Deep Learning for Natural Language Processing

Introduction to embeddings

UNIVERSITY OF GOTHENBURG

CHALMERS

WASP WALLENBERG AI, AUTONOMOUS SYSTEMS AND SOFTWARE PROGRAM

Richard Johansson

richard.johansson@gu.se

drawbacks of one-hot encoding

- in NLP, we deal with **discrete-valued** features all the time
  - most notably, words
- we previously saw how to **one-hot encode** discrete-valued features

2739 ("cucumber") $\longrightarrow$ [0, 0, ..., 1, ..., 0, 0]

- but this has some drawbacks
  - vocabularies are large
  - large vectors with many zeros $\rightarrow$ lots of useless computation
  - words are "atomic"

embedding layers

- in a neural network, an **embedding layer** represents a symbol (coded as an integer) as a continuous vector

2739 ("cucumber") ⟶ [0.7, -1.2, ..., -0.1]

- note: an embedding layer is mathematically equivalent to a one-hot encoding followed by a linear layer
  - but in practice, implemented as a lookup table
  - computationally much more efficient

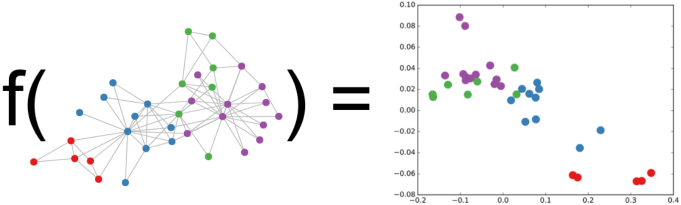
$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 8 & 2 & 1 & 9 \\ 6 & 5 & 4 & 0 \\ 7 & 1 & 6 & 2 \\ 1 & 3 & 5 & 8 \\ 0 & 4 & 9 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 & 8 \end{bmatrix}$$

One-hot vector

Embedding Weight Matrix

why "embedding"?

- in mathematics, an *embedding* is a **structure-preserving** function
- for instance: *embedding a graph* in a vector space
  - e.g. distances in vector space equal to weights in the graph



[source]

embeddings in ML generally and in NLP

- in ML generally, embeddings are used to represent discrete-valued features
- examples of symbols in NLP:
  - words
  - word combinations: bigrams, trigrams, ...
  - pieces of words
  - characters
  - linguistic symbols: phonemes, PoS tags, phrase types, ...

training embeddings in NLP

- there are two main approaches to training embeddings:
  - **end-to-end training**: we learn specialized embeddings in tandem with all other parts of the model
  - **pre-training**: we learn generally useful embeddings that we can "plug" into different tasks
- for the moment, we will consider the first approach

embeddings in PyTorch

- `torch.nn.Embedding`
- need to specify vocabulary size and dimension
- maps a N-dimensional `LongTensor` to a $N+1$-dimensional `FloatTensor`

embeddings in PyTorch, example

```
In [1]: import torch
        from torch import nn
```

```
In [2]: emb = nn.Embedding(8, 3)

        emb.weight
```

```
Out[2]: Parameter containing:
        tensor([[ 0.0733,  0.9253,  0.6466],
                [ 1.9724,  0.9734, -0.7239],
                [-0.4438, -0.2104,  1.6736],
                [-1.0462, -0.4576,  1.2713],
                [-1.5249,  0.8861,  0.9804],
                [-2.0849,  1.5713, -0.0222],
                [ 0.7230,  0.5150, -1.1710],
                [-1.6432, -1.3719, -1.1472]], requires_grad=True)
```

```
In [3]: docs = torch.LongTensor([[1, 5], [7, 5]])

        docs
```

```
Out[3]: tensor([[1, 5],
                [7, 5]])
```

```
In [4]: emb(docs)
```

```
Out[4]: tensor([[[ 1.9724,  0.9734, -0.7239],
                 [-2.0849,  1.5713, -0.0222]],

                [[-1.6432, -1.3719, -1.1472],
                 [-2.0849,  1.5713, -0.0222]]], grad_fn=<EmbeddingBackward>)
```

so what should we use the embeddings for?

- so far, we haven't said **how to use** the embeddings and how they are **trained**
- next, let's return to the topic of **categorization** and see how embeddings can be used and trained