Deep learning

12.3. Word embeddings and translation

François Fleuret
https://fleuret.org/dlc/
Dec 20, 2020

Word embeddings and CBOW
An important application domain for machine intelligence is Natural Language Processing (NLP).

- Speech and (hand)writing recognition,
- translation,
- question answering.
- part-of-speech tagging,
- sentiment analysis,
- auto-captioning.

While language modeling was historically addressed with formal methods, in particular generative grammars, state-of-the-art and deployed methods are now heavily based on statistical learning and deep learning.

A core difficulty of Natural Language Processing is to devise a proper density model for sequences of words.

Since a vocabulary is usually of the order of $10^4 - 10^6$ words, empirical distributions can not be estimated for more than triplets of words.
The standard strategy to mitigate this problem is to embed words into a geometrical space and exploit regularities for further [statistical] modeling.

The geometry after embedding should account for synonymy, but also for identical word classes, etc. E.g. we would like such an embedding to make “cat” and “tiger” close, but also “red” and “blue”, or “eat” and “work”, etc.

Even though they are not “deep”, classical word embedding models are key elements of NLP with deep-learning.

Let

\[ k_t \in \{1, \ldots, W\}, \ t = 1, \ldots, T \]

be a training sequence of \( T \) words, encoded as IDs from a \( W \) words vocabulary.

Given an embedding dimension \( D \), the objective is to learn vectors

\[ E_k \in \mathbb{R}^D, \ k \in \{1, \ldots, W\} \]

so that “similar” words are embedded with “similar” vectors.
A common word embedding is the Continuous Bag of Words (CBOW) version of word2vec (Mikolov et al., 2013a).

**In this model, the embedding vectors are chosen so that a word can be [linearly] predicted from the sum of the embeddings of words around it.**

More formally, let \( C \in \mathbb{N}^* \) be a “context size”, and

\[
\mathcal{C}_t = (k_{t-C}, \ldots, k_{t-1}, k_{t+1}, \ldots, k_{t+C})
\]

be the “context” around \( k_t \), that is the indexes of words around it.
The embeddings vectors
\[ E_k \in \mathbb{R}^D, \ k = 1, \ldots, W, \]
are optimized jointly with an array
\[ M \in \mathbb{R}^{W \times D} \]
so that the predicted vector of \( W \) scores
\[ \psi(t) = M \sum_{k \in C_t} E_k \]
is a good predictor of the value of \( k_t \).

Ideally we would minimize the cross-entropy between the vector of scores \( \psi(t) \in \mathbb{R}^W \) and the class \( k_t \)
\[ \sum_t - \log \left( \frac{\exp \psi(t)_{k_t}}{\sum_{k=1}^{W} \exp \psi(t)_k} \right). \]
However, given the vocabulary size, doing so is numerically unstable and computationally demanding.
The “negative sampling” approach uses the prediction for the correct class $k_t$ and only $Q \ll W$ incorrect classes $\kappa_t, 1, \ldots, \kappa_t, Q$ sampled at random.

In our implementation we take the later uniformly in $\{1, \ldots, W\}$ and use the same loss as Mikolov et al. (2013b):

$$\sum_t \log \left(1 + e^{-\psi(t)k_t}\right) + \sum_{q=1}^{Q} \log \left(1 + e^{\psi(t)\kappa_t, q}\right).$$

We want $\psi(t)k_t$ to be large and all the $\psi(t)\kappa_t, q$ to be small.

Although the operation

$$x \mapsto E_x$$

could be implemented as the product between a one-hot vector and a matrix, it is far more efficient to use an actual lookup table.
The PyTorch module `nn.Embedding` does precisely that. It is parametrized with a number $N$ of words to embed, and an embedding dimension $D$.

It gets as input an integer tensor of arbitrary dimension $A_1 \times \cdots \times A_U$, containing values in $\{0, \ldots, N - 1\}$ and it returns a float tensor of dimension $A_1 \times \cdots \times A_U \times D$.

If $w$ are the embedding vectors, $x$ the input tensor, $y$ the result, we have

$$y[a_1, \ldots, a_U, d] = w[x[a_1, \ldots, a_U]][d].$$

```python
>>> e = nn.Embedding(num_embeddings = 10, embedding_dim = 3)
>>> x = torch.tensor([[1, 1, 2, 2], [0, 1, 9, 9]])
>>> y = e(x)
>>> y.size()
torch.Size([2, 4, 3])
>>> y
tensor([[ 0.1310,  0.7812, -0.8920],
         [ 0.1310,  0.7812, -0.8920],
         [ 0.9378, -2.5204,  0.3979],
         [ 0.9378, -2.5204,  0.3979]], grad_fn=<EmbeddingBackward>)
```
Our CBOW model has as parameters two embeddings

\[ E \in \mathbb{R}^{W \times D} \quad \text{and} \quad M \in \mathbb{R}^{W \times D}. \]

Its forward gets as input a pair of integer tensors corresponding to a batch of size \( B \):

- \( c \) of size \( B \times 2C \) contains the IDs of the words in a context, and
- \( d \) of size \( B \times R \) contains the IDs, for each of the \( B \) contexts, of the \( R \) words for which we want the prediction score (that will be the correct one and \( Q \) negative ones).

It returns a tensor \( y \) of size \( B \times R \) containing the dot products.

\[
y[n, j] = \frac{1}{D} M[d[n, j]] \cdot \left( \sum_i E[c[n, i]] \right).\]

class CBOW(nn.Module):
    def __init__(self, voc_size = 0, embed_dim = 0):
        super().__init__()
        self.embed_dim = embed_dim
        self.embed_E = nn.Embedding(voc_size, embed_dim)
        self.embed_M = nn.Embedding(voc_size, embed_dim)

    def forward(self, c, d):
        sum_w_E = self.embed_E(c).sum(1, keepdim = True).transpose(1, 2)
        w_M = self.embed_M(d)
        return w_M.bmm(sum_w_E).squeeze(2) / self.embed_dim
Regarding the loss, we can use `nn.BCEWithLogitsLoss` which implements
\[
\sum_t y_t \log(1 + \exp(-x_t)) + (1 - y_t) \log(1 + \exp(x_t)).
\]
It takes care in particular of the numerical problem that may arise for large values of \(x_t\) if implemented “naively”.

Before training a model, we need to prepare data tensors of word IDs from a text file. We will use a 100Mb text file taken from Wikipedia and

- make it lower-cap,
- remove all non-letter characters,
- replace all words that appear less than 100 times with '*',
- associate to each word a unique id.

From the resulting sequence of length \(T\) stored in a integer tensor, and the context size \(C\), we will generate mini-batches, each of two tensors

- a 'context' integer tensor \(c\) of dimension \(B \times 2C\), and
- a 'word' integer tensor \(w\) of dimension \(B\).
If the corpus is “The black cat plays with the black ball.”, we will get the following word IDs:

- the: 0
- black: 1
- cat: 2
- plays: 3
- with: 4
- ball: 5

The corpus will be encoded as

```
the black cat plays with the black ball
0 1 2 3 4 0 1 5
```

and the data and label tensors will be

<table>
<thead>
<tr>
<th>Words</th>
<th>IDs</th>
<th>c</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>the black cat plays with</td>
<td>0 1 2 3 4</td>
<td>0,1,3,4</td>
<td>2</td>
</tr>
<tr>
<td>black cat plays with the</td>
<td>1 2 3 4 0</td>
<td>1,2,4,0</td>
<td>3</td>
</tr>
<tr>
<td>cat plays with the black</td>
<td>2 3 4 0 1</td>
<td>2,3,0,1</td>
<td>4</td>
</tr>
<tr>
<td>plays with the black ball</td>
<td>3 4 0 1 5</td>
<td>3,4,1,5</td>
<td>0</td>
</tr>
</tbody>
</table>

We can train the model for an epoch with:

```python
for k in range(0, id_seq.size(0) - 2 * context_size - batch_size, batch_size):
c, w = extract_batch(id_seq, k, batch_size, context_size)
d = torch.randint(voc_size, (w.size(0), 1 + nb_neg_samples))
d[:, 0] = w
target = torch.zeros(d.size())
target[:, 0] = 1.0
output = model(c, d)
loss = bce_loss(output, target)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```
As results, some nearest-neighbors for the cosine distance between the embeddings

\[ d(w, w') = \frac{E_w \cdot E_{w'}}{\|E_w\| \|E_{w'}\|} \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>paris</td>
<td>bike</td>
<td>cat</td>
<td>fortress</td>
<td>powerful</td>
</tr>
<tr>
<td>0.61</td>
<td>0.61</td>
<td>0.55</td>
<td>0.61</td>
<td>0.47</td>
</tr>
<tr>
<td>0.59</td>
<td>france</td>
<td>0.51</td>
<td>0.54</td>
<td>0.55</td>
</tr>
<tr>
<td>0.55</td>
<td>brussels</td>
<td>0.51</td>
<td>0.49</td>
<td>0.55</td>
</tr>
<tr>
<td>0.53</td>
<td>bordeaux</td>
<td>0.49</td>
<td>0.44</td>
<td>0.52</td>
</tr>
<tr>
<td>0.51</td>
<td>toulouse</td>
<td>0.47</td>
<td>0.42</td>
<td>0.51</td>
</tr>
<tr>
<td>0.51</td>
<td>vienna</td>
<td>0.43</td>
<td>0.40</td>
<td>0.50</td>
</tr>
<tr>
<td>0.51</td>
<td>strasbourg</td>
<td>0.42</td>
<td>0.40</td>
<td>0.49</td>
</tr>
<tr>
<td>0.49</td>
<td>munich</td>
<td>0.41</td>
<td>0.39</td>
<td>0.49</td>
</tr>
<tr>
<td>0.49</td>
<td>marseille</td>
<td>0.41</td>
<td>0.39</td>
<td>0.48</td>
</tr>
<tr>
<td>0.48</td>
<td>rouen</td>
<td>0.41</td>
<td>0.38</td>
<td>0.48</td>
</tr>
</tbody>
</table>

An alternative algorithm is the skip-gram model, which optimizes the embedding so that a word can be predicted by any individual word in its context (Mikolov et al., 2013a).
Trained on large corpora, such models reflect semantic relations in the linear structure of the embedding space. E.g.

\[ \mathbf{E}_{[\text{paris}]} - \mathbf{E}_{[\text{france}]} + \mathbf{E}_{[\text{italy}]} \approx \mathbf{E}_{[\text{rome}]} \]

Table 8: Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).

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

(Mikolov et al., 2013a)

The main benefit of word embeddings is that they are trained with unsupervised corpora, hence possibly extremely large.

This modeling can then be leveraged for small-corpora tasks such as

- sentiment analysis,
- question answering,
- topic classification,
- etc.
Sequence-to-sequence translation

Figure 1: Our model reads an input sentence “ABC” and produces “WXYZ” as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

(Sutskever et al., 2014)
English to French translation.

Training:

- corpus 12M sentences, 348M French words, 304M English words,
- LSTM with 4 layers, one for encoding, one for decoding,
- 160,000 words input vocabulary, 80,000 output vocabulary,
- 1,000 dimensions word embedding, 384M parameters total,
- input sentence is reversed,
- gradient clipping.

The hidden state that contains the information to generate the translation is of dimension 8,000.

Inference is done with a “beam search”, that consists of greedily increasing the size of the predicted sequence while keeping a bag of $K$ best ones.

Comparing a produced sentence to a reference one is complex, since it is related to their semantic content.

A widely used measure is the BLEU score, that counts the fraction of groups of one, two, three and four words (aka “n-grams”) from the generated sentence that appear in the reference translations (Papineni et al., 2002).

The exact definition is complex, and the validity of this score is disputable since it poorly accounts for semantic.
Method | test BLEU score (ntst14)
--- | ---
Bahdanau et al. [2] | 28.45
Baseline System [29] | 33.30
Single forward LSTM, beam size 12 | 26.17
Single reversed LSTM, beam size 12 | 30.59
Ensemble of 5 reversed LSTMs, beam size 1 | 33.00
Ensemble of 2 reversed LSTMs, beam size 12 | 33.27
Ensemble of 5 reversed LSTMs, beam size 2 | 34.50
Ensemble of 5 reversed LSTMs, beam size 12 | 34.81

Table 1: The performance of the LSTM on WMT’14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

(Sutskever et al., 2014)
Our model Ulrich UNK, membre du conseil d'administration du constructeur automobile Audi, affirme qu’…

Figure 2: The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the figures. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

(Sutskever et al., 2014)

Figure 3: The left plot shows the performance of our system as a function of sentence length, where the x-axis corresponds to the test sentences sorted by their length and is marked by the actual sequence lengths. There is no degradation on sentences with less than 35 words, there is only a minor degradation on the longest sentences. The right plot shows the LSTM’s performance on sentences with progressively more rare words, where the x-axis corresponds to the test sentences sorted by their “average word frequency rank”.

(Sutskever et al., 2014)
References


