CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 25
Neural Approaches to NLP

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Where we're at

Lecture 25: Word Embeddings and neural LMs
Lecture 26: Recurrent networks
Lecture 27: Sequence labeling and Seq2Seq
Lecture 28: Review for the final exam
Lecture 29: In-class final exam

Motivation

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.

Neural approaches to NLP

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.

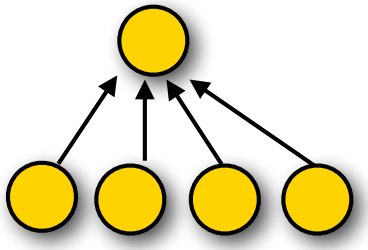
Why have they suddenly made a comeback?

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

What are neural nets?

Simplest variant: single-layer feedforward net

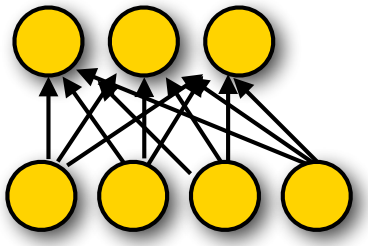
For binary classification tasks:
Single output unit
Return 1 if $y > 0.5$
Return 0 otherwise



Output unit: scalar *y*

Input layer: vector **x**

For multiclass classification tasks:
K output units (a vector)
Each output unit
y_i = class i
Return $\text{argmax}_i(y_i)$



Output layer: vector **y**

Input layer: vector **x**

Multiclass models: softmax(y_i)

Multiclass classification = predict one of K classes.

Return the class i with the highest score: $\text{argmax}_i(y_i)$

In neural networks, this is typically done by using the **softmax** function, which maps real-valued vectors in R^N into a distribution over the N outputs

For a vector $\mathbf{z} = (z_0 \ldots z_K)$: $P(i) = \text{softmax}(z_i) = \exp(z_i) \; / \; \sum_{k=0..K} \exp(z_k)$

(NB: This is just logistic regression)

Single-layer feedforward networks

Single-layer (linear) feedforward network

$y = \mathbf{wx} + b$ (binary classification)

w is a weight vector, b is a bias term (a scalar)

This is just a linear classifier (aka Perceptron)
(the output y is a linear function of the input **x**)

Single-layer non-linear feedforward networks:

Pass **wx** + b through a non-linear activation function,
e.g. $y = \tanh(\mathbf{wx} + b)$

Nonlinear activation functions

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

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

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

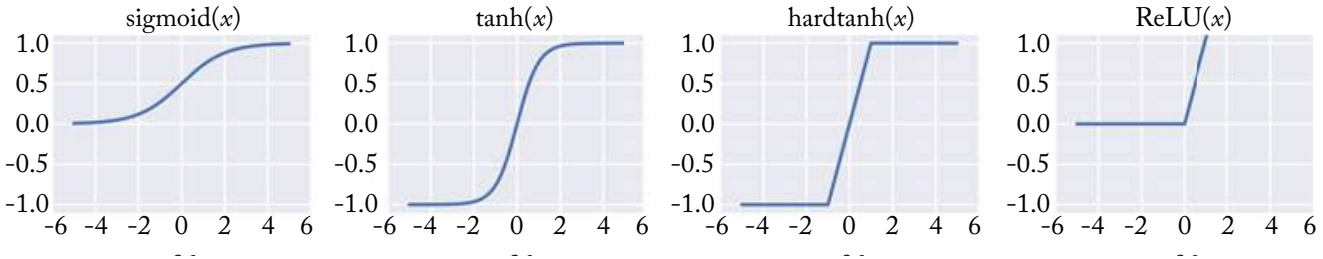
Useful for internal units: [-1,1] range

Hard tanh (approximates tanh)

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

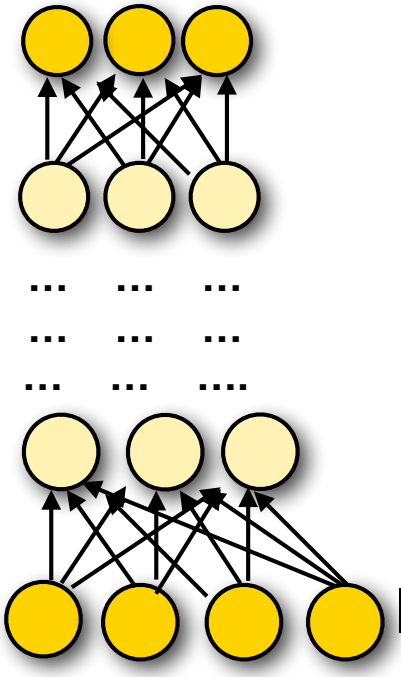
Rectified Linear Unit: $\text{ReLU}(x) = \max(0, x)$

Useful for internal units



Multi-layer feedforward networks

We can generalize this to multi-layer feedforward nets



Output layer: vector **y**

Hidden layer: vector $\mathbf{h_n}$

Hidden layer: vector $\mathbf{h_1}$

Input layer: vector **x**

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

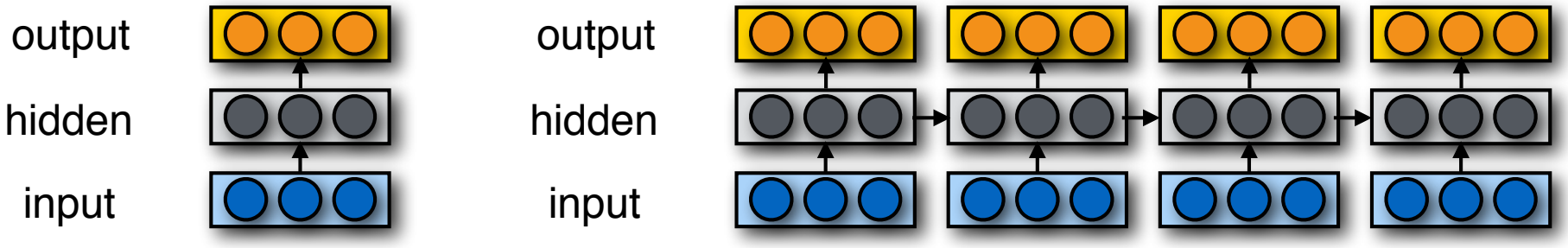
Input: Represent each preceding word by its d-dimensional embedding.

- 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)

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.

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:

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

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⟩

Shortcomings of traditional n-gram models

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

Solutions offered by neural models:

- Do not represent context words as distinct, discrete symbols, but use a dense vector representation where similar words have similar vectors [today's class]
- Use recurrent nets that can encode variable-lengths contexts [next class]

Neural n-gram models

Task: Represent $P(w \mid w_1 \ldots w_k)$ with a neural net

Assumptions:

- We'll assume each word $w_i \in V$ in the context is a dense vector $v(w)$: $v(w) \in R^{dim(emb)}$
- V is a finite vocabulary, containing UNK, BOS, EOS tokens.
- We'll use a feedforward net with one hidden layer $\mathbf{h}$

The input $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ to the NN represents the context $w_1 \ldots w_k$

Each $w_i \in V$ is a dense vector $v(w)$

The output layer is a softmax:

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

Neural n-gram models

Architecture:

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

$v(w) = E_{[w]}$

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

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

Parameters:

Embedding matrix: $\mathbf{E} \in \mathrm{R}^{|V| \times \dim(\text{emb})}$

Weight matrices and biases:

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

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

Neural n-gram models

Advantages over traditional n-gram models:

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

  $\mathbf{W}^1$ goes from $R^{k \cdot \dim(emb) \times \dim(\mathbf{h})}$ to $R^{(k+1) \cdot \dim(emb) \times \dim(\mathbf{h})}$
- Increasing the number of words in the vocabulary also leads only to a linear increase in the vocabulary
- Easy to incorporate more context: just add more input units
- Easy to generalize across contexts (embeddings!)

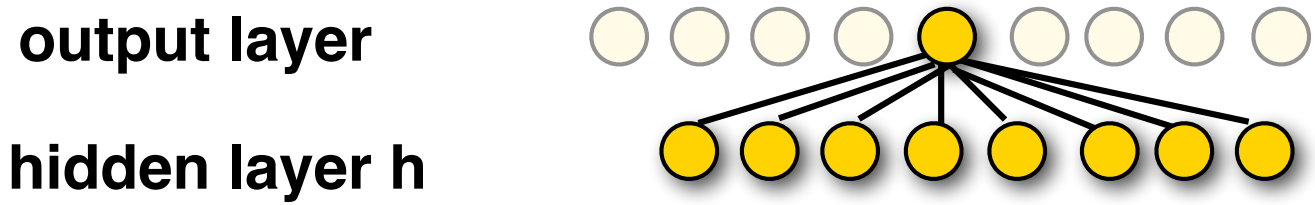
Computing softmax over large V is expensive:

requires matrix-vector multiplication with $\mathbf{W}^2$,
followed by |V| exponentiations

Solution (during training): only sample a subset of the vocabulary (or use hierarchical softmax)

Word representations as by-product of neural LMs

Output embeddings: Each column in $\mathbf{W}^2$ is a dim(**h**)-dimensional vector that is associated with a vocabulary item $w \in V$



h is a dense (non-linear) representation of the context
Words that are similar appear in similar contexts.
Hence their columns in $\mathbf{W}^2$ should be similar.

Input embeddings: each row in the embedding matrix is a representation of a word.

Modifications to neural LM

If we want good word representations (rather than good language models), we can modify this model:

1) We can also take the words that follow into account:
compute $P(w_3 \mid w_1 w_2 \_ w_4 w_5)$ instead of $P(w_5 \mid w_1 w_2 w_3 w_4)$
Now, the input context $c = w_1 w_2 \_ w_4 w_5$, not $w_1 w_2 w_3 w_4$
2) We don't need a distribution over the output word.
We just want the correct output word to have a higher score $s(w,c)$ than other words w'. We remove the softmax, define $s(w,c)$ as the activation of the output unit for w with input context c and use a (margin-based) ranking loss:
Loss for predicting a random word w' instead of w:

$$L(w, c, w') = \max(0, 1 - (s(w, c) - s(w', c))$$

Obtaining Word Embeddings

Word2Vec (Mikolov et al. 2013)

Modification of neural LM:

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

Task: train a classifier to predict a word from its context (or the context from a word)

Idea: Use the dense vector representation that this classifier uses as the embedding of the word.

CBOW vs Skip-Gram

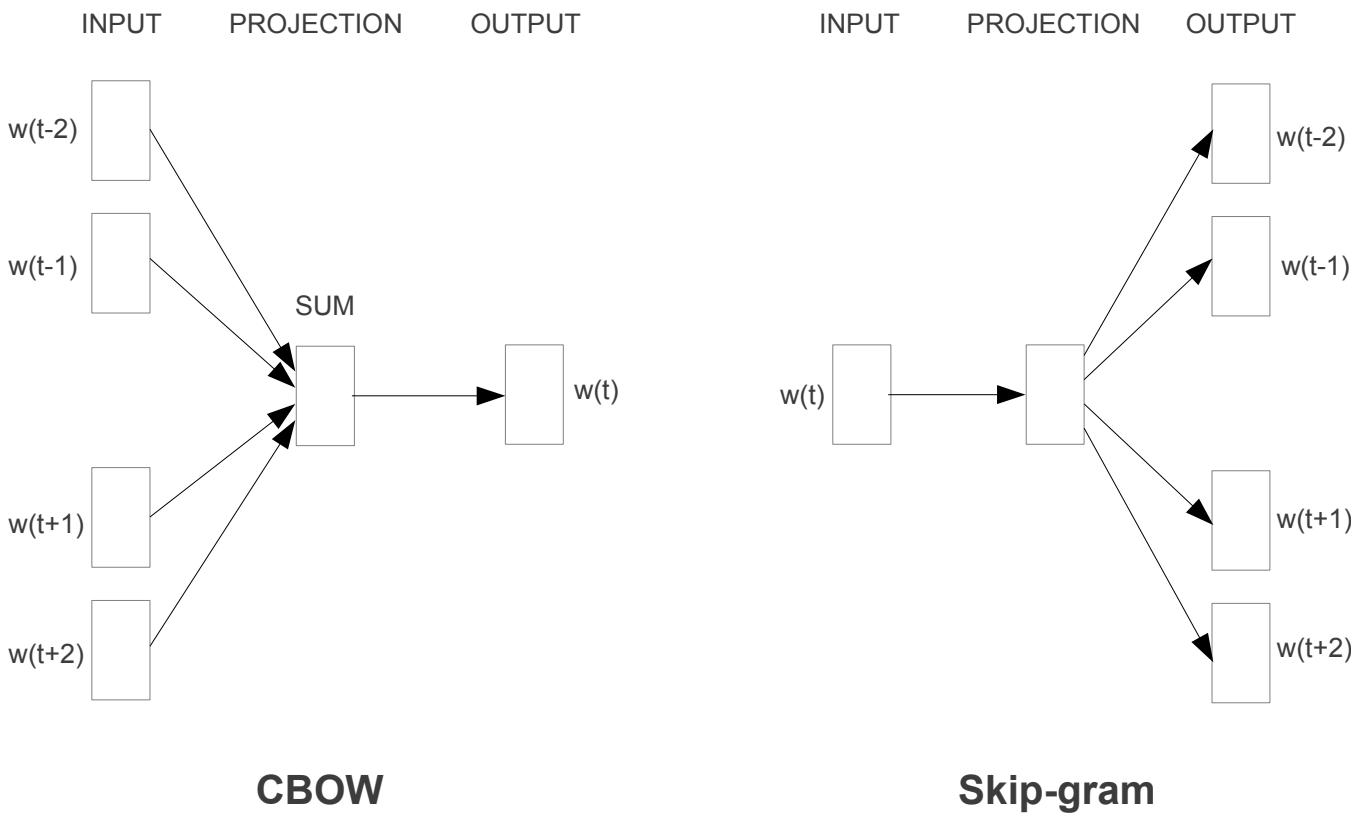


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

Word2Vec: CBOW

CBOW = Continuous Bag of Words

Remove the hidden layer, and the order information of the context.
Define context vector **c** as a sum of the embedding vectors of each context word c_i, and score s(**t**,**c**) as **tc**

$$\mathbf{c} = \sum_{i=1\ldots k} c_i$$

$$s(\mathbf{t}, \mathbf{c}) = \mathbf{tc}$$

$$P(+|t,c) = \frac{1}{1 + exp(-(t \cdot c_1 + t \cdot c_2 + \ldots + t \cdot c_k)}$$

Word2Vec: SkipGram

Don’t predict the current word based on its context, but predict the context based on the current word.

Predict surrounding C words (here, typically C = 10).
Each context word is one training example

Skip-gram algorithm

1. Treat the target word and a neighboring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the weights as the embeddings

Word2Vec: Negative Sampling

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

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

Skip-Gram Training Data

Training sentence:

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

c1 c2 target c3 c4

Asssume context words are those in +/- 2 word window

Skip-Gram Goal

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

(*apricot*, *jam*)
(*apricot*, *aardvark*)

Return the probability that c is a real context word:
$P(D = + \mid t, c)$
$P(D = - \mid t, c) = 1 - P(D = + \mid t, c)$

How to compute p(+ | t, c)?

Intuition:

Words are likely to appear near similar words

Model similarity with dot-product!

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

Problem:

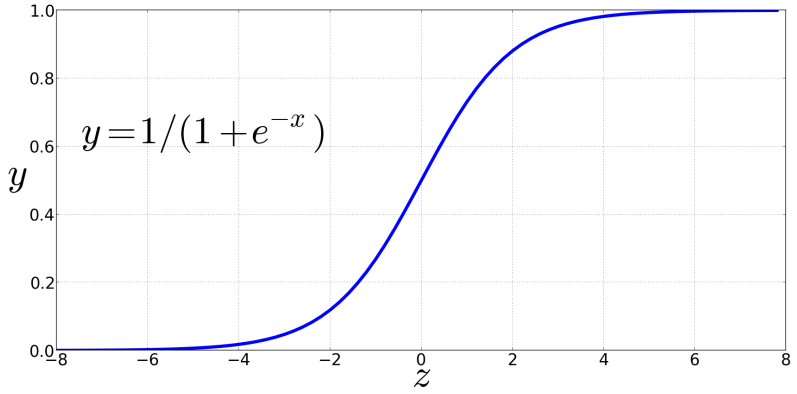
Dot product is not a probability!

(Neither is cosine)

Turning the dot product into a probability

The sigmoid lies between 0 and 1:

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



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

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

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 $(t,c) \in D-$

For all the context words

Assume all context words $c_{1:k}$ are independent:

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

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

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) = freq(w)$^{0.75}$

(This gives more weight to rare words)

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

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)…

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

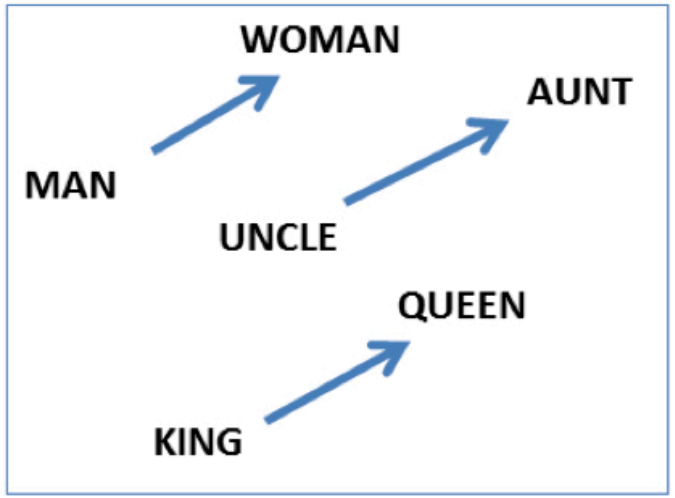
Malfoy

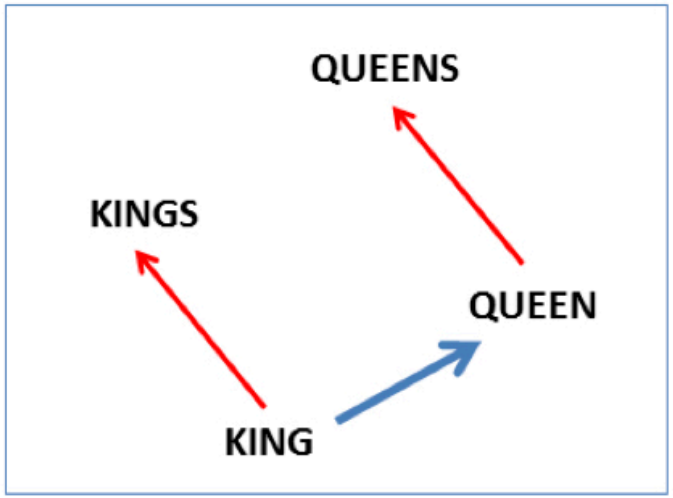
halfblood

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 $T \in$ 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/