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

Lecture 9: Word2Vec and basic intro to RNNs

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center

Class Admin

Assignments:
MP1: due 11:59pm Monday, Sept 30
MP2: will be released later today.

Midterm: Friday, Oct 11 in class
Closed book exam, short questions

4th Credit: Proposal due Friday, Oct 4
(via Compass)
We'll release a template later today.
We want to make sure that you have a topic,
that you've started to look at relevant papers,
and that your project is realistic.

Words as input to neural models

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

Naive solution: one-hot encoding ($\dim(\mathbf{x}) = |V|$)

“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

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 (counts or PMI values)

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 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

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

Word2Vec Embeddings

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

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)

Skip-Gram with negative sampling

Train a **binary logistic regression classifier** to decide whether target word t does or doesn't appear in the context of words $c_{1..k}$

- **"Context":** the set of k words near (surrounding) t
- **Positive (+) examples**: t and any word c in its context
- **Negative (–) examples**: t and *randomly sampled* words c'
- **Training objective:** maximize the probability of the correct label $P(+ \mid t, c)$ or $P(- \mid t, c)$ of these examples
- This classifier represents t and c as **vectors (embeddings)**

  It has two sets of parameters:

  a) a matrix of **target embeddings** to represent target words,

  b) a matrix of **context embeddings** to represent context words
- $P(+ \mid t, c) = \dfrac{1}{1 + \exp(-t \cdot c)}$ depends on **similarity** (dot product) of t, c

Use the **target embeddings** as word embeddings.

Skip-Gram Goal (during training)

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 (Negative Sampling)

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 (for target *apricot*)
(*apricot*, *tablespoon*), (*apricot*, of), (*apricot*, *jam*), (*apricot*, *a*)

Negative examples (for target *apricot*)
For each positive example, sample k **negative examples**, using noise words: (*apricot, aardvark*), (*apricot*, *puddle*)…
Noise words can be sampled according to corpus frequency or according to a smoothed variant where $\text{freq}'(w) = \text{freq}(w)^{0.75}$
(This gives more weight to rare words)

Word2Vec: Negative Sampling

D^+: all positive training examples,
D^-: all negative trainng examples

Training objective:
Maximize log-likelihood of training data $D^+ \cup D^-$:

$$\mathscr{L}(D) = \sum_{(t,c)\in D^+} \log P(D=+\mid t,c) + \sum_{(t,c)\in D^-} \log P(D=-\mid t,c)$$

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

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.

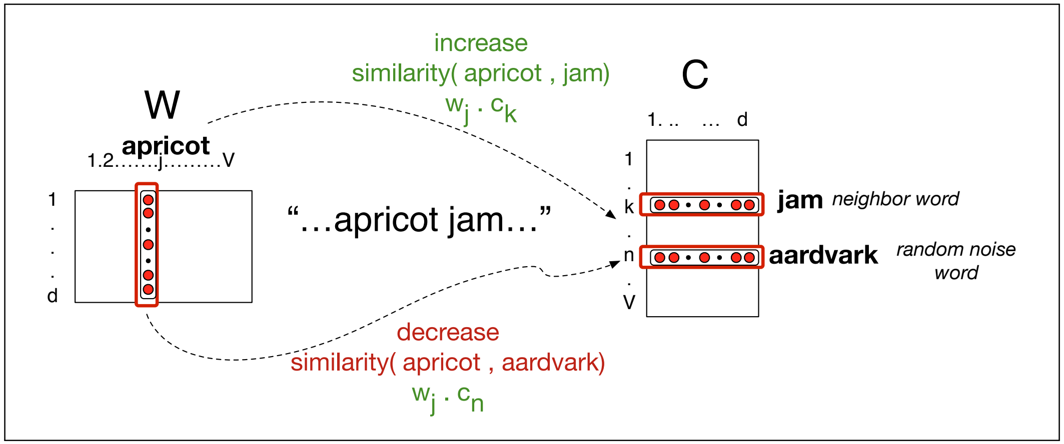


Figure 6.13 The skip-gram model tries to shift embeddings so the target embedding (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (have a lower dot product with) context embeddings for words that don't occur nearby (here *aardvark*).

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

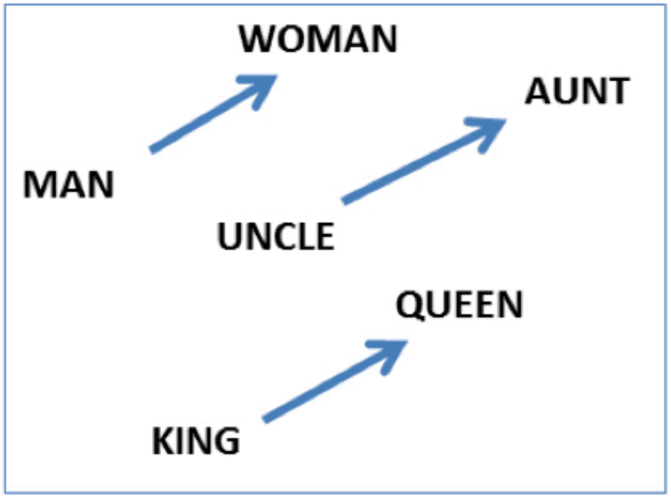
Malfoy

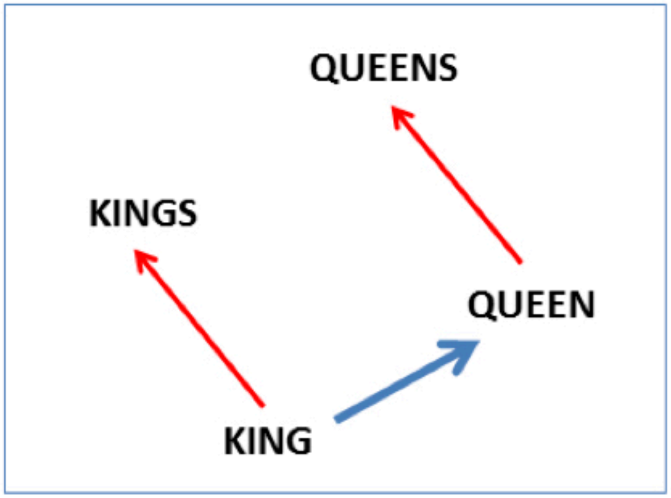
halfblood

Analogy: Embeddings capture relational meaning!

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

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





Dense embeddings you can download!

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

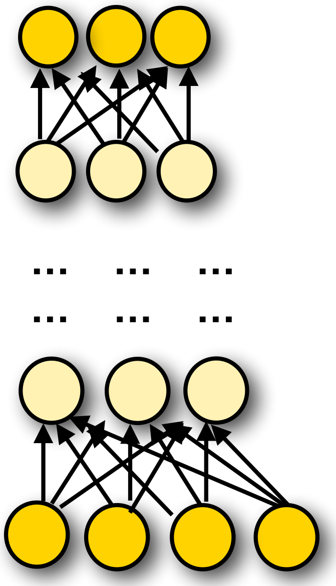
Glove (Pennington, Socher, Manning)
http://nlp.stanford.edu/projects/glove/

Recurrent Neural Nets (RNNs)

Recap: 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)



Output layer: vector **y**

Hidden layer: vector $\mathbf{h}_n$

Hidden layer: vector $\mathbf{h}_1$

Input layer: vector **x**

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

Recurrent Neural Nets (RNNs)

The input to a feedforward net has a fixed size.

How do we handle variable length inputs?
In particular, how do we handle variable length sequences?

RNNs handle variable length sequences

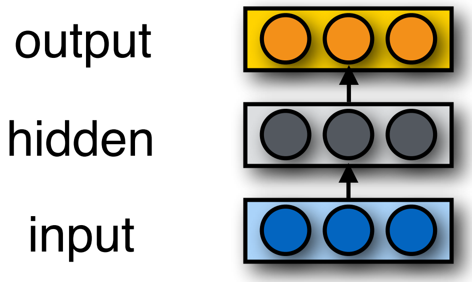
There are 3 main variants of RNNs, which differ in their internal structure:

basic RNNs (Elman nets)
LSTMs
GRUs

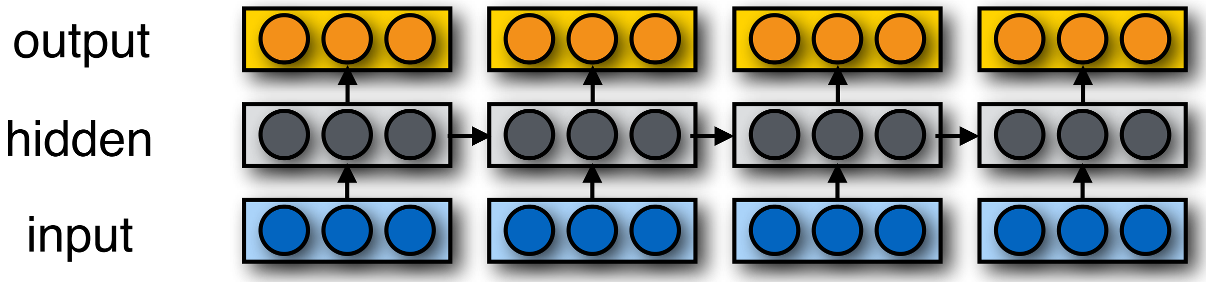
Recurrent neural networks (RNNs)

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.



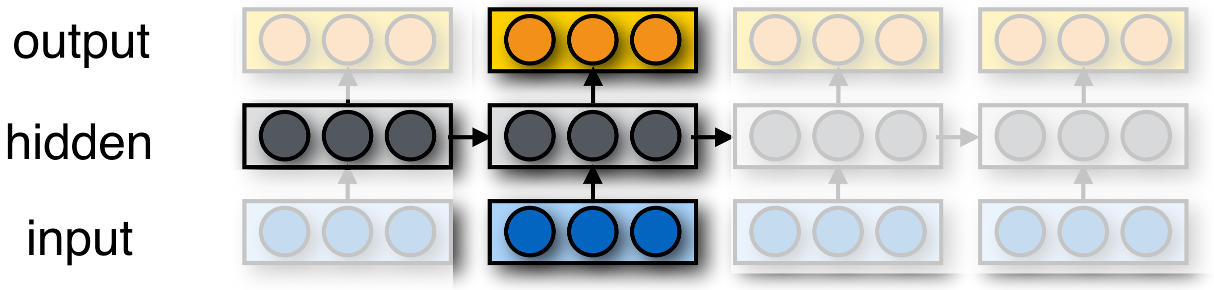
Feedforward Net



Recurrent Net

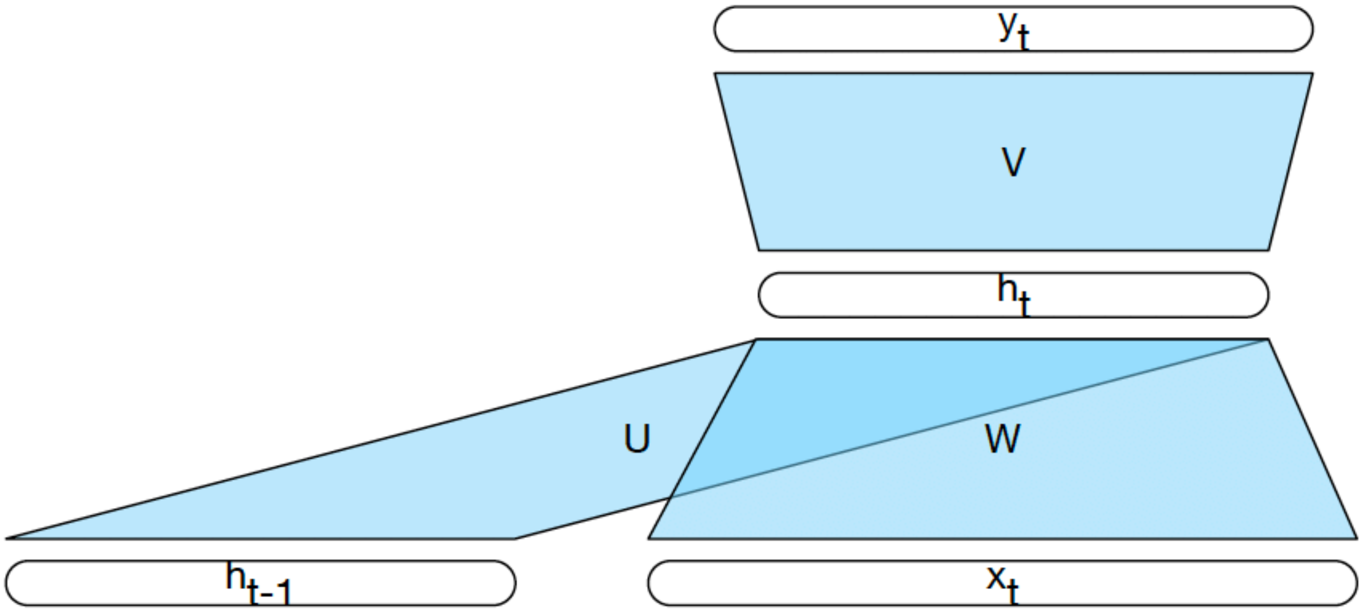
Basic RNNs

Each time step corresponds to a feedforward net where the hidden layer gets its input not just from the layer below but also from the activations of the hidden layer at the previous time step

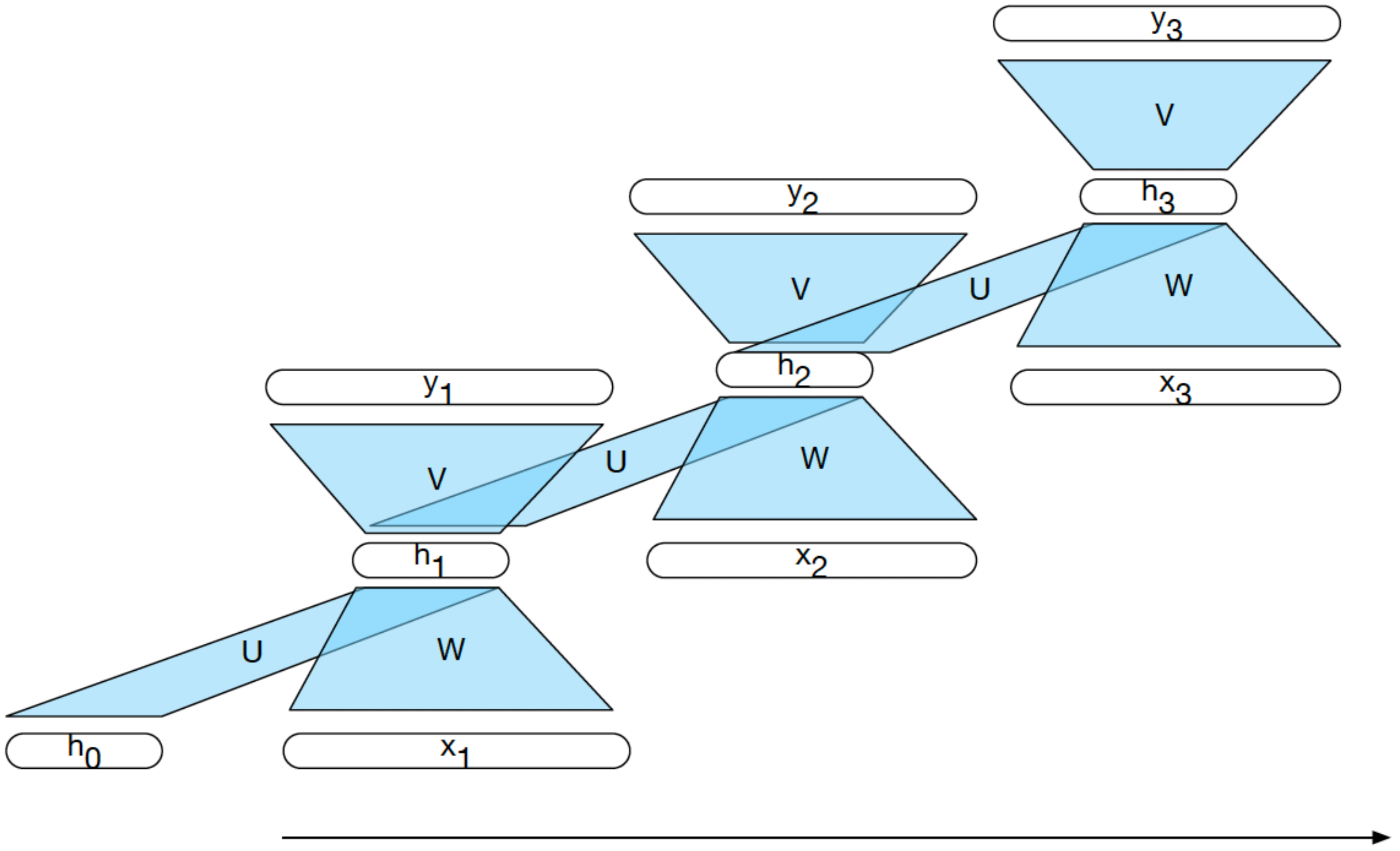


Basic RNNs

Each time step corresponds to a feedforward net where the hidden layer gets its input not just from the layer below but also from the activations of the hidden layer at the previous time step



A basic RNN unrolled in time



RNNs for language modeling

If our vocabulary consists of V words, the output layer (at each time step) has V units, one for each word.

The softmax gives a distribution over the V words for the next word.

To compute the probability of a string $w_0 w_1 \ldots w_n\ w_{n+1}$ (where $w_0 = \text{<s>}$, and $w_{n+1} = \text{<\textbackslash s>}$), feed in w_i as input at time step i and compute

$$\prod_{i=1..n+1} P(w_i \mid w_0 \ldots w_{i-1})$$

RNNs for language generation

To generate a string $w_0 w_1 \ldots w_n\ w_{n+1}$ (where w_0 = <s>, and w_{n+1} = <\s>), give w_0 as first input, and then pick the next word according to the computed probability

$$P(w_i \mid w_0 \ldots w_{i-1})$$

Feed this word in as input into the next layer.

Greedy decoding: always pick the word with the highest probability

(this only generates a single sentence — why?)

Sampling: sample according to the given distribution

RNNs for sequence classification

If we just want to assign a label to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

We can use the hidden state of the last word in the sequence as input to a feedforward net: