Skipgram (Word2Vec): Practical implementation

Benjamin Roth

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian-Universität München
beroth@cis.uni-muenchen.de

Negative Log-likelihood

- Likelihood:
  - Probability (P) of the training data (labels) as a function of the parameters.
  - Product of the Ps of the individual training instances[1]:

$$\mathcal{L}(\theta) = \prod_i P(y^{(i)}|x^{(i)};\theta)$$

- Likelihood should be maximized $\Leftrightarrow$ Negative Log-likelihood should be minimized:

$$NLL(\theta) = -\log \mathcal{L}(\theta) = -\sum_i \log P(y^{(i)}|x^{(i)};\theta)$$

- What does the respective components at Skipgram correspond to?
  - $x^{(i)}$
  - $y^{(i)}$ (label)
  - θ (parameter)
  - $P(...)$

[1] Assuming that the data is i.i.d. (*identically independently distributed*)

Negative Log-likelihood

- What does the respective components at Skipgram correspond to?
  - $x^{(i)}$
    Word pair: a pair that occurred in the corpus OR artificially created negative pair (sampling)
  - $y^{(i)}$ (label)
    Indicator if the word pair is co-occurrence from the corpus (True) OR if it has been sampled (False).
  - θ (parameter)
    word-embeddings for context and target words ($\boldsymbol{v}$ bzw $\boldsymbol{w}$).
  - $P(...)$
    Logistic sigmoid function. Specifies the probability that word pair is co-occurrence from the corpus. Converts dot product to P: $\sigma(\boldsymbol{v}^T \boldsymbol{w})$

Skipgram probabilities

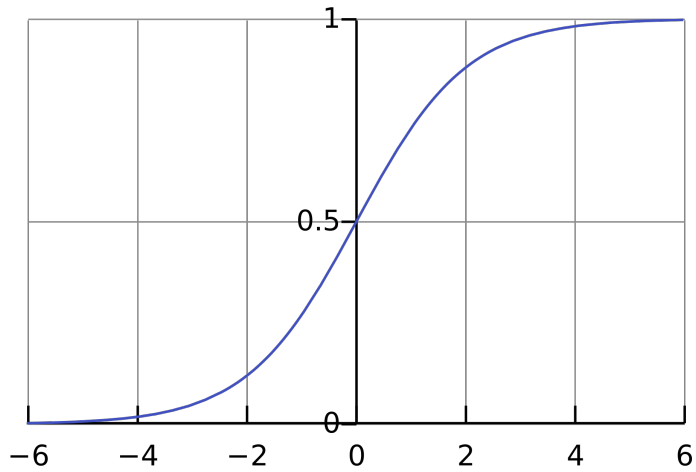
- With Skipgram with negative sampling, the P is estimated for a word pair if it is a co-occurrence from the corpus. For example:

$$P(True|orange, juice) = \sigma(\boldsymbol{v}_{orange}^{T} \boldsymbol{w}_{juice})$$

- The P that the pair does not belong to the corpus:

$$P(False|orange, upends) = 1 - \sigma(\boldsymbol{v}_{orange}^{T} \boldsymbol{w}_{upends})$$

- Sigmoid function: $\sigma(z) = \frac{1}{1+exp(-z)}$



Generation of positive word-pairs

- Corpus:
  the cat sat on the mat
- Co-occurrences (in the defined window):
  `(target_word, context_word, True)`
  - *Target Word*: word *"in the middle"*
  - *Context Word*: Word is at most `max_distance` positions away from target word.
  - *True*: Word-pair comes from the corpus.
- Z.B.: (the, cat, True) (cat, the, True) (cat, sat, True) (sat, cat, True) (sat, on, True) (on, sat, True) (on, the, True) (the, on, True) (the, mat, True) (mat, the, True)
- In the real implementation, each word is represented by its line number in the embedding matrices.

Generation of negative word-pairs

- A negative tuple is generated from each positive tuple (Number `neg_samples_factor` is a hyperparameter)
  `(target_word, random_word, False)`
  - *Target Word*: Will be taken from positive pair
  - *Random Word*: Is taken randomly from the entire vocabulary
  - *False*: Word pair comes **not** from the corpus, but was created randomly.[2]
- Generation of negative pairs (sampling and replacing the context word):
  (the, cat, True) (the, on, False) (the, the, False) (cat, the, True) (cat, mat, False) (cat, sat, False) (cat, sat, True) (cat, on, False) (cat, the, False) (sat, cat, True) (sat, the, False) (sat, mat, False) ...

[2](But it may be that another identical word pair comes from the corpus)

Embedding matrices

- One $n \times d$ matrix per context or target-embeddingvektor ($\boldsymbol{V}$ or $\boldsymbol{W}$). (n: vocabulary size; d: dimensions of word vectors)
- word vectors for context word i and target word j:
  - $\boldsymbol{v}^{(i)T} = \boldsymbol{V}[i,:]$
  - $\boldsymbol{w}^{(j)T} = \boldsymbol{W}[j,:]$
- The entries of the matrices are optimized by *stochastic gradient descent* to optimize the (negative log) likelihood of the positive and negative training instances.

Stochastic Gradient Descent

- Gradient: **Vector**, contains the derivatives of a function with respect to several of its variables.
- In our case (*stochastic* gradient descent)
  - Function: NLL of an instance (so for example $-\log P(False|cat, mat)$)
  - Derivation concerning: Representation of the context word or the target word
- Formulas for calculating the gradients in our case [3]:

$$\nabla_{\boldsymbol{v}^{(i)}} NLL = -\left(label - \sigma(\boldsymbol{v}^{(i)T}\boldsymbol{w}^{(j)})\right)\boldsymbol{w}^{(j)}$$

$$\nabla_{\boldsymbol{w}^{(j)}} NLL = -\left(label - \sigma(\boldsymbol{v}^{(i)T}\boldsymbol{w}^{(j)})\right)\boldsymbol{v}^{(i)}$$

[3] The label True corresponds to 1, False corresponds to 0

Stochastic Gradient Descent

- optimization step for one instance:

$$\boldsymbol{v}^{(i)}_{updated} = \boldsymbol{v}^{(i)} + \eta \left(label - \sigma(\boldsymbol{v}^{(i)T} \boldsymbol{w}^{(j)}) \right) \boldsymbol{w}^{(j)}$$

$$\boldsymbol{w}^{(j)}_{updated} = \boldsymbol{w}^{(j)} + \eta \left(label - \sigma(\boldsymbol{v}^{(i)T} \boldsymbol{w}^{(j)}) \right) \boldsymbol{v}^{(i)}$$

- The learning rate η is a hyperparameter.
- Questions:
  - When are the vectors of a word pair made more similar? When more dissimilar?
  - When does an update make a big change when a small one?
  - Similarities and differences to the perceptron?

Stochastic Gradient Descent

- When are the vectors of a word pair made more similar? When more dissimilar?
  If the label is positive, the other vector is added to it, which makes the vectors more similar (the dot product gets bigger). If the label is negative, it is subtracted and the vectors are made more dissimilar.
- When does an update make a big change when a small one?
  The amount of change results from how close the prediction of the label was to the real value (0 or 1).
- Similarities and differences to the perceptron?
  In the case of the perceptron too, the weights are adjusted as a function of the prediction by addition or subtraction. Differences: (1) With the perceptron the adaptation is binary, with Skipgram it is always adjusted weighted. (2) In the perceptron, features and weights are separate, with Skipgram every weight is characteristic and vice versa.

Implementation of Skipgram

- First, co-occurrences and negative samples must be generated from the corpus and the matrices initialized.
  (`positive_and_negative_cooccurrences(...)`: Exercise sheet).
- Das Modell wird in mehreren Iterationen trainiert.
  - Jede Iteration führt für alle Instanzen im Korpus die Updates aus.
  - Vor jeder Iteration: Mischen (shufflen) der Daten!
- Word similarity may be calculated after training with one of the embedding matrices (e.g., the target word matrix)
  (`DenseSimilarityMatrix(...)`, last exercise sheet)

Implementation of Skipgram

```
class SkipGram:
    def __init__(self, tokens, vocab_size=10000, num_dims=50):
        self.word_to_id = # TODO: create dict word -> id
        self.pos_neg_list = \
            positive_and_negative_cooccurrences(tokens, ...)
        rows = len(self.word_to_id)
        self.target_word_matrix = np.random.rand(rows, num_dims)
        self.context_word_matrix = np.random.rand(rows, num_dims)

    def update(self, target_id, context_id, label, learning_rate):
        # TODO: update self.context_word_matrix[context_id]
        # TODO: update self.target_word_matrix[target_id]

    def train_iter(self, learning_rate=0.1):
        random.shuffle(self.pos_neg_list)
        for tgt_id, ctxt_id, lbl in self.pos_neg_list:
            self.update(tgt_id, ctxt_id, lbl, learning_rate)

    def toDenseSimilarityMatrix(self):
        return DenseSimilarityMatrix( \
            self.target_word_matrix, self.word_to_id)
```

Summary

- Negative Log-Likelihood
- Sigmoid function
- Sampling of negative word-pairs
- Update of Embedding Matrices: Gradient Descent