Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

08 - Text Processing

François Pitié

Ussher Assistant Professor in Media Signal Processing
Department of Electronic & Electrical Engineering, Trinity College Dublin

[4C16] Deep Learning and its Applications — 2018/2019

At the core of Neural Networks is the use of linear algebra to combine feature vectors. Using Neural Networks for text processing is therefore problematic as text is a sequence of words and symbols and not a vector of real numbers.

In this handout we will look at how text can be represented with real-valued **word vectors**. This transformation is a key step in using machine learning for **natural language processing** and is called encoding or embedding.

One-Hot

The simplest word vector is the **one-hot** vector. This is the same kind of encoding that is found in digital circuits to indicate the state of a state machine. Each word is represented as a vector with all 0's and one 1 at the index of that word in the application dictionary.

Example

Consider the following alphabetically sorted dictionary: `{a, cat, is, sitting}`. The dictionary size is 4 and the word vector sizes are 4×1. The one-hot encodings for `cat` and `sitting` could be as follows:

$$\mathbf{x}_{\texttt{cat}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{x}_{\texttt{sitting}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Note that one-hot encoding can be used to encode more than just words and can be used to encode sequences of any kind of symbols.

Example

For instance say we want to encode the 26 letters from the alphabet. Each character can be represented as follows:

$$\mathbf{x}_a = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{x}_b = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}, \quad \mathbf{x}_c = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \cdots$$

The problem with one-hot word vectors is that there are about 200,000 words in current use in the English language, which altogether create about 13 million different tokens. One word (*e.g.* `have`) can have many tokens (*e.g.*: `having`, `had`, `has`, `'s`). Word vectors generated by one-hot would be embedded in a vector space of dimension 13 millions!

In Neural Nets, this would mean using input vectors of dimension 13 millions and thus would require training a huge amount of weights.

We need to find a more compact representation.

Another problem is that, in Machine Learning, we assume that the input vectors live in an **Euclidean vector space**. That is:

1. not only we need to define a vector representation,
2. but also we need to be able to add these vectors,

   e.g. $\mathbf{x_1} + 3\mathbf{x_2} - 0.2\mathbf{x_3}$ should make some sense.

3. and finally we need to be able to compare vectors with an inner product: $\mathbf{x_1}^\top \mathbf{x_2}$. In fact, text similarity is usually done through the derived cosine similarity metric:

   $$\textbf{cosine similarity} = \frac{\mathbf{x_1}^\top \mathbf{x_2}}{\|\mathbf{x_1}\| \|\mathbf{x_2}\|}$$

   A cosine similarity of +1 means that both words are similar and a similarity of -1 means that the words are very dissimilar.

Let's look at the word vectors we have defined with one-hot. In theory we could add the word vectors as follows:

$$2\mathbf{x}_{\texttt{cat}} + 3.2\mathbf{x}_{\texttt{sitting}} - 1.5\mathbf{x}_{\texttt{is}}$$

but we will end up with vectors that are difficult to interpret. For instance, what would be the meaning of

$$2\mathbf{x}_{\texttt{cat}} + 3.2\mathbf{x}_{\texttt{sitting}} - 1.5\mathbf{x}_{\texttt{is}} = \begin{bmatrix} 0 \\ 2 \\ -1.5 \\ 3.2 \end{bmatrix} = \mathbf{x}_{?}$$

Similarly, comparing one-hot word vectors using the inner product doesn't seem to make much sense.

In fact, by definition of one-hot encoding, for any two distinct dictionary entries $\mathbf{x}_1$ and $\mathbf{x}_2$, and regardless of their meaning, we have $\mathbf{x}_1^\mathsf{T}\mathbf{x}_2 = 0$ and thus $\textbf{cosine similarly}(\mathbf{x}_1, \mathbf{x}_2) = 0$.

We could do with a better word encoding.

In the following, we will look at 3 word encoding techniques: **SVD**, **GloVe** (2014) and **Word2Vec** (2013). All of these techniques rely on exploiting the co-occurrence statistics of words.

[GloVe] GloVe: Global Vectors for Word Representation.
J. Pennington, R. Socher, C. Manning., 2014 [https://goo.gl/8WFNkx]

[Word2Vec] Efficient Estimation of Word Representations in Vector Space.
T. Mikolov, K. Chen, G. Corrado, J. Dean, 2013 [https://arxiv.org/abs/1301.3781]

[SVD] Blog Post on SVD: “Stop Using word2vec”
C. Moody, 2017 [https://goo.gl/k2QV4G]

Context Window

Consider the sentence below:

a cat is sitting on a suitcase

For the word `sitting`, its surrounding words (`cat`, `is`, `on`, `a`) define its context window.

Research in word encoding is interested in the co-occurence statistics, that is looking at the frequencies of particular word appearing in the context window of a particular word (*e.g.* what is the frequency to have `cat` and `sitting` in the same context window).

Consider the following corpus:

A cat is sitting on a coach.
A person is sitting on a chair.
We are all is sitting on a sofa.

Paris is the capital of France.
Rome is the capital of Italy
Dublin is the capital of Ireland

I want to cook.
They wish to cook.

The words `coach`, `chair`, `sofa` have similar contexts, so have `Paris`, `Rome` and `Dublin`, and so have `want` and `wish`.

It is striking that words that are used in similar contexts have similar functions and their vector representations should probably be close in the Euclidean word vector space.

So let us combine all this and try to define word vectors $\mathbf{x}_i$ by how their co-occurrence statistics in a context window.

Denote as $\mathbf{x}_i$ the word vector for the dictionary entry i.

We define $p_{i,j}$ as the probability (frequency) that both $\mathbf{x}_i$ and $\mathbf{x}_j$ co-occur in a context window.

Alongside the word vector $\mathbf{x}_i$, we also define a **context vector** $\tilde{\mathbf{x}}_j$, which encodes the context of word j. The idea is that this vector encodes all the information about the co-occurrence frequencies for that word.

The argument is that word vectors $\mathbf{x}_i$ and context vectors $\tilde{\mathbf{x}}_j$ should be defined so as to satisfy the following relationship:

$$\mathbf{x}_i^\top \tilde{\mathbf{x}}_j = p_{i,j} \tag{1}$$

This is a bit abstract but, using a one-hot encoding for the word vectors, we can easily see how we could define suitable context vectors:

$$\mathbf{x}_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \tilde{\mathbf{x}}_j = \begin{bmatrix} p_{1,j} \\ \vdots \\ p_{i,j} \\ \vdots \\ p_{n,j} \end{bmatrix}$$

then we would have indeed $\mathbf{x}_i^\top \tilde{\mathbf{x}}_j = p_{i,j}$

Of course this is still a one-hot word encoding and we need to go further.

What we need to do is to reduce the dimension of the word and context vectors from the dictionary size n to a smaller embedding size r, whilst still preserving the relationship of equation (1).

One way of doing this is to use Low Rank Matrix Factorisation.

Denote as $\mathbf{P}$ the matrix of all co-occurence frequencies, $\mathbf{W}$ the matrix stacking all word vectors $\mathbf{W} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n]^\mathsf{T}$ and $\widetilde{\mathbf{W}}$ the matrix stacking all context vectors $\widetilde{\mathbf{W}} = [\widetilde{\mathbf{w}}_1, \widetilde{\mathbf{w}}_2, \cdots, \widetilde{\mathbf{w}}_n]^\mathsf{T}$.

We can rewrite eq. (1) in a matrix form as:

$$\mathbf{P} = \mathbf{W}\widetilde{\mathbf{W}}^\mathsf{T}$$

The matrix factorisation aims at finding $\mathbf{W}$ and $\widetilde{\mathbf{W}}$ such that both matrices $\mathbf{W}$ and $\widetilde{\mathbf{W}}$ are of size $n \times r$, with $r \ll n$.

One way of achieving this matrix factorisation is to use the singular value decomposition (SVD), which factorises the matrix $\mathbf{P}$ as follows:

$$\mathbf{P} = \mathbf{U}\Sigma\mathbf{V}^{\mathsf{T}}$$

with $\mathbf{U}$ and $\mathbf{V}$ orthogonal matrices and Σ a diagonal matrix containing the eigenvalues of $\mathbf{P}$. By keeping the r largest values (the other singular values are replaced by zero), we can get a rank r approximation.

$$\mathbf{W} = \begin{pmatrix} \mathbf{u}_{1,1} & \cdots & \mathbf{u}_{1,r} \\ \vdots & & \vdots \\ \mathbf{u}_{n,1} & \cdots & \mathbf{u}_{n,r} \end{pmatrix} \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{pmatrix}, \quad \widetilde{\mathbf{W}} = \begin{pmatrix} \mathbf{v}_{1,1} & \cdots & \mathbf{v}_{1,r} \\ \vdots & & \vdots \\ \mathbf{v}_{n,1} & \cdots & \mathbf{v}_{n,r} \end{pmatrix}$$

Recent work has shown that instead of directly factorising the co-occurrence matrix, it is a better idea to factorise some of its variants.

In particular, a good alternative is to factorise the Positive Pointwise Mutual Information (ppmi) matrix. The model becomes:

$$\mathbf{x}_i^\top \tilde{\mathbf{x}}_j = \text{PPMI}(\mathbf{x}_i, \mathbf{x}_j) = \max\left(\log \frac{p(\mathbf{x}_i, \mathbf{x}_j)}{p(\mathbf{x}_i)p(\mathbf{x}_j)}, 0\right)$$

There is evidence/discussions that this is quite close to what Word2Vec actually optimises.

In Glove, the authors propose to factorise the log of the co-occurence probabilities:

$$\mathbf{x}_i^\top \tilde{\mathbf{x}}_j = \log(p(\mathbf{x}_i, \mathbf{x}_j))$$

GloVe

Let's look at some of the results. In GloVe, the nearest neighbours of the word vector for `frog` are:

1. `toad`

2. `Litoria`

3. `Leptodactylidae`

4. `Rana`

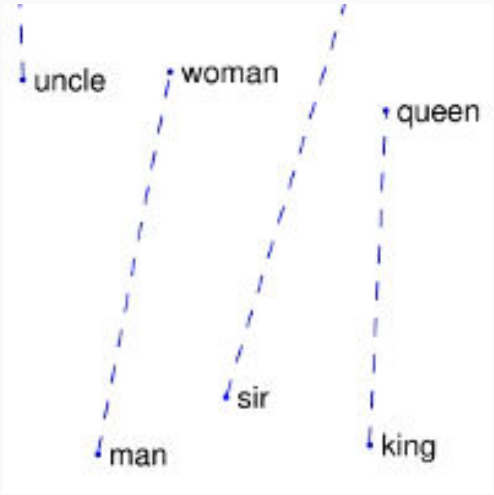
5. `lizard`

6. `Eleutherodactylus`

...and this is only based on text analysis, not image processing!

GloVe

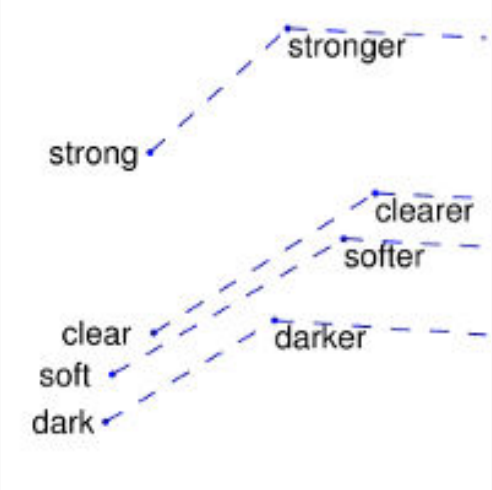
Interestingly, this embedding is also able to model some linear substructures. Below are the projections of some gender related GloVe word vectors:



$$\begin{aligned}\mathbf{x}_{\text{man}} - \mathbf{x}_{\text{woman}} &\approx \mathbf{x}_{\text{sir}} - \mathbf{x}_{\text{madam}} \\ &\approx \mathbf{x}_{\text{uncle}} - \mathbf{x}_{\text{aunt}} \\ &\approx \mathbf{x}_{\text{king}} - \mathbf{x}_{\text{queen}}\end{aligned}$$

GloVe

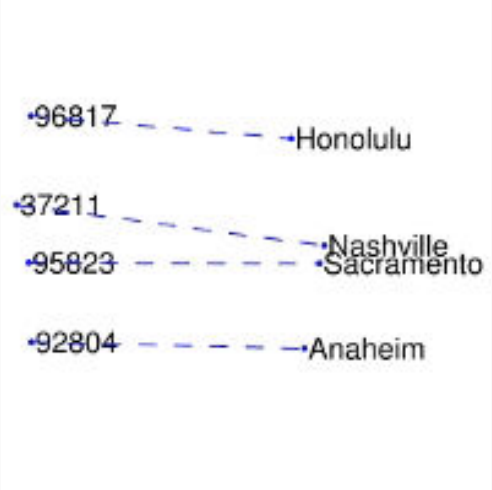
Below are the projections of some of the GloVe word vectors for comparative and superlatives:



$$\mathbf{x}_{\text{strong}} - \mathbf{x}_{\text{stronger}} \approx \mathbf{x}_{\text{clear}} - \mathbf{x}_{\text{clearer}}$$
$$\approx \mathbf{x}_{\text{soft}} - \mathbf{x}_{\text{softer}}$$
$$\approx \mathbf{x}_{\text{dark}} - \mathbf{x}_{\text{darker}}$$

GloVe

keeping in mind that numbers are also words, below are the projections of some of the word vectors for zip codes and city names:



$$\begin{aligned} \mathbf{x}_{\text{Anaheim}} - \mathbf{x}_{92804} &\approx \mathbf{x}_{\text{Honolulu}} - \mathbf{x}_{96817} \\ &\approx \mathbf{x}_{\text{nashville}} - \mathbf{x}_{37211} \\ &\approx \mathbf{x}_{\text{Sacramento}} - \mathbf{x}_{95823} \end{aligned}$$

Word2Vec

Word2Vec (2013) is the first neural embedding model which gained popularity. It is still used by many researchers and has been the basis of a number of derived works.

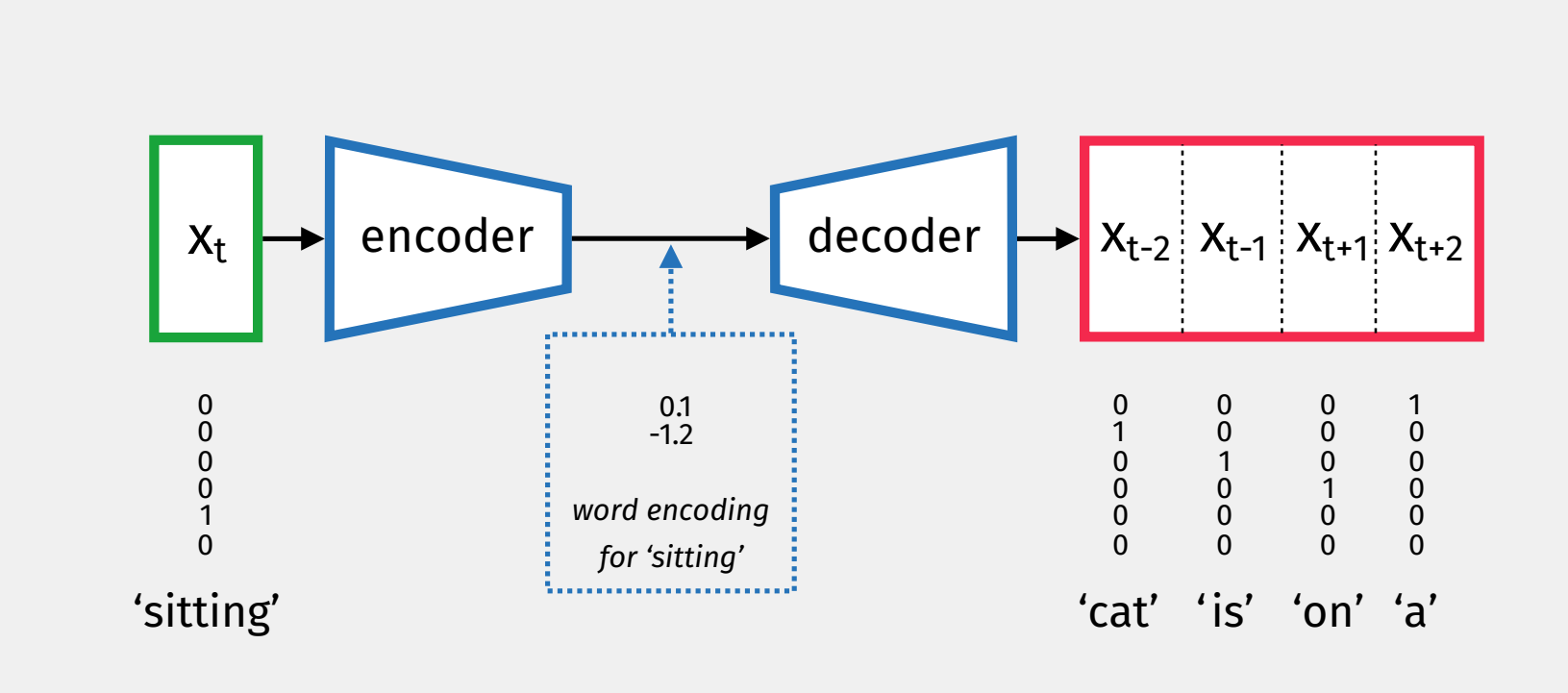
The paper actually proposes two models, one based on using the context to predict a target word (a method known as continuous bag of words, or CBOW), and the other model, called skip-gram, tries to predict a target context from a target word. We use the latter method because it produces more accurate results on large datasets.

Word2Vec: Skipgram

Recall that the context refers to the neighbouring words. If we are using a 5 words window and say the target word is indexed $\mathbf{x}_t$, then the corresponding context could be made of $\mathbf{x}_{t-2}$, $\mathbf{x}_{t-1}$, $\mathbf{x}_{t+1}$ and $\mathbf{x}_{t+2}$.

In the skipgram approach of Word2Vec, the input is $\mathbf{x}_t$ and we are trying to predict the 4 vectors $[\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}]$.

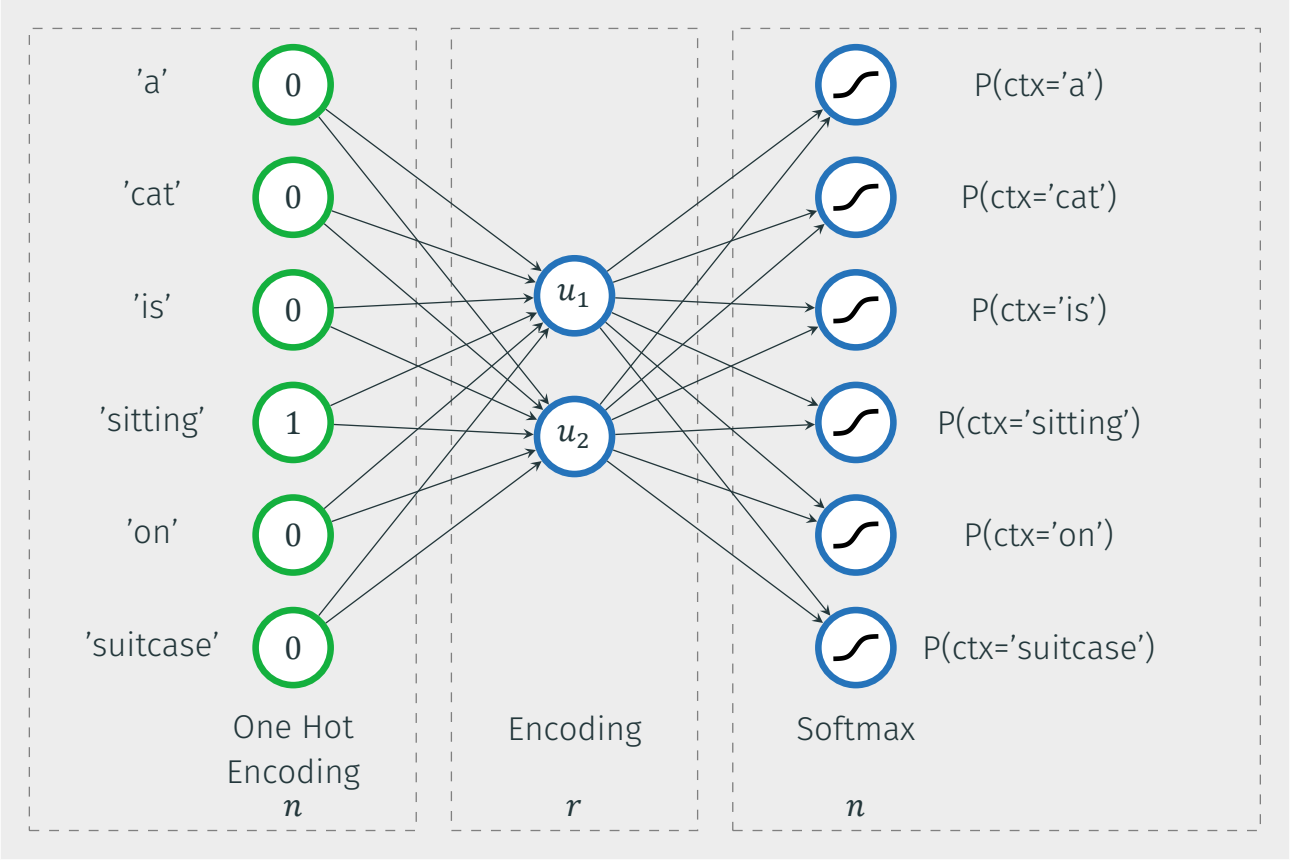
Word2Vec



The model follows an encoder-decoder architecture. If the input size is n (the size of the dictionary), the output of the encoder is of size r.

Word2Vec

In detail, the encoder-decoder networks are very simple. Below is an example for a single context word prediction. The encoder only consists of a single dense layer of size r with linear activations. The decoding layer is simply a softmax layer of size n.



Word2Vec

In practice this is a hard problem to solve as the size of the dictionary is still very high. In particular the last layer which gives us the actual context words predictions uses softmax functions but **the softmax functions require the use of exponentials for potentially millions of entries**. This is a numerical challenge (see original paper for discussion about this).

Example: Text Analysis

Below is described an example of how to do text classification using pre-trained word embeddings and a convolutional neural network in Keras.

https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html

Character vs Word Embedding

When developing an application, we have the choice of starting from a one-hot character encoding or a word encoding such as word2vec. It is still an open question whether one approach is better than the other.

Starting with word encoding can give higher accuracy at a lower computational cost than with character encoding. But with more data and longer training times, character encoding seem to eventually outperform word based encoding. Also, character encoding can be applied to a wider variety of text material and languages. For instance word encoding will not work well on morphologically rich languages such as Finish, Turkish or Russian.