50.039 – Theory and practice of deep learning

Alex

Week 08: Word Embeddings

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Key takeaways

- be able to explain how beam search works
- be able to explain how the training Continuous bag of words works
- be able to explain how the training of skip-gram models works
- be able to explain the rough structure of ELMO word embeddings
- be able to recapitulate the usage of *prince/princess = man/woman* analogies to measure similarities in word embeddings

1 an important tool for decoding: Beam search

Setting: generating a sequence by feeding into RNN the last output x_{t-1} and then as the softmax obtaining $P(x_t|x_{t-1}, x_{t-2}, \ldots, x_3, x_2, x_1)$

Remember: the softmax at step t encodes $P(x_t|x_{t-1}, x_{t-2}, \ldots, x_3, x_2, x_1)$
When creating a sequence, one can choose at step t merely the output word x_t with the highest probability: $\hat{x}_t = \text{argmax}_z P(x_t = z|x_{t-1}, x_{t-2}, \ldots, x_3, x_2, x_1)$

This creates one sequence. What can go wrong? Early high probability candidates can result in end sequences with low probabilities: $0.9, 0.9, 0.1, 0.1, 0.1$ (for later sequence elements all choices have low probabilities, no peak)

alternative: Beam search with beam width n – maintain the greedy best n candidates

- at step t keep track of n candidate sequences

$$
\begin{aligned}
seq_{t-1}^{(1)} &= x_{t-1}^{(1)}, x_{t-2}^{(1)}, \dots x_3^{(1)}, x_2^{(1)}, x_1^{(1)} \\
seq_{t-1}^{(2)} &= x_{t-1}^{(2)}, x_{t-2}^{(2)}, \dots x_3^{(2)}, x_2^{(2)}, x_1^{(2)} \\
seq_{t-1}^{(3)} &= x_{t-1}^{(3)}, x_{t-2}^{(3)}, \dots x_3^{(3)}, x_2^{(3)}, x_1^{(3)} \\
&\dots \\
seq_{t-1}^{(n)} &= x_{t-1}^{(n)}, x_{t-2}^{(n)}, \dots x_3^{(n)}, x_2^{(n)}, x_1^{(n)}
\end{aligned}
$$

- at step t generate K candidates with highest probability by applying the argmax for every sequence

$$
\begin{aligned}
\hat{x}_t^{(1)} &= \text{argmax}_z P(x_t = z | x_{t-1}^{(1)}, x_{t-2}^{(1)}, \dots, x_3^{(1)}, x_2^{(1)}, x_1^{(1)}) \\
\hat{x}_t^{(2)} &= \text{argmax}_z P(x_t = z | x_{t-1}^{(2)}, x_{t-2}^{(2)}, \dots, x_3^{(2)}, x_2^{(2)}, x_1^{(2)}) \\
\hat{x}_t^{(3)} &= \text{argmax}_z P(x_t = z | x_{t-1}^{(3)}, x_{t-2}^{(3)}, \dots, x_3^{(3)}, x_2^{(3)}, x_1^{(3)})
\end{aligned}
$$

  This is for every index a different probability distribution, because each $s_{t-1}^{(k)}$ is a different sequence

- use this to generate the n candidate sequences of length t
- keeps track of the n greedy best candidates so far, no guarantee to find the optimal one. better than greedy 1 best

`https://hackernoon.com/beam-search-a-search-strategy-5d92fb7817f`

2 Some reading on word embeddings

`http://ruder.io/word-embeddings-1/index.html` `http://nlp.fast.ai/classification/2018/05/15/introducting-ulmfit.html`

3 Simple approaches to Word embeddings

Recurrent neural networks, when used for processing texts, are often fed in with word embeddings. To understand this:

We need to feed in words in a mathematical form. Can encode words by one hot embedding.

Problem: Then all words equally similar/ dissimilar under the inner product of one-hot vectors. This ignores similarity between words that can be present. Examples:
king - queen
Horse - ride
Rice - Wheat

Appendicitis - Cholecystitis

The idea of word embeddings is: map each word into a vector which encodes somehow similarities between words. Similarity can be task specific: In a novel the similarity between two types of medicine does not matter. In medical texts ???

How to generate a word embedding?

- Train a neural network for some "meaningful" (whatever!) tasks over words, use a hidden layer of that task as a feature representation.
- Use the nn to train a feature extractor.

Two simple models are:

- CBoW (continuous bag of words)
- Skip gram.

Both were introduced in the word2vec paper by Mikolov et al. `https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-composition.pdf`
Note: Using word embeddings is a form of transfer learning - Preset weights in some layers learnt from another model.

3.1 CBoW:

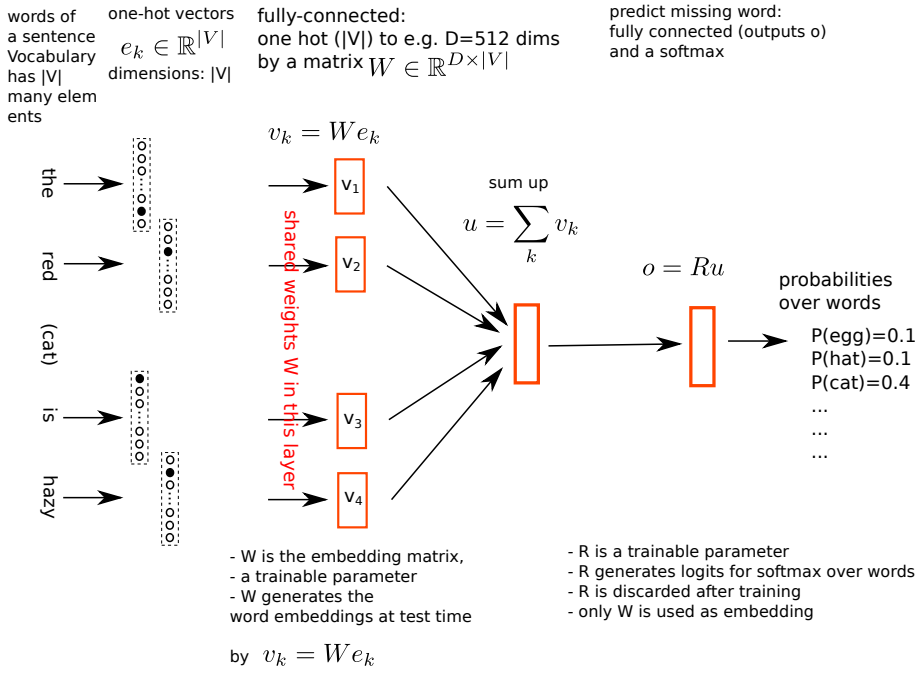
Take a large text corpus. Then take a sequence of $2k + 1$ consecutive words. Task to be solved: Predict the word in the middle (has index k) from the other $2k$ words around it.

Example: $k = 2$:

```
But at length when the tide struck the barque.
It floated away like a sharque,
And hereafter he'll steer
Of that spot very clear
And look out for a low water marque.
```

So you want to predict the word `away` from the inputs: `It floated like a sharque`.

A model for this can look like that:



Training objective for one word at position t? suppose the output o_t should generate the word w_t encoded by one-hot vector $e(w_t)$. you know that the softmax outputs a vector of probabilities:

$$\begin{pmatrix} p(o_t = w_1 \ |w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}) \\ p(o_t = w_2 \ |w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}) \\ p(o_t = w_3 \ |w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}) \\ \vdots \\ p(o_t = w_{|V|} \ |w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}) \end{pmatrix}$$

Then the loss is the neg logarithm for the right word w_t:

$$L_t = -\log p(o_t = w_t \ |w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k})$$

The full training objective is a sum over all possible positions for missing words:

$$L = \frac{1}{T - 2(k+1)} \sum_{t=k+1}^{T-(k+1)} L_t$$

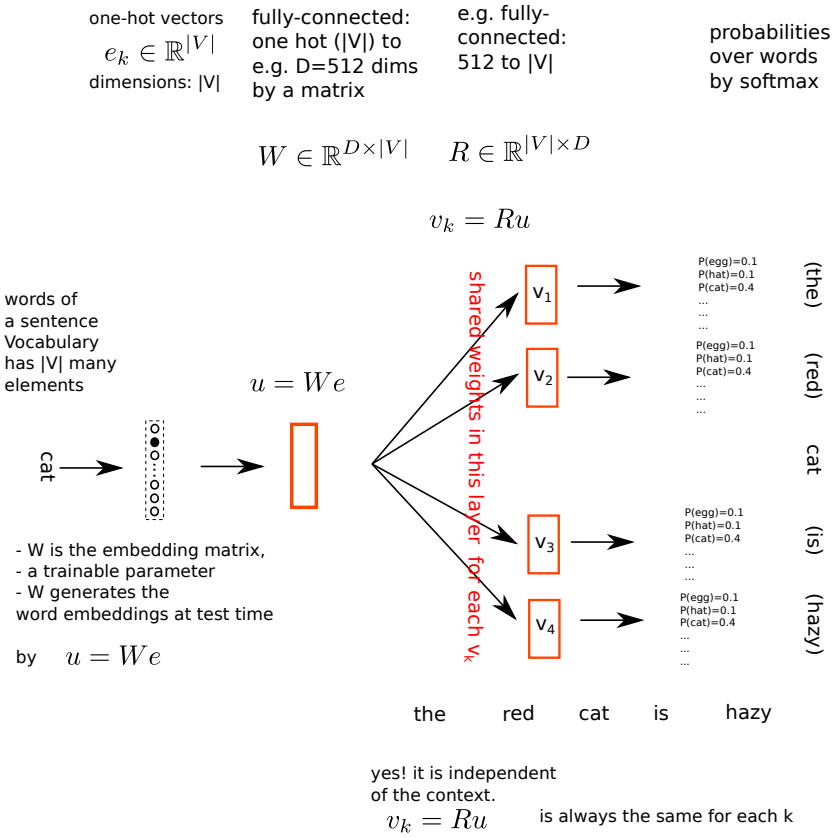
There are variations on this (e.g. predict the word at the end). Note that the weights in the first layer are shared. All four neurons are initialized with the same weights, the gradient updates for these four neurons are summed across the four neurons, and applied to each neuron in the same way, ensuring that the weights stay the same.

3.2 Skip-Gram:

Take a large text corpus. Then take a sequence of $2k + 1$ consecutive words. Task to be solved: Predict the words around the middle (has index k) word

from the middle word. – The other way round

Often worse when trained om small datasets and better for bigger dataset. Surprised? Its the more complicated task.



Be warned

Note: $v_k^{'} = Ru$ is the same output for each k. This is okay because **we do not want to generate** a valid output sentence $w_{t-k}, \ldots, w_{t-1}, w_t, w_{t+1}, \ldots, w_{t+k}$ around word w_t!
All we want is to train a good embedding W. So the order of output words does not matter. We only care that the probability is high for the right $2k$ output words

$$w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k}$$

which are around w_t. These probabilities are high when the rows for the right $2k$ words in $v_k = Ru$ have high logits scores. See:
`http://ruder.io/word-embeddings-1/index.html`
`https://rwalk.xyz/word2vec-skip-gram-feedforward-architecture/`.
Some other sources online get this point wrong (Mikolov's paper is a great achievement, it does not mean it is clearly written about every aspect...).
As a side consequence, the skip-gram model is not suitable to predict/output a sequence of words.

In Ruders notation $v_{w_i}^{'} = R_{w_i,:}^{\top}$ is those row of matrix R (transposed) which corresponds to word w_i, then:

$$R_{w_i,:}u = u \cdot R_{w_i,:}^{\top} = u \cdot v_{w_i}^{'} = v_{w_t} \cdot v_{w_i}^{'}$$

where

$$v_{w_t} = We_{w_t}$$

where e_{w_t} is the one hot vector for word w_t

The training loss is the neg log probability for the right word w_{t+n} around our input word w_t, where $u = We_{w_t}$:

$$\begin{aligned} L_t &= \sum_{n=-2k}^{n=2k} -\log P(o_{t+n} = w_{t+n} | w_t) \\ &= \sum_{n=-2k}^{n=2k} -\log \frac{\exp(u \cdot R_{(w_{t+n},:)}^{\top})}{\sum_{w \in V} \exp(u \cdot R_{(w,:)}^{\top})} \\ &= \sum_{n=-2k}^{n=2k} -\log \frac{\exp(R_{(w_{t+n},:)}u)}{\sum_{w \in V} \exp(R_{(w,:)}u)} \\ L &= \frac{1}{T - 2(k+1)} \sum_{t=k+1}^{T-(k+1)} L_t \end{aligned}$$

The Word2vec paper uses a number of additional tricks (out of class) to facilitate training `https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-`

pdf , see: http://ruder.io/secret-word2vec/index.html for an overview.

What does Skip-gram learn?

- a word embedding that contains knowledge what other words occur around the word to be embedded (because it is able to predict them somewhat!)
- describe a word by information about the words that surround it
- intuitively that can be used to measure the similarity of two words – one compares the information about the words that surround these two

3.3 On efficient computation of the softmax

What is the problem? Log-soft max for one word w_t is:

$$\begin{aligned} L(w_t) &= -\log \frac{\exp(h \cdot R_{(w_t,:)})}{\sum_{w_i \in V} \exp(h \cdot R_{(w_i,:)})} \\ &= -h \cdot R_{(w_t,:)} + \log \left(\sum_{w_i \in V} \exp(h \cdot R_{(w_i,:)}) \right) \end{aligned}$$

This is very costly as it involves a sum over the whole vobabulary. What are the alternatives?

Skip Gram with Negative sampling – solving a different prediction problem:

- idea: avoid the softmax over $|V|$ words, do not try to predict the next word. Skip-gram is anyway not suitable for that.
- define: let context c be the input word w_t in a skip-gram model, and w a word for which to predict whether it appears close to c in the text or not ?
- consider a binary classification problem: is a pair (c, w) present in your training text corpus or not?
- define label $D = 1$ if (w, c) is present in your training text corpus
- need for training (likely) negative labels: sample for a word w, k times likely false contexts $\hat{c}$: $\hat{c} \sim P^{3/4}_{contexts}$, then take each of these k pairs $(w, \hat{c})$ as having negative label $D = 0$.

Let h_c be the embedding vector of a context c. Suppose (w, c) is a pair which is in the training corpus, and thus should have positive label. Suppose $(w, \hat{c})$ is negative labeled because $\hat{c} \sim P^{3/4}_{contexts}$.

Then using a sigmoid σ for two class classification (as in logistic regression) over $h_c \cdot R_{(w,:)}$ yields as classification objective

$$L = -\log(\sigma(h_c \cdot R_{(w,:)})) + \sum_{\hat{c} \sim P^{3/4}_{contexts}, k \text{ times}} -\log(\sigma(-h_{\hat{c}} \cdot R_{(w,:)}))$$

This is negative sampling. Here the softmax over $|V|$ words was circumvented.

`https://arxiv.org/pdf/1402.3722.pdf` gives a good explanation on this.
Hierarchical softmax:

- group words in a groups in a tree
- leaves are single words
- at every node one sigmoid with a binary decision: go left or go right, every node computes $P(\text{go left}|node, \text{inputs})$
- encode a word by a binary sequence 01101011 corresponding to the left/right decisions of the path in the tree from the root to the word, the probability of the word is the product of the probabilities of these decisions $P(01101011) = P(0|node0, inputs)P(1|node1, inputs)P(1|node2, inputs)P(0|node3, inputs)\ldots$

3.4 Problems with word embeddings

- out of vocabulary words
- biases in embeddings

3.5 A more recent word embedding - Elmo

Code `https://github.com/allenai/allennlp/blob/master/tutorials/how_to/elmo.md`

For explanation: `https://www.mihaileric.com/posts/deep-contextualized-word-representations-el`

A step by step explanation

- training setup is predicting the next word and training loss using the neg log probability of the correct next word w_t:

$$L = \sum_t -\log P(o_t = w_t|w_{t-1}, w_{t-2}, \ldots, w_1)$$

- uses at the top RNNs to process words
- input a whole sentence to obtain the embedding of a word: One inputs the whole sentence (not only the k-th word!) into the RNN

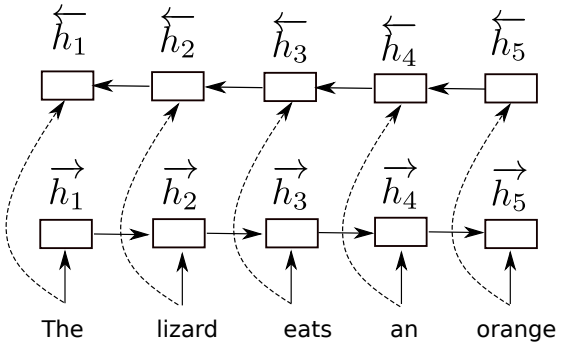
  $$w_1, \ldots, w_k, \ldots, w_n,$$

  but when one is interested for the k-th word w_k, ones takes the hidden state h_k corresponding to the k-th word w_k

- a word embedding which is context aware and potentially is a function of the whole sentence

- use at the top as RNNs bidirectional LSTMs.

$$h_k = \left[\overleftarrow{h_k}, \overrightarrow{h_k}\right]$$



Each direction $\overleftarrow{\cdot}$, $\overrightarrow{\cdot}$ of the top bidirectional RNN produces for every word w_k two hidden states $\overleftarrow{h_k}$ and $\overrightarrow{h_k}$. Therefore one uses for the k-th word the concatenation of both $h_k = \left[\overleftarrow{h_k}, \overrightarrow{h_k}\right]$

- uses multiple layers (l) of bidirectional LSTMs. Thus, for one word one obtains a set of hidden states for two directions and multiple layers $h_k^{(l)} = \left[\overleftarrow{h_k^{(l)}}, \overrightarrow{h_k^{(l)}}\right], l = 1, 2, ...$

- the neural network is made of
  - at first a simple character embedding $y = Qx$, followed by character level CNN on the embedded vectors y,
  - then a character level RNN,
  - then multiple layers of bidirectional RNNs (LSTMs) which take as input hidden states from the character level RNN, aggregated for every word

- Idea 1: atomic input is not a word but a character. Input a word by feeding a sequence of characters into the character level CNN
  - input to the multiple layers of bidirectional RNNs (LSTMs) are hidden states from a character level RNNs, aggregated at the end of each word

- Idea 2: use as representation for the k-th word w_k the concatenation of
  - its base one-hot embedding e_{w_k}
  - its character level embedding x_k of one word (output of the character level RNN)

  - with all hidden states $\left[\overleftarrow{h_k^{(l)}}, \overrightarrow{h_k^{(l)}}\right]$ for all layers (l) for the same word w_k from the word level bidirectional RNNs

- the character level RNN is a two-layer highway network `https://arxiv.org/pdf/1505.00387.pdf`, see eq (8) in `https://arxiv.org/pdf/1508.06615.pdf` for a concise description of a pipeline of word embedding–CNN–global maxpool–highway network over the maxpooled filter responses. A highway network is similar to LSTM memory cells and resnets.
- achieves state of the art in many tasks
- downside: low speed! use fasttext `https://github.com/facebookresearch/fastText` if you need speed

A note on using character level embeddings:

- character level embeddings can deal naturally with out of vocabulary words. fails only with out of vocabulary characters, e.g. Kyrillic on Latin
- character level embeddings allow to capture similarity between words like *love* and *loving* when one of them is not present in the vocabulary, but they share a word stem
- Warning: in some applications a silent treatment of out-of-vocabulary words by feeding them in as a sequence of characters is undesired. Sometimes flagging unknown words out as UNK for a human to check can be good. E.g. critical applications like military or medicine.

It is an open research question how much semantic meaning is carried by out of vocabulary words (e.g. medicine, location names, or family names specific to a language) when processed by character level RNNs. It is possibly measurable by similarities between out-of-vocab words to known similar in-vocab words versus random words from the vocabulary, or words with similar characters ...

Question is: if we have tons of diseases in the vocabulary like Appendicitis, Pleuritis, Nephritis – Will an out-of-vocab disease like Angina be more similar to diseases or to random words like Angela?

For out of class reading on better understanding of this model: `https://arxiv.org/pdf/1508.06615.pdf`

3.6 More:

Word embeddings like *Word2Vec* `https://code.google.com/archive/p/word2vec/`, *GloVe* `https://nlp.stanford.edu/projects/glove/`, *Conceptnet Numberbatch* `https://github.com/commonsense/conceptnet-numberbatch` map words into a vector space such that similar words should be close by euclidean distance.

- How to measure their quality?

  - better: evaluating their usefulness for your task at hand
  - ok: measuring the similarities they captured

- you can list for every embedded word the most similar vectors. The embedding of *ladybug* should have words close by like: *bee,bug, butterfly, insect*, and words like *ocean,AI,grammar,toothbrush* should be more far away from that, and take a manual sanity check

- measuring similarity in a vector space by the inner product between two vectors, or the cosine angle!

$$cos(u, v) = \frac{u \cdot v}{\|u\|\|v\|}$$

- Similarities can be also evaluated for example by pairwise analogies:

  Queen to Woman = X to Man.

  Vector spaces are not suitable for dividing vectors, but you can subtract vectors and compute similarities between differences of vectors. What is most similar in terms of inner product to ?

$$vec(Princess) - vec(Woman) \approx X - vec(Man)$$

$$cos(u, v) = \frac{u \cdot v}{\|u\|\|v\|}$$

$$\text{word} = \text{argmax}_w cos(vec(w), vec(Princess) - vec(Woman) + vec(Man)\ )$$

  Idea here is: if I take princess, subtract woman from it and add man to it, what is the nearest word to that?

  out of class: `http://www.aclweb.org/anthology/W14-1618` suggests a different evaluation (see footnote 7 in this paper):

$$\text{word} = \text{argmax}_w \frac{cos(vec(w),\ vec(Princess))cos(vec(w), vec(Man))}{cos(vec(w), vec(woman)) + 0.001}$$

- To see similarities of word embeddings, you can look at for example: `https://blog.conceptnet.io/2016/05/25/conceptnet-numberbatch-a-new-name-for-the-best-` - they compute a simple similarity: the inner product between the name of an actor ("Cumberbatch") and the word "actor" and then a similarity between the

- Be careful when using text corpora for training of e.g. chatbots – many datasets have biases: `https://blog.conceptnet.io/2017/04/24/conceptnet-numberbatch-17-04-` An algorithm that you train with those, will easily pick them up `https://en.wikipedia.org/wiki/Tay_(bot)`.

That can a risk for your startup ? It will not kill Microsoft for sure. Think about such things before deploying AI outcomes.

- A side question: do you really want chatbots for talking to elderly people? Do you really want chatbots for customer service?

4 Further reading on neural machine translation

A long tutorial:

Graham Neubig, *Neural Machine Translation and Sequence-to-sequence Models: A Tutorial*
`https://arxiv.org/pdf/1703.01619.pdf`

google for Tree-LSTMs. So far what was ignored here was grammar structure. Better models fuse deep learning with grammar rules as prior knowledge.