CS447: Natural Language Processing

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

Lecture 26
Word Embeddings and Recurrent Nets

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Where we're at

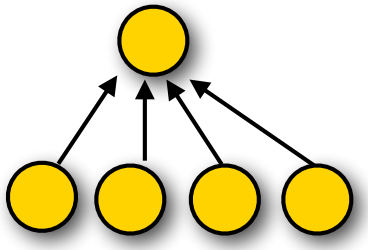
Lecture 25: Word Embeddings and neural LMs
Lecture 26: Recurrent networks
Lecture 27: Sequence labeling and Seq2Seq
Lecture 28: Review for the final exam
Lecture 29: In-class final exam

Recap

What are neural nets?

Simplest variant: single-layer feedforward net

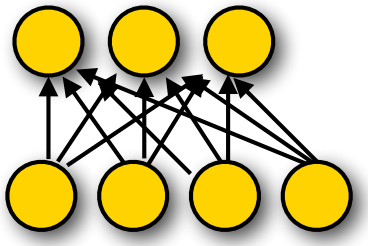
For binary classification tasks:
Single output unit
Return 1 if $y > 0.5$
Return 0 otherwise



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 $\text{argmax}_i(y_i)$

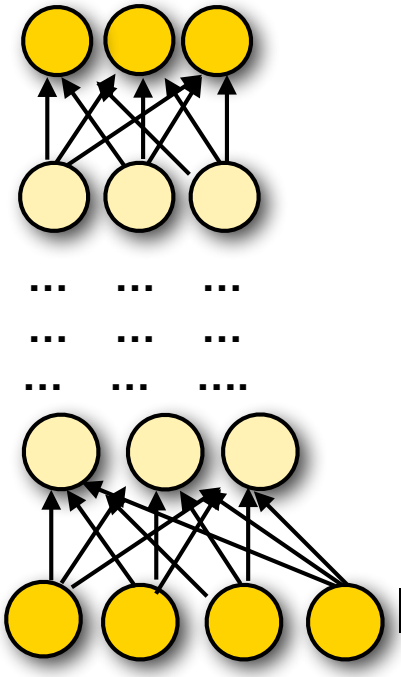


Output layer: vector **y**

Input layer: vector **x**

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**

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)

Neural Language Models

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**

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

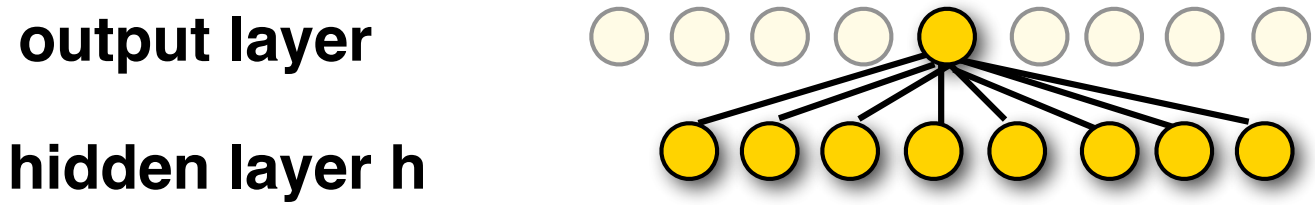
Weight matrices and biases:

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

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.

Obtaining Word Embeddings

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)

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

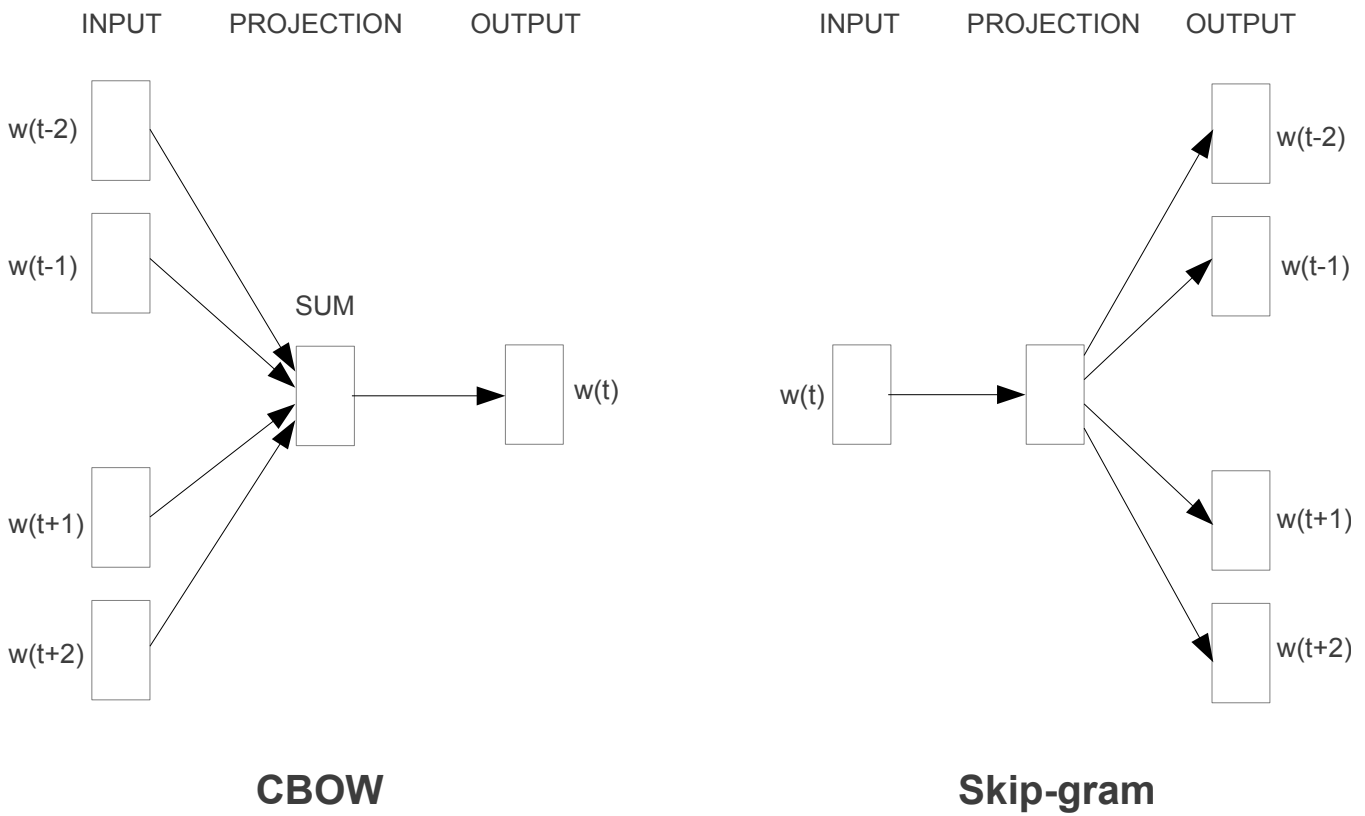


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

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:

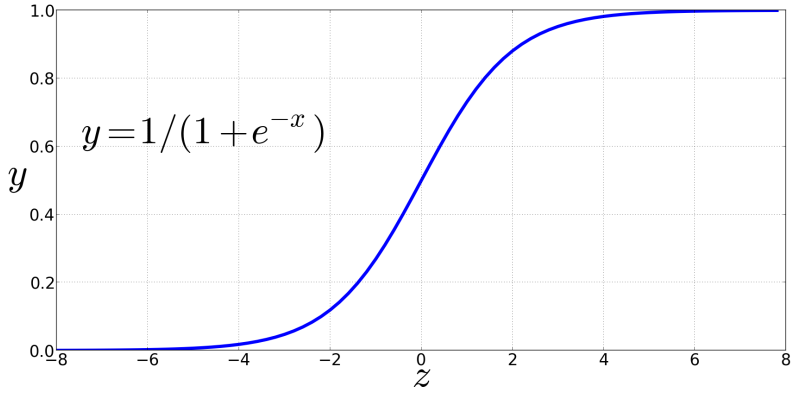
Dot product is not a probability!

(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

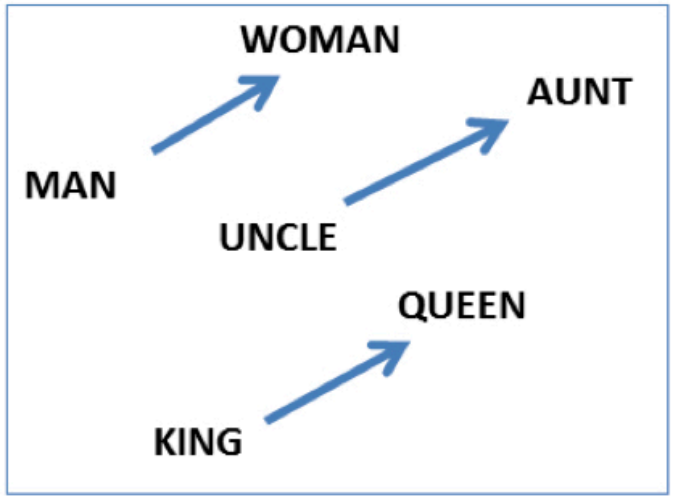
Malfoy

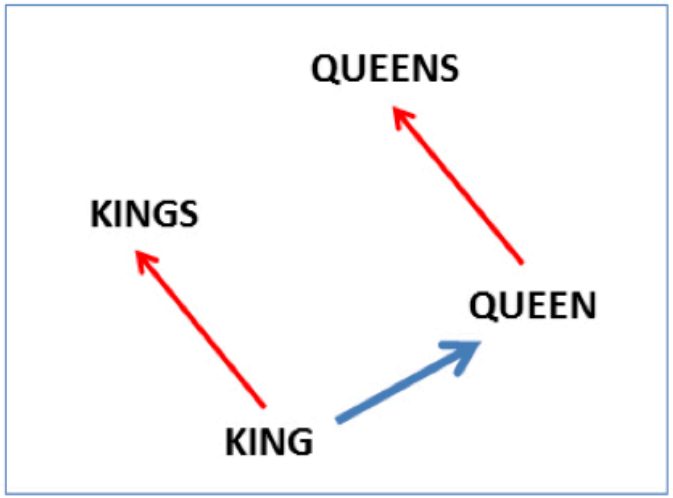
halfblood

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/

Recurrent Neural Nets (RNNs)

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

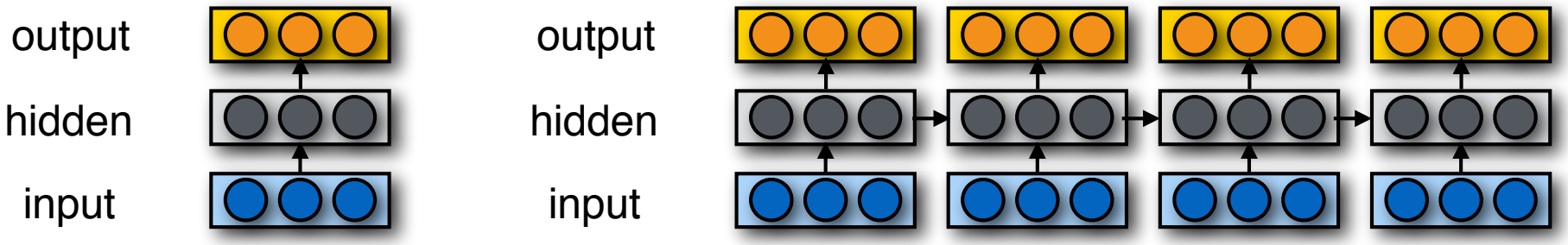
There are 3 main variants of RNNs, which differ in their internal structure:

- 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}).

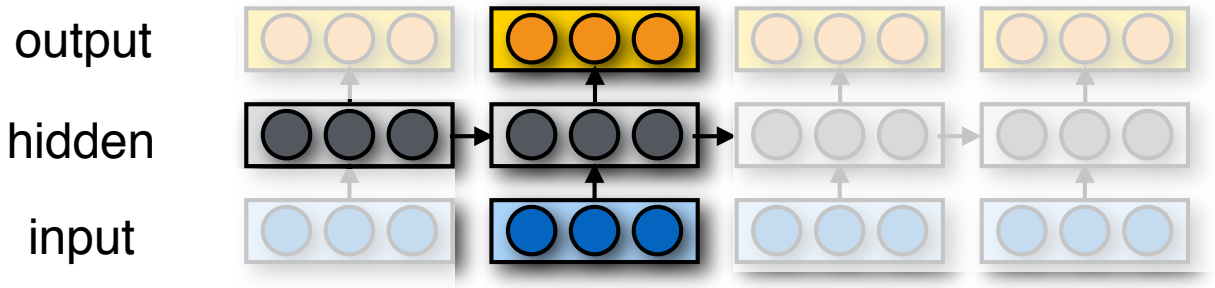
"Output" — typically (the last) hidden layer.



Feedforward Net **Recurrent Net**

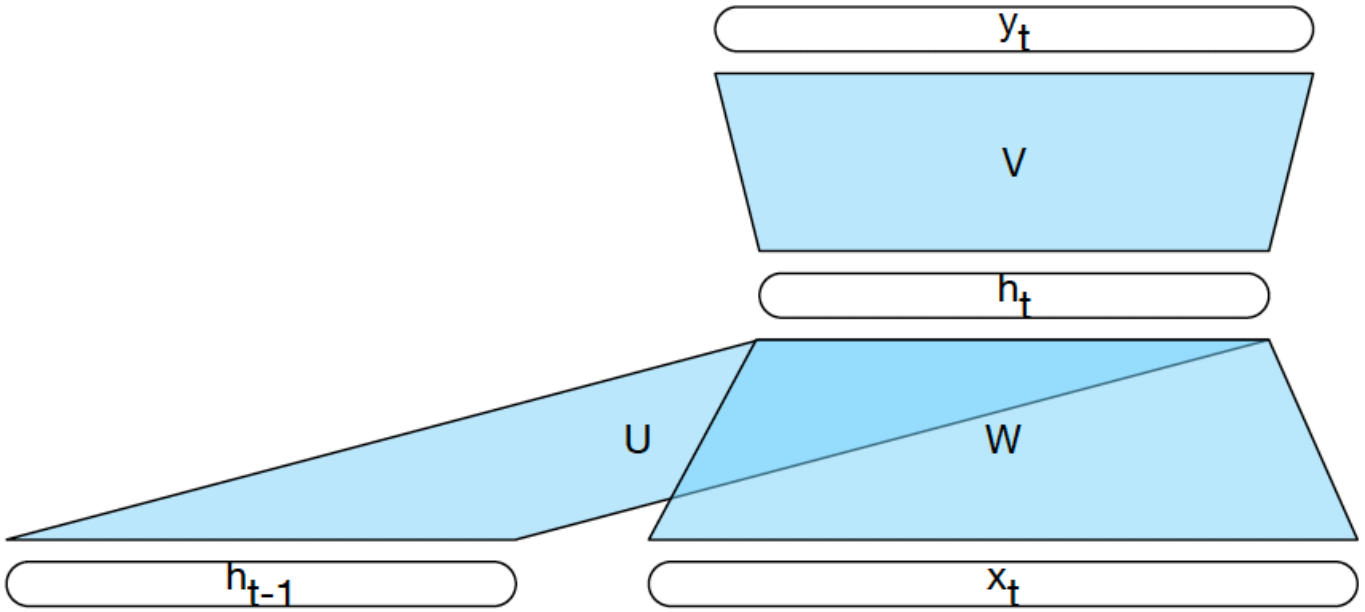
Basic RNNs

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

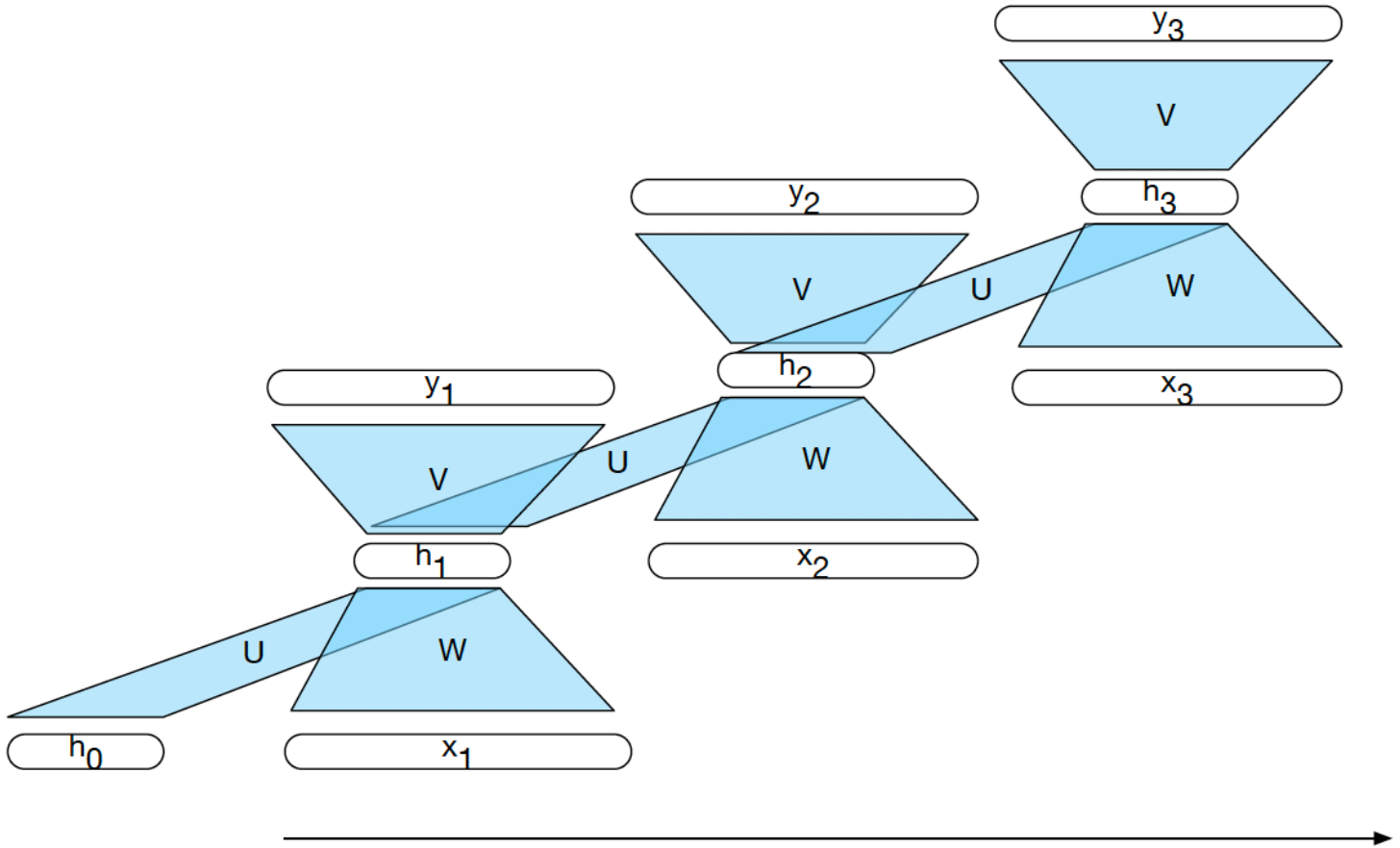


Basic RNNs

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 = <s>, and w_{n+1} = <\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 labeling

In sequence labeling, we want to assign a label or tag t_i to each word w_i

Now the output layer gives a distribution over the T possible tags.

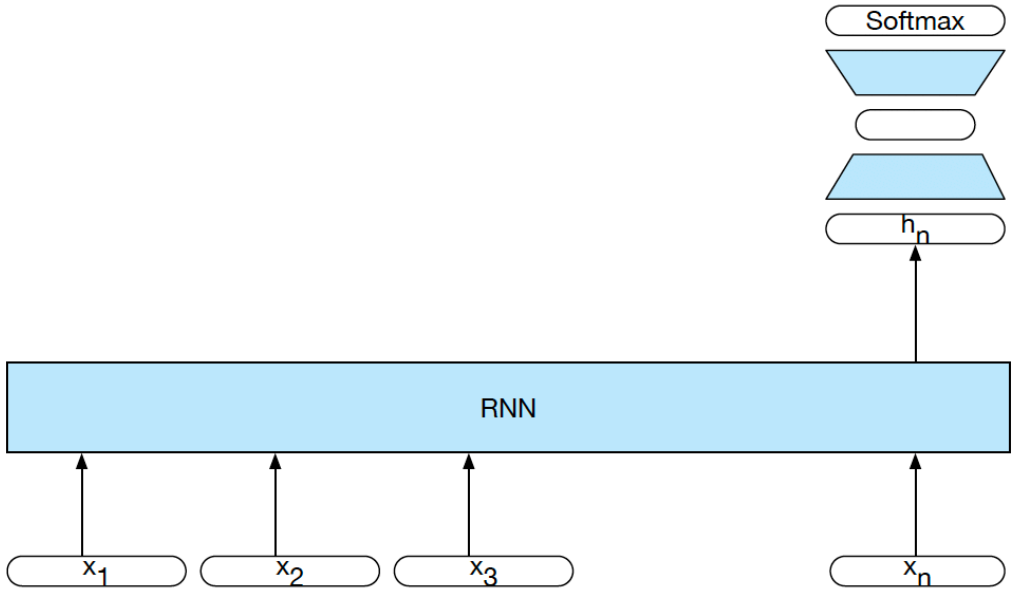
The hidden layer contains information about the previous words and the previous tags.

To compute the probability of a tag sequence $t_1 \ldots t_n$ for a given string $w_1 \ldots w_n$ feed in w_i (and possibly t_{i-1}) as input at time step i and compute $P(t_i \mid w_1 \ldots w_{i-1}, t_1 \ldots t_{i-1})$

RNNs for sequence classification

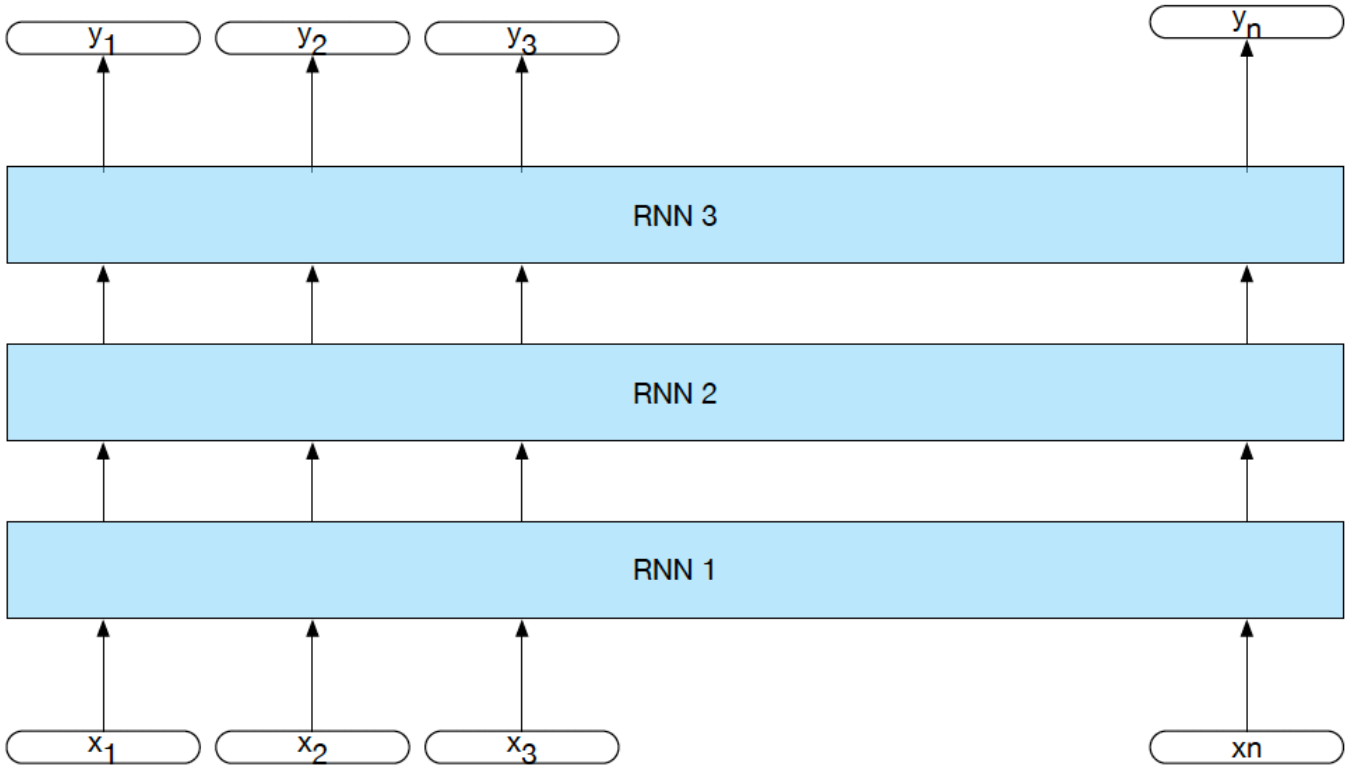
If we just want to assign a label to the entire sequence, we don't need to produce output at each time step, so we can use a simpler architecture.

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



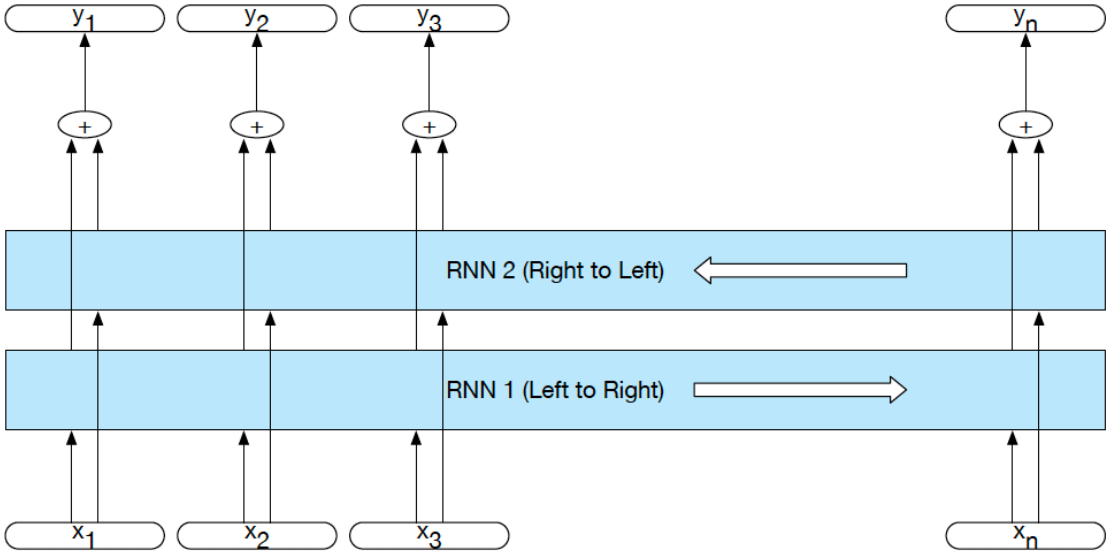
Stacked RNNs

We can create an RNN that has “vertical” depth (at each time step) by stacking:



Bidirectional RNNs

Unless we need to generate a sequence, we can run two RNNs over the input sequence — one in the forward direction, and one in the backward direction. Their hidden states will capture different context information.



Further extensions

Character and substring embeddings

- We can also learn embeddings for individual letters.
- This helps generalize better to rare words, typos, etc.
- These embeddings can be combined with word embeddings (or used instead of an UNK embedding)

Context-dependent embeddings (ELMO, BERT, ….)

- Word2Vec etc. are static embeddings: they induce a type-based lexicon that doesn't handle polysemy etc.
- Context-dependent embeddings produce token-specific embeddings that depend on the particular context in which a word appears.