From features to neural networks

Lecture, Feb 14
CS 690N, Spring 2017
Advanced Natural Language Processing
http://people.cs.umass.edu/~brenocon/anlp2017/

Brendan O’Connor
College of Information and Computer Sciences
University of Massachusetts Amherst
MaxEnt / Log-Linear models

- **x**: input (all previous words)
- **y**: output (next word)
- **f(x,y)** $\rightarrow \mathbb{R}^d$ feature function [[domain knowledge here!]]
- **v**: $\mathbb{R}^d$ parameter vector (weights)

For any $x \in X$, $y \in Y$, the model defines a conditional probability:

$$p(y|x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in Y} \exp(v \cdot f(x, y'))}$$

Application to history-based LM:

$$P(w_1..w_T) = \prod_t P(w_t \mid w_1..w_{t-1})$$

$$= \prod_t \frac{\exp(v \cdot f(w_1..w_{t-1}, w_t))}{\sum_{w \in \mathcal{V}} \exp(v \cdot f(w_1..w_{t-1}, w))}$$
Too many features

• Millions to billions of features: performance often keeps improving!
• Engineering issue: feature name=>number mapping
• Feature selection ... mixed results
  • Count cutoffs: great computational benefits; typically not for performance
  • Features seen only once at training time typically help (!), or even features not seen at training time
  • Predictive value: mutual info. / info. gain / chi-square
  • L1 regularization: encourages $\theta$ sparsity, but not always better than L2
    • [structured sparsity more interesting: Yogatama, Martins tutorial]
• Personal opinion: feature-based models just want a high diversity of weak signals
Feature hashing

- Feature hashing: make e.g. $N(u,v,w)$ mapping random with collisions (!) (Weinberger et al. 2009)
  - Accuracy loss low since collisions are rare (since features are sparse). Works well, great for large-scale data (memory usage constant!)
  - Practically: use a fast string hashing function (e.g. murmurhash or Python’s internal one)
- This is a type of randomized projection $Ax$. Typically not better than the original representation.
- Instead of randomized embeddings, better generalization from learning them
Dense representations

• Feature hashing as dense representation

\[ P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}}) \]

• Saul and Pereira 1997 as dense representation

\[ P(w_{next} \mid w_{prev}) = A_{w_{prev}} \cdot B_{w_{next}} \]

• Mnih and Hinton 2007: log-bilinear model [similar: “word2vec” Mikolov et al.]

\[ P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}}) \]

• Learn with gradient descent
• (this is simplified from their version)
Neural networks

- Idea: learn distributed representations of concepts
- Nonlinear functions seem to help

---

**Diagram 1.** The basic components of a parallel distributed processing system.

[Diagrams from: Rumelhart and McClelland (ed.) 1986, *Parallel Distributed Processing*]
Bengio et al. 2003: N-gram multilayer perceptron

\[ f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t|w_1^{t-1}) \]

\( i \)-th output = \( P(w_t = i | context) \)

Learn: \( C, W, U, H, d \) (chain rule)

\( C(i) \in \mathbb{R}^m \) Word embedding parameters

\( x = (C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1})) \)

Lookup layer with concatenation:

(kind) hidden layer size \((n-1)m\)

another hidden layer, size \(h\)

\( y = b + Wx + U \tanh(d + Hx) \)

Vocab output: log-probs size \(V\)

\( \hat{P}(w_t|w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{yw_t}}{\sum_i e^{yi}}. \)

Output layer (softmax / log-linear)
• stopped here 2/14
Word/feature embeddings

• “Lookup layer”: from discrete input features (words, ngrams, etc.) to continuous vectors
  • Anything that was directly used in log-linear models, move to using vectors
• As model parameters: learn them like everything else
• As external information: use pretrained embeddings
  • Common in practice: use a faster-to-train model on very large, perhaps different, dataset [e.g. word2vec, glove pretrained word vectors]
• Shared representations for domain adaptation and multitask learning