial intelligence (GOFAI) for decades. The main problem is that rationality was equated with intelligence, and this meant that the higher faculties, where the one that embodied intelligence. Hans Moravec [10] discovered that higher faculties (such as chess playing and theorem proving) where in fact easier than recognizing cats on unlabelled photos, and this caused the AI community to rethink the previously accepted concept of intelligence, and with it ideas of *low faculty reasoning* became interesting.

To explain what low faculty reasoning is we turn to an example. If you consider two sentences 'a tomato is a vegetable' and 'a tomato is a suspension bridge', you might conclude that they are both false, and you would technically be right. But most people (and intelligent animals) endorse an idea of fuzziness which takes into account the degree of wrongness. You are less wrong by uttering 'a tomato is a vegetable' than 'a tomato is a suspension bridge'. Note also that these are not sentences of natural phenomena, but sentences about linguistic classification and the social conventions on language use. You are not referring to objects (except for 'tomato'), but to classes defined by descriptions (composed of properties) or examples (which share to a degree a number of common properties). Notice that you are using singular terms in all three cases, and the only symbolic part is '_is a_', which is irrelevant.

If an agent were locked in a room and given only books in a foreign language to read, we would consider her intelligent if she would be able to find patters, such as a word which denote places and words that denote people. So if she would classify two sentences 'Luca frequenta la scuola elementare Pedagna' and 'Marco frequenta la scuola elementare Zolino' as being similar, she would display a certain degree of intelligence. She might even go so far to say that in this context 'Luca' is to 'Pedagna' as 'Marco' is to 'Zolino'. If she was given a new sentence 'Luca vive in Pedagna', she might infer the sentence 'Marco vive in Zolino', and she might hit it spot on. The question of semantically similar terms very quickly became a question of reasoning.

We can actually find similarities of terms in our datasets an even reason with them in this fashion using Word2vec. To see how, let us return to our code. The following code goes immediately after the code from the last section (in the same Python file). We will use the `embedding_weight_matrix` to find an interesting way to measure word similarities (actually word vector clusterings) and to calculate and reason with words with the help of word vectors. To do this, we first run

embedding_weight_matrix through PCA and keep just the first two dimensions,[3] and then simply draw the results to a file:

```
pca = PCA(n_components=2)
pca.fit(embedding_weight_matrix)
results = pca.transform(embedding_weight_matrix)
x = np.transpose(results).tolist()[0]
y = np.transpose(results).tolist()[1]
n = list(word2index.keys())
fig, ax = plt.subplots()
ax.scatter(x, y)
for i, txt in enumerate(n):
    ax.annotate(txt,(x[i],y[i]))
plt.savefig('word_vectors_in_2D_space.png')
plt.show()
```

This produces Fig. 9.2. Note that we need a significantly larger dataset than our nine word sentence to be able to learn similarities (and to see them in the plot), but you can experiment with different datasets using the parser we used with recurrent neural networks.

Reasoning with word vectors is also quite straightforward. We need to take the corresponding vectors from embedding_weight_matrix and do simple arithmetic with them. They are all of the same dimensionality, which means it is quite easy to add and subtract them. Let *w2v(someword)* denote the trained word embedding

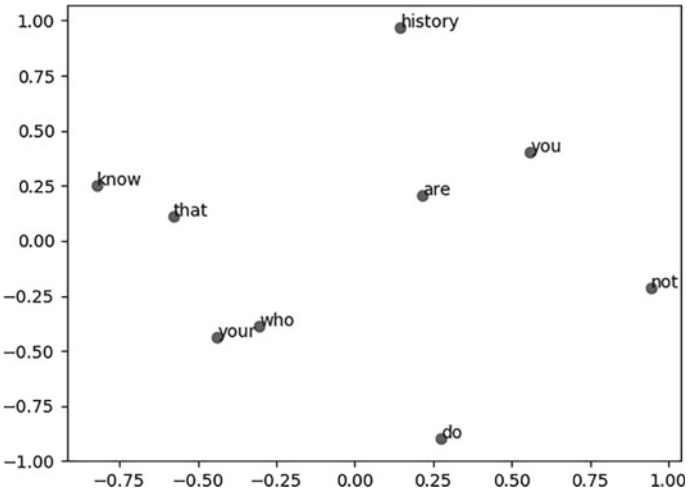

Fig. 9.2 Word similarity clusters in transformed 2D space

[3] More precisely: to transform the matrix into a decorrelated matrix whose columns are arranged in descending variance and then keep the first two columns.

for the word 'someword'. To recreate the classic example, take *w2v(king)*, subtract from it *w2v(man)* add to it *w2v(woman)* and the result would be near *w2v(queen)*. The same holds even if we use PCA to transform the vectors and keep just the first two or three components, although it is sometimes more distorted. This depends on the quality and size of the dataset, and we suggest the reader to try to make a script which does this over a large dataset as an exercise.

References

1. R.W. Hamming, Error detecting and error correcting codes. Bell Syst. Tech. J. **29**(2), 147–160 (1950)
2. V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. **10**(8), 707–710 (1966)
3. M.A. Jaro, Advances in record linkage methodology as applied to the 1985 census of tampa florida. J. Am. Stat. Assoc. **84**(406), 414–420 (1989)
4. W.E. Winkler, String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage, in *Proceedings of the Section on Survey Research Methods* (American Statistical Association, 1990), pp. 354–359
5. A. Singhal, Modern information retrieval: a brief overview. Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. **24**(4), 35–43 (2001)
6. T. Mikolov, T. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in *ICLR Workshop* (2013), arXiv:1301.3781
7. Z. Harris, Distributional structure. Word **10**(23), 146–162 (1954)
8. J.R. Firth, A synopsis of linguistic theory 1930–1955, in *Studies in Linguistic Analysis* (Philological Society, 1957), pp. 1–32
9. L. Wittgenstein, *Philosophical Investigations* (MacMillan Publishing Company, London, 1953)
10. H. Moravec, *Mind Children: The Future of Robot and Human Intelligence* (Harvard University Press, Cambridge, 1988)