C2_W4_lecture_nb_1_data_prep

July 4, 2025

1 Word Embeddings First Steps: Data Preparation

In this series of ungraded notebooks, you'll try out all the individual techniques that you learned about in the lectures. Practicing on small examples will prepare you for the graded assignment, where you will combine the techniques in more advanced ways to create word embeddings from a real-life corpus.

This notebook focuses on data preparation, which is the first step of any machine learning algorithm. It is a very important step because models are only as good as the data they are trained on and the models used require the data to have a particular structure to process it properly.

To get started, import and initialize all the libraries you will need.

[1]:

```python
import re
import nltk

nltk.download('punkt')

import emoji
import numpy as np
from nltk.tokenize import word_tokenize
from utils2 import get_dict
```

```
[nltk_data] Downloading package punkt to /home/jovyan/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

2 Data preparation

In the data preparation phase, starting with a corpus of text, you will:

- Clean and tokenize the corpus.
- Extract the pairs of context words and center word that will make up the training data set for the CBOW model. The context words are the features that will be fed into the model, and the center words are the target values that the model will learn to predict.
- Create simple vector representations of the context words (features) and center words (targets) that can be used by the neural network of the CBOW model.

2.1 Cleaning and tokenization

To demonstrate the cleaning and tokenization process, consider a corpus that contains emojis and various punctuation signs.

[2]:
```
# Define a corpus
corpus = 'Who  "word embeddings" in 2020? I do!!!'
```

First, replace all interrupting punctuation signs — such as commas and exclamation marks — with periods.

[3]:
```
# Print original corpus
print(f'Corpus:  {corpus}')

# Do the substitution
data = re.sub(r'[,!?;-]+', '.', corpus)

# Print cleaned corpus
print(f'After cleaning punctuation:  {data}')
```

```
Corpus:  Who  "word embeddings" in 2020? I do!!!
After cleaning punctuation:  Who  "word embeddings" in 2020. I do.
```

Next, use NLTK's tokenization engine to split the corpus into individual tokens.

[4]:
```
# Print cleaned corpus
print(f'Initial string:  {data}')

# Tokenize the cleaned corpus
data = nltk.word_tokenize(data)

# Print the tokenized version of the corpus
print(f'After tokenization:  {data}')
```

```
Initial string:  Who  "word embeddings" in 2020. I do.
After tokenization:  ['Who', ' ', '``', 'word', 'embeddings', "''", 'in',
'2020', '.', 'I', 'do', '.']
```

Finally, as you saw in the lecture, get rid of numbers and punctuation other than periods, and convert all the remaining tokens to lowercase.

[5]:
```
# Print the tokenized version of the corpus
print(f'Initial list of tokens:  {data}')

# Filter tokenized corpus using list comprehension
data = [ ch.lower() for ch in data
         if ch.isalpha()
         or ch == '.'
         or emoji.get_emoji_regexp().search(ch)
```

```
        ]

# Print the tokenized and filtered version of the corpus
print(f'After cleaning:  {data}')
```

```
Initial list of tokens:  ['Who', ' ', '``', 'word', 'embeddings', "''", 'in',
'2020', '.', 'I', 'do', '.']
After cleaning:  ['who', ' ', 'word', 'embeddings', 'in', '.', 'i', 'do', '.']
```

Note that the heart emoji is considered as a token just like any normal word.

Now let's streamline the cleaning and tokenization process by wrapping the previous steps in a function.

[6]:

```
# Define the 'tokenize' function that will include the steps previously seen
def tokenize(corpus):
    data = re.sub(r'[,!?;-]+', '.', corpus)
    data = nltk.word_tokenize(data)  # tokenize string to words
    data = [ ch.lower() for ch in data
             if ch.isalpha()
             or ch == '.'
             or emoji.get_emoji_regexp().search(ch)
           ]
    return data
```

Apply this function to the corpus that you'll be working on in the rest of this notebook: "I am happy because I am learning"

[7]:

```
# Define new corpus
corpus = 'I am happy because I am learning'

# Print new corpus
print(f'Corpus:  {corpus}')

# Save tokenized version of corpus into 'words' variable
words = tokenize(corpus)

# Print the tokenized version of the corpus
print(f'Words (tokens):  {words}')
```

```
Corpus:  I am happy because I am learning
Words (tokens):  ['i', 'am', 'happy', 'because', 'i', 'am', 'learning']
```

Now try it out yourself with your own sentence.

[8]:

```
# Run this with any sentence
tokenize("Now it's your turn: try with your own sentence!")
```

```
[8]: ['now', 'it', 'your', 'turn', 'try', 'with', 'your', 'own', 'sentence', '.']
```

2.2 Sliding window of words

Now that you have transformed the corpus into a list of clean tokens, you can slide a window of words across this list. For each window you can extract a center word and the context words.

The `get_windows` function in the next cell was introduced in the lecture.

[9]:

```
# Define the 'get_windows' function
def get_windows(words, C):
    i = C
    while i < len(words) - C:
        center_word = words[i]
        context_words = words[(i - C):i] + words[(i+1):(i+C+1)]
        yield context_words, center_word
        i += 1
```

The first argument of this function is a list of words (or tokens). The second argument, `C`, is the context half-size. Recall that for a given center word, the context words are made of `C` words to the left and `C` words to the right of the center word.

Here is how you can use this function to extract context words and center words from a list of tokens. These context and center words will make up the training set that you will use to train the CBOW model.

[10]:

```
# Print 'context_words' and 'center_word' for the new corpus with a 'context␣
 ↪half-size' of 2
for x, y in get_windows(['i', 'am', 'happy', 'because', 'i', 'am', 'learning'],␣
 ↪2):
    print(f'{x}\t{y}')
```

```
['i', 'am', 'because', 'i']     happy
['am', 'happy', 'i', 'am']      because
['happy', 'because', 'am', 'learning']  i
```

The first example of the training set is made of:

- the context words “i”, “am”, “because”, “i”,
- and the center word to be predicted: “happy”.

Now try it out yourself. In the next cell, you can change both the sentence and the context half-size.

[11]:

```
# Print 'context_words' and 'center_word' for any sentence with a 'context␣
 ↪half-size' of 1
for x, y in get_windows(tokenize("Now it's your turn: try with your own␣
 ↪sentence!"), 1):
    print(f'{x}\t{y}')
```

```
['now', 'your'] it
['it', 'turn']  your
```

```
['your', 'try'] turn
['turn', 'with']        try
['try', 'your'] with
['with', 'own'] your
['your', 'sentence']    own
['own', '.']    sentence
```

2.3 Transforming words into vectors for the training set

To finish preparing the training set, you need to transform the context words and center words into vectors.

2.3.1 Mapping words to indices and indices to words

The center words will be represented as one-hot vectors, and the vectors that represent context words are also based on one-hot vectors.

To create one-hot word vectors, you can start by mapping each unique word to a unique integer (or index). We have provided a helper function, `get_dict`, that creates a Python dictionary that maps words to integers and back.

[12]:

```python
# Get 'word2Ind' and 'Ind2word' dictionaries for the tokenized corpus
word2Ind, Ind2word = get_dict(words)
```

Here's the dictionary that maps words to numeric indices.

[13]:

```python
# Print 'word2Ind' dictionary
word2Ind
```

[13]:

```
{'am': 0, 'because': 1, 'happy': 2, 'i': 3, 'learning': 4}
```

You can use this dictionary to get the index of a word.

[14]:

```python
# Print value for the key 'i' within word2Ind dictionary
print("Index of the word 'i':  ",word2Ind['i'])
```

```
Index of the word 'i':   3
```

And conversely, here's the dictionary that maps indices to words.

[15]:

```python
# Print 'Ind2word' dictionary
Ind2word
```

[15]:

```
{0: 'am', 1: 'because', 2: 'happy', 3: 'i', 4: 'learning'}
```

[16]:

```python
# Print value for the key '2' within Ind2word dictionary
print("Word which has index 2:  ",Ind2word[2] )
```

```
Word which has index 2:   happy
```

Finally, get the length of either of these dictionaries to get the size of the vocabulary of your corpus, in other words the number of different words making up the corpus.

[17]:
```
# Save length of word2Ind dictionary into the 'V' variable
V = len(word2Ind)

# Print length of word2Ind dictionary
print("Size of vocabulary: ", V)
```

```
Size of vocabulary:  5
```

2.3.2 Getting one-hot word vectors

Recall from the lecture that you can easily convert an integer, n, into a one-hot vector.

Consider the word "happy". First, retrieve its numeric index.

[18]:
```
# Save index of word 'happy' into the 'n' variable
n = word2Ind['happy']

# Print index of word 'happy'
n
```

[18]:
```
2
```

Now create a vector with the size of the vocabulary, and fill it with zeros.

[19]:
```
# Create vector with the same length as the vocabulary, filled with zeros
center_word_vector = np.zeros(V)

# Print vector
center_word_vector
```

[19]:
```
array([0., 0., 0., 0., 0.])
```

You can confirm that the vector has the right size.

[20]:
```
# Assert that the length of the vector is the same as the size of the vocabulary
len(center_word_vector) == V
```

[20]:
```
True
```

Next, replace the 0 of the n-th element with a 1.

[21]:
```
# Replace element number 'n' with a 1
center_word_vector[n] = 1
```

And you have your one-hot word vector.

[22]:
```
# Print vector
center_word_vector
```

[22]:
```
array([0., 0., 1., 0., 0.])
```

You can now group all of these steps in a convenient function, which takes as parameters: a word to be encoded, a dictionary that maps words to indices, and the size of the vocabulary.

[23]:
```
# Define the 'word_to_one_hot_vector' function that will include the steps␣
 ↪previously seen
def word_to_one_hot_vector(word, word2Ind, V):
    one_hot_vector = np.zeros(V)
    one_hot_vector[word2Ind[word]] = 1
    return one_hot_vector
```

Check that it works as intended.

[24]:
```
# Print output of 'word_to_one_hot_vector' function for word 'happy'
word_to_one_hot_vector('happy', word2Ind, V)
```

[24]:
```
array([0., 0., 1., 0., 0.])
```

What is the word vector for "learning"?

[25]:
```
# Print output of 'word_to_one_hot_vector' function for word 'learning'
word_to_one_hot_vector('learning', word2Ind, V)
```

[25]:
```
array([0., 0., 0., 0., 1.])
```

Expected output:

```
array([0., 0., 0., 0., 1.])
```

2.3.3 Getting context word vectors

To create the vectors that represent context words, you will calculate the average of the one-hot vectors representing the individual words.

Let's start with a list of context words.

[26]:
```
# Define list containing context words
context_words = ['i', 'am', 'because', 'i']
```

Using Python's list comprehension construct and the `word_to_one_hot_vector` function that you created in the previous section, you can create a list of one-hot vectors representing each of the context words.

[27]:
```
# Create one-hot vectors for each context word using list comprehension
context_words_vectors = [word_to_one_hot_vector(w, word2Ind, V) for w in␣
 ↪context_words]

# Print one-hot vectors for each context word
context_words_vectors
```

[27]:
```
[array([0., 0., 0., 1., 0.]),
 array([1., 0., 0., 0., 0.]),
 array([0., 1., 0., 0., 0.]),
 array([0., 0., 0., 1., 0.])]
```

And you can now simply get the average of these vectors using numpy's `mean` function, to get the vector representation of the context words.

[28]:
```
# Compute mean of the vectors using numpy
np.mean(context_words_vectors, axis=0)
```

[28]:
```
array([0.25, 0.25, 0.  , 0.5 , 0.  ])
```

Note the `axis=0` parameter that tells `mean` to calculate the average of the rows (if you had wanted the average of the columns, you would have used `axis=1`).

Now create the `context_words_to_vector` function that takes in a list of context words, a word-to-index dictionary, and a vocabulary size, and outputs the vector representation of the context words.

[29]:
```
# Define the 'context_words_to_vector' function that will include the steps␣
 ↪previously seen
def context_words_to_vector(context_words, word2Ind, V):
    context_words_vectors = [word_to_one_hot_vector(w, word2Ind, V) for w in␣
 ↪context_words]
    context_words_vectors = np.mean(context_words_vectors, axis=0)
    return context_words_vectors
```

And check that you obtain the same output as the manual approach above.

[30]:
```
# Print output of 'context_words_to_vector' function for context words: 'i',␣
 ↪'am', 'because', 'i'
context_words_to_vector(['i', 'am', 'because', 'i'], word2Ind, V)
```

[30]:
```
array([0.25, 0.25, 0.  , 0.5 , 0.  ])
```

What is the vector representation of the context words "am happy i am"?

[31]:
```
# Print output of 'context_words_to_vector' function for context words: 'am',␣
 ↪'happy', 'i', 'am'
context_words_to_vector(['am', 'happy', 'i', 'am'], word2Ind, V)
```

[31]:
```
array([0.5 , 0.  , 0.25, 0.25, 0.  ])
```

Expected output:

```
array([0.5 , 0.  , 0.25, 0.25, 0.  ])
```

2.4 Building the training set

You can now combine the functions that you created in the previous sections, to build a training set for the CBOW model, starting from the following tokenized corpus.

[32]:
```
# Print corpus
words
```

[32]:
```
['i', 'am', 'happy', 'because', 'i', 'am', 'learning']
```

To do this you need to use the sliding window function (`get_windows`) to extract the context words and center words, and you then convert these sets of words into a basic vector representation using `word_to_one_hot_vector` and `context_words_to_vector`.

[33]:
```
# Print vectors associated to center and context words for corpus
for context_words, center_word in get_windows(words, 2):  # reminder: 2 is the context half-size
    print(f'Context words:  {context_words} -> {context_words_to_vector(context_words, word2Ind, V)}')
    print(f'Center word:  {center_word} -> {word_to_one_hot_vector(center_word, word2Ind, V)}')
    print()
```

```
Context words:  ['i', 'am', 'because', 'i'] -> [0.25 0.25 0.   0.5  0.  ]
Center word:  happy -> [0. 0. 1. 0. 0.]

Context words:  ['am', 'happy', 'i', 'am'] -> [0.5  0.   0.25 0.25 0.  ]
Center word:  because -> [0. 1. 0. 0. 0.]

Context words:  ['happy', 'because', 'am', 'learning'] -> [0.25 0.25 0.25 0.
0.25]
Center word:  i -> [0. 0. 0. 1. 0.]
```

In this practice notebook you'll be performing a single iteration of training using a single example, but in this week's assignment you'll train the CBOW model using several iterations and batches of example. Here is how you would use a Python generator function (remember the `yield` keyword from the lecture?) to make it easier to iterate over a set of examples.

[34]:
```
# Define the generator function 'get_training_example'
def get_training_example(words, C, word2Ind, V):
    for context_words, center_word in get_windows(words, C):
        yield context_words_to_vector(context_words, word2Ind, V),␣
↪word_to_one_hot_vector(center_word, word2Ind, V)
```

The output of this function can be iterated on to get successive context word vectors and center word vectors, as demonstrated in the next cell.

[35]:
```
# Print vectors associated to center and context words for corpus using the␣
↪generator function
for context_words_vector, center_word_vector in get_training_example(words, 2,␣
↪word2Ind, V):
    print(f'Context words vector:  {context_words_vector}')
    print(f'Center word vector:  {center_word_vector}')
    print()
```

```
Context words vector:  [0.25 0.25 0.   0.5  0.  ]
Center word vector:  [0. 0. 1. 0. 0.]

Context words vector:  [0.5  0.   0.25 0.25 0.  ]
Center word vector:  [0. 1. 0. 0. 0.]

Context words vector:  [0.25 0.25 0.25 0.   0.25]
Center word vector:  [0. 0. 0. 1. 0.]
```

Your training set is ready, you can now move on to the CBOW model itself which will be covered in the next lecture notebook.

Congratulations on finishing this lecture notebook! Hopefully you now have a better understanding of how to prepare your data before feeding it to a continuous bag-of-words model.

Keep it up!