Computer-Linguistische Anwendungen

CLA | B.Sc. | LMU

Skipgram (Word2Vec): Practical Implementation

Negative Log-Likelihood

Likelihood:

- Wahrscheinlichkeit (WK) der Trainings-Daten (Labels) als Funktion der Parameter.
- Produkt der WKen der einzelnen Trainings-Instanzen (unter der Annahme, dass die Daten i.i.d. (identically independently distributed) sind.

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)};\theta)$$

Negative Log-Likelihood

- Likelihood soll maximiert werden ←→ Negative Log-likelihood soll minimiert werden:

$$NLL(\theta) = -\log \mathcal{L}(\theta) = -\sum_{i=1}^{n} \log P(y^{(i)}|x^{(i)};\theta)$$

- Viele Optimierungsalgorithmen sind darauf ausgelegt, Funktionen zu **minimieren**.
- Die negative Log-Likelihood-Funktion ermöglicht das Minimieren statt Maximieren der Funktion.
- Dadurch wird das gleiche Optimierungsziel erreicht, nämlich Parameter zu finden, die die Likelihood maximieren.

- Ein weiterer Vorteil ist die **numerische Stabilität**, insbesondere bei großen Datensätzen.
- Die Likelihood-Funktion kann für einige Parameterwerte sehr klein werden und zu numerischen Problemen führen. Durch die Anwendung des **Logarithmus** auf die **Likelihood und Negation** können diese numerischen Probleme vermieden werden.

Negative Log-Likelihood

- Likelihood soll maximiert werden ←→ Negative Log-likelihood soll minimiert werden:

$$NLL(\theta) = -\log \mathcal{L}(\theta) = -\sum_{i=1}^{n} \log P(y^{(i)}|x^{(i)};\theta)$$

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

Negative Log-Likelihood

- Was entspricht bei Skipgram den jeweiligen Komponenten?
  - $x^{(i)}$ (Input data)

    Wort-Paar: im Korpus vorkommendes "positives" Paar ODER künstlich erzeugtes, zufälliges, "negatives" Paar (sampling)

  - $y^{(i)}$ (Label)

    Indikator ob das Wort-Paar Coocurrence aus dem Korpus ist (True) ODER ob es gesampelt wurde (False).

  - θ (Parameter)

    Wort-Embeddings für Kontext und Ziel-Wörter (v bzw w).

  - $P(\dots)$

    Logistic Sigmoid Funktion. Gibt die Wahrscheinlichkeit an, dass das Wortpaar "positiv", also eine Cooccurrence aus dem Korpus, ist.

Skipgram Wahrscheinlichkeiten

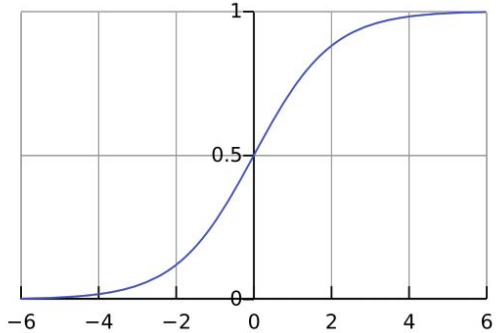
Bei Skipgram mit Negative Sampling wird für ein Wortpaar die WK geschätzt, ob es eine Co-okkurrenz aus dem Korpus ist. Z.B.:

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

Die WK, dass ein Paar nicht zum Korpus gehört:

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

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



Erzeugen der positiven Wort-Paare

- Corpus:
  the cat sat on the mat

- Coocurrences (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.

Erzeugen der positiven Wort-Paare

- Corpus:
  the cat sat on the mat

- Coocurrences (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

Erzeugen der negative Wort-Paare (**negative sampling**)

- 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.
    - Es kann aber sein, dass ein anderes identisches Wortpaar aus dem Korpus kommt

- 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) ...

Erzeugen der negative Wort-Paare (**negative sampling**)

- Je eine $n \times d$ Matrix für Kontext- bzw Ziel-Embeddingvektoren ($\vec{V}$ bzw. $\vec{W}$). (n: Vokabulargröße; d: Dimension der Wortvektoren)
- Wortvektoren für Kontextwort i und Zielwort j:
  - $\vec{v}^{(i)T} = \vec{V}[i,:]$
  - $\vec{w}^{(j)T} = \vec{W}[j,:]$
- 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 :

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

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

→ Das Label True entspricht der Zahl 1, False entspricht 0

Stochastic Gradient Descent

- Optimierungsschritt für eine Instanz:

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

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

- 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 von 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 Cooccurrences 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)
```