CS546: Machine Learning in NLP (Spring 2020)

http://courses.engr.illinois.edu/cs546/

Lecture 3: From neural language models to static word embeddings

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Office hours: Monday, 11am—12:30pm

Today's lecture

How does NLP use neural nets (wrap-up)

Neural language models:

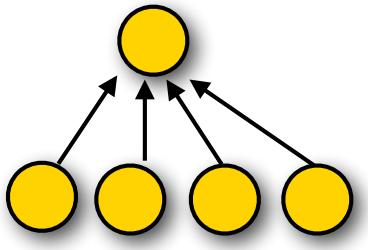
- Feedforward nets
- Recurrent nets

From words to vectors: Word2Vec

What are neural nets?

Simplest variant: single-layer feedforward net

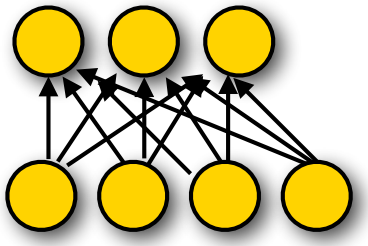
For binary classification tasks:
Single output unit
Return 1 if y > 0.5
Return 0 otherwise



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

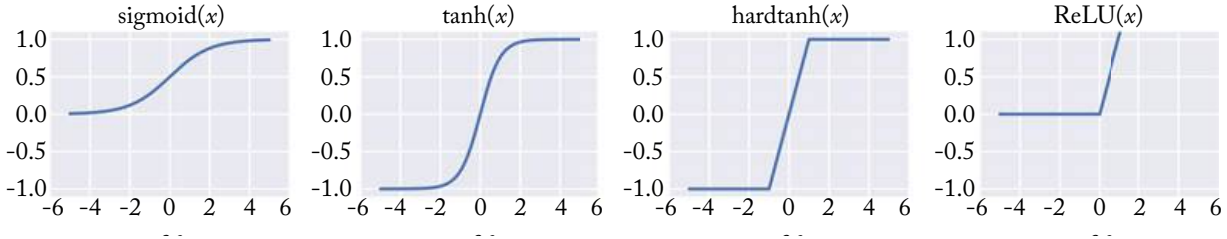
Useful for internal units: [-1,1] range

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

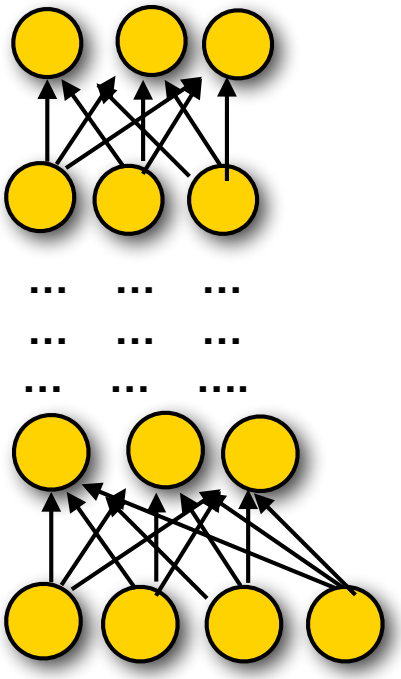


Softmax: $\text{softmax}(z_i) = \exp(z_i) \ / \ \sum_{k=0..K} \exp(z_k)$

Special case for output units (multiclass classification)

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

In NLP, the 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?

- With 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.;
- With convolutional nets: use a sliding (fixed-length) window)

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) to get a dense vector representation of each word

Neural language models:

Use recurrent neural networks (RNNs, GRUs, LSTMs) to predict word sequences (or to obtain context-sensitive embeddings (ELMO)

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.

Convolutional neural nets

Used e.g. for text classification

Transformers

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⟩

A naive neural n-gram model $P(w \mid w_1 \ldots w_{n-1})$

— The **vocabulary** V contains v types (incl. UNK, BOS, EOS)
— We want to condition each word on n-1 preceding words

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

$$P(w \mid w_1 \ldots w_{n-1}) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

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

Naive neural n-gram model

Architecture:

Input Layer: $\mathbf{x} = [v(w_1)\ldots v(w_{n-1})]$

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

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

Output Layer: $P(w \mid w_1 \ldots w_{n-1}) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

Parameters:

Weight matrices and biases:

first layer: $\mathbf{W}^1 \in \mathbb{R}^{(n-1)\times v \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

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

How many parameters do we need to learn?

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

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

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

#parameters going to hidden layer: $(n-1)\cdot\dim(V)\cdot\dim(\mathbf{h})$,
with $\dim(\mathbf{h}) = 300$, dim(V) = 10,000 and n-1=2: 6,000,000

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 (n-1) · dim(emb)×dim(**h**) to n · dim(emb)×dim(**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]

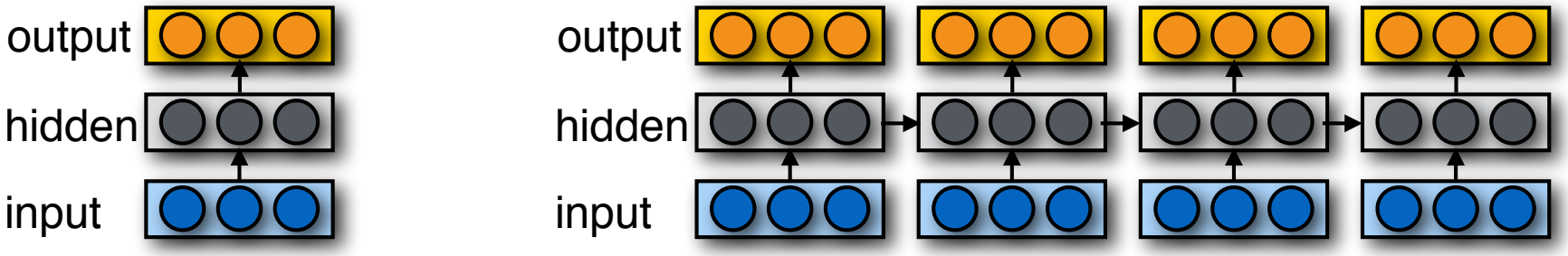
… using **recurrent nets** instead of feedforward nets:

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

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) will have very similar vector representations.

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

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 classifier to predict which words appear in the context of (i.e. near) a target word (or vice versa)
This classifier induces a dense vector representation of words (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

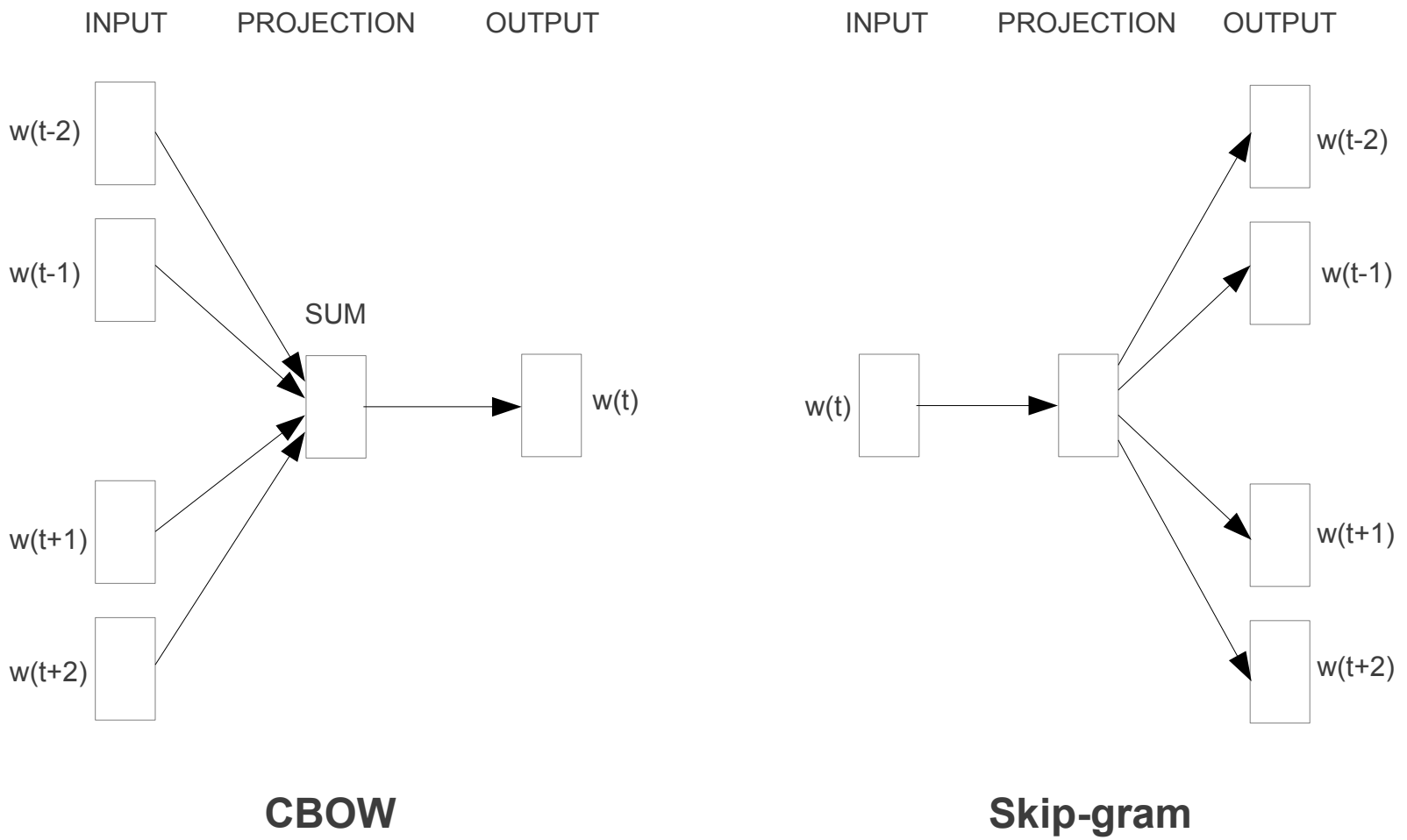


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

CBOW

(CBOW=Continuous Bag of Words)

Training sentence:

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

c1 c2 t c3 c4

Given the surrounding context words (tablespoon, of, jam, a), predict the target word (apricot).

Input: each context word is a one-hot vector
Projection layer: map each one-hot vector down to a dense D-dimensional vector, and average these vectors
Output: predict the target word with 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:

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

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

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 (in reality: use +/- 10 words)

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)…

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)

Word2Vec: Negative Sampling

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

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

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:

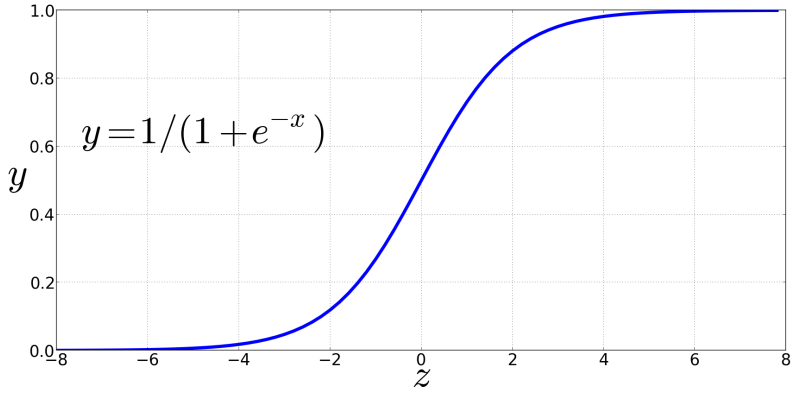
Dot product is not a probability!

(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 $(t,c) \in D-$

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 (gradient descent):
Assume an initial set of vectors, and then adjust them during training to maximize the probability of the training examples.

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

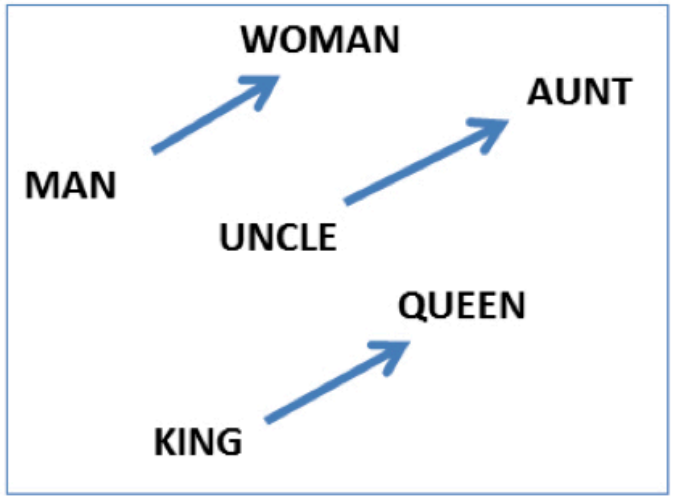
Malfoy

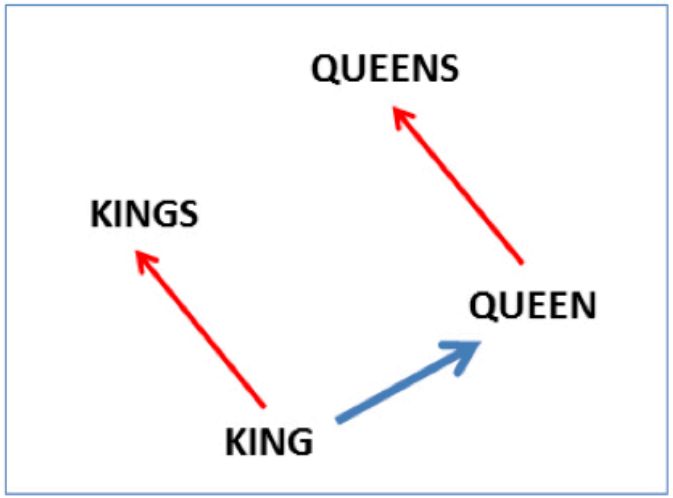
halfblood

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 $T \in \text{dim(emb)} \times \text{dim(emb)}$ and use rows of $E' = ET$ (adapts all embeddings, not specific words)
- Option 4: Keep E fixed, but learn matrix $\Delta \in R^{|V| \times \text{dim(emb)}}$ and use $E' = E + \Delta$ or $E' = ET + \Delta$ (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/