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-liklihood 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 ($\mathbf{v}$ *bzw* $\mathbf{w}$*).*
  - $P(...)$
    Logistic Sigmoid Funktion. Gibt die Wahrscheinlichkeit an, dass Wortpaar Co-Okkurrenz aus dem Korpus ist. Wandelt Dot-Produkt in WK um: $\sigma(\mathbf{v}^T\mathbf{w})$

Skipgram Wahrscheinlichkeiten

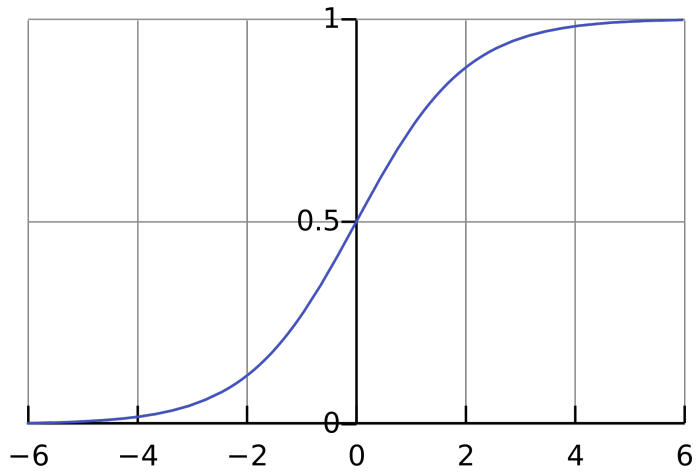
- Bei Skipgram mit Negative Sampling wird für ein Wortpaar die WK geschätzt, ob es eine Co-okkurrenz aus dem Korpus ist. Z.B.:

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

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

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

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



Erzeugung der Positiven und negativen Wort-Paare

- Corpus:
  the cat sat on the mat
- Co-Okkurrenzen (in definiertem Fenster):
  (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)
- Erzeugen der negativen Paare (samplen des Context-Wortes):
  (the, cat, True) (the, the, False) (the, sat, False) (cat, the, True) (cat, mat, False) (cat, on, False) (cat, sat, True) (cat, the, False) (cat, cat, False) (sat, cat, True) (sat, mat, False) (sat, the, False) (sat, on, True) (sat, the, False) (sat, sat, False) (on, sat, True) (on, mat, False) (on, cat, False) (on, the, True) (on, mat, False) (on, on, False) (the, on, True) (the, cat, False) (the, sat, False) (the, mat, True) (the, sat, False) (the, on, False) (mat, the, True) (mat, on, False) (mat, cat, False)
- In der echten Implementierung wird jedes Wort durch seine Zeilennummer in den Embedding-Matrizen repräsentiert.
- Hinweis: die Anzahl der negative samples ist ein Hyper-Parameter. Mehr negative samples bringen oft bessere Ergebnisse, brauchen aber auch mehr Speicherplatz und Trainingszeit.

Embedding-Matrizen

- Je eine $n \times d$ Matrix für Kontext- bzw Ziel-Embeddingvektoren ($\mathbf{V}$ bzw. $\mathbf{W}$). (n: Vokabulargröße; d: Dimension der Wortvektoren)
- Wortvektoren für Kontextwort i und Zielwort j:
  - $\mathbf{v}^{(i)T} = \mathbf{V}[i,:]$
  - $\mathbf{w}^{(j)T} = \mathbf{W}[j,:]$
- Die Einträge der Matrizen werden durch *stochastic gradient descent* (Gradienten-Abstiegs-Methode) optimiert, damit Sie die 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[2]:

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

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

- Hinweis: In PyTorch müssen wir den Gradient nicht explizit berechnen, da er automatisch für jede `autograd.Variable` berechnet werden kann.

[2] Das Label True entspricht der Zahl 1, False entspricht 0

Stochastic Gradient Descent

- Optimierungsschritt für eine Instanz:

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

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

- 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?

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.

Implementierung von Skipgram

- Zunächst müssen Co-Okkurrenzen und Negative-Samples aus dem Korpus erzeugt, und die Matrizen initialisiert werden.
- 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 Context-Wort-Matrix) berechnet werden
- $\Rightarrow$ Cosinus Ähnlichkeit

Some more Pytorch...

- Embedding layer:
  - Contains learnable parameter tensor
    `torch.nn.Embedding(num_embeddings, embedding_dim)`
  - Embedding vectors can be obtained by indexing with LongTensor/Variable:

```
emb = Embedding(num_embeddings, embedding_dim)
selected_inds = Variable(LongTensor(num_select, 1))
selected_vecs = emb(selected_inds)
```

- Batch training (mini-batch SGD):
  - Do updates for a number of instances at a time (instead of just one).
  - Many Pytorch methods interpret first axis of Tensors/Variables as listing the items in the batch.
  - E.g. Batch matrix multiplication:
    `matrixC` torch.bmm(matrixA, matrixB)=
  - $\Rightarrow$ matrixA has size `(batch_size,k,m)`, matrixB has size `(batch_size,m,n)`, and the resulting matrixC has size `(batch_size,m,n)`

Some more Pytorch...

- Mini-batch training:
  - The DataLoader class allows for convenient creation of mini-batches:

```
train = TensorDataset(data_tensor, target_tensor)
train_loader = DataLoader(train, batch_size=512, shuffle=True)
for inputs_batch, labels_batch in train_loader:
    # ... do training ...
    outputs_batch = my_model.forward(Variable(inputs_batch))
    # ...
```

  - Note: Tensors need to have at least two axis in order to work with DataLoader. For example, a vector of predictions should be represented as a matrix with size `(num_items, 1)`, not as a vector of size `(num_items,)`
- Use `tensor.view(axis1_size, axis2_size, axis3_...)` to reshape your Tensors/Variables.

```
my_vec_for_dataloader = my_vec.view(-1,1)
```

Some more Pytorch...

- Log-Likelihood for binary prediction with sigmoid, aka *Binary Crossentropy*:

```
prediction = F.sigmoid(score)
...
criterion = nn.BCELoss()
```

- Numerically more stable: don't apply sigmoid for training and use special loss function:

```
prediction = score
...
criterion = nn.BCEWithLogitsLoss()
```

Zusammenfassung

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