Data Science 2

Text Embeddings

Ondřej Týbl

Charles University, Prague

Outline

Network inputs are tensors only.

In computer vision we had a natural encoding using a tensor of size $3 \times H \times W$ that represents RGB pixels.

For text, we do not have an obvious choice for encoding and different techniques with pros and cons have arisen.

Outline

One-hot encoding

Words

A naive approach for text encoding would be to take a whole vocabulary of all existing words $w_1, \ldots, w_N$ in given language and use a *word one-hot encoding*, that is

$$w_j \sim \mathbf{e}_j, \quad j = 1, \ldots, N,$$

where e_j is the j-th basis vector in $\mathbb{R}^N$.

Words

Word one-hot encoding drawback I

We use vectors in $\mathbb{R}^N$, where N is the vocabulary size: tens of thousands for English and hundreds of thousands for Czech.

A network that could process such vectors on input would typically have too many parameters in the first layer (e.g. for a fully connected layer the number of parameters grows linearly with the input size).

Words

Word one-hot encoding drawback II

The output of neural networks changes continuously with the input vector (in the end, it is a continuous mapping).

But with the word one-hot encoding all the input vectors would be equally distant and orthogonal as

$$\|e_i - e_j\| = 1 - \delta_{i,j} \quad \text{and} \quad e_i^T e_j = \delta_{i,j}$$

We would then loose the ability of the network to produce similar results for similar inputs as for example the representations for *dog* and *cat* would differ in the same way as for *dog* and *rocket*.

Words

Word one-hot encoding drawback III

The encoding has no ability to adapt as if a new word (unseen during the architecture design phase) comes. With each new word the whole network architecture has to change.

Characters

The easiest way to deal with the drawback III is to use *Character one-hot encoding*.

Now each word is splitted into characters for which we use one-hot encoding.

For a word of length N in alphabet with K characters, we obtain a representation $a_1, \dots a_N \in \mathbb{R}^K$.

Characters

Now we can represent any word, however, new problems arise

- the representation depends on the length of a word[1],
- there are words with very similar spelling while the meaning is completely different: e.g. *word* and *worm*

[1]small obstacle, some aggregation such as average can reduce the word length dimensionality

Subwords

We may want find a compromise between two extreme cases: words and characters.

We search for a set of *subwords* so that each word can be written as a combination of subwords while keeping this set reasonably small.

We need to specify

- what is the vocabulary of subwords
- how to split a word into subwords (usually more then one option is available)

Subwords

Algorithm (Byte Pair Encoding, see [5])

```
1: Initialize vocabulary $S = (s_1, \dots, s_{n_0})$ with all characters and a special start-of-word symbol
2: Set target vocabulary size $N > |S|$
3: Split training text into characters: $s_1, \dots, s_{n_0}$
4: while $|S| < N$ do
5:     Find the most frequent consecutive pair $(s_i, s_j)$ in the training text for which the start-of-word symbol is not inside
6:     Add new subword $s_i s_j$ to $S$
7:     Replace all occurrences of $(s_i, s_j)$ in the training text with $s_i s_j$
8: end while
```

Subwords

For example, for a text *low lowest newer wider* we would begin with

$$S = \{\mathrm{l}, \mathrm{o}, \mathrm{w}, \mathrm{e}, \mathrm{s}, \mathrm{t}, \mathrm{n}, \mathrm{r}, \mathrm{i}, \mathrm{d}, \cdot\}$$

Then we would add

$$\mathrm{r}\cdot, \quad \mathrm{er}\cdot, \quad \mathrm{lo}, \quad \mathrm{low}, \quad \ldots$$

Subwords

Suppose we have a network that was trained to process natural language with a training set that has been splitted into subwords $(s_1, \dots, s_n)$.

Now in an inference mode a new word w from test set comes. First, we need to split it into a subword $s_{i_1}, \dots, s_{i_k}$.

But for consistency between train and test data, we want our split to be as similar to the train split as possible. **This is not achieved by a simple iteration over subwords**.

Algorithm (Byte Pair Encoding for inference, see [5])

```
    We have a vocabulary $S = (s_1, \ldots, s_N)$ with subwords ordered as
    they have been added during the training. We have a word $w =$
    $s_{i_1}, \ldots s_{i_k}$.[2].
2:  for $s \in S$ do
        if $s = s_{i_a} s_{i_{a+1}} \ldots s_{i_{a+|s|}}$ for some $a$ then
4:          replace $s_{i_a}, s_{i_{a+1}}, \ldots, s_{i_{a+|s|}}$ by $s$
        end if
6:  end for
```

In this way, w is splitted using the same rules as if it was part of the train set.

[2]We can always write w using some subwords as S contains all characters

Subwords

- For example, GPT-4 model uses variant with $|S| = 100256$ subwords.
- Some literature refers to subwords by *tokens*.
- Byte Pair Encoding is very simple, yet currently dominant in natural language processing[3].

[3] Previously, *WordPiece* from [6] was the dominating method. It compares frequency of each tuples $p_{(s_i, s_j)}$ to independent sampling $p_{s_i} p_{s_j}$

Supervised Encodings

Supervised Encodings

Idea: for each word w, its encoding $a_w \in \mathbb{R}^K$ could represent some features.

Example: for an animal-related problem we set features

- number of legs,
- indicator if its a pet,
- number of hours it sleeps each day on average.

Then we would obtain

- $dog \sim (4, 1, 12)$,
- $cat \sim (4, 1, 14)$,
- $snake \sim (0, 0, 16)$
- ...

Supervised Encodings

Then each word is represented with vector of small, fixed size. Even new animals or objects can be encoded and similar animals have similar encodings (if the set of features is descriptive enough).

However, this is very difficult and often requires manual assignments.

We thus aim for a *data-based* method that would do the job for us.

Supervised Encodings

Suppose a network that takes as an input sequences word encoded by one-hot encoding, each of them processed by a (shared) fully connected layer

$$\mathbb{R}^{L \times N} \ni (x_1, \dots, x_L) \mapsto (Wx_1, \dots, Wx_L) \in \mathbb{R}^{L \times d}$$

where L is the sequence length, N is the vocabulary size and $W \in \mathbb{R}^{d \times N}$ is so called *embedding matrix* of dimension d.

We shall then add some layers and create a network for some supervised task, i.e. sentiment of the sentence, and obtain optimal weights W for this task.

Supervised Encodings

Then in fact the *j*th column of *W* can be thought of as a feature vector for a *j*th word in our vocabulary. We call *W* an *embedding* matrix.

By training the network, we thus jointly define and find the features implicitly. We obtain features which cannot be interpreted, however, are descriptive[4].

[4]With respect to the supervised task on which the network has been trained.

Supervised Encodings

We initially argued that the vocabulary size N is too large and thus W would have too many parameters.

However,

- we train W just once and then re-use it for other tasks,
- we always multiply W by the basis vectors e_j only and the product We_j can be implemented efficiently as it is in fact just indexing

Supervised Encodings

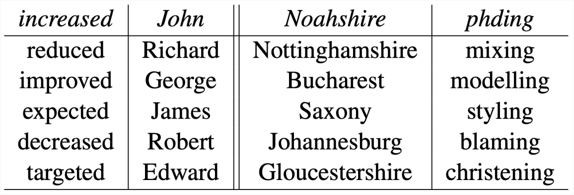
Our embedding matrix W then solves the first two problems of word one-hot encoding:

- too high dimensionality ✓
- similarity for similar words ✓
- ability to adapt for new word

We still have to solve the adaptation capability though.

Supervised Encodings

Solution: create also an character-level embeddig $W_{\text{character}}$ matrix using the same procedure as for the word embedding W and then combine these; if a word w is missing in W then we use a aggregation of the character embeddings $W_{\text{character}}$.



increased	*John*	*Noahshire*	*phding*
reduced	Richard	Nottinghamshire	mixing
improved	George	Bucharest	modelling
expected	James	Saxony	styling
decreased	Robert	Johannesburg	blaming
targeted	Edward	Gloucestershire	christening

Figure 1: Most-similar words: the two query words on the left are in the training vocabulary (word embedding used), those on the right are not (character embedding used). Taken from [3].

Supervised Encodings

That is, we prefer a word embedding if available as for example words like *are* has a meaning completely unrelated to the characters *a, r, e*.

Self-Supervised Encodings

Self-Supervised Encodings

It is difficult to get enough data to train embeddings W with supervised encodings.

Methods where our loss function does not depend on manually labeled data is desirable.

Self-Supervised Encodings

Distributional hypothesis (see [2])

You shall know a word by a company it keeps.

Self-Supervised Encodings

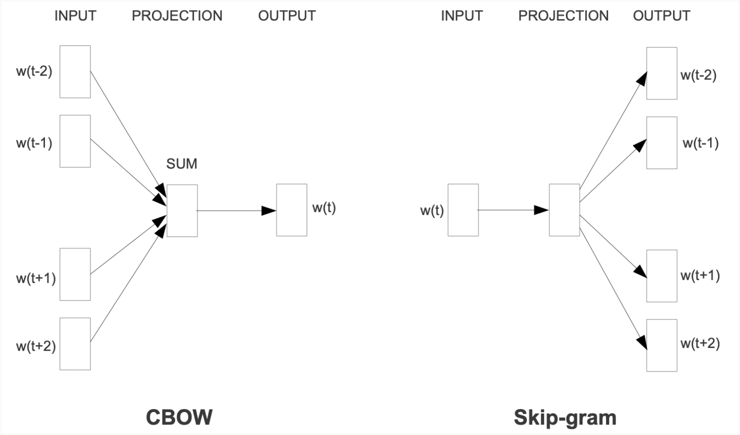


Figure 2: The Continuous Bag of Words (CBOW) architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word. Taken from [4].

Self-Supervised Encodings

Simple architecture of CBOW takes 4 words with their word one-hot encoding, embedds using a common matrix $W \in \mathbb{R}^{d \times N}$, sums[5] and then multiplies by V^T, where $V \in \mathbb{R}^{d \times N}$ and compares with the word one-hot encoding of the middle word.

Fully linear model and therefore can be written efficiently to process enormous texts.

[5] That is, the word order is lost

Self-Supervised Encodings

Skip-gram works in the opposite direction: for each word we predict four words from its neighborhood[6].

Skip-gram is corresponds to a model for conditional probability of *i*th word given the *j*th word from the vocabulary

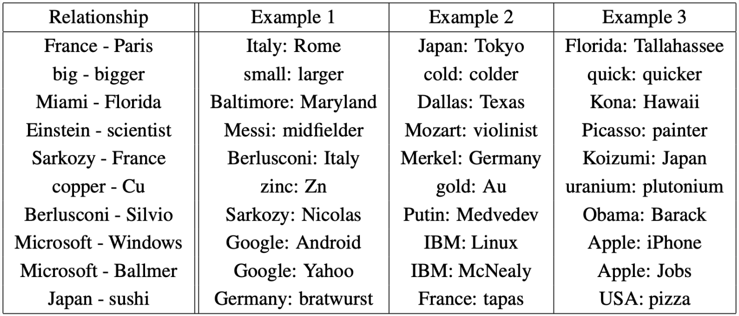
$$p\left(i|j\right)=\frac{e^{V_j^T W_i}}{\sum_{k=1}^{N} e^{V_j^T W_k}} \tag{1}$$

where $W, V \in \mathbb{R}^{d\times N}$. Then the matrix V is the final embedding.[7]

[6]Usually works better if the text size is small as each word generates four training examples.

[7]One could use also *W* and the results would be similar.

Self-Supervised Encodings



Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Figure 3: Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality). Taken from [4].

Self-Supervised Encodings

The key to the efficiency is that in the denominator of (2) we do not sum over all N words but we use *negative sampling instead*.

That is, instead of a classification into N classes, we randomly choose $N' < N$ classes and do binary classifications (in practice, we take one positive example and K negative ones). That is, we have a model

$$p\left(i|j\right) = \frac{e^{V_j^T W_i}}{e^{V_j^T W_i} + \sum_{k=1}^{K} e^{V_j^T W_{i_k}}} \tag{2}$$

for randomly chosen $i_1, \ldots i_K$ indices[8].

[8]A rigorous (and technical) description would involve expectation over empirical measure in the denominator.

Self-Supervised Encodings

In practice, even $K = 2$ is enough if the text is large enough.

In (2) we can either sample K negative samples using uniform distribution over the vocabulary, or we can use frequencies of the words from the training text (so-called *unigram probability P*). However, it has been empirically shown that the probability $P^{3/4}$ works the best[9].

[9] That is, we take the unigram distribution, take powers of all probabilities and then re-normalize

Self-Supervised Encodings

Again, only words seen during training can be embedded using *W*. However, due to the fast implementation and no labels needed we can take huge texts that cover most of the words.

We therefore also train so-called *n-gram* embeddings, where we process the same large text dataset using again Skip-gram (or CBOW) but now instead of words, we input character sequences of some length range (typically 3 to 6).

The final embedding for given word is then a sum of its word embedding and embeddings of all its character subsequences of given lengths.

Self-Supervised Encodings

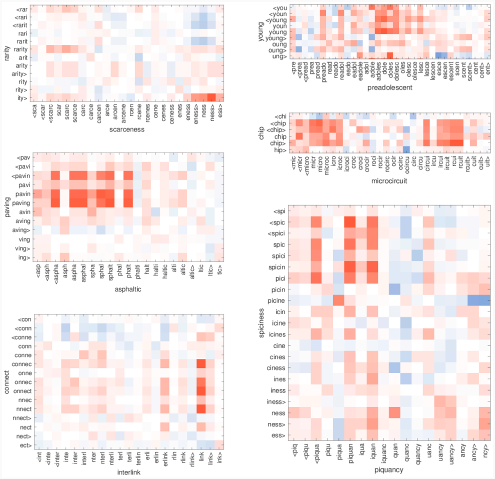


Figure 4: Illustration of the similarity between character n-grams in out-of-vocabulary words. For each pair, only one word is OOV, and is shown on the xaxis. Red indicates positive cosine similarity, while blue negative. Taken from [1].

References i

P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov.
Enriching word vectors with subword information.
Transactions of the association for computational linguistics, 5:135–146, 2017.

J. Firth.
A synopsis of linguistic theory, 1930-1955.
Studies in linguistic analysis, pages 10–32, 1957.

W. Ling, T. Luís, L. Marujo, R. F. Astudillo, S. Amir, C. Dyer, A. W. Black, and I. Trancoso.
Finding function in form: Compositional character models for open vocabulary word representation.
arXiv preprint arXiv:1508.02096, 2015.

References ii

- T. Mikolov, K. Chen, G. Corrado, and J. Dean.
  Efficient estimation of word representations in vector space.
  arXiv preprint arXiv:1301.3781, 2013.

- R. Sennrich, B. Haddow, and A. Birch.
  Neural machine translation of rare words with subword units.
  arXiv preprint arXiv:1508.07909, 2015.

- Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al.
  Google's neural machine translation system: Bridging the gap between human and machine translation.
  arXiv preprint arXiv:1609.08144, 2016.