CS447: Natural Language Processing
http://courses.engr.illinois.edu/cs447

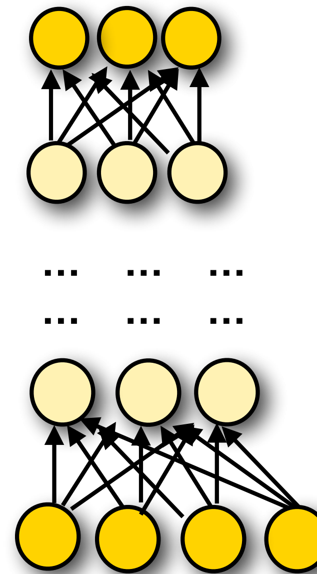
Lecture 7: More on neural nets for NLP

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center

Fully connected feedforward nets

Three kinds of layers, arranged in sequence:

- **Input layer** (what's fed into the net)
- **Hidden layers:** (intermediate computations)
- **Output layer**: (what the net returns)



Each layer consists of a number of **units**.

- Each unit computes a ***real-valued*** **activation**
- In a ***feedforward*** net, each (hidden/output) unit receives inputs from the units in the **immediately *preceding* layer**
- In a ***fully connected*** feedforward net, each unit receives inputs from ***all*** **units** in the immediately preceding layer

  Additional "*Highway connections*" from layers in earlier layers can be useful

Feedforward computations

The **activation** x_{ij} **of unit** *j* **in layer** *i* is computed as

$$x_{ij} = f(\mathbf{w}_{ij} \cdot \mathbf{x}_{i-1} + b_{ij})$$

where

— $\mathbf{w}_{ij} = (w_{ij1}, \ldots, w_{ijK})$ is a (unit-specific) **weight vector**
(K = #units in (i-1)-th layer)
because each connection is associated with one real-valued weight

— b_{ij} is a (unit-specific) real-valued **bias term**

— $f()$ is a (layer-specific) **non-linear activation function**

Each **layer** is defined by its number of units, N, a non-linear activation function $f()$, a learned matrix of weights $\mathbf{W}$, and a learned bias vector $\mathbf{b}$.

Nonlinear activation functions

Sigmoid (logistic function): $\sigma(x) = 1/(1 + e^{-x})$

Useful for output units (probabilities) [0,1] range

Hyperbolic tangent: $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$

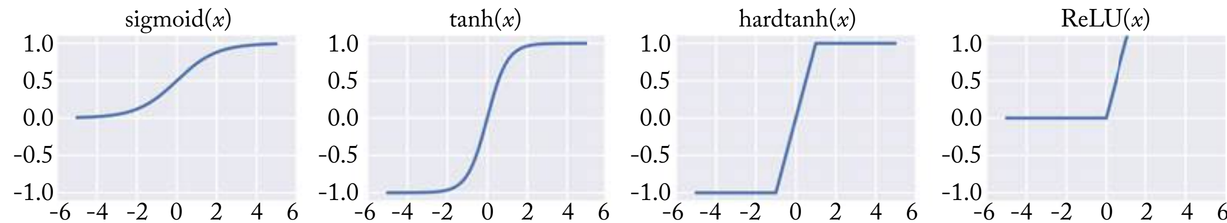
Useful for internal units: [-1,1] range

Hard tanh (approximates tanh)

$\text{htanh}(x) = -1$ for $x < -1$, 1 for $x > 1$, x otherwise

Rectified Linear Unit: $\text{ReLU}(x) = \max(0, x)$

Useful for internal units



Softmax: $\text{softmax}(z_i) = \exp(z_i) \;/\; \sum_{k=0..K} \exp(z_k)$

Special case for output units (multiclass classification)

Neural Language Models

What is a language model?

Probability distribution over the strings in a language, typically factored into distributions $P(w_i \mid \ldots)$ for each word:

$$P(\mathbf{w}) = P(w_1 \ldots w_n) = \prod_i P(w_i \mid w_1 \ldots w_{i-1})$$

N-gram models assume each word depends only preceding n−1 words:

$$P(w_i \mid w_1 \ldots w_{i-1}) =_{def} P(w_i \mid w_{i-n+1} \ldots w_{i-1})$$

To handle variable length strings, we assume each string starts with n−1 start-of-sentence symbols (BOS), or ⟨S⟩ and ends in a special end-of-sentence symbol (EOS) or ⟨\S⟩

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (naively)

— The **vocabulary** V contains *n* types (incl. UNK, BOS, EOS)
— We want to condition each word on *k* preceding words

— [Naive] Each **input word** $w_i \in V$ (that we're conditioning on) is an ***n*-dimensional one-hot vector** $v(w) = (0,\ldots0, 1,0\ldots.0)$
— Our **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $n \times k$ elements
— To predict the probability over output words, the **output layer** is a softmax over n elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{hW}^2 + \mathbf{b}^2)$$

With (say) one hidden layer $\mathbf{h}$ we'll need two sets of parameters, one for **h** and one for the output

Naive neural n-gram model

Architecture:

Input Layer: $\mathbf{x} = [v(w_1)....v(w_k)]$

$v(w)$: a one-hot vector of size $\dim(V) = |V|$

Hidden Layer: $\mathbf{h} = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)$

Output Layer: $P(w \mid w1...wk) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

Parameters:

Weight matrices and biases:

first layer: $\mathbf{W}^1 \in R^{k \cdot \dim(V) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in R^{\dim(\mathbf{h})}$

second layer: $\mathbf{W}^2 \in R^{\dim(\mathbf{h}) \times |V|}$ $\mathbf{b}^2 \in R^{|V|}$

How many parameters do we need to learn?

Traditional n-gram model: $\dim(V)^k$ parameters

With dim(V) = 10,000 and k=3: 1,000,000,000,000

Naive neural n-gram model (one-hot encoding of vocabulary):

#parameters going to hidden layer: $k \cdot \dim(V) \cdot \dim(\mathbf{h})$,

with $\dim(\mathbf{h})$ = 300, dim(V) = 10,000 and k=3: 9,000,000

plus #parameters going to output layer: $\dim(\mathbf{h}) \cdot \dim(V)$

with $\dim(\mathbf{h})$ = 300, dim(V) = 10,000: 3,000,000

The neural model requires still a lot of parameters,
but far fewer than the traditional n-gram model

Naive neural n-gram models

Advantages over traditional n-gram models:

— The hidden layer captures interactions among context words

— Increasing the order of the n-gram requires only a small linear increase in the number of parameters.

dim($\mathbf{W}^1$) goes from $k \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$ to $(k+1) \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$

A traditional k-gram model requires $\dim(V)^k$ parameters

— Increasing the vocabulary also leads only to a linear increase in the number of parameters

Better neural language models

Naive neural models have similar shortcomings as standard n-gram models

— Models get very large (and sparse) as n increases
— We can't generalize across similar contexts
— N-gram Markov (independence) assumptions are too strict

Better neural language models overcome these by…

… using **word embeddings** instead of one-hots as input:
Instead of representing context words as distinct, discrete symbols (i.e. very high-dimensional one-hot vectors), use a dense low-dimensional vector representation where similar words have similar vectors [next]

… using **recurrent nets** instead of feedforward nets:
Instead of a fixed-length (n-gram) context, use recurrent nets to encode variable-lengths contexts [later class]

Motivation for neural NLP

NLP research questions redux

How do you represent (or predict) words?

Do you treat words in the input as atomic categories, as continuous vectors, or as structured objects?

How do you handle rare/unseen words, typos, spelling variants, morphological information?

Lexical semantics: do you capture word meanings/senses?

How do you represent (or predict) word sequences?

Sequences = sentences, paragraphs, documents, dialogs,…

As a vector, or as a structured object?

How do you represent (or predict) structures?

Structures = labeled sequences, trees, graphs, formal languages (e.g. DB records/queries, logical representations)

How do you represent “meaning”?

Two core problems for NLP

Ambiguity: Natural language is highly ambiguous

- Words have multiple senses and different POS
- Sentences have a myriad of possible parses
- etc.

Coverage (compounded by Zipf's Law)

- Any (wide-coverage) NLP system will come across words or constructions that did not occur during training.
- We need to be able to generalize from the seen events during training to unseen events that occur during testing (i.e. when we actually use the system).
- We typically have very little labeled training data

Statistical models for NLP

NLP makes heavy use of statistical models as a way to handle both the ambiguity and the coverage issues.

- Probabilistic models (e.g. HMMs, MEMMs, CRFs, PCFGs)
- Other machine learning-based classifiers

Basic approach:

- Decide which output is desired
  (may depend on available labeled training data)
- Decide what kind of model to use
- Define features that could be useful (this may require further processing steps, i.e. a pipeline)
- Train and evaluate the model.
- Iterate: refine/improve the model and/or the features, etc.

Example: Language Modeling

A language model defines a distribution $P(\mathbf{w})$ over the strings $\mathbf{w} = w_1 w_2 .. w_i \ldots$ in a language

Typically we factor $P(\mathbf{w})$ such that we compute the probability word by word:

$$P(\mathbf{w}) = P(w_1)\, P(w_2 \mid w_1) \ldots\, P(w_i \mid w_1 \ldots w_{i-1})$$

Standard n-gram models make the Markov assumption that w_i depends only on the preceding n−1 words:

$$P(w_i \mid w_1 \ldots w_{i-1}) := P(w_i \mid w_{i-n+1} \ldots w_{i-1})$$

We know that this independence assumption is invalid (there are many long-range dependencies), but it is computationally and statistically necessary

(we can’t store or estimate larger models)

Motivation for neural approaches to NLP: Markov assumptions

Traditional sequence models (n-gram language models, HMMs, MEMMs, CRFs) make rigid Markov assumptions (bigram/trigram/n-gram).

Recurrent neural nets (RNNs, LSTMs) can capture arbitrary-length histories without requiring more parameters.

Features for NLP

Many systems use explicit features:

- Words (does the word “river” occur in this sentence?)
- POS tags
- Chunk information, NER labels
- Parse trees or syntactic dependencies
  (e.g. for semantic role labeling, etc.)

Feature design is usually a big component of building any particular NLP system.

Which features are useful for a particular task and model typically requires experimentation, but there are a number of commonly used ones (words, POS tags, syntactic dependencies, NER labels, etc.)

Features define equivalence classes of data points.

Assumption: they provide useful abstractions & generalizations

Features may be noisy

(because they are compute by other NLP systems)

Motivation for neural approaches to NLP: Features can be brittle

Word-based features:

How do we handle unseen/rare words?

Many features are produced by other NLP systems (POS tags, dependencies, NER output, etc.)

These systems are often trained on labeled data.

Producing labeled data can be very expensive.

We typically don't have enough labeled data from the domain of interest.

We might not get accurate features for our domain of interest.

Features in neural approaches

Many of the current successful neural approaches to NLP do not use traditional discrete features.

Words in the input are often represented as dense vectors (aka. word embeddings, e.g. word2vec)

- Traditional approaches: each word in the vocabulary is a separate feature. No generalization across words that have similar meanings.
- Neural approaches: Words with similar meanings have similar vectors. Models generalize across words with similar meanings

Other kinds of features (POS tags, dependencies, etc.) are often ignored.

What is “deep learning”?

Neural networks, typically with several hidden layers

- (depth = # of hidden layers)
- Single-layer neural nets are linear classifiers
- Multi-layer neural nets are more expressive

Very impressive performance gains in computer vision (ImageNet) and speech recognition over the last several years.

Neural nets have been around for decades.
Why have they suddenly made a comeback?

- Fast computers (GPUs!) and (very) large datasets have made it possible to train these very complex models.

Challenges in using NNs for NLP

Our input and output variables are discrete:
words, labels, structures.

NNs work best with continuous vectors.

We typically want to learn a mapping (embedding) from discrete words (input) to dense vectors.
We can do this with (simple) neural nets and related methods.

The input to a NN is (traditionally) a fixed-length vector. How do you represent a variable-length sequence as a vector?

Use recurrent neural nets: read in one word at the time to predict a vector, use that vector and the next word to predict a new vector, etc.

How does NLP use NNs?

Word embeddings (word2vec, Glove, etc.)

Train a NN to predict a word from its context (or the context from a word).

This gives a dense vector representation of each word

Neural language models:

Use recurrent neural networks (RNNs) to predict word sequences

More advanced: use LSTMs (special case of RNNs)

Sequence-to-sequence (seq2seq) models:

From machine translation: use one RNN to encode source string, and another RNN to decode this into a target string.

Also used for automatic image captioning, etc.

Recursive neural networks:

Used for parsing

Neural Language Models

LMs define a distribution over strings: $P(w_1 \ldots w_k)$

LMs factor $P(w_1 \ldots w_k)$ into the probability of each word:

$$P(w_1 \ldots w_k) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1 w_2) \cdot \ldots \cdot P(w_k \mid w_1 \ldots w_{k-1})$$

A neural LM needs to define a distribution over the V words in the vocabulary, conditioned on the preceding words.

Output layer: V units (one per word in the vocabulary) with softmax to get a distribution

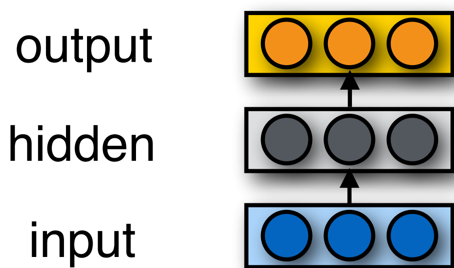
Input: Represent each preceding word by its d-dimensional embedding.

- Fixed-length history (n-gram): use preceding n–1 words
- Variable-length history: use a recurrent **neural net**

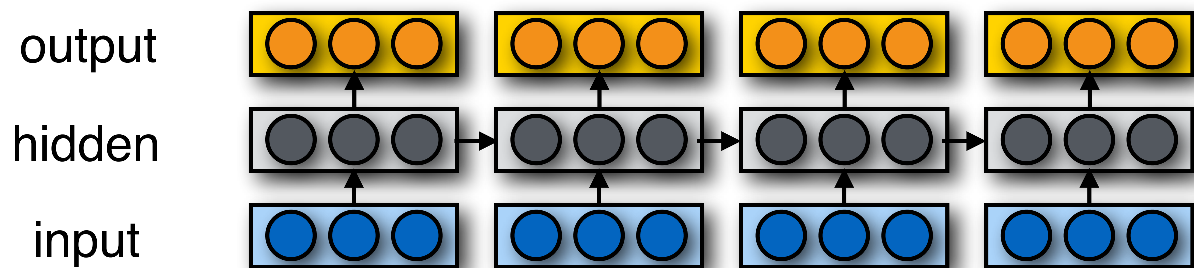
Recurrent neural networks (RNNs)

Basic RNN: Modify the standard feedforward architecture (which predicts a string $w_0 \ldots w_n$ one word at a time) such that the output of the current step (w_i) is given as additional input to the next time step (when predicting the output for w_{i+1}).

"Output" — typically (the last) hidden layer.



Feedforward Net



Recurrent Net

Word Embeddings (e.g. word2vec)

Main idea:
If you use a feedforward network to predict the probability of words that appear in the context of (near) an input word, the hidden layer of that network provides a dense vector representation of the input word.

Words that appear in similar contexts (that have high distributional similarity) wils have very similar vector representations.

These models can be trained on large amounts of raw text (and pretrained embeddings can be downloaded)

Sequence-to-sequence (seq2seq) models

Task (e.g. machine translation):

Given one variable length sequence as input,
return another variable length sequence as output

Main idea:

Use one RNN to encode the input sequence (“encoder”)
Feed the last hidden state as input to a second RNN (“decoder”) that then generates the output sequence.

From words to vectors

From words to vectors

We typically think of words as atomic symbols, but neural nets require input in vector form.

Naive solution: one-hot encoding (dim(**x**) = |V|)

- “a” = (1,0,0,…0), “aardvark” = (0,1,0,…,0), ….
- Very high-dimensional, very sparse vectors (most elements 0)
- No ability to generalize across similar words
- Still requires a lot of parameters.

How do we obtain low-dimensional, dense vectors?

- Low-dimensional => our models need far fewer parameters
- Dense => lots of elements are non-zero
- We also want words that are similar to have similar vectors

The Distributional Hypothesis

Zellig Harris (1954):

“oculist and eye-doctor … occur in almost the same environments”

“If A and B have almost identical environments we say that they are synonyms.”

John R. Firth 1957:

You shall know a word by the company it keeps.

The contexts in which a word appears tells us a lot about what it means.

Words that **appear in similar contexts** have similar meanings

Why do we care about word contexts?

What is tezgüino?

A bottle of tezgüino is on the table.
Everybody likes tezgüino.
Tezgüino makes you drunk.
We make tezgüino out of corn.
(Lin, 1998; Nida, 1975)

The contexts in which a word appears tells us a lot about what it means.

Vector representations of words

"Traditional" distributional similarity approaches represent words as sparse vectors

- Each dimension represents one specific context
- Vector entries are based on word-context co-occurrence statistics

Alternative, dense vector representations:

- We can use Singular Value Decomposition to turn these sparse vectors into dense vectors (Latent Semantic Analysis)
- We can also use classifiers or neural models to explicitly learn a dense vector representation (embedding) (word2vec, Glove, etc.)

  Sparse vectors = most entries are zero

  Dense vectors = most entries are non-zero

(Static) Word Embeddings

A (static) word embedding is a function that maps each word type to a single vector

— these vectors are typically dense and have much lower dimensionality than the size of the vocabulary

— this mapping function typically ignores that the same string of letters may have different senses (dining table vs. a table of contents) or parts of speech (to table a motion vs. a table)

— this mapping function typically assumes a fixed size vocabulary (so an UNK token is still required)

Word2Vec Embeddings

Main idea:
Use a binary classifier to predict which words appear in the context of (i.e. near) a target word.
The parameters of that classifier provide a dense vector representation of the target word (embedding)

Words that appear in similar contexts (that have high distributional similarity) will have very similar vector representations.

These models can be trained on large amounts of raw text (and pre-trained embeddings can be downloaded)

Word2Vec (Mikolov et al. 2013)

The first really influential dense word embeddings

Two ways to think about Word2Vec:

— a simplification of neural language models
— a binary logistic regression classifier

Variants of Word2Vec

— Two different context representations: CBOW or Skip-Gram
— Two different optimization objectives:
Negative sampling (NS) or hierarchical softmax

Skip-Gram with negative sampling

Train a binary classifier that decides whether a target word *t* appears in the context of other words $c_{1..k}$

- **Context**: the set of k words near (surrounding) *t*
- Treat the target word *t* and any word that *actually* appears in its context in a real corpus as **positive** examples
- Treat the target word *t* and *randomly sampled* words that don't appear in its context as **negative** examples
- Train a **binary logistic regression** classifier to distinguish these cases
- The **weights** of this classifier depend on the **similarity** of t and the words in $c_{1..k}$

Use the weights of this classifier as embeddings for *t*

Skip-Gram Goal

Given a tuple (t,c) = target, context

(*apricot*, *jam*)

(*apricot*, *aardvark*)

Return the probability that c is a real context word:

$P(D = + \mid t, c)$

$P(D = - \mid t, c) = 1 - P(D = + \mid t, c)$

Skip-Gram Training data

Training sentence:

... lemon, a tablespoon of **apricot** jam a pinch ...

c1 c2 t c3 c4

Training data: input/output pairs centering on *apricot*

Assume a +/- 2 word window

Positive examples:
(apricot, tablespoon), (apricot, of), (apricot, jam), (apricot, a)

For each positive example, create k **negative examples**, using noise words:

(apricot, aardvark), (apricot, puddle)…

The Skip-Gram classifier

Use logistic regression to predict whether the pair (t, c) (target word t and a context word c), is a positive or negative example:

$$P(+|t,c) = \frac{1}{1+e^{-t \cdot c}}$$

$$\begin{aligned} P(-|t,c) &= 1 - P(+|t,c) \\ &= \frac{e^{-t \cdot c}}{1+e^{-t \cdot c}} \end{aligned}$$

Assume that t and c are represented as vectors,
so that their dot product tc captures their similarity

To capture the entire context window $c_{1..k}$, assume the words in $c_{1:k}$ are independent (multiply) and take the log:

$$P(+|t,c_{1:k}) = \prod_{i=1}^{k} \frac{1}{1+e^{-t \cdot c_i}}$$

$$\log P(+|t,c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1+e^{-t \cdot c_i}}$$

Where do we get vectors t, c from?

Iterative approach:
Assume an initial set of vectors, and then adjust them during training to maximize the probability of the training examples.

Word2Vec: Negative Sampling

Training data: D+ ∪ D-

D+ = actual examples from training data

Where do we get D- from?

Lots of options.

Word2Vec: for each good pair (w,c), sample k words and add each w_i as a negative example (w_i,c) to D'

(D' is k times as large as D)

Words can be sampled according to corpus frequency or according to smoothed variant where freq'(w) = $\text{freq}(w)^{0.75}$

(This gives more weight to rare words)

Word2Vec: Negative Sampling

Training objective:
Maximize log-likelihood of training data D+ ∪ D-:

$$\mathscr{L}(\Theta, D, D') = \sum_{(w,c)\in D} \log P(D=1|w,c) + \sum_{(w,c)\in D'} \log P(D=0|w,c)$$

How to compute $p(+ \mid t, c)$?

Intuition:

Words are likely to appear near similar words

Model similarity with dot-product!

Similarity(t,c) $\propto t \cdot c$

Problem:

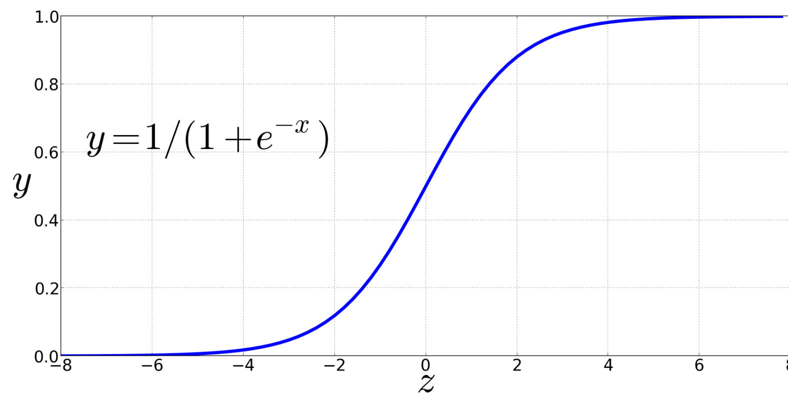
Dot product is not a probability!

(Neither is cosine)

Turning the dot product into a probability

The sigmoid lies between 0 and 1:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



$$P(+ \mid t, c) = \frac{1}{1 + exp(-t \cdot c)}$$

$$P(- \mid t, c) = 1 - \frac{1}{1 + exp(-t \cdot c)} = \frac{exp(-t \cdot c)}{1 + exp(-t \cdot c)}$$

Word2Vec: Negative Sampling

Distinguish "good" (correct) word-context pairs (D=1), from "bad" ones (D=0)

Probabilistic objective:

$P(D = 1 \mid t, c)$ defined by sigmoid:

$$P(D=1|w,c) = \frac{1}{1+exp(-s(w,c))}$$

$P(D = 0 \mid t, c) = 1 - P(D = 0 \mid t, c)$

$P(D = 1 \mid t, c)$ should be high when $(t, c) \in$ D+, and low when $(t,c) \in$ D-

Summary: How to learn word2vec (skip-gram) embeddings

For a vocabulary of size V: Start with V random 300-dimensional vectors as initial embeddings

Train a logistic regression classifier to distinguish words that co-occur in corpus from those that don’t

- Pairs of words that co-occur are positive examples
- Pairs of words that don't co-occur are negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance

Throw away the classifier code and keep the embeddings.

Evaluating embeddings

Compare to human scores on word similarity-type tasks:

WordSim-353 (Finkelstein et al., 2002)

SimLex-999 (Hill et al., 2015)

Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012)

TOEFL dataset: *Levied is closest in meaning to: imposed, believed, requested, correlated*

Properties of embeddings

Similarity depends on window size C

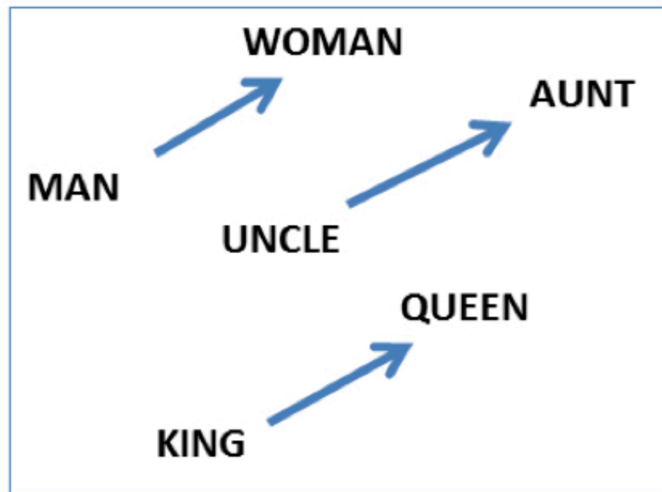
C = ±2 The nearest words to *Hogwarts:*
Sunnydale
Evernight

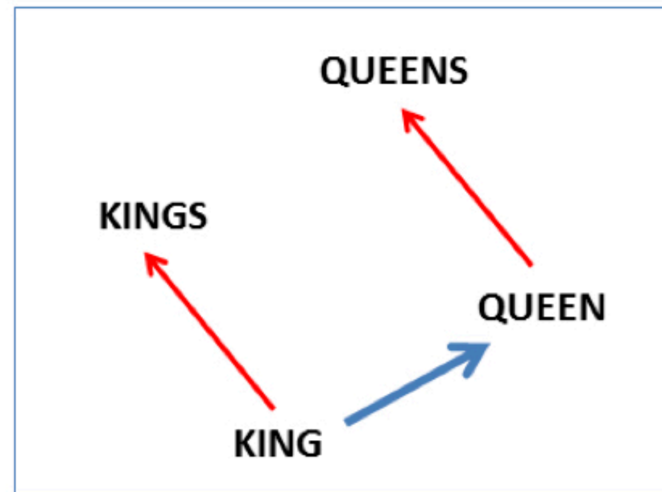
C = ±5 The nearest words to *Hogwarts:*
Dumbledore
Malfoy
halfblood

Analogy: Embeddings capture relational meaning!

vector(‘*king*’) - vector(‘*man*’) + vector(‘*woman*’) = vector(‘queen’)

vector(‘*Paris*’) - vector(‘*France*’) + vector(‘*Italy*’) = vector(‘*Rome*’)





Using Word Embeddings

Using pre-trained embeddings

Assume you have pre-trained embeddings E.
How do you use them in your model?

- Option 1: Adapt E during training
Disadvantage: only words in training data will be affected.
- Option 2: Keep E fixed, but add another hidden layer that is learned for your task
- Option 3: Learn matrix T ∈ dim(emb)×dim(emb) and use rows of E' = ET (adapts all embeddings, not specific words)
- Option 4: Keep E fixed, but learn matrix $\Delta \in R^{|V| \times dim(emb)}$ and use E' = E + Δ or E' = ET + Δ (this learns to adapt specific words)

More on embeddings

Embeddings aren't just for words!

You can take any discrete input feature (with a fixed number of K outcomes, e.g. POS tags, etc.) and learn an embedding matrix for that feature.

Where do we get the input embeddings from?

We can learn the embedding matrix during training.

Initialization matters: use random weights, but in special range (e.g. [-1/(2d), +(1/2d)] for d-dimensional embeddings), or use Xavier initialization

We can also use pre-trained embeddings

LM-based embeddings are useful for many NLP task

Dense embeddings you can download!

Word2vec (Mikolov et al.)
https://code.google.com/archive/p/word2vec/
Fasttext http://www.fasttext.cc/

Glove (Pennington, Socher, Manning)
http://nlp.stanford.edu/projects/glove/