CS546: Machine Learning in NLP (Spring 2018)

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

Lecture 3
More background/ motivation

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Office hours: Tue/Thu 2pm-3pm

Today's lecture

Super-quick recap of "traditional" statistical NLP:

- Distributional similarities (words as vectors)
- Language modeling (predicting word sequences)
- Sequence labeling
- Parsing with CFGs and with dependencies

How does neural NLP differ from these models?

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”?

Review: Language modeling with n-grams

N-gram models

Unigram model	$P(w_1)P(w_2)...P(w_i)$
Bigram model	$P(w_1)P(w_2\|w_1)...P(w_i\|w_{i-1})$
Trigram model	$P(w_1)P(w_2\|w_1)...P(w_i\|w_{i-2}w_{i-1})$
N-gram model	$P(w_1)P(w_2\|w_1)...P(w_i\|w_{i-n-1}...w_{i-1})$

N-gram models assume each word (event) depends only on the previous n–1 words (events).
Such independence assumptions are called Markov assumptions (of order n–1).

$$P(w_i|w_1...w_{i-1}) :\approx P(w_i|w_{i-n-1}...w_{i-1})$$

Parameter estimation (training)

Parameters: the actual probabilities

$$P(w_i = \text{`the'} \mid w_{i-1} = \text{`on'}) = ???$$

We need (a large amount of) text as training data to estimate the parameters of a language model.

The most basic estimation technique:
relative frequency estimation (= counts)

$$P(w_i = \text{`the'} \mid w_{i-1} = \text{`on'}) = C(\text{`on the'}) / C(\text{`on'})$$

Also called Maximum Likelihood Estimation (MLE)

MLE assigns *all* probability mass to events that occur in the training corpus.

How do we use language models?

Independently of any application, we can use a language model as a random sentence generator
(i.e we sample sentences according to their language model probability)

Systems for applications such as machine translation, speech recognition, spell-checking, generation, often produce multiple candidate sentences as output.

- We prefer output sentences S_{Out} that have a higher probability
- We can use a language model $P(S_{Out})$ to score and rank these different candidate output sentences, e.g. as follows:

$$\text{argmax}_{SOut}\ P(S_{Out} \mid \text{Input}) = \text{argmax}_{SOut}\ P(\text{Input} \mid S_{Out})P(S_{Out})$$

Generating Shakespeare

Unigram	• To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have • Every enter now severally so, let • Hill he late speaks; or! a more to leg less first you enter • Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like
Bigram	• What means, sir. I confess she? then all sorts, he is trim, captain. •Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. •What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman? •Enter Menenius, if it so many good direction found'st thou art a strong upon command of fear not a liberal largess given away, Falstaff! Exeunt
Trigram	• Sweet prince, Falstaff shall die. Harry of Monmouth's grave. • This shall forbid it should be branded, if renown made it empty. • Indeed the duke; and had a very good friend. • Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.
Quadrigram	• King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; • Will you not tell me who I am? • It cannot be but so. • Indeed the short and the long. Marry, 'tis a noble Lepidus.

MLE doesn't capture unseen events

We estimated a model on 440K word tokens, but:

Only 30,000 word types occur in the training data
Any word that does not occur in the training data has zero probability!

Only 0.04% of all possible bigrams (over 30K word types) occur in the training data
Any bigram that does not occur in the training data has zero probability (even if we have seen both words in the bigram)

How do we evaluate models?

Define an evaluation metric (scoring function).

We will want to measure how similar the predictions of the model are to real text.

Train the model on a *'seen'* training set

Perhaps: tune some parameters based on held-out data (disjoint from the training data, meant to emulate unseen data)

Test the model on an *unseen* test set

(usually from the same source (e.g. WSJ) as the training data)

Test data must be disjoint from training and held-out data

Compare models by their scores

Handling unseen events

Traditional language models require sophisticated smoothing techniques to

— predict the probabilities of unseen words (in seen or unseen contexts)

— predict the probabilities of (known) words in unseen contexts

Review: Distributional Similarities

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)

Distributional hypothesis:
You shall know a word by the company it keeps.
(Firth 1957)

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

Distributional similarities

Distributional similarities use the set of contexts in which words appear to measure their similarity.

They represent each word *w* as a vector $\mathbf{w}$

$$\mathbf{w} = (w_1, \ldots, w_N) \in \mathbf{R}^N$$

in an N-dimensional vector space.

- Each dimension corresponds to a particular context c_n
- Each element w_n of $\mathbf{w}$ captures the degree to which the word *w* is associated with the context c_n.
- w_n depends on the co-occurrence counts of *w* and c_n

The similarity of words *w* and *u* is given by the similarity of their vectors $\mathbf{w}$ and $\mathbf{u}$

What is a 'context'?

There are many different definitions of context that yield different kinds of similarities:

Contexts defined by nearby words:

How often does *w* appear near the word *drink*?

Near = "*drink* appears within a window of ±k words of *w*", or "*drink* appears in the same sentence as *w*"

This yields fairly broad thematic similarities.

Contexts defined by grammatical relations:

How often is (the noun) *w* used as the subject (object) of the verb *drink*? (Requires a parser).

This gives more fine-grained similarities.

Using nearby words as contexts

- Decide on a fixed vocabulary of N context words $c_1..c_N$
  Context words should occur frequently enough in your corpus that you get reliable co-occurrence counts, but you should ignore words that are too common ('stop words': *a*, *the, on, in, and, or, is, have,* etc.)

- Define what 'nearby' means
  For example: *w* appears near *c if* c appears within ±5 words of *w*

- Get co-occurrence counts of words *w* and contexts *c*

- Define how to transform co-occurrence counts of words *w* and contexts *c* into vector elements w_n
  For example: compute PMI of words and contexts

- Define how to compute the similarity of word vectors
  For example: use the cosine of their angles.

Computing PMI of *w* and *c*: Using a fixed window of ± k words

$$PMI(w,c) = \log \frac{p(w,c)}{p(w)p(c)}$$

N: How many tokens does the corpus contain?

$f(w) \leq N$: How often does w occur?

$f(w, c) \leq f(w,)$: How often does w occur with c in its window?

$f(\mathrm{c}) = \sum_w f(w, c) \leq N$: How many tokens have c in their window?

$p(w) = f(w)/N$

$p(\mathrm{c}) = f(c)/N$

$p(w, c) = f(w, c)/N$

Vector similarity: Cosine

One way to define the similarity of two vectors is to use the cosine of their angle.

The cosine of two vectors is their dot product, divided by the product of their lengths:

$$\text{sim}_{cos}(\vec{x},\vec{y}) = \frac{\sum_{i=1}^{N} x_i \times y_i}{\sqrt{\sum_{i=1}^{N} x_i^2}\sqrt{\sum_{i=1}^{N} y_i^2}} = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$$

sim(**w**, **u**) = 1: **w** and **u** point in the same direction
sim(**w**, **u**) = 0: **w** and **u** are orthogonal
sim(**w**, **u**) = –1: **w** and **u** point in the opposite direction

Distributional similarities

Distributional similarities capture the fact that the meanings of words are related.

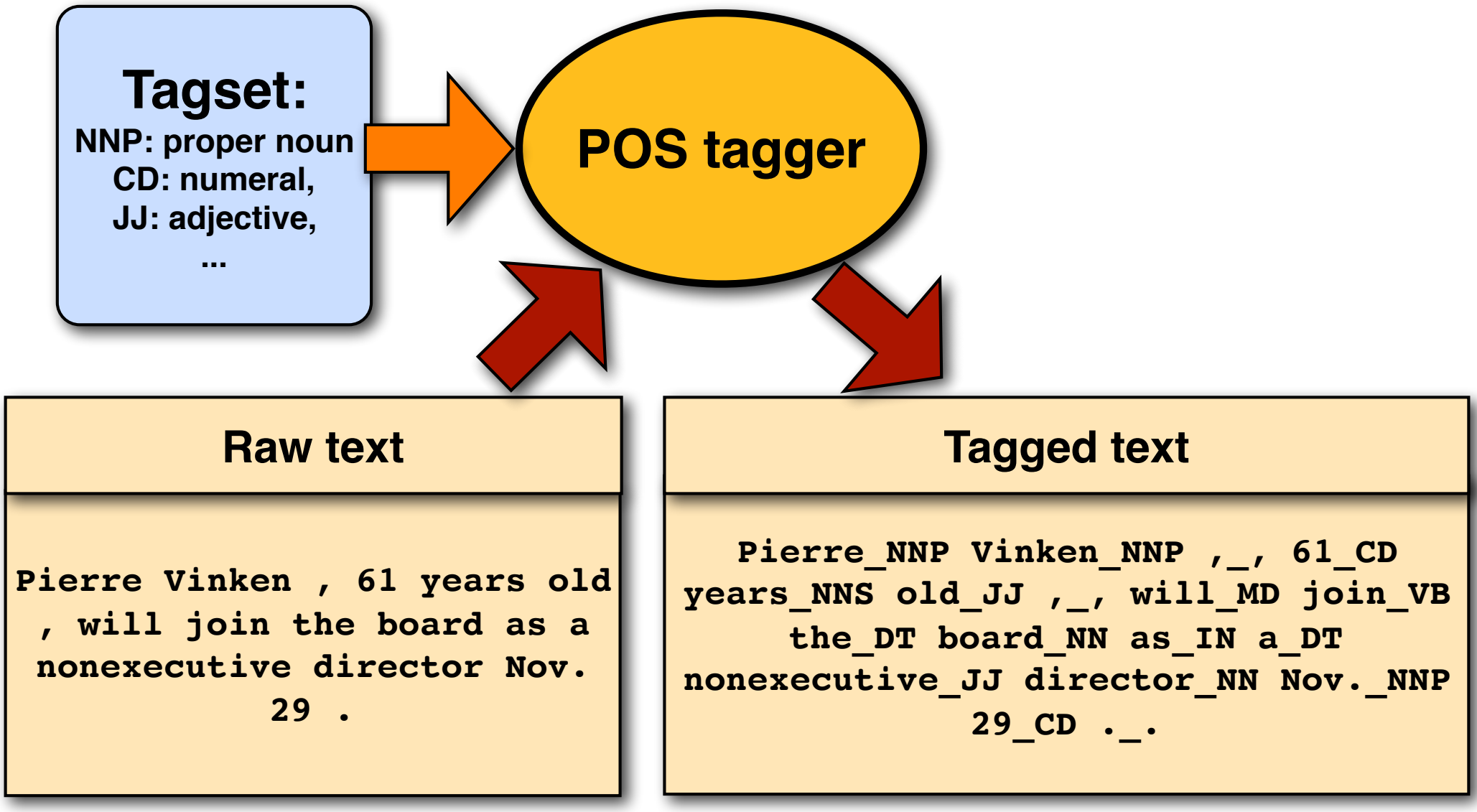
- Words are not just atomic symbols
- Words whose vectors are close to each other have related meanings
  (what does “related meaning” mean?)

Distributional similarities yield sparse vectors:

- There are very many possible contexts.
- Each word (token or type) occurs only in a very small fraction of those.

Review: Sequence Labeling

POS tagging


Tagset:
NNP: proper noun
CD: numeral,
JJ: adjective,
...
POS tagger
Raw text
Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .
Tagged text
Pierre_NNP Vinken_NNP ,_, 61_CD years_NNS old_JJ ,_, will_MD join_VB the_DT board_NN as_IN a_DT nonexecutive_JJ director_NN Nov._NNP 29_CD ._.

Statistical POS tagging with HMMs

What is the most likely sequence of tags **t** for the given sequence of words **w** ?

$$\underset{\mathbf{t}}{\operatorname{argmax}} P(\mathbf{t}|\mathbf{w}) = \underset{\mathbf{t}}{\operatorname{argmax}} P(\mathbf{t})P(\mathbf{w}|\mathbf{t})$$

Hidden Markov Models define $P(\mathbf{t})$ and $P(\mathbf{w}|\mathbf{t})$ as:

Transition probabilities:

$P(\mathbf{t}) = \prod_i P(t_i \mid t_{i-1})$ [bigram HMM]

or $P(\mathbf{t}) = \prod_i P(t_i \mid t_{i-1}, t_{i-2})$ [trigram HMM]

Emission probabilities:

$P(\mathbf{w} \mid \mathbf{t}) = \prod_i P(w_i \mid t_i)$

An example HMM

Transition Matrix A

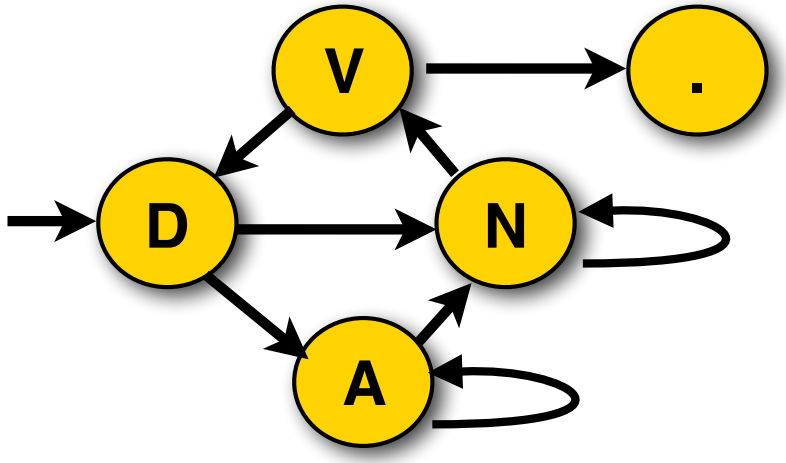
	D	N	V	A	.
D		0.8		0.2	
N		0.7	0.3		
V	0.6				0.4
A		0.8		0.2	
.					

Emission Matrix B

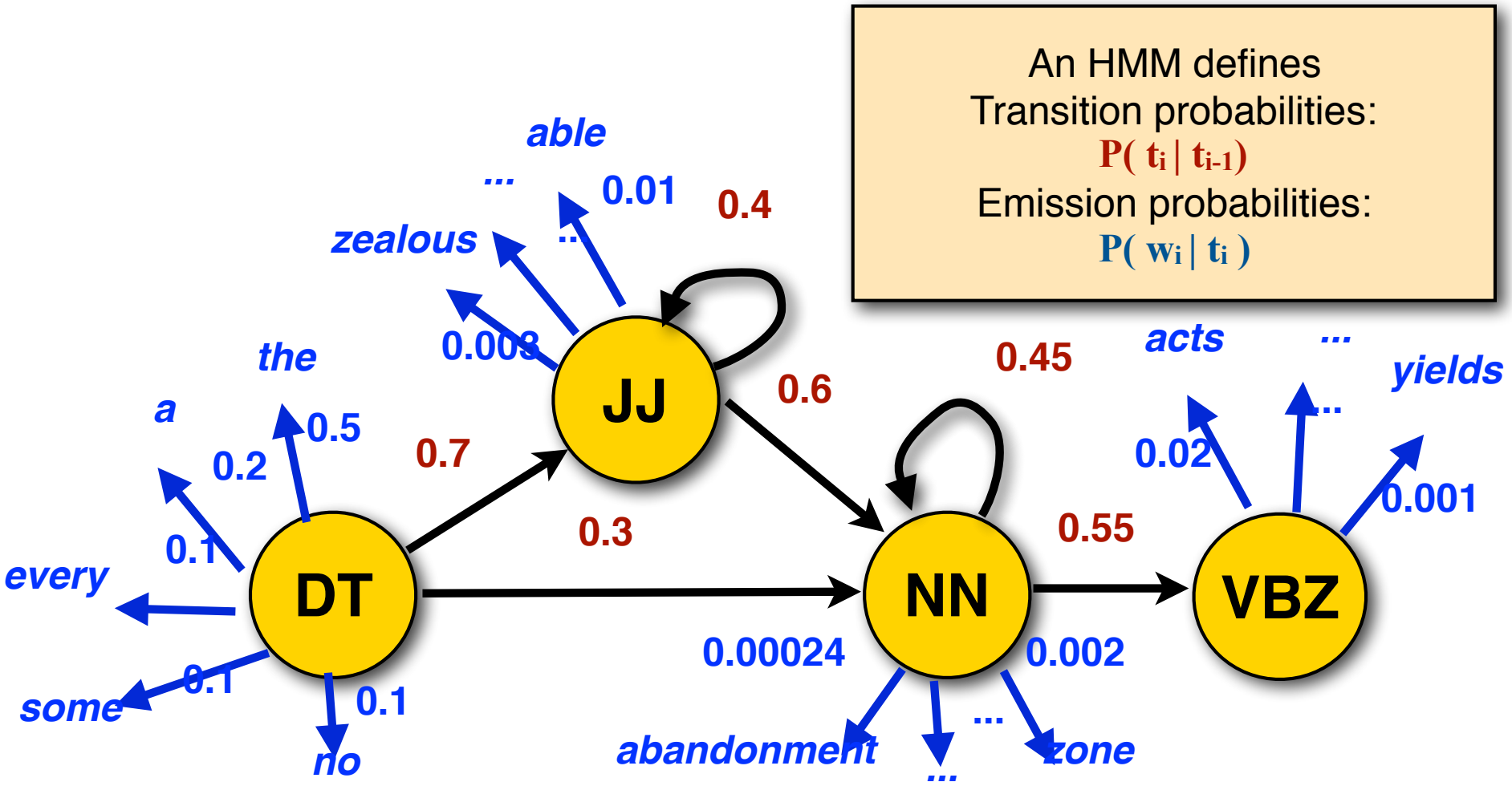
	the	*man*	*ball*	*throws*	*sees*	*red*	*blue*	.
D	1							
N		0.7	0.3					
V				0.6	0.4			
A						0.8	0.2	
.								1

Initial state vector π

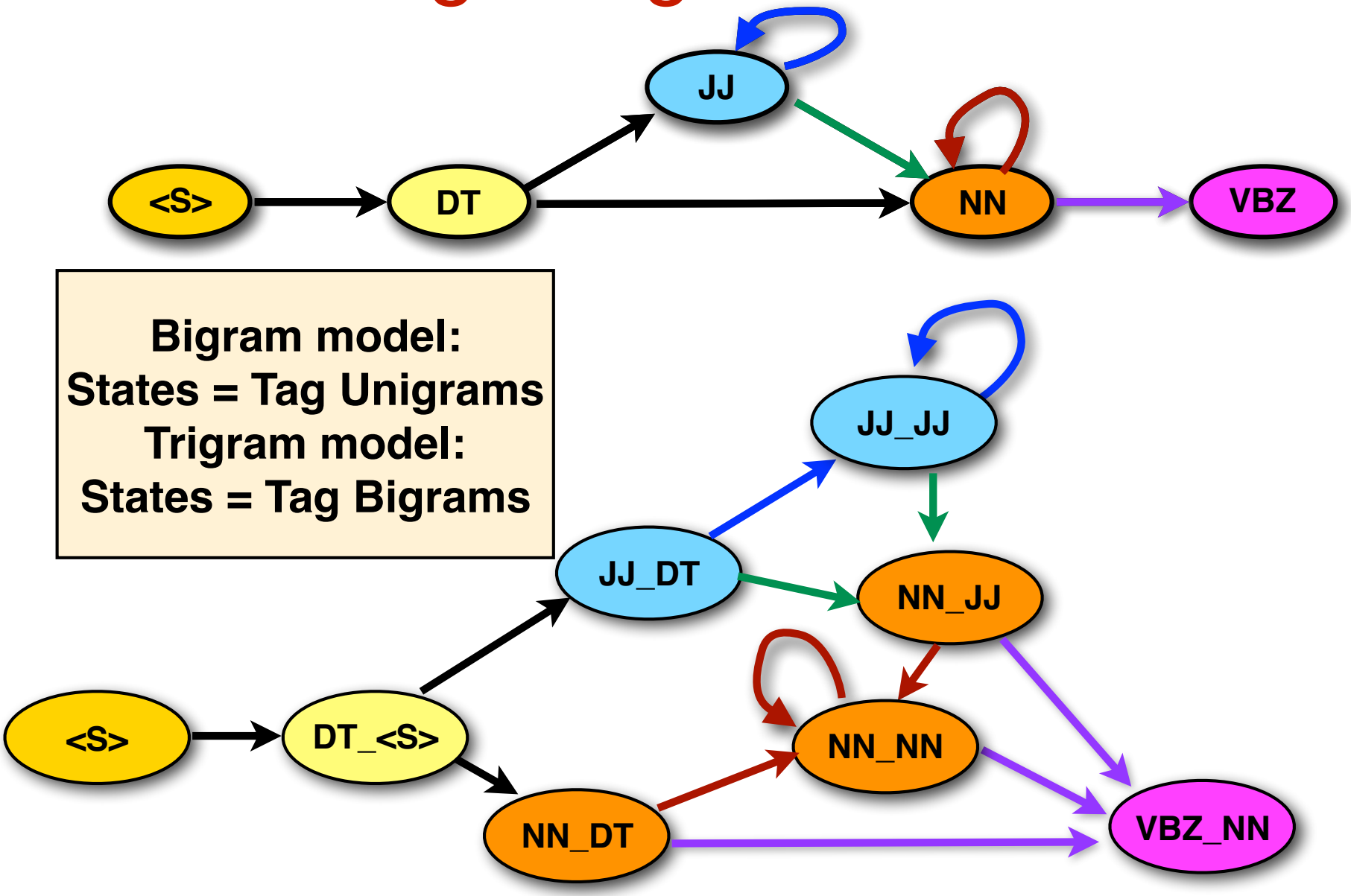
	D	N	V	A	.
π	1				



HMMs as probabilistic automata



Encoding a trigram model as FSA



Dynamic programming

Dynamic programming is a general technique to solve certain complex search problems by memoization

1.) Recursively decompose the large search problem into smaller subproblems that can be solved efficiently
- There is only a polynomial number of subproblems.

2.) Store (memoize) the solution of each subproblem in a common data structure
- Processing this data structure takes polynomial time

Dynamic programming algorithms for HMMs

I. Likelihood of the input:

Compute $P(\boldsymbol{w}|\ \lambda\ )$ for the input $\boldsymbol{w}$ and HMM λ

⇒ **Forward algorithm**

II. Decoding (=tagging) the input:

Find best tags $\boldsymbol{t}^* = argmax_{\boldsymbol{t}}\ P(\boldsymbol{t}\ |\ \boldsymbol{w},\lambda)$ for the input $\boldsymbol{w}$ and HMM λ

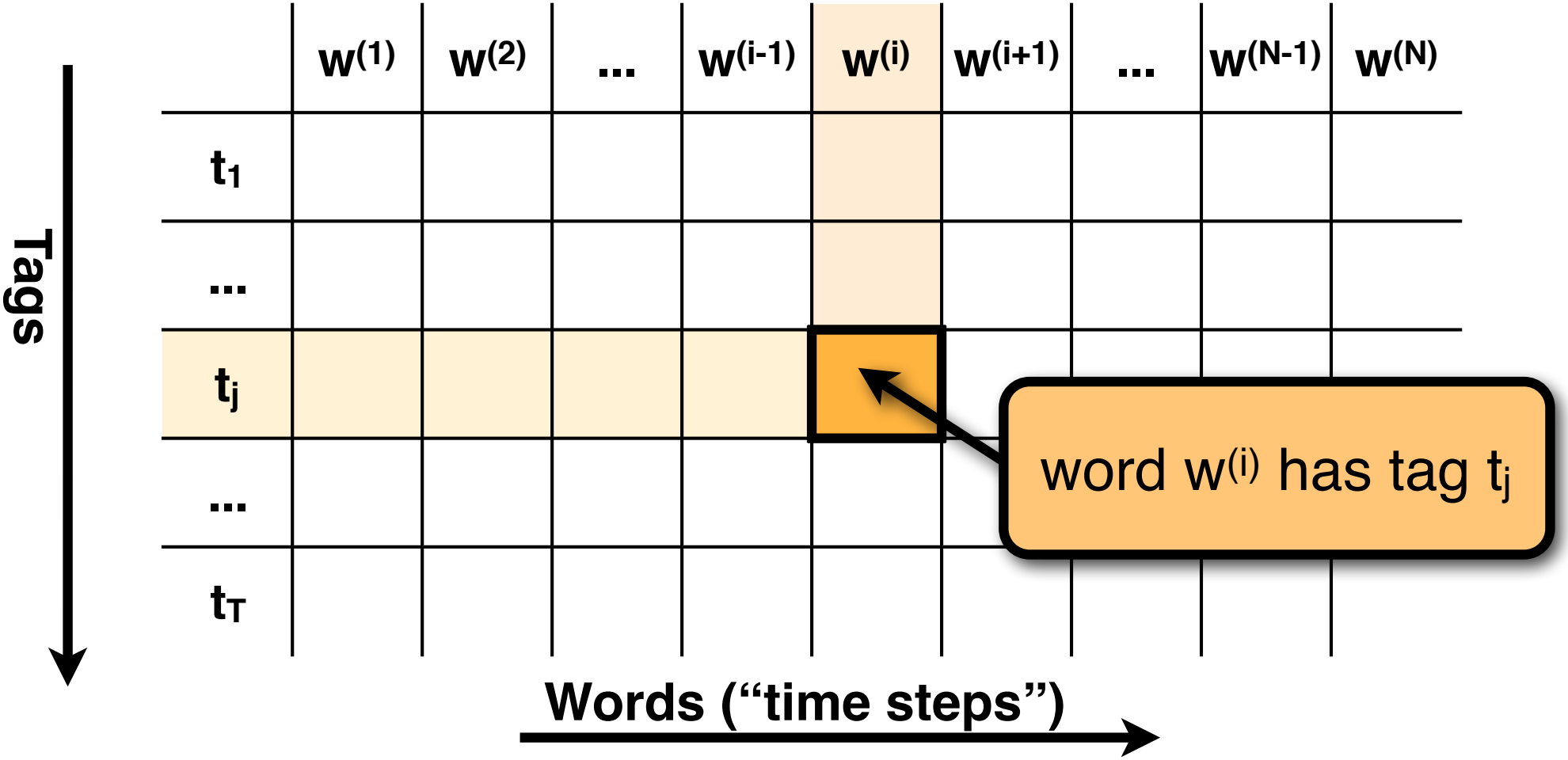
⇒ **Viterbi algorithm**

III. Estimation (=learning the model):

Find best model parameters $\lambda^* = argmax_{\lambda}\ P(\boldsymbol{t},\ \boldsymbol{w}\ |\ \lambda)$ for training data $\boldsymbol{w}$

⇒ **Forward-Backward algorithm**

Bookkeeping: the trellis

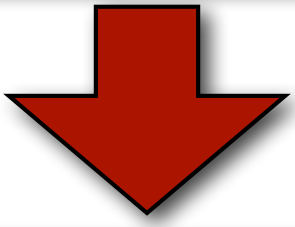


We use a N×T table ("**trellis**") to keep track of the HMM.
The HMM can assign one of the T tags to each of the N words.

Sequence labeling tasks

POS tagging

```
Pierre Vinken , 61 years old , will join IBM 's board
as a nonexecutive director Nov. 29 .
```

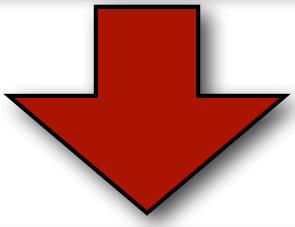


```
Pierre_NNP Vinken_NNP ,_, 61_CD years_NNS old_JJ ,_,
will_MD join_VB IBM_NNP 's_POS board_NN as_IN a_DT
nonexecutive_JJ director_NN Nov._NNP 29_CD ._.
```

Task: assign POS tags to words

Noun phrase (NP) chunking

```
Pierre Vinken , 61 years old , will join IBM 's board
as a nonexecutive director Nov. 29 .
```



```
[NP Pierre Vinken] , [NP 61 years] old , will join
[NP IBM] 's [NP board] as [NP a nonexecutive director]
[NP Nov. 2] .
```

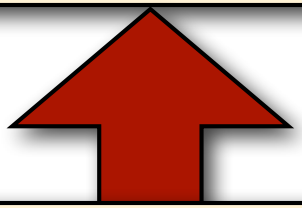
Task: identify all non-recursive NP chunks

The BIO encoding

We define three new tags:

- **B-NP**: beginning of a noun phrase chunk
- **I-NP**: inside of a noun phrase chunk
- **O**: outside of a noun phrase chunk

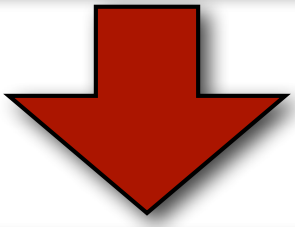
```
[NP Pierre Vinken] , [NP 61 years] old , will join
[NP IBM] 's [NP board] as [NP a nonexecutive director]
[NP Nov. 2] .
```



```
Pierre_B-NP Vinken_I-NP ,_O 61_B-NP years_I-NP
old_O ,_O will_O join_O IBM_B-NP 's_O board_B-NP as_O
a_B-NP nonexecutive_I-NP director_I-NP Nov._B-NP
29_I-NP ._O
```

Named Entity Recognition

```
Pierre Vinken , 61 years old , will join IBM 's board
as a nonexecutive director Nov. 29 .
```



```
[PERS Pierre Vinken] , 61 years old , will join
[ORG IBM] 's board as a nonexecutive director
[DATE Nov. 2] .
```

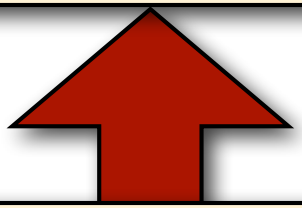
Task: identify all mentions of named entities (people, organizations, locations, dates)

The BIO encoding for NER

We define many new tags:

- **B-PERS**, **B-DATE, …:** beginning of a mention of a person/date...
- **I-PERS**, **I-DATE, …:** inside of a mention of a person/date...
- **O**: outside of any mention of a named entity

```
[PERS Pierre Vinken] , 61 years old , will join
[ORG IBM] 's board as a nonexecutive director
[DATE Nov. 2] .
```



```
Pierre_B-PERS Vinken_I-PERS ,_O 61_O years_O old_O ,_O
will_O join_O IBM_B-ORG 's_O board_O as_O a_O
nonexecutive_O director_O Nov._B-DATE 29_I-DATE ._O
```

Many NLP tasks are sequence labeling tasks

Input: a sequence of tokens/words:

```
Pierre Vinken , 61 years old , will join IBM 's board
as a nonexecutive director Nov. 29 .
```

Output: a sequence of **labeled** tokens/words:

POS-tagging:
```
Pierre_NNP Vinken_NNP ,_, 61_CD years_NNS
old_JJ ,_, will_MD join_VB IBM_NNP 's_POS board_NN
as_IN a_DT nonexecutive_JJ director_NN Nov._NNP
29_CD ._.
```

Named Entity Recognition:
```
Pierre_B-PERS Vinken_I-PERS ,_O
61_O years_O old_O ,_O will_O join_O IBM_B-ORG 's_O
board_O as_O a_O nonexecutive_O director_O Nov._B-DATE
29_I-DATE ._O
```

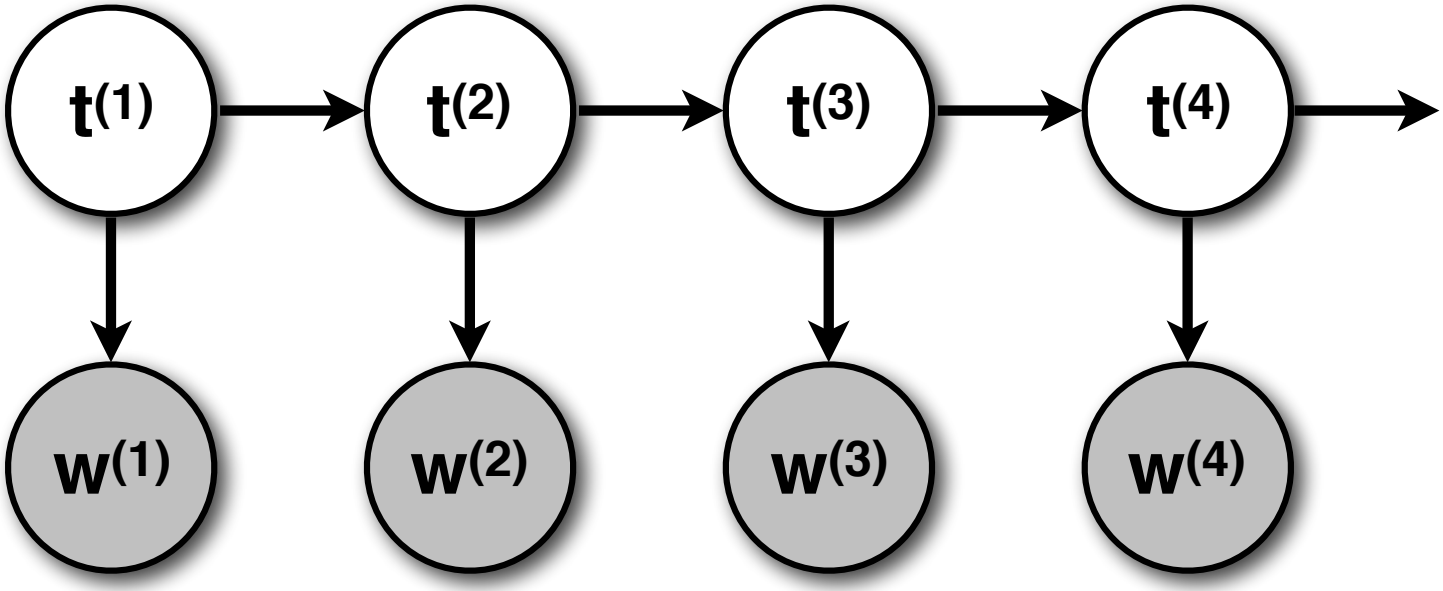
Graphical models for sequence labeling

HMMs as graphical models

HMMs are **generative** models of the observed input string **w**

They 'generate' **w** with $P(\mathbf{w},\mathbf{t}) = \prod_i P(t^{(i)}|\ t^{(i-1)})P(w^{(i)}|\ t^{(i)})$

When we use an HMM to tag, we observe **w**, and need to find **t**



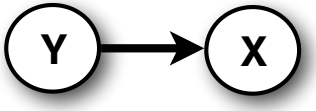
Directed graphical models

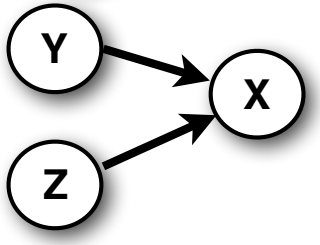
Graphical models are a **notation for probability models**.

In a **directed** graphical model, **each node** represents a distribution over a random variable:

- $P(X)$ = 

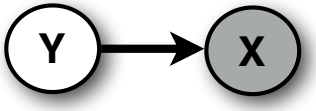
Arrows represent dependencies (they define what other random variables the current node is conditioned on)

- $P(Y)\ P(X \mid Y)$ = 

- $P(Y)\ P(Z)\ P(X \mid Y, Z)$ = 

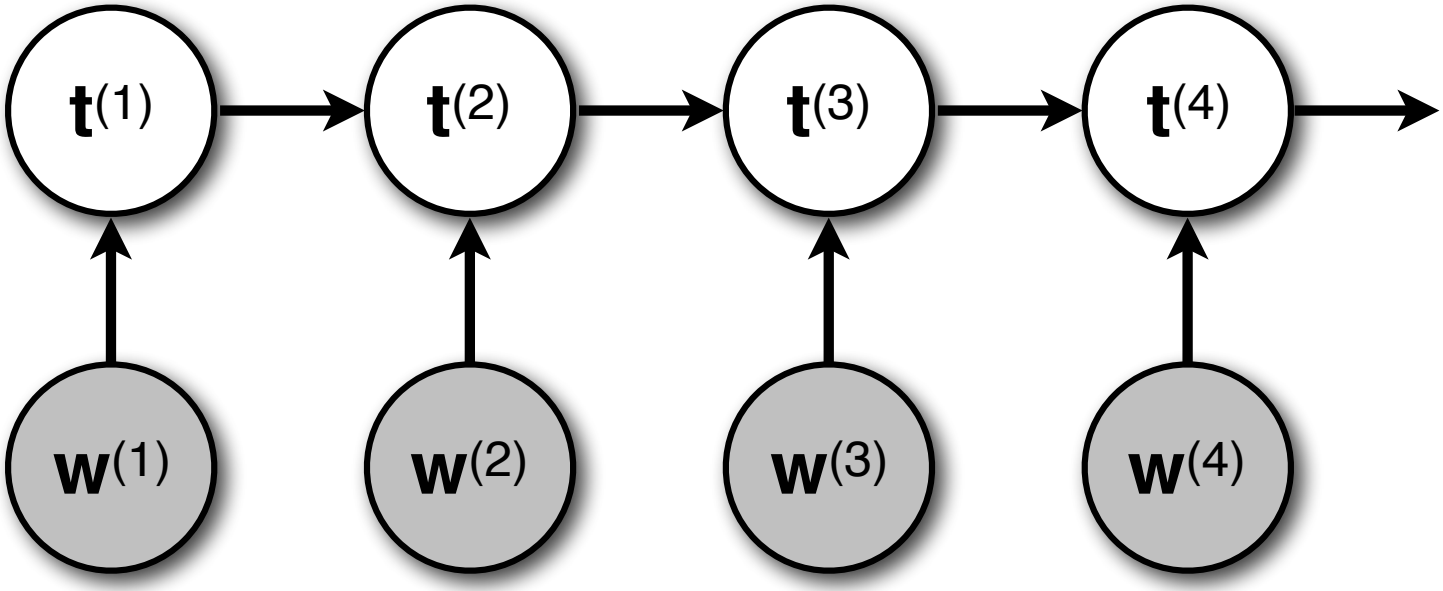
Shaded nodes represent observed variables.

White nodes represent hidden variables

- $P(Y)\ P(X \mid Y)$ with Y hidden and X observed = 

Discriminative probability models

A discriminative or **conditional** model of the labels **t** given the observed input string **w** models $P(\mathbf{t} \mid \mathbf{w}) = \prod_i P(t^{(i)} \mid w^{(i)}, t^{(i-1)})$ directly.



Advantages of discriminative models

We're usually not really interested in $P(\mathbf{w} \mid \mathbf{t})$.

– $\mathbf{w}$ is given. We don't need to predict it!

Why not model what we're actually interested in: $P(\mathbf{t} \mid \mathbf{w})$

Modeling $P(\mathbf{w} \mid \mathbf{t})$ well is quite difficult:

– Prefixes (capital letters) or suffixes are good predictors for certain classes of $\mathbf{t}$ (proper nouns, adverbs,…)

– These features may also help us deal with unknown words

– But these features may not be independent (e.g. they are overlapping)

Modeling $P(\mathbf{t} \mid \mathbf{w})$ should be easier:

– Now we can incorporate arbitrary features of the word, because we don't need to predict $\mathbf{w}$ anymore

Discriminative models

There are two main types of discriminative probability models for sequence labeling:

- Maximum Entropy Markov Models (MEMMs)
- Conditional Random Fields (CRFs)

MEMMs and CRFs:

- are both based on logistic regression
- have the same graphical model
- require the Viterbi algorithm for tagging
- differ in that MEMMs consist of independently learned distributions, while CRFs are trained to maximize the probability of the entire sequence

Probabilistic classification

Classification:

Predict a class (label) c for an input $\mathbf{x}$

There are only a (small) finite number of possible class labels

Probabilistic classification:

– Model the probability $P(c \mid \mathbf{x})$

$P(c|\mathbf{x})$ is a probability if $0 \leq P(c_i \mid \mathbf{x}) \leq 1$, and $\sum_i P(c_i \mid \mathbf{x}) = 1$

– Return the class $c^* = \text{argmax}_i P(c_i \mid \mathbf{x})$ that has the highest probability

There are different ways to model $P(c \mid \mathbf{x})$.

MEMMs and CRFs are based on logistic regression

Terminology

Models that are of the form

$$P(c \mid \mathbf{x}) = \text{score}(\mathbf{x},c) / \sum_j \text{score}(\mathbf{x},c_j)$$
$$= \exp(\sum_i w_{ic} f_i(\mathbf{x})) / \sum_j \exp(\sum_i w_{ij} f_i(\mathbf{x}))$$

are also called loglinear models, Maximum Entropy (MaxEnt) models, or multinomial logistic regression models.

CS446 and CS546 should give you more details about these.

The normalizing term $\sum_j \exp(\sum_i w_{ij} f_i(\mathbf{x}))$ is also called the partition function and is often abbreviated as Z

Using features

Think of feature functions as useful questions you can ask about the input $\mathbf{x}$:

- **Binary feature functions**:

  $f_{\text{first-letter-capitalized}}(\mathbf{Urbana}) = 1$

  $f_{\text{first-letter-capitalized}}(\mathbf{computer}) = 0$

- **Integer (or real-valued) features**:

  $f_{\text{number-of-vowels}}(\mathbf{Urbana}) = 3$

Which specific feature functions are useful will depend on your task (and your training data).

From features to probabilities

We associate a real-valued weight w_{ic} with each feature function $f_i(\mathbf{x})$ and output class c

Note that the feature function $f_i(\mathbf{x})$ does not have to depend on c as long as the weight does (note the double index w_{ic})

This gives us a real-valued score for predicting class c for input $\mathbf{x}$: $\text{score}(\mathbf{x},c) = \sum_i w_{ic} f_i(\mathbf{x})$

This score could be negative, so we exponentiate it:
$\text{score}(\mathbf{x},c) = \exp(\sum_i w_{ic} f_i(\mathbf{x}))$

To get a probability distribution over all classes c, we renormalize these scores:

$$P(c \mid \mathbf{x}) = \text{score}(\mathbf{x},c) / \sum_j \text{score}(\mathbf{x},c_j)$$
$$= \exp(\sum_i w_{ic} f_i(\mathbf{x})) / \sum_j \exp(\sum_i w_{ij} f_i(\mathbf{x}))$$

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.)

Structure prediction (e.g. syntactic parsing)

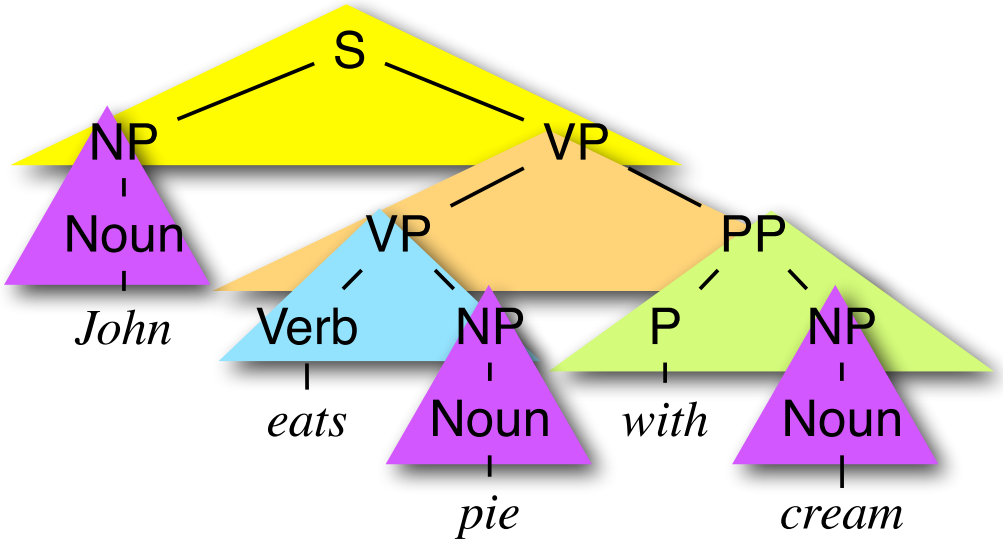
Probabilistic Context-Free Grammars

For every nonterminal X, define a probability distribution $P(X \rightarrow \alpha \mid X)$ over all rules with the same LHS symbol X:

S	$\rightarrow$	NP VP	0.8
S	$\rightarrow$	S conj S	0.2
NP	$\rightarrow$	Noun	0.2
NP	$\rightarrow$	Det Noun	0.4
NP	$\rightarrow$	NP PP	0.2
NP	$\rightarrow$	NP conj NP	0.2
VP	$\rightarrow$	Verb	0.4
VP	$\rightarrow$	Verb NP	0.3
VP	$\rightarrow$	Verb NP NP	0.1
VP	$\rightarrow$	VP PP	0.2
PP	$\rightarrow$	P NP	1.0

Computing P(τ) with a PCFG

The probability of a tree τ is the product of the probabilities of all its rules:



S	→	NP VP	0.8
S	→	S conj S	0.2
NP	→	Noun	0.2
NP	→	Det Noun	0.4
NP	→	NP PP	0.2
NP	→	NP conj NP	0.2
VP	→	Verb	0.4
VP	→	Verb NP	0.3
VP	→	Verb NP NP	0.1
VP	→	VP PP	0.2
PP	→	P NP	1.0

$P(\tau) = 0.8 \times 0.3 \times 0.2 \times 1.0 \times 0.2^3$

$= \mathbf{0.00384}$

Probabilistic CKY

Input: POS-tagged sentence

`John_N eats_V pie_N with_P cream_N`

John	eats	pie	with	cream	
N 1.0, NP 0.2	S 0.8 · 0.2 · 0.3	S 0.8 · 0.2 · 0.06		S 0.2 · 0.0036 · 0.8	John
	V 1.0, VP 0.3	VP 1 · 0.3 · 0.2 = 0.06		VP max(1.0 · 0.008 · 0.3, 0.06 · 0.2 · 0.3)	eats
		N 1.0, NP 0.2		NP 0.2 · 0.2 · 0.2 = 0.008	pie
			P 1.0	PP 1 · 1 · 0.2	with
				N 1.0, NP 0.2	cream

S	→	NP VP	0.8
S	→	S conj S	0.2
NP	→	Noun	0.2
NP	→	Det Noun	0.4
NP	→	NP PP	0.2
NP	→	NP conj NP	0.2
VP	→	Verb	0.3
VP	→	Verb NP	0.3
VP	→	Verb NP NP	0.1
VP	→	VP PP	0.3
PP	→	P NP	1.0

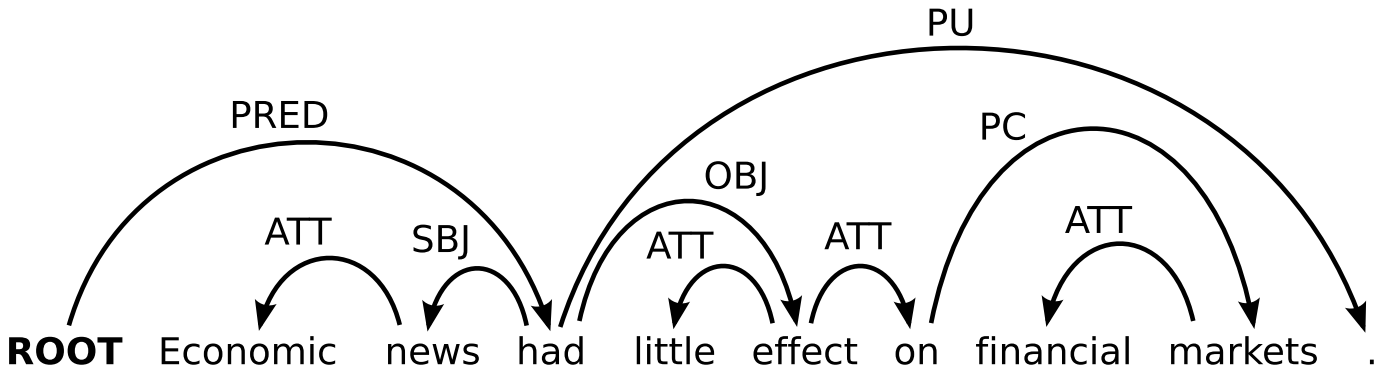
(P)CFG parsing

Probabilistic CFGs are the simplest statistical model for phrase-structure grammars

More advanced models are based on words, and other features of the tree

Most approaches still require a parse chart, and dynamic programming and (either Viterbi or A*)

A dependency parse



Transition-based parsing

Transition-based shift-reduce parsing processes the sentence $S = w_0 w_1 ... w_n$ from left to right.

Unlike CKY, it constructs a **single tree**.

N.B: this only works for projective dependency trees

Notation:

w_0 is a special ROOT token.

$V_S = \{w_0, w_1, ..., w_n\}$ is the vocabulary of the sentence

R is a set of dependency relations

The parser uses three data structures:

σ: a **stack** of partially processed words $w_i \in V_S$

β: a **buffer** of remaining input words $w_i \in V_S$

A: a **set of dependency arcs** $(w_i, r, w_j) \in V_S \times R \times V_S$

Economic news had little effect on financial markets .

Transition	Configuration		
	([ROOT],	[Economic, . . . , .],	$\emptyset$)
SH $\Rightarrow$	([ROOT, Economic],	[news, . . . , .],	$\emptyset$)
$\text{LA}_{\text{ATT}} \Rightarrow$	([ROOT],	[news, . . . , .],	$A_1 = \{(\text{news, ATT, Economic})\}$)
SH $\Rightarrow$	([ROOT, news],	[had, . . . , .],	A_1)
$\text{LA}_{\text{SBJ}} \Rightarrow$	([ROOT],	[had, . . . , .],	$A_2 = A_1 \cup \{(\text{had, SBJ, news})\}$)
SH $\Rightarrow$	([ROOT, had],	[little, . . . , .],	A_2)
SH $\Rightarrow$	([ROOT, had, little],	[effect, . . . , .],	A_2)
$\text{LA}_{\text{ATT}} \Rightarrow$	([ROOT, had],	[effect, . . . , .],	$A_3 = A_2 \cup \{(\text{effect, ATT, little})\}$)
SH $\Rightarrow$	([ROOT, had, effect],	[on, . . . , .],	A_3)
SH $\Rightarrow$	([ROOT, . . . on],	[financial, markets, .],	A_3)
SH $\Rightarrow$	([ROOT, . . . , financial],	[markets, .],	A_3)
$\text{LA}_{\text{ATT}} \Rightarrow$	([ROOT, . . . on],	[markets, .],	$A_4 = A_3 \cup \{(\text{markets, ATT, financial})\}$)
$\text{RA}_{\text{PC}} \Rightarrow$	([ROOT, had, effect],	[on, .],	$A_5 = A_4 \cup \{(\text{on, PC, markets})\}$)
$\text{RA}_{\text{ATT}} \Rightarrow$	([ROOT, had],	[effect, .],	$A_6 = A_5 \cup \{(\text{effect, ATT, on})\}$)
$\text{RA}_{\text{OBJ}} \Rightarrow$	([ROOT],	[had, .],	$A_7 = A_6 \cup \{(\text{had, OBJ, effect})\}$)
SH $\Rightarrow$	([ROOT, had],	[.],	A_7)
$\text{RA}_{\text{PU}} \Rightarrow$	([ROOT],	[had],	$A_8 = A_7 \cup \{(\text{had, PU, .})\}$)
$\text{RA}_{\text{PRED}} \Rightarrow$	([ ],	[ROOT],	$A_9 = A_8 \cup \{(\text{ROOT, PRED, had})\}$)
SH $\Rightarrow$	([ROOT],	[ ],	A_9)

Dependency parsing

Increasingly popular representation, especially for languages other than English.

Transition-based parsing still requires a learned model (to predict the next action).
Other parsing algorithms are also possible.

Why neural approaches to NLP?

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.

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.

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.

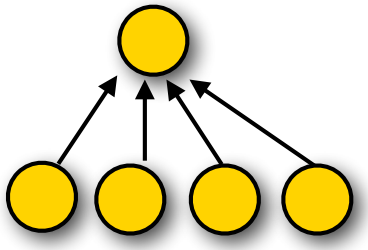
Why have they suddenly made a comeback?

- 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

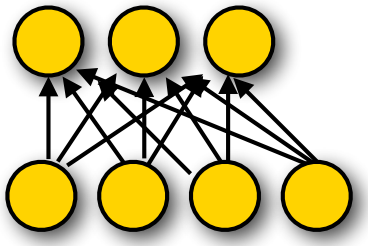
For binary classification tasks:
Single output unit
Return 1 if $y > 0.5$
Return 0 otherwise



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)

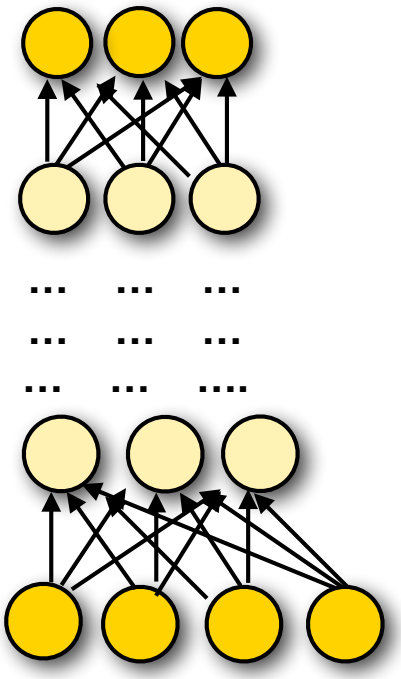
This is just a linear classifier (aka Perceptron)
(the output y is a linear function of the input **x**)

Single-layer non-linear feedforward networks:

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

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

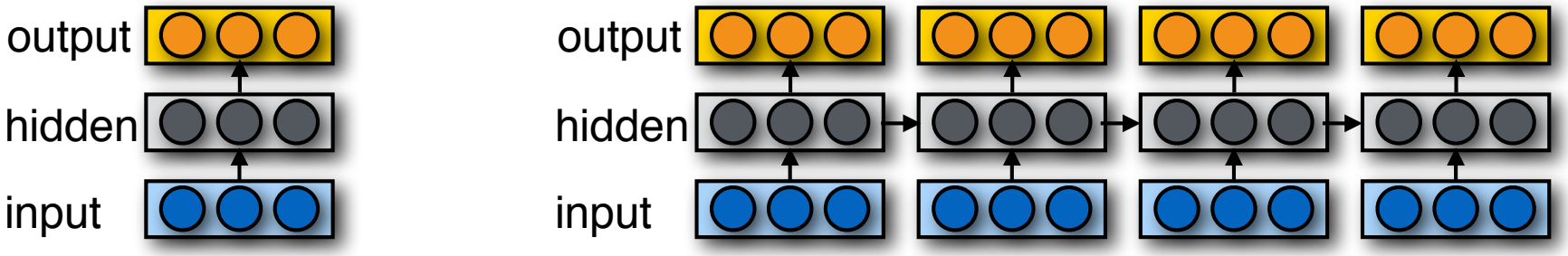
Input: Represent each preceding word by its d-dimensional embedding.

- 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) 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.