Deep Learning for Text

2021.7.13

11.1 Natural Language Processing

Using machine learning and large datasets to give computers the ability to ingest a piece of language as input and return something useful, like predicting:

What's the topic of this text: text classification

Does this text contain abuse: content filtering

Does this text sound positive or negative: sentiment analysis

What would be the next word in this incomplete sentence: language modeling

How would you say this in German: translation

How would you summarize this article in one paragraph: summarization

11.2 Preparing the text data

Vectorizing text is the process of transforming text into numeric tensors. Text vectorization processes come in many shapes and form, but they all follow the same template:

First, standarizing the text to make it easier to process, for instance by converting it to lowercase or removing puncation.

Second, spliting the text into units(called tokens), such as characters, words, or groups of words. This is called tokenization.

Third, converting each such token into a numerical vector. This will involve first **indexing** all tokens present in the data.

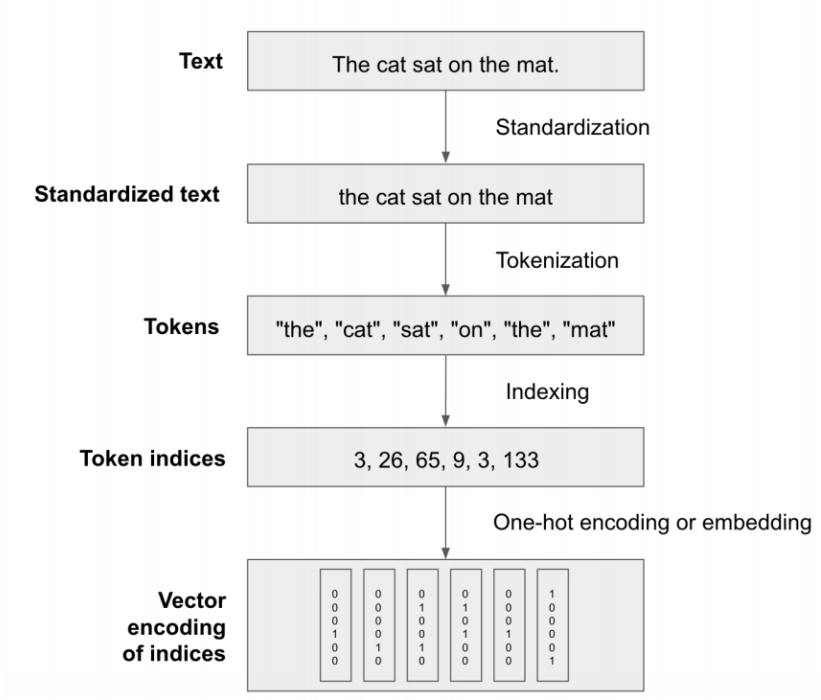

Figure 11.1 From raw text to vectors

11.2.1 Text Standarization

Text standarization is a basic form of feature engineering that aims to erase encoding differences that you don't want model to have to deal with.

One of the simplest and most widespread standardization scheme is "convert to lowercase and remove punctuation characters".

Another common transfer is to convert special characters to a standard form.
A much more advanced standarization pattern, that is more rarely used is **stemming**: converting variations of a term into a single shared represnetations, like turning caught and been catching into [catch].
Of course, standardization may also be erasing some amount of information. For instance, when writing a model that extracts questions from interview articles, it should definitely treat **?** as a separate token instead of dropping it.

11.2.2 Text Spliting (tokenization)

Tokenization in two different ways:

Word-level tokenization: where toknes are space-seperated substrings. A variant of this is to further split words into subword when applicable.

N-gram tokenization: where tokens are group of N consecutive words. For instance, "the cat" or "he was" would be 2-gram tokens.
There are two kinds of text-preprocessing models: sequence model and bag-of-words model. When building a sequence model, using word-level tokenization, and if building a bag-of-words models, using N-gram tokenization.

11.2.3 Vocabulary indexing

Once text is split into tokens, we need to encode each token into a numerical representation. In practice, the way we'd do about it is to build an index of all terms found in the training data, and assign a unique integer to each entry in the vocabulary.

```
vocabulary = {}
for text in dataset:
    text = standardize(text)
    tokens = tokenize(text)
    for token in tokens:
        if token not in vocabulary:
            vocabulary[token] = len(vocabulary)
```

Then, we can convert that integer into a vector encoding that can be processed by a neural network:

```
def one_hot_encode_token(token):
    vector = np.zeros((len(vocabulary),))
    token_index = vocabulary[token]
    vector[token_index] = 1
    return vector
```

To index the vocabulary of a text corpus, just call the *adapt*() method of the layer with a *Dataset* object, that yield strings, or just with on list of Python strings.
The first two entries in the vocabulary are the mask token (index 0) and the OOV token (index 1). Entries in the vocabulary list are sorted by frequency.

11.3 Representing groups of words: sets and sequences

A much more problematic question is how to encode the way ***words are weaved into sentences***: word order
Historically, most early applications of machine learning to NLP just involved bag-of-words models. Interest in sequence models only started rising in 2015, with the rebirth of recurrent nn. Today, both approaches remain relevant. Let's see how they work, and when to leverage which. In this chapter, we will process the raw IMDB text data, just like we would do when approaching a new text classification problem in the real world.

11.3.3 Preprocessing words as a sequence: the Sequence model approach

Clearly, using one-hot encoding to turn words into vectors, which was the simplest thing we could do. There's a better way: *word embeddings*
One-hot encoding, you're injecting into your model a fundamental assumption about the structure of your feature space: *the different tokens you're encoding are all independent from each other*.
And in the case of words, that assumption is clearly wrong. To get a bit more abstract, the *geometric relationship* between two word vectors should reflect the *semantic relationship* between these words. For instance, we should expect **synonyms** to be embedded into similar word vectors, and in general, we would expect the geometric distance between any two word vectors to relate to the "semantic distance" between the associated words. Words that mean different things should lie far away from each other, whereas related words should be closer.

Word embeddings are vector representations of words that achieve this: they map human language into a structured geometric space.

Words embeddings are low-dimension floating-point vectors. Figure 11.2. It's common to see word embeddings that are 256-dim, 512-dim, or 1024-dim when dealing with very large vocabularies.

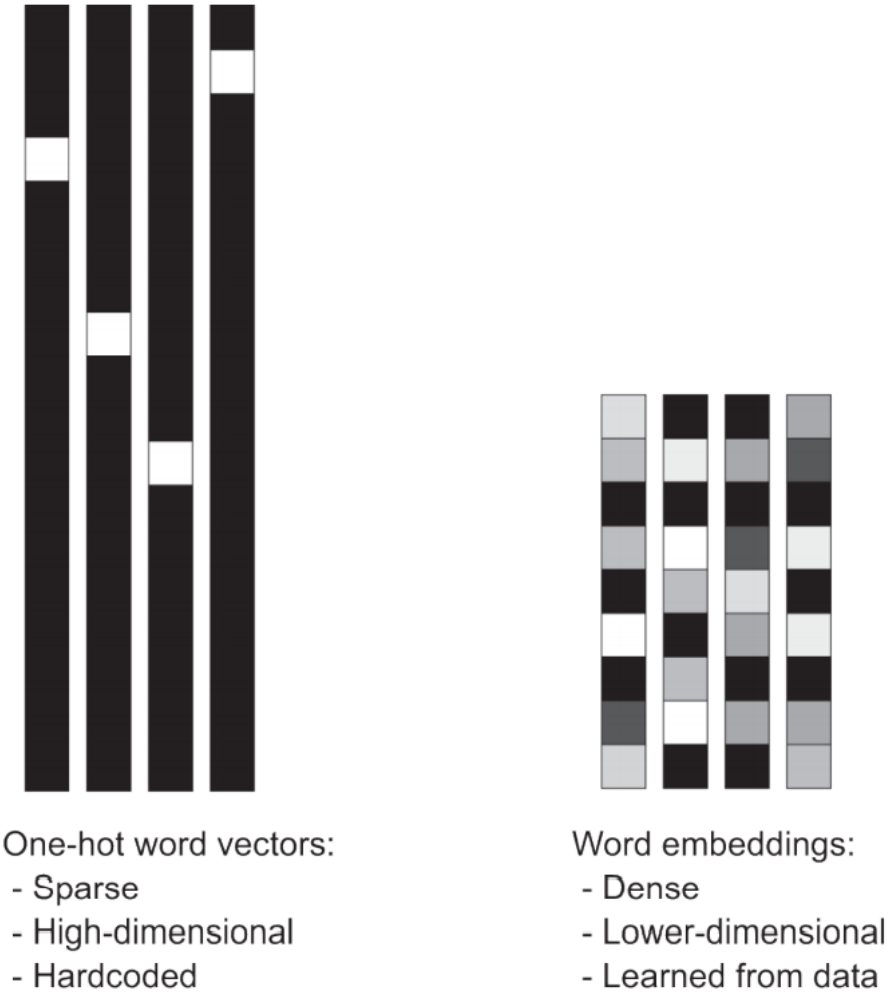

Figure 11.2 Whereas word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded, word embeddings are dense, relatively low-dimensional, and learned from data.

Besides being *dense*, word embeddings are also structured representations, and their structure is learned from data. Similar words get embedded in close location, and further, specific *directions* in the embedding space are meaningful.
In Figure 11.3, four words are embeded on a 2D plane: cat, dog, wolf, and tiger.

With the vector representations we choose here, some semantic relationships between these words can be encoded as geometric transformations. For instance, the same vector allow us to go from cat to *tiger* and from dog to *wolf*. This vector can interpreted as the "from pet to wild animal" vector. Similarly, another vector lets us go from dog to *cat* and from wolf to *tiger*, which could be interpreted as a "from canine to feline" vector.

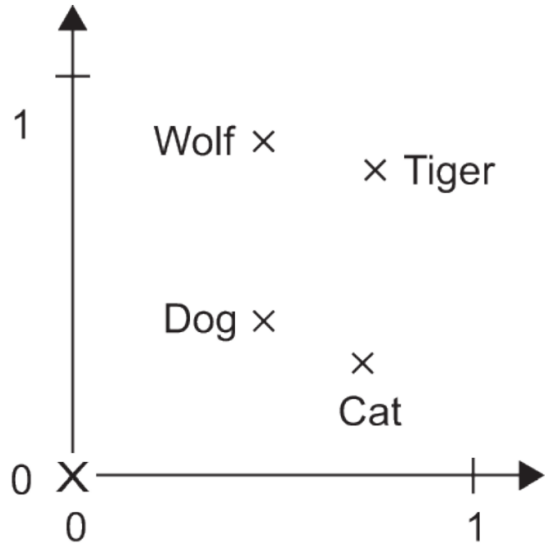

Figure 11.3 A toy example of a word-embedding space

For instance, by adding a "female" vector to the vector "king", we obtain the vector "queen".
There are two ways to obtain word embeddings:
1. Learn word embeddings jointly with the main task we care about (such as document classification or sentiment prediction). In this setup, we start with random word vectors and then learn word vectors in the same way we learn the weights of a neural network.
2. Load into our model word embeddings that were precomputed using a different machine-learning task than the one we're trying to solve. These are called *pretrained word embeddings*.

Let's go back to coding part