Skipgram (Word2Vec): Praktische Implementierung

Benjamin Roth

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

Negative Log-likelihood

- Likelihood:
  - Wahrscheinlichkeit (WK) der Trainings-Daten (Labels) als Funktion der Parameter.
  - Produkt der WKen der einzelnen Trainings-Instanzen[1]:

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

- Likelihood soll maximiert werden $\Leftrightarrow$ Negative Log-likelihood soll minimiert werden:

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

- Was entspricht bei Skipgram den jeweiligen Komponenten?
  - $x^{(i)}$
  - $y^{(i)}$ (Label)
  - θ (Parameter)
  - $P(...)$

[1] Unter der Annahme, dass die Daten i.i.d. (*identically independently distributed*) sind

Negative Log-likelihood

- Was entspricht bei Skipgram den jeweiligen Komponenten?
  - $x^{(i)}$
    Wort-Paar: im Korpus vorgekommenes ODER künstlich erzeugtes negatives Paar (sampling)
  - $y^{(i)}$ (Label)
    Indikator ob das Wort-Paar Co-okkurrenz aus dem Korpus ist (True) ODER ob es gesampelt wurde (False).
  - θ (Parameter)
    Word-Embedings für Kontext und Ziel-Wörter ($\boldsymbol{v}$ bzw $\boldsymbol{w}$).
  - $P(...)$
    Logistic Sigmoid Funktion. Gibt die Wahrscheinlichkeit an, dass Wortpaar Co-Okkurrenz aus dem Korpus ist. Wandelt Dot-Produkt in WK um: $\sigma(\boldsymbol{v}^T \boldsymbol{w})$

Skipgram Wahrscheinlichkeiten

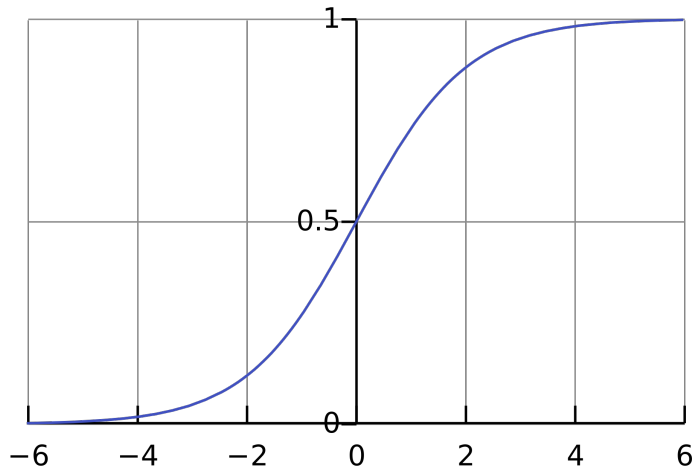
- Bei Skipgram mit Negative Sampling wird für ein Wortpaar die WK geschätzt, ob es eine Co-okkurrenz aus dem Korpus ist. Z.B.:

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

- Die WK, dass das Paar nicht zum Korpus gehört:

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

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



Erzeugung der positiven Wort-Paare

- Corpus:
  the cat sat on the mat
- Co-Okkurrenzen (in definiertem Fenster):
  `(target_word, context_word, True)`
  - *Target Word*: Wort *"in der Mitte"*
  - *Context Word*: Word das höchstens `max_distance` Positionen von Target Word entfernt ist.
  - *True*: Wort-Paar kommt aus dem Korpus.
- 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 der echten Implementierung wird jedes Wort durch seine Zeilennummer in den Embedding-Matrizen repräsentiert.

Erzeugung der negativen Wort-Paare

- Aus jedem positiven Tupel werden negative Tupel erzeugt (Anzahl `neg_samples_factor` ist ein Hyper-Parameter)
  `(target_word, random_word, False)`
  - *Target Word*: Wird aus positivem Paar übernommen
  - *Random Word*: Wird zufällig aus dem gesamten Vokabular übernommen
  - *False*: Wort-Paar kommt **nicht** aus dem Korpus, sondern wurde zufällig erzeugt.[2]
- Erzeugen der negativen Paare (samplen und ersetzen des Kontext Wortes):
  (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] (Es kann aber sein, dass ein anderes identisches Wortpaar aus dem Korpus kommt)

Embedding-Matrizen

- Je eine $n \times d$ Matrix für Kontext- bzw Ziel-Embeddingvektoren ($\boldsymbol{V}$ bzw. $\boldsymbol{W}$). (n: Vokabulargröße; d: Dimension der Wortvektoren)
- Wortvektoren für Kontextwort i und Zielwort j:
  - $\boldsymbol{v}^{(i)T} = \boldsymbol{V}[i,:]$
  - $\boldsymbol{w}^{(j)T} = \boldsymbol{W}[j,:]$
- Die Einträge der Matrizen werden durch *stochastic gradient descent* (Gradienten-Abstiegs-Methode) optimiert, damit sie die (negative Log-) Likelihood der positiven und negativen Trainings-Instanzen optimieren.

Stochastic Gradient Descent

- Gradient: **Vektor**, der Ableitungen einer Funktion bezüglich mehrerer ihrer Variablen enthält.
- In unserem Fall (*Stochastic* Gradient Descent)
  - Funktion: NLL einer Instanz (also z.B. $-\log P(False|cat, mat)$)
  - Ableitung bezüglich: Repäsentation des Kontext-Wortes bzw. des Ziel-Wortes
- Formeln zur Berechnung der Gradienten in unserem Fall[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] Das Label True entspricht der Zahl 1, False entspricht 0

Stochastic Gradient Descent

- Optimierungsschritt für eine Instanz:

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

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

- Wobei die Lernrate η ein Hyper-parameter ist.
- Fragen:
  - Wann werden die Vektoren eines Wort-Paares einander ähnlicher gemacht? Wann unähnlicher?
  - Wann ergibt ein Update eine große Veränderung, wann eine kleine?
  - Ähnlichkeiten und Unterschiede zum Perzeptron?

Stochastic Gradient Descent

- Wann werden die Vektoren eines Wort-Paares einander ähnlicher gemacht? Wann unähnlicher?
  Wenn das Label positiv ist, wird der jeweils andere Vektor dazu-addiert, dadurch werden die Vektoren ähnlicher (das Dot-Produkt wird größer). Ist das Label negativ, wird subtrahiert, und die Vektoren werden unähnlicher gemacht.
- Wann ergibt ein Update eine große Veränderung, wann eine kleine?
  Der Betrag der Änderung ergibt sich daraus, wie nah die Vorhersage des Labels am wirklichen Wert (0 bzw 1) war.
- Ähnlichkeiten und Unterschiede zum Perzeptron?
  Auch beim Perzeptron werden die Gewichte in Abhängigkeit der Vorhersage durch Addition oder Subtraktion angepasst. Unterschiede: (1) Beim Perzeptron ist die Anpassung binär, bei Skipgram wird immer gewichtet angepasst. (2) Beim Perzeptron sind Merkmale und Gewichte separat, bei Skipgram ist jedes Gewicht auch Merkmal und umgekehrt.

Implementierung von Skipgram

- Zunächst müssen Co-Okkurrenzen und Negative-Samples aus dem Korpus erzeugt, und die Matrizen initialisiert werden. (`positive_and_negative_cooccurrences(...)`: Übungsblatt).
- 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!
- Wort-Ähnlichkeit kann nach dem Training mit einer der Embedding-Matrizen (z.B. der Ziel-Wort-Matrix) berechnet werden (`DenseSimilarityMatrix(...)`, letztes Übungsblatt)

Implementierung von 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)
```

Zusammenfassung

- Negative Log-Likelihood
- Sigmoid Funktion
- Sampling von negativen Wort-Paaren
- Update der Embedding Matrizen: Gradient Descent