Natural Language Processing with Neural Nets

Mark Hasegawa-Johnson

5/2022

Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Semantics: Montague grammar

Richard Montague, 1930-1971
photograph © Richard Thomason

Richard Montague defined formal semantics as follows:

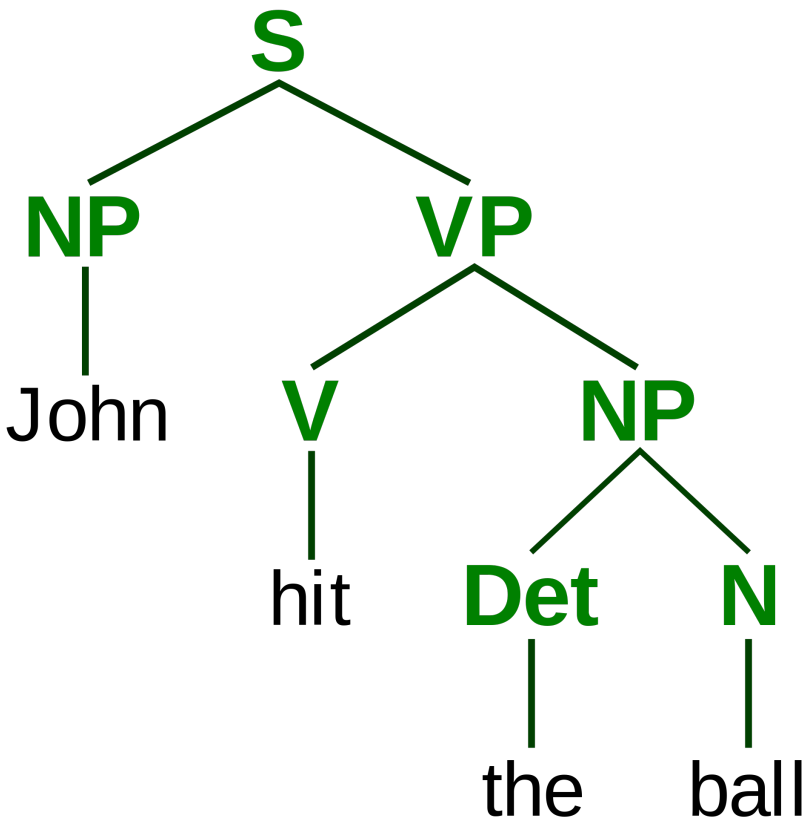
- ”Understanding a sentence” means that you can specify the conditions under which the sentence would be true
- The meaning of a sentence is composed of the meanings of its words. For example:

Logical form	Example	Meaning
some(P,Q)	“some people sing”	$\exists x\colon ((Px) \wedge (Qx))$
a(P,Q)	“a bird sings”	$\exists x\colon ((Px) \wedge (Qx))$
every(P,Q)	“every bird sings”	$\forall x\colon ((Px) \rightarrow (Qx))$
no(P,Q)	“no bird snores”	$\forall x\colon ((Px) \rightarrow \neg(Qx))$

Syntax

Syntax is the study of how words combine.

- Syntax is a descriptive science: it simply describes how words combine when people are using them naturally.
- Compositional semantics studies the meanings of those combinations.

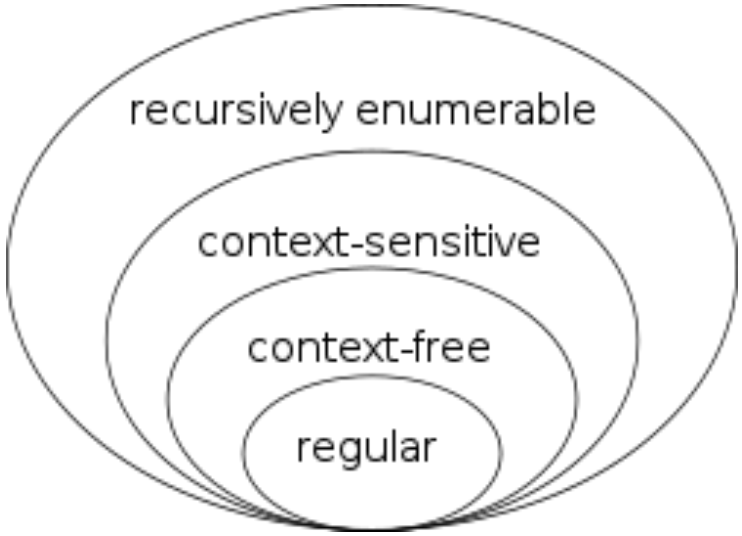


ParseTree.svg. Public domain image, Stannered, 2007

Grammar

A **grammar** is a mathematical specification of the set of all word sequences that form valid sentences in a language (e.g., English).

- **Recursively enumerable**: any grammar that can be decided by a Turing machine
- **Context-sensitive**: phrase A is expanded into phrases B and C using rules of the form $\alpha A \beta \rightarrow \alpha B C \beta$ for specified contexts α and β.
- **Context-free**: phrase A is expanded into phrases B and C using context-free rules: $A \rightarrow BC$.
- **Regular**: phrase A can only be expanded into a word followed by another phrase: $A \rightarrow aB$.

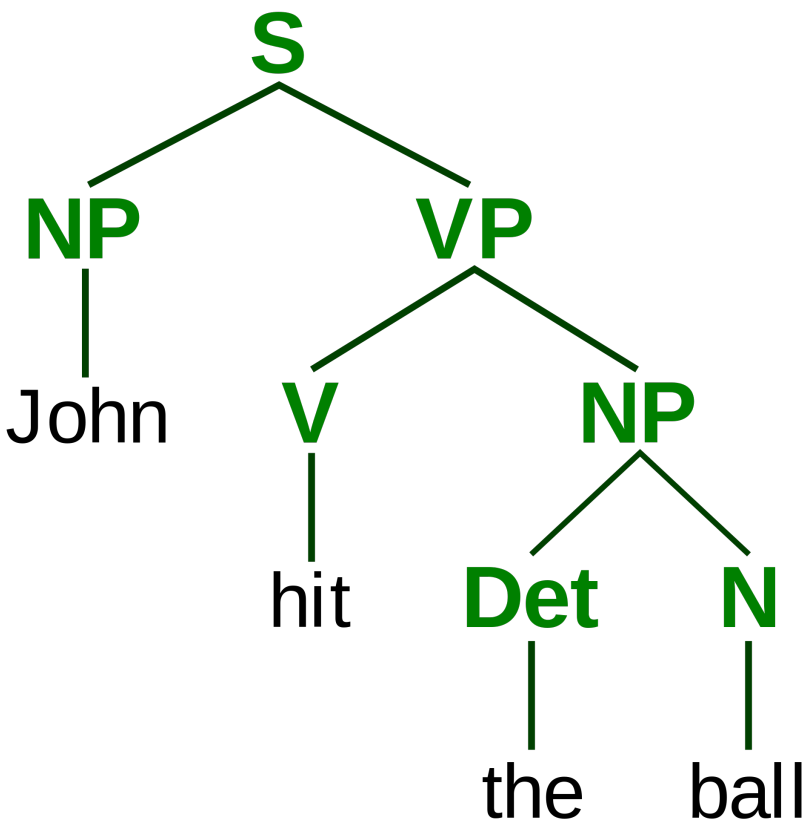


Chomsky-hierarchy.svg. CC-SA 3.0, J. Finkelstein, 2010

Grammar

Humans usually think of natural language using context-free grammar (CFG). For example,

$$S \rightarrow NP\ VP$$
$$VP \rightarrow V\ NP$$
$$NP \rightarrow Det\ N$$
$$NP \rightarrow \text{John}$$
$$V \rightarrow \text{hit}$$
$$Det \rightarrow \text{the}$$
$$N \rightarrow \text{ball}$$

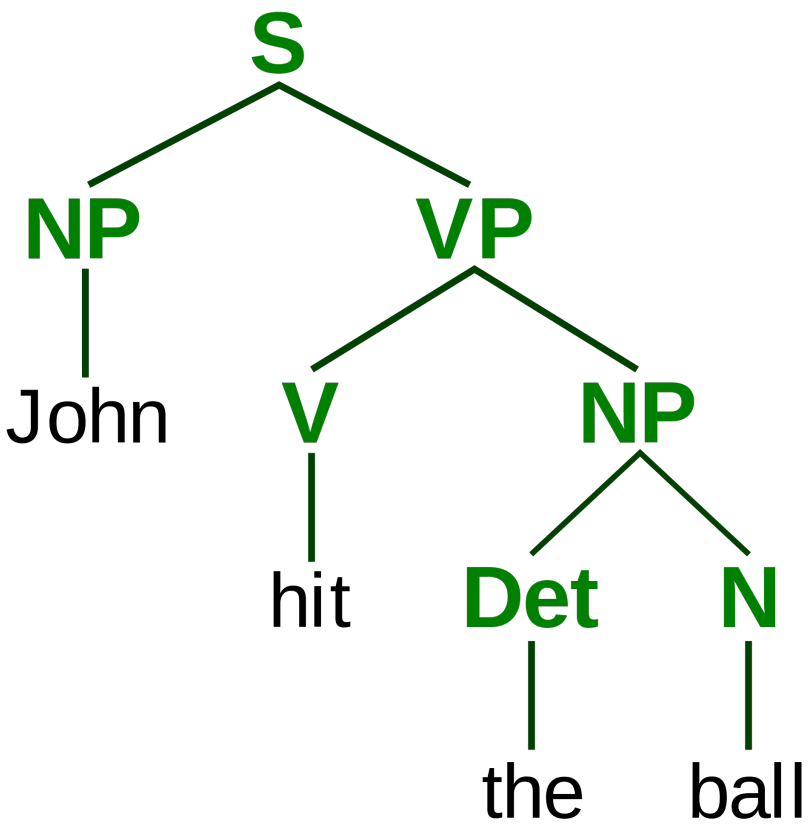


ParseTree.svg. Public domain image, Stannered, 2007

Grammar

A CFG with finite recursion depth can be written as a regular grammar. For example:

$$S \rightarrow \text{John } VP$$
$$VP \rightarrow \text{hit } NP$$
$$NP \rightarrow \text{the } N$$
$$N \rightarrow \text{ball}$$



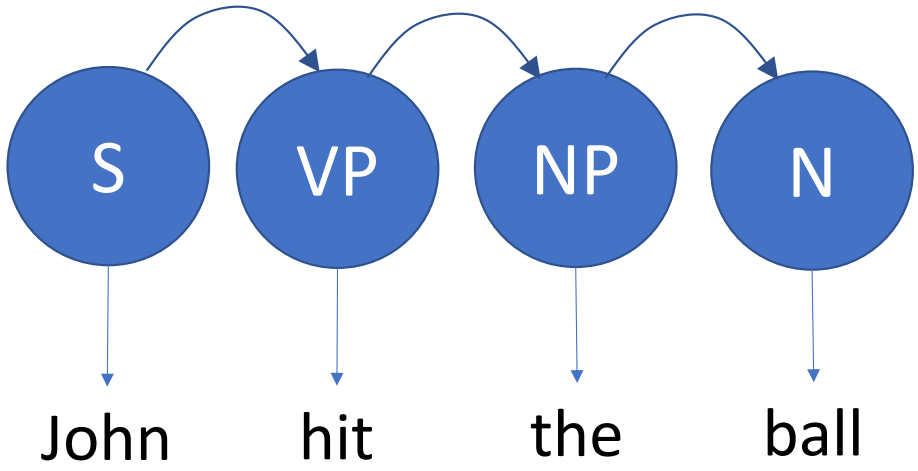
ParseTree.svg. Public domain image, Stannered, 2007

Grammar

A regular grammar can be written using an HMM.

- The phrase is the state variable
- The word is the observed variable

$$S \rightarrow \text{John } VP$$
$$VP \rightarrow \text{hit } NP$$
$$NP \rightarrow \text{the } N$$
$$N \rightarrow \text{ball}$$



Key concepts: syntax and semantics

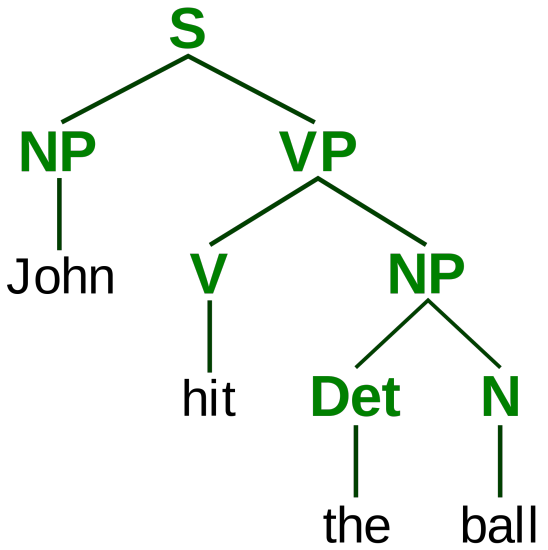
- Compositional semantics studies how sentence meaning is computed from word meanings.
- Syntax studies the ways in which words combine.
- A grammar is a mathematical specification of the sequences of words that form valid sentences in a language.
- A context-free grammar with finite recursion depth can be written as a regular grammar.
- A regular grammar can be written as an HMM.

Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

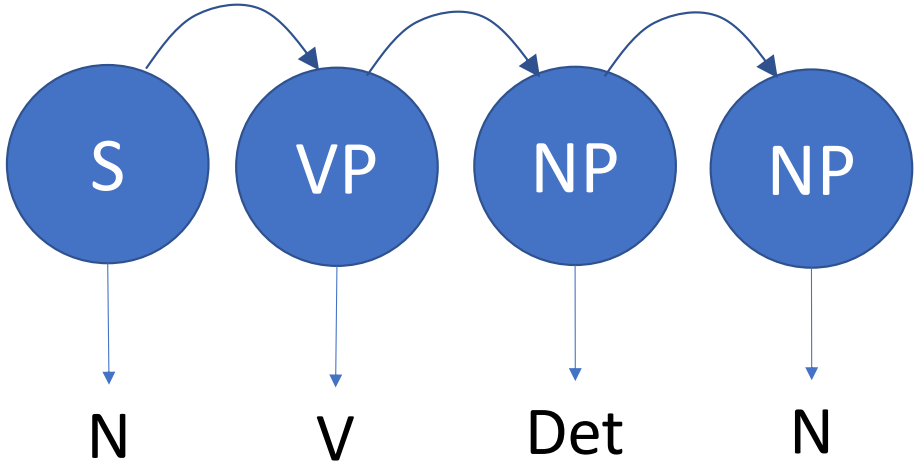
Parts of speech

Many grammars are written in terms of parts of speech, to make them a bit more general. For example, this one…



ParseTree.svg. Public domain image, Stannered, 2007

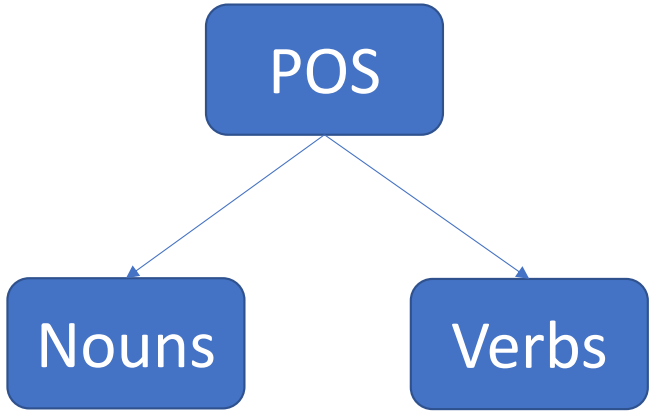
…could be generalized like this…



Parts of speech

For some reason, most of the part-of-speech (POS) systems proposed by philosophers have 2^N parts of speech, for some value of N.

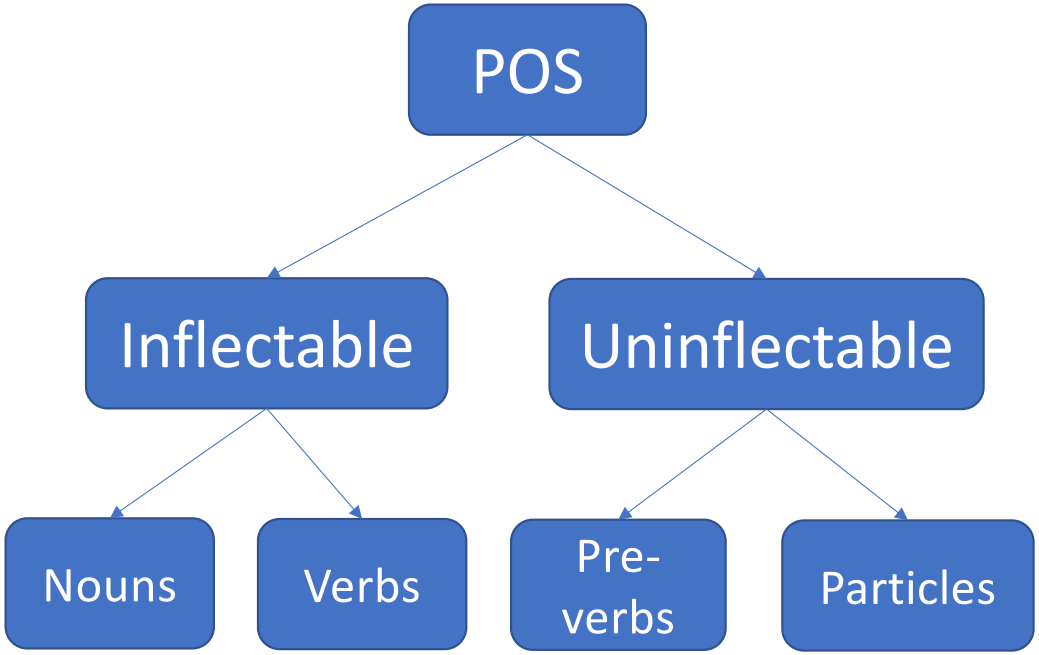
- Plato (350BC) proposed that there are 2 parts of speech: nouns and verbs.



Parts of speech

For some reason, most of the part-of-speech (POS) systems proposed by philosophers have 2^N parts of speech, for some value of N.

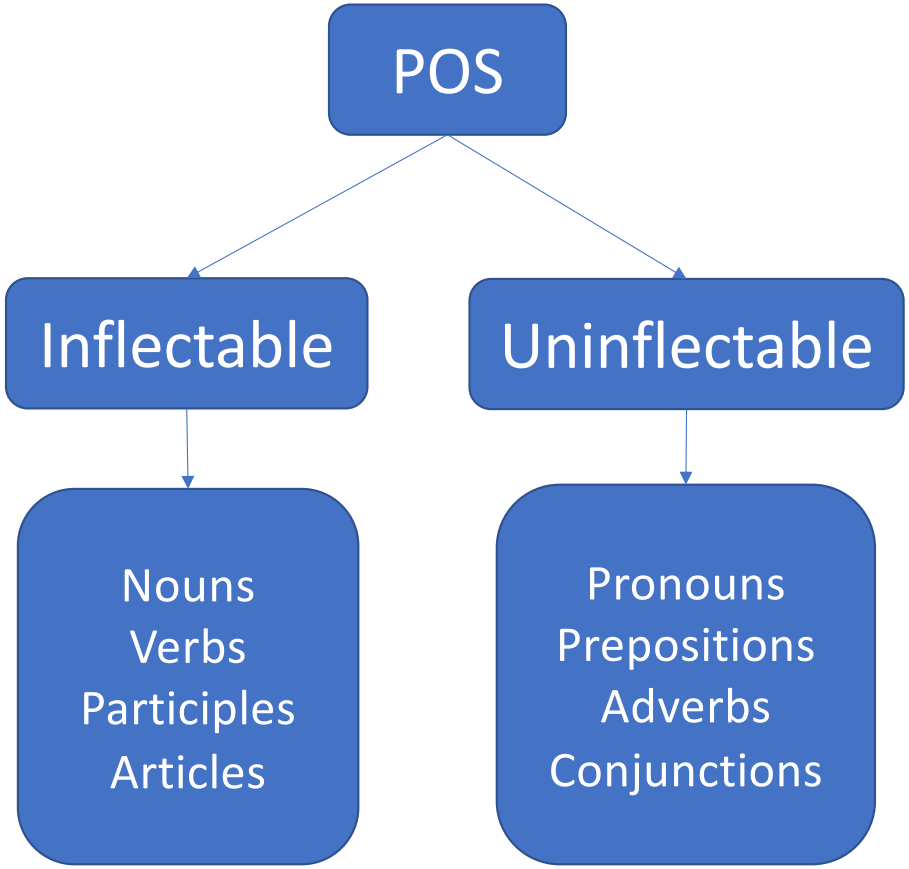
- Plato (350BC) proposed that there are 2 parts of speech: nouns and verbs.
- Yāska (600BC) proposed that there are 4 parts of speech.



Parts of speech

For some reason, most of the part-of-speech (POS) systems proposed by philosophers have 2^N parts of speech, for some value of N.

- Plato (350BC) proposed that there are 2 parts of speech: nouns and verbs.
- Yāska (600BC) proposed that there are 4 parts of speech.
- Dionysus Thrax (100BC) proposed 8 parts of speech.



Parts of speech

Most modern English dictionaries use these POS tags.

- **Open-class words** (anybody can make up a new word, in any of these classes, at any time): nouns, verbs, adjectives, adverbs, interjections
- **Closed-class words** (it's hard to make up a new word in these classes): pronouns, prepositions, conjunctions, determiners

Most published, tagged data use POS tags that are finer-grained than the nine tags listed above. For example, the next few slides describe the Penn Treebank POS tag set.

Nouns

The Penn Treebank noun categories are:

- NN (singular or mass common noun): llama, thought, communism
- NNS (plural common noun): llamas, thoughts
- NNP (singular proper noun): Jane, IBM, Mexico
- NNPS (plural proper noun): Osbournes, Carolinas
- VBG (gerund): eating

Verbs

The Penn Treebank verb categories are:

- VB (verb base form): eat
- VBD (verb past tense): ate
- VBP (verb non-3sg present): eat
- VBZ (verb 3sg present): eats
- MD (modal): can as in “can lift”, should as in “should go”
- RP (particle): up as in “get up,” off as in “take off”

Adjectives

The Penn Treebank has several categories that might be considered types of adjectives:

- CD (cardinal number --- use this tag regardless of whether the number is being used as a noun or adjective): one, two, twenty
- JJ (adjective): yellow, exceptional, tall
- JJR (comparative adjective): yellower, taller
- JJS (superlative adjective): yellowest, tallest
- PRP$ (possessive pronoun): your, one’s
- VBN (verb past participle): eaten, compiled
- WP$ (wh-possessive): whose

Determiners, Prepositions and Conjunctions

The Penn Treebank has a lot of things that look like determiners, prepositions, or conjunctions:

- CC (coordinating conjunction): and, but, or
- DT (determiner): a, the
- IN (preposition or subordinating conjunction): of, in, by
- PDT (predeterminer): all, both
- POS (possessive ending): ‘s, as in “Bob’s dog”
- TO (any use of the word “to”): to
- WDT (wh-determiner): which, that

Why do POS tagging?

- Because it's highly accurate, typically 97%. That means you can run a POS tagger as a pre-processing step, before doing harder natural language understanding tasks.
- Because it's necessary, if you want to know what the words in the sentence mean.

Will Will ? Will will . Will will will Will 's will to Will .

MD NNP SYM NNP MD SYM NNP MD VB NNP POS NN TO NNP SYM

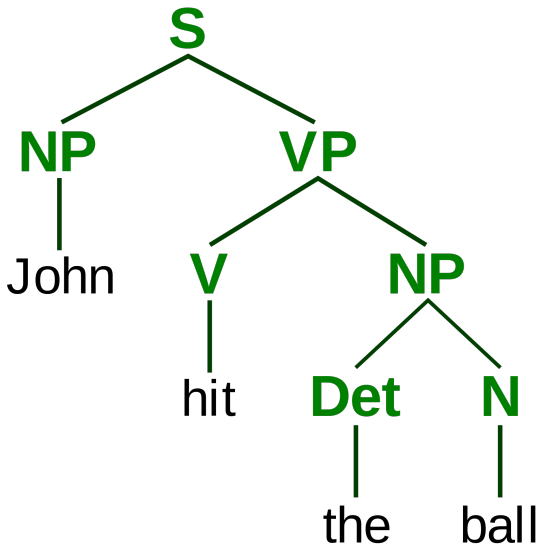
Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

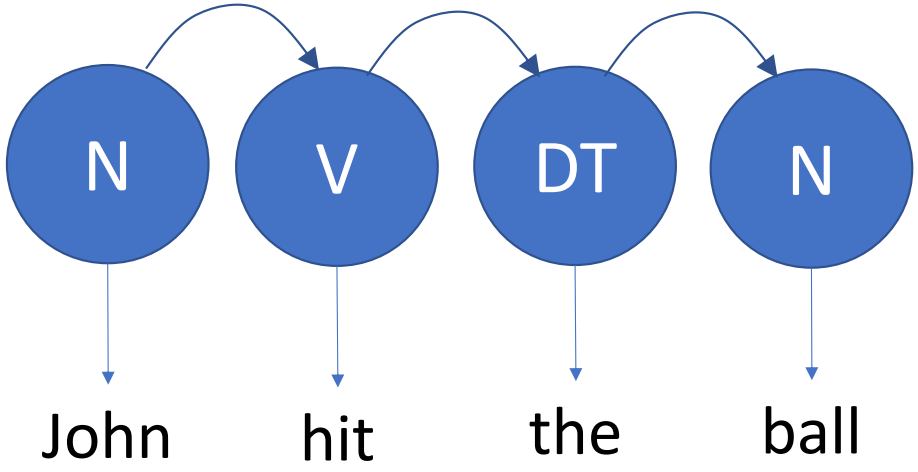
An HMM for POS tagging

The basic idea of an HMM POS tagger is:

- Treat the part of speech as the hidden state variable
- Treat the word as observed



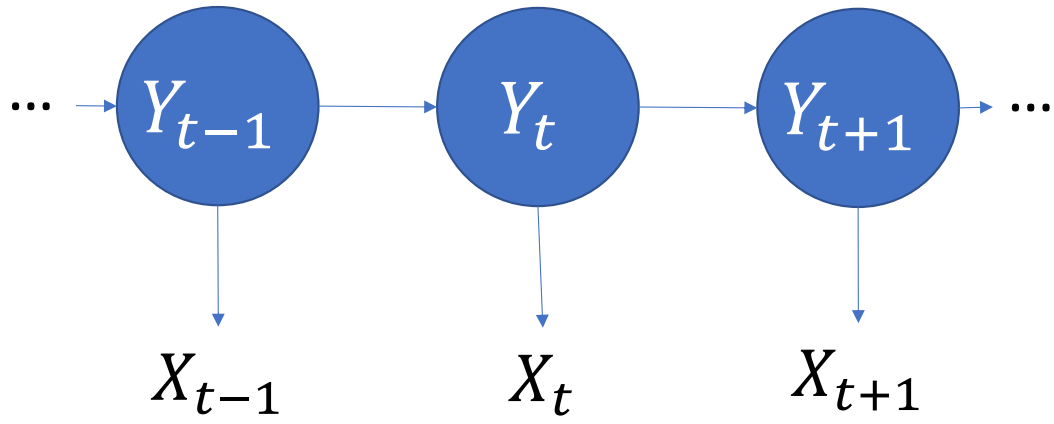
ParseTree.svg. Public domain image, Stannered, 2007



HMM as a Bayes Net

This slide shows an HMM as a Bayes Net. You should remember the graph semantics of a Bayes net:

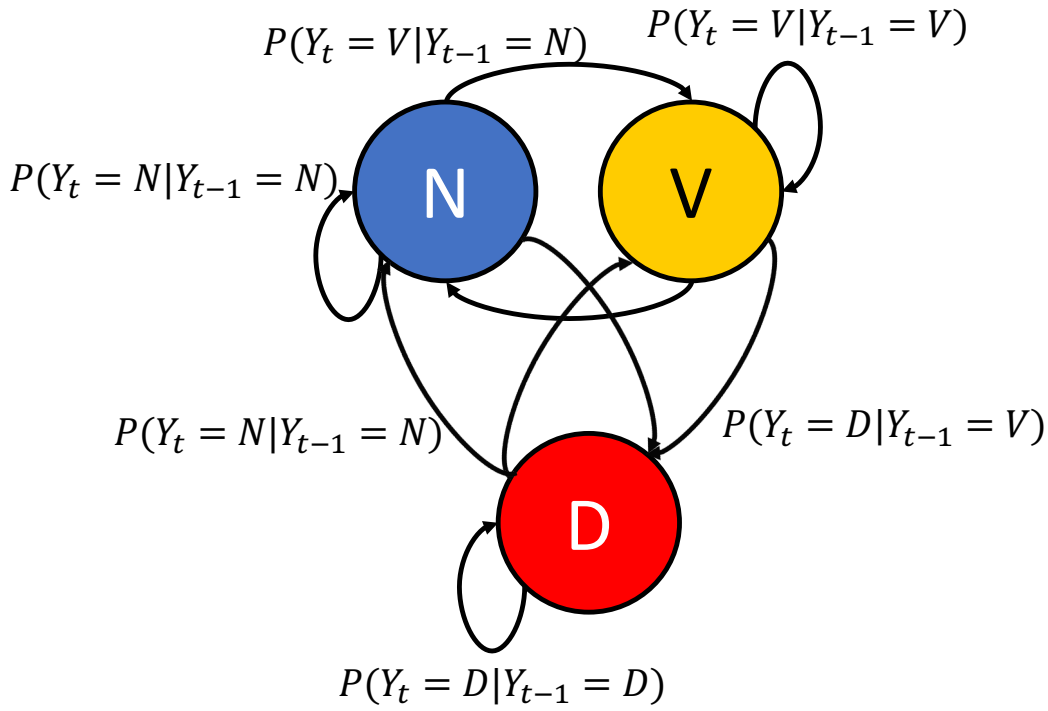
- Nodes are random variables.
- Edges denote stochastic dependence.



HMM as a Finite State Machine

This slide shows ***exactly the same HMM,*** viewed in a totally different way. Here, we show it as a finite state machine:

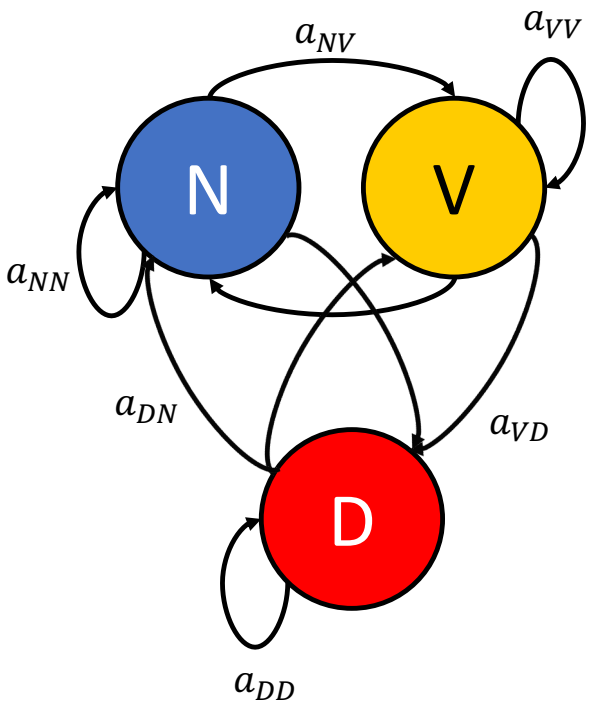
- Nodes denote states.
- Edges denote possible transitions between the states.



Parameters of an HMM

Suppose that there are N distinct POS tags, and V distinct words. Then the parameters of an HMM are:

- $\pi_j = P(Y_1 = j)$. There are N of these.
- $a_{ij} = P(Y_t = j | Y_{t-1} = i)$. There are N^2 of these.
- $b_{jk} = P(X_t = k | Y_t = j)$. There are NV of these.

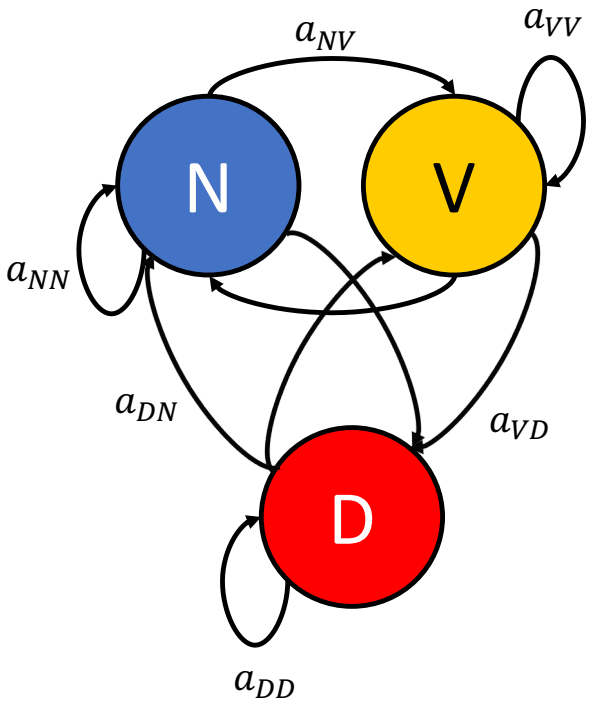


Estimating the Parameters of an HMM

$$\pi_j = \frac{(\#\text{sentences that start with POS } j) + k}{(\#\text{sentences in the training corpus}) + kN}$$

$$a_{ij} = \frac{(\#\text{times } j \text{ follows } i) + k}{(\#\text{times tag } i \text{ occurs in training corpus}) + kN}$$

$$b_{jk} = \frac{(\#\text{times tag } j \text{ is matched to word } k) + k}{(\#\text{times tag } j \text{ occurs in training corpus}) + k(V+1)}$$



Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Viterbi Algorithm: Key concepts

Nodes and edges have numbers attached to them:

- **Edge Probability**: Probability of taking that transition, and then generating the next observed output

$$a_{i,j}b_{j,k} = P(Y_t = j, X_t = k | Y_{t-1} = i)$$

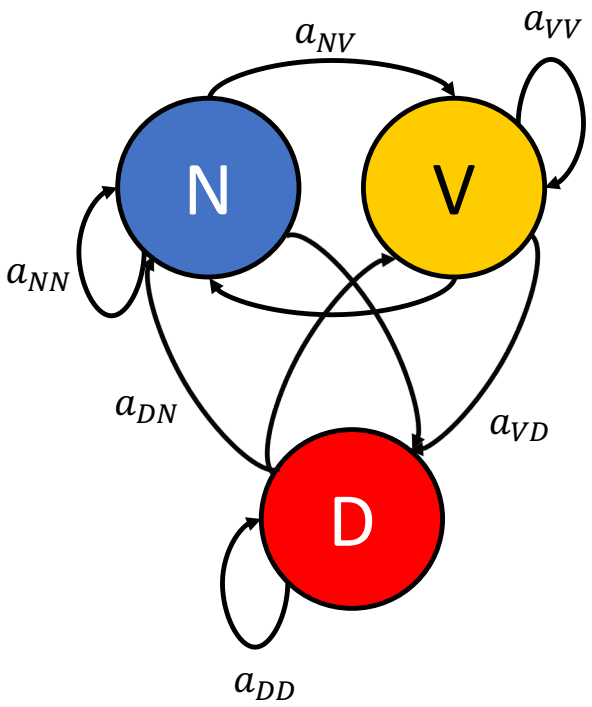
- **Node Probability**: Probability of the best path until node j at time t

$$v_{j,t} = \max_{y_1,\dots,y_{t-1}} P(X_1 = x_1 \dots, X_t = x_t, Y_1 = y_1, \dots, Y_t = j)$$

Viterbi Algorithm for POS tagging

Initial Node Probability: Probability of starting in a particular node:

$$v_{j,1} = P(X_1 = x_1, Y_1 = j)$$
$$= P(Y_1 = j)P(X_t = x_1 | Y_t = j)$$
$$= \pi_j b_{j,x_1}$$



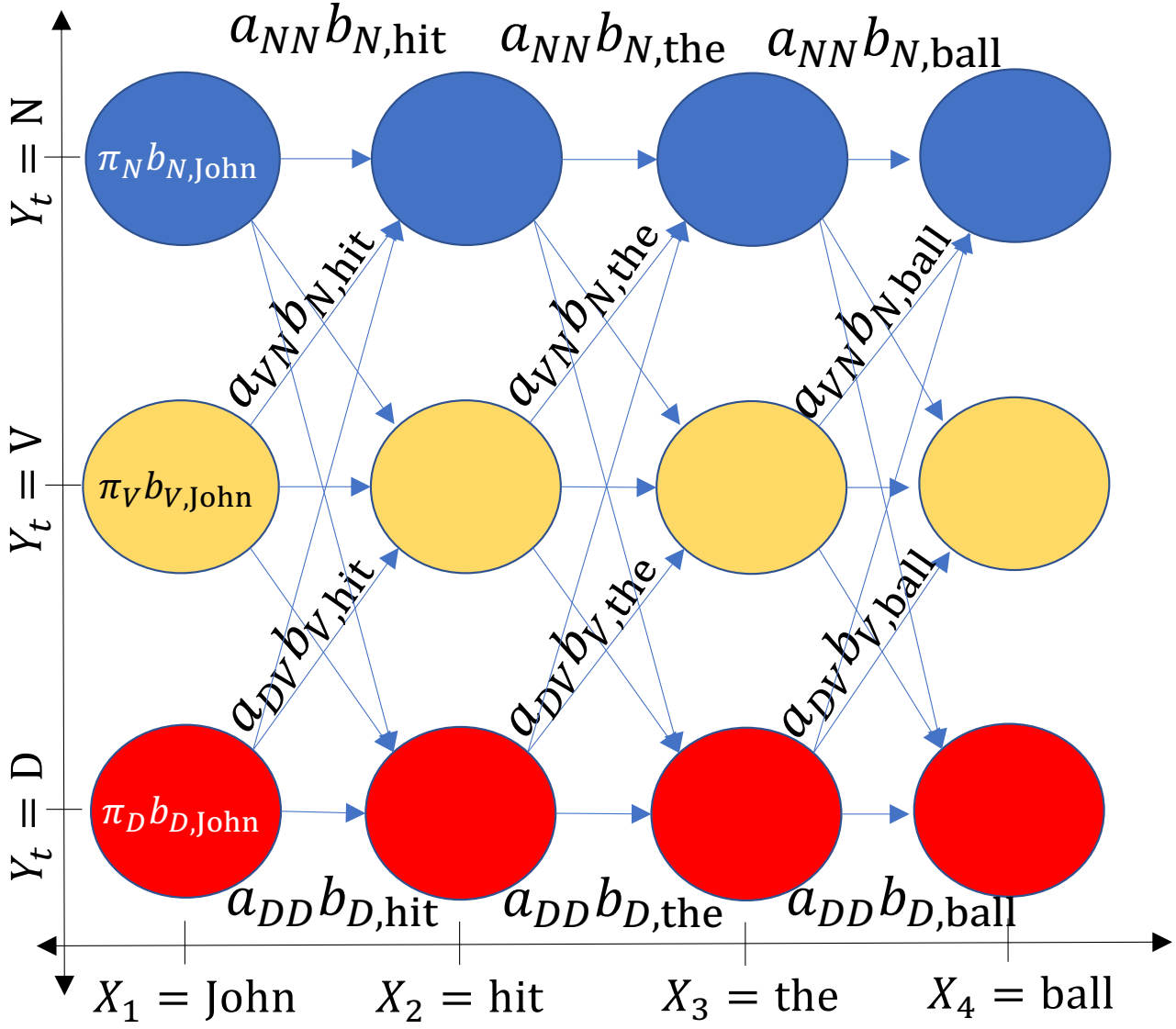
Trellis

Initial Node Probability:

$$v_{j,1} = \pi_j b_{j,x_1}$$

Edge Probability:

$$a_{i,j} b_{j,x_t}$$



Trellis

Initial Node Probability:

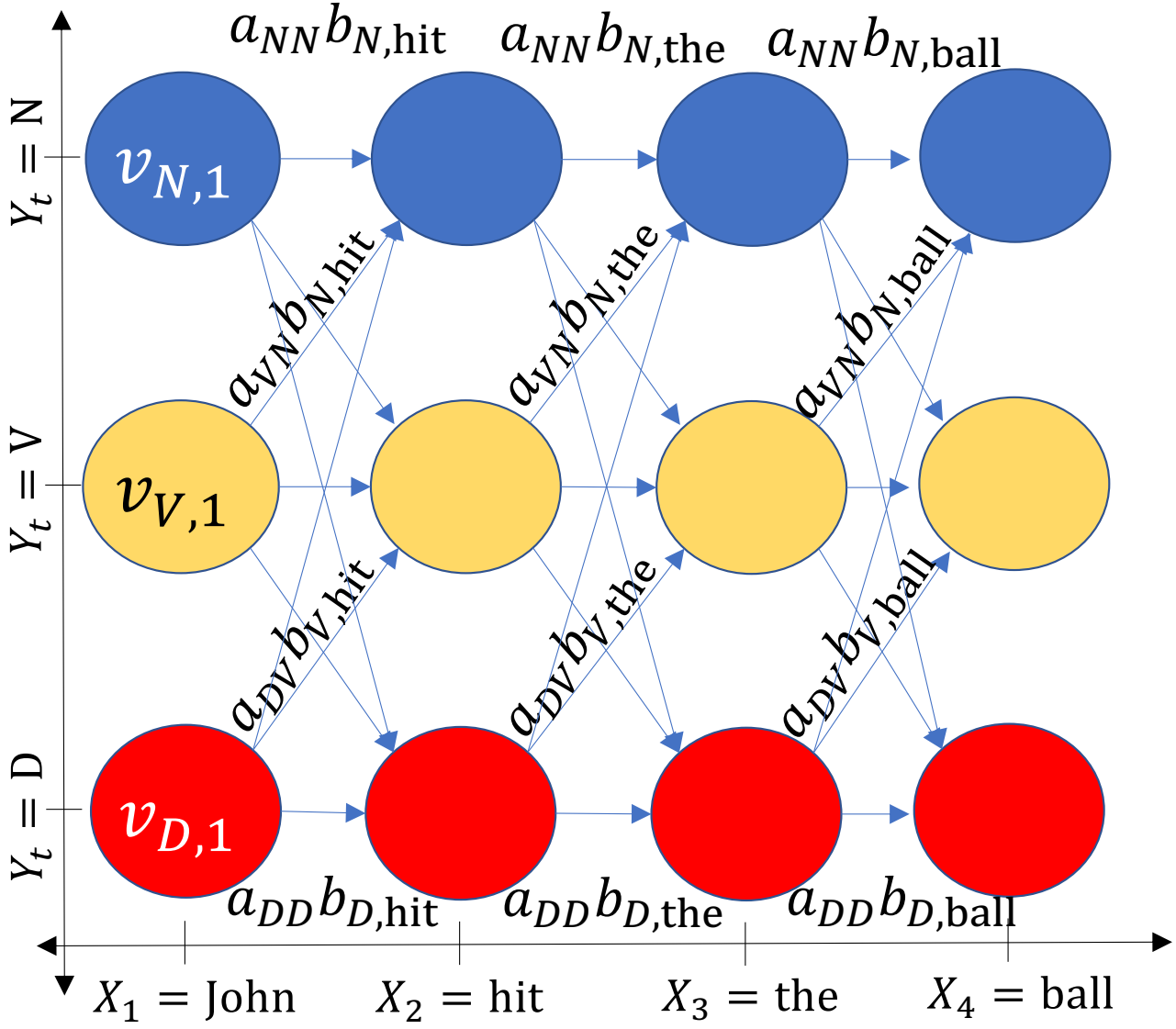
$$v_{j,1} = \pi_j b_{j,x_1}$$

Edge Probability:

$$a_{i,j} b_{j,x_t}$$

Node Probability:

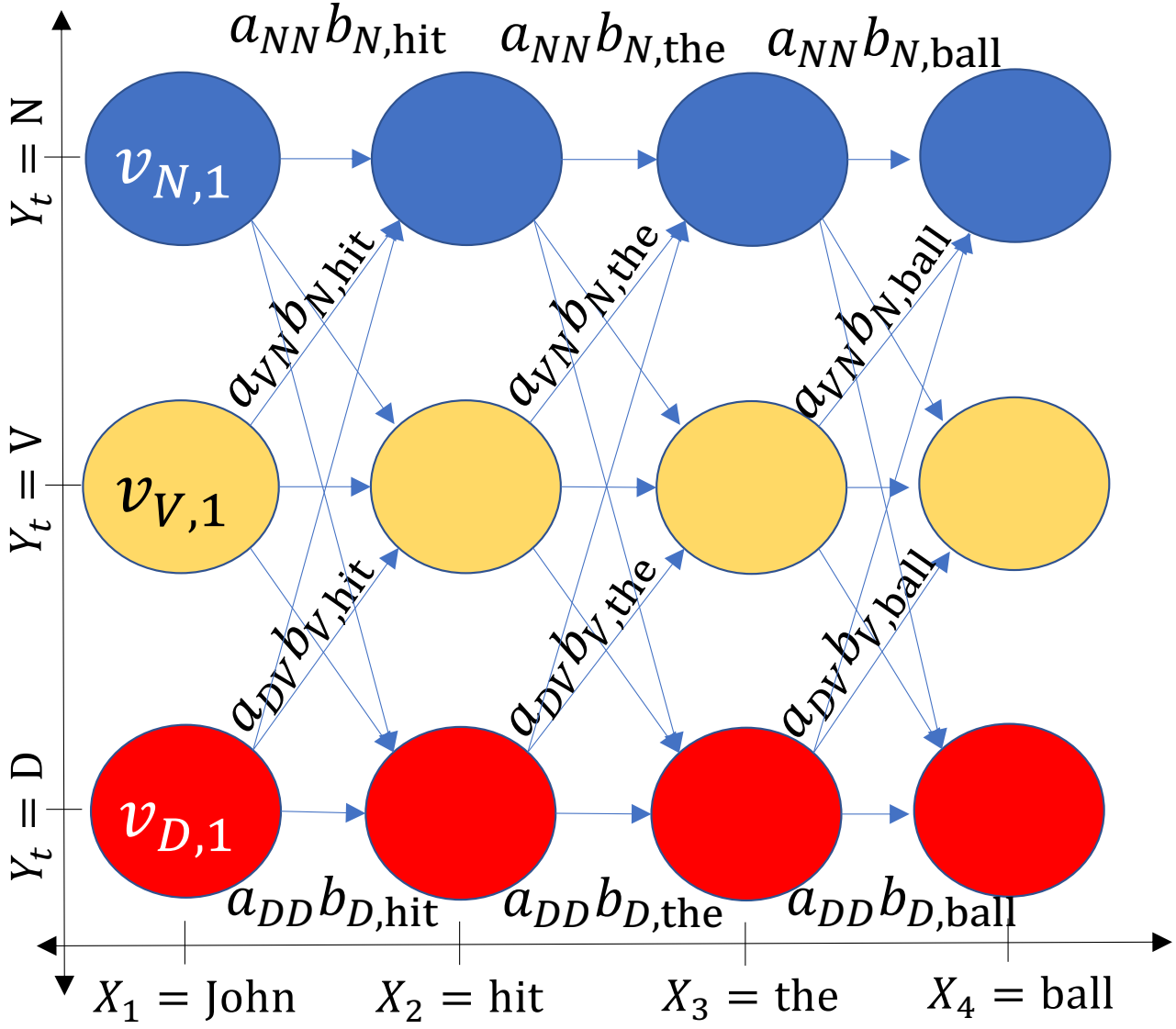
$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$



Trellis

Node Probability:

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$



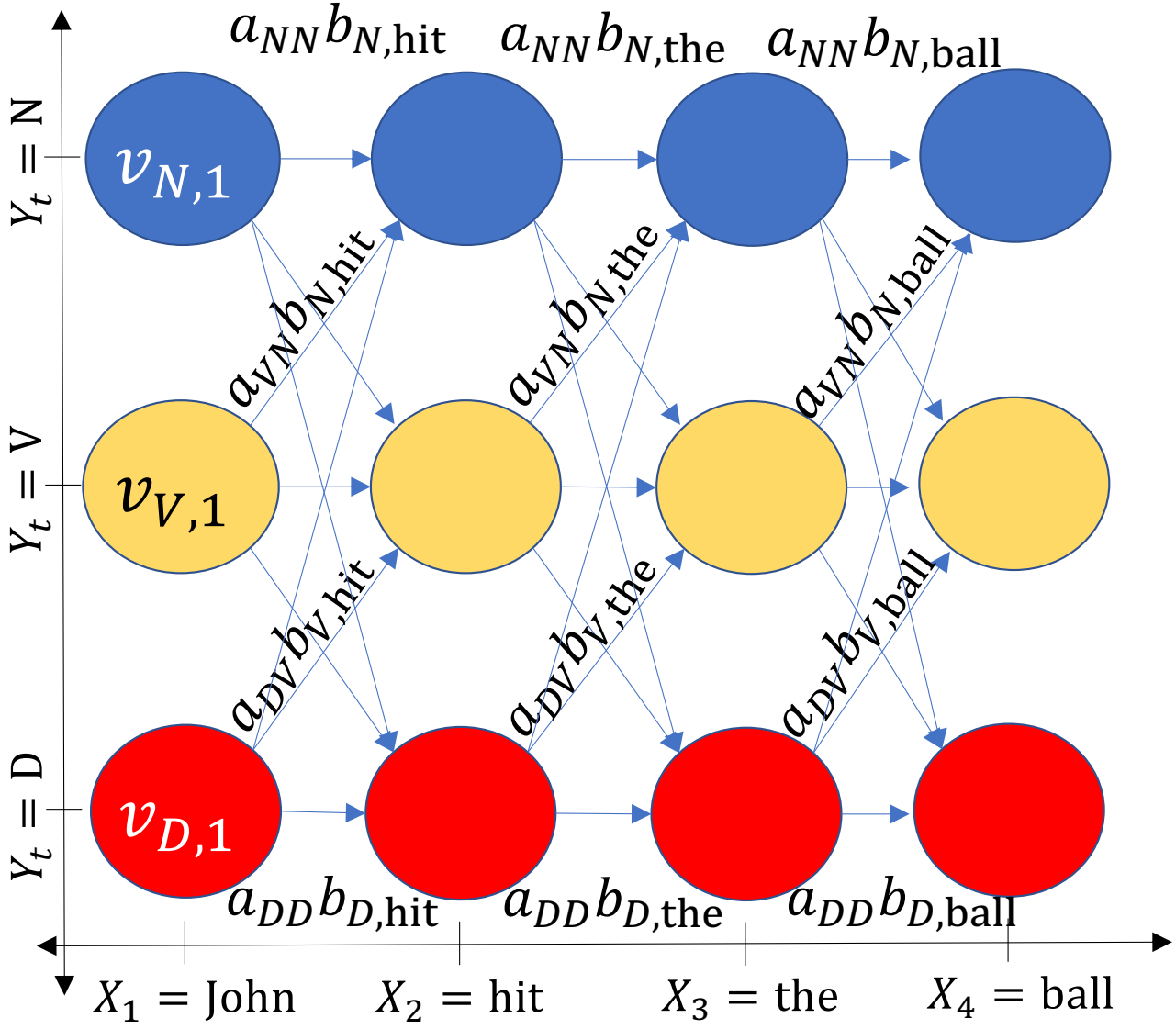
Trellis

Node Probability:

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$

For example, suppose

$$\max_i v_{i,1} a_{i,N} b_{N,\text{hit}} = v_{V,1} a_{V,N} b_{N,\text{hit}}$$



Trellis

Node Probability:

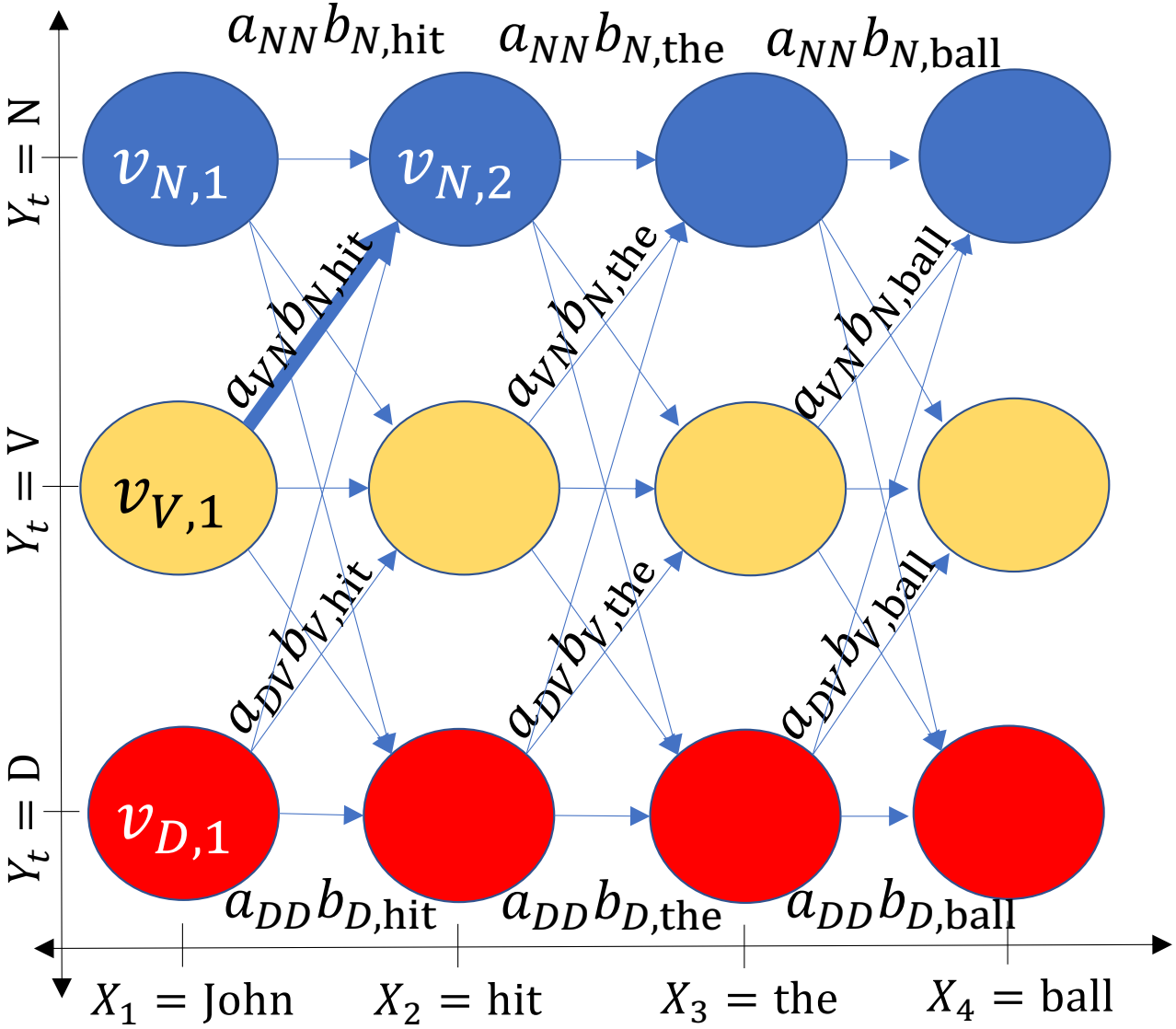
$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$

For example, suppose

$$\max_i v_{i,1} a_{i,N} b_{N,\text{hit}} = v_{V,1} a_{V,N} b_{N,\text{hit}}$$

Therefore

$$v_{N,2} = v_{V,1} a_{V,N} b_{N,\text{hit}}$$



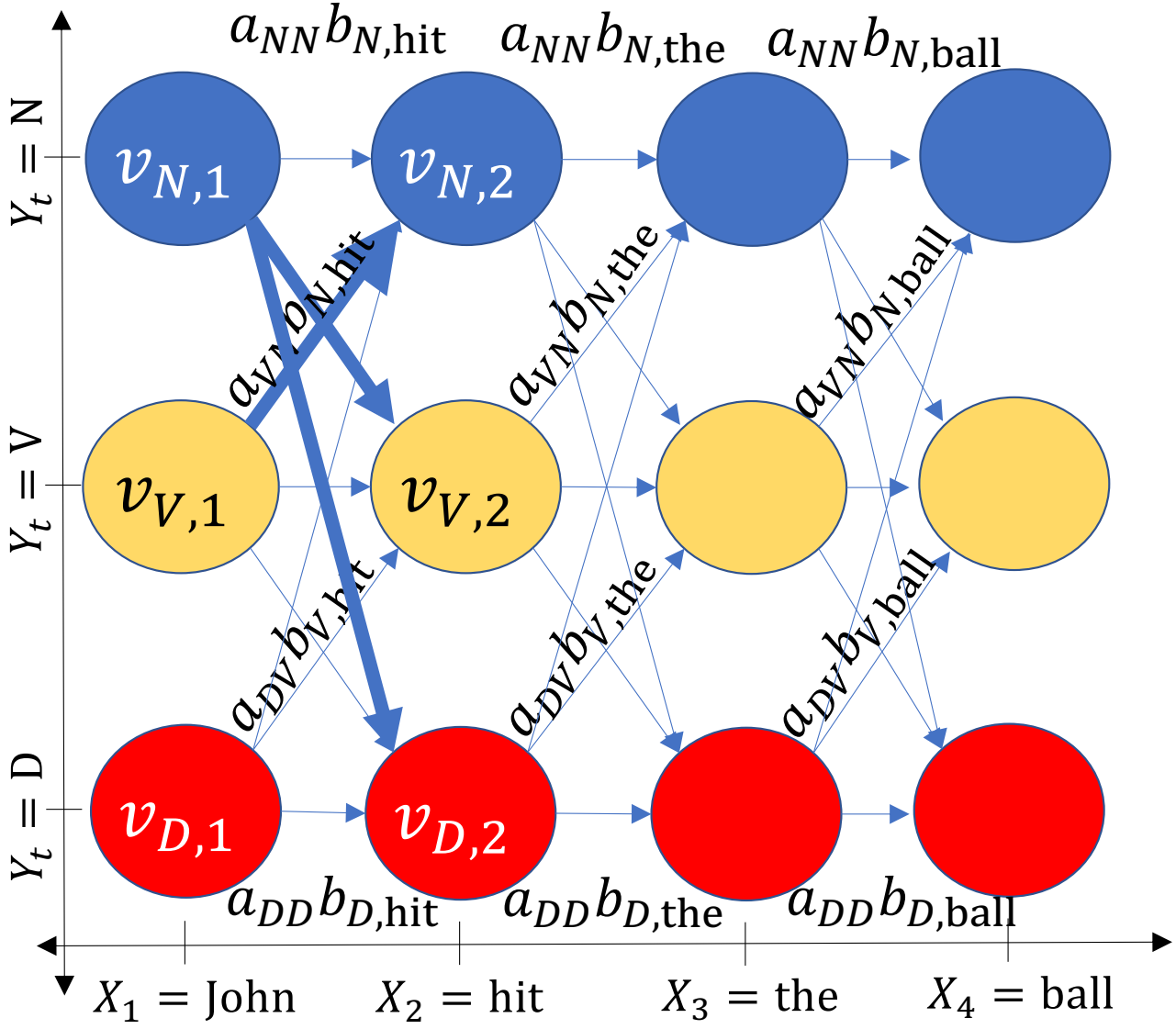
Trellis

Node Probability:

Similarly, we calculate

$$v_{j,2} = \max_i v_{i,1} a_{i,j} b_{j,\text{hit}}$$

…for all of the other nodes at time 2.



Trellis

Node Probability:

Similarly, we calculate

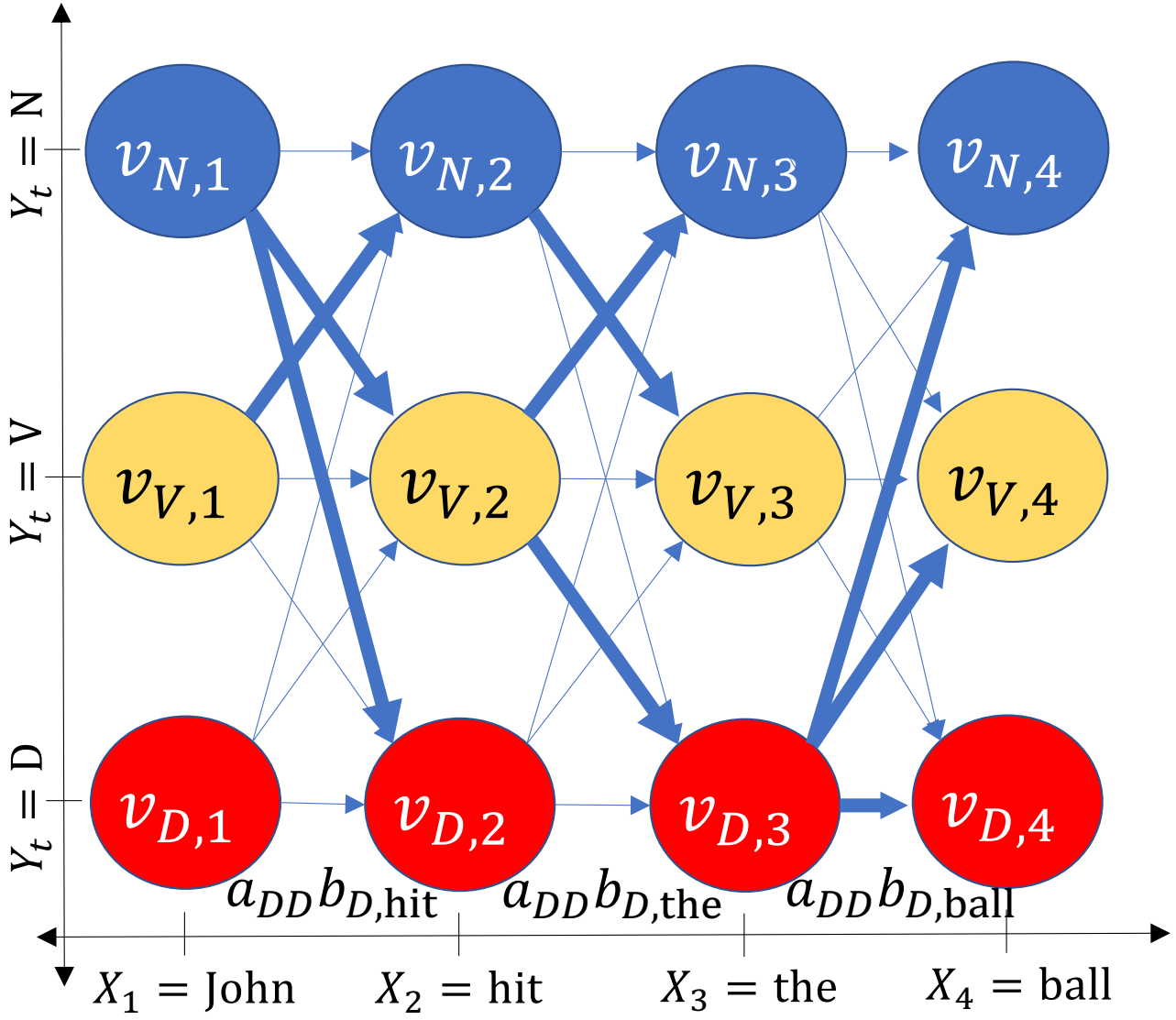
$$v_{j,2} = \max_i v_{i,1} a_{i,j} b_{j,\text{hit}}$$

…for all of the other nodes at time 2.

…and then calculate

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$

…for every other time step.



Trellis

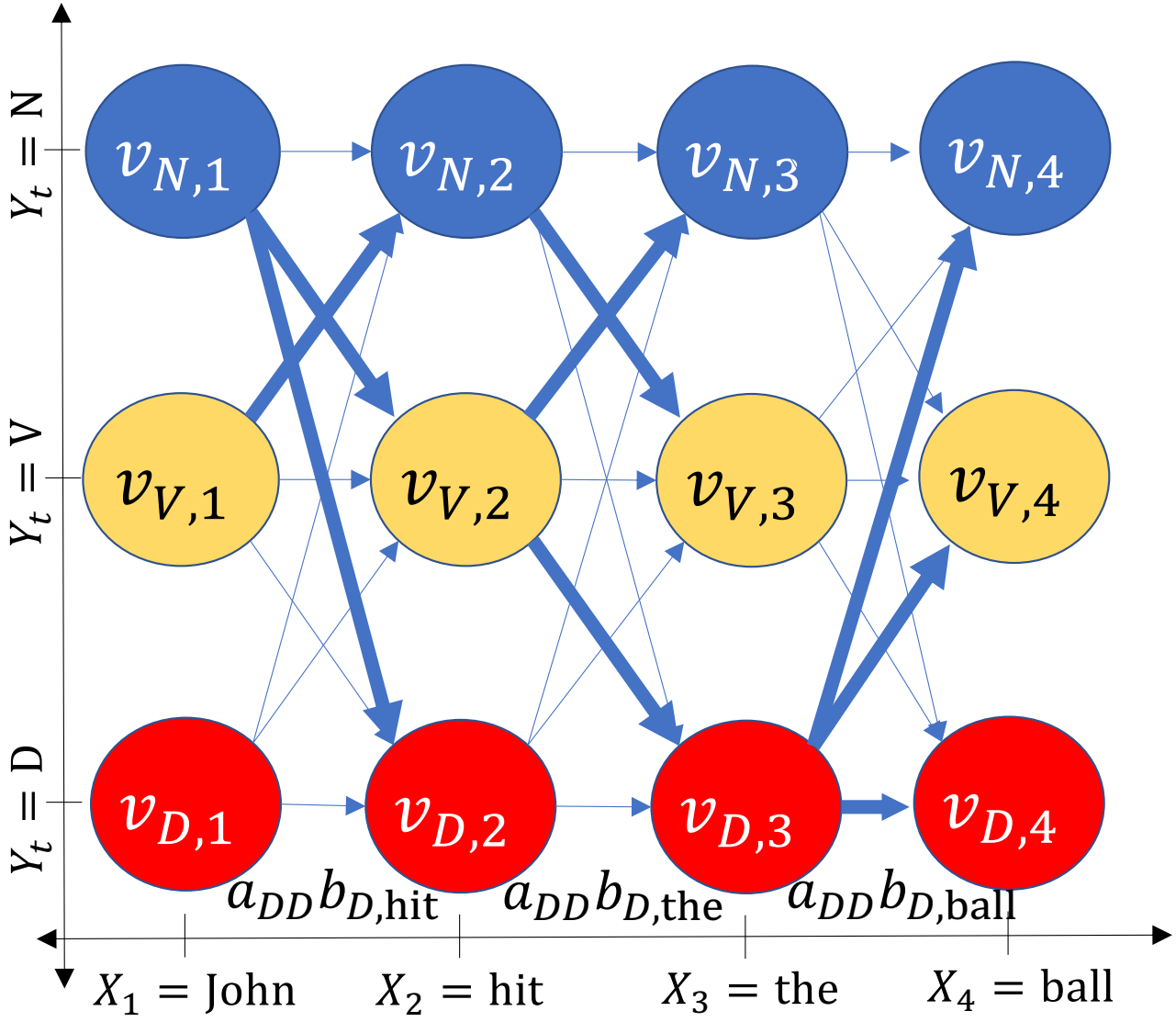
Node Probability:

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$

Backpointer:

$$i^*_{j,t} = \operatorname{argmax}_i v_{i,t-1} a_{ij} b_{j,x_t}$$

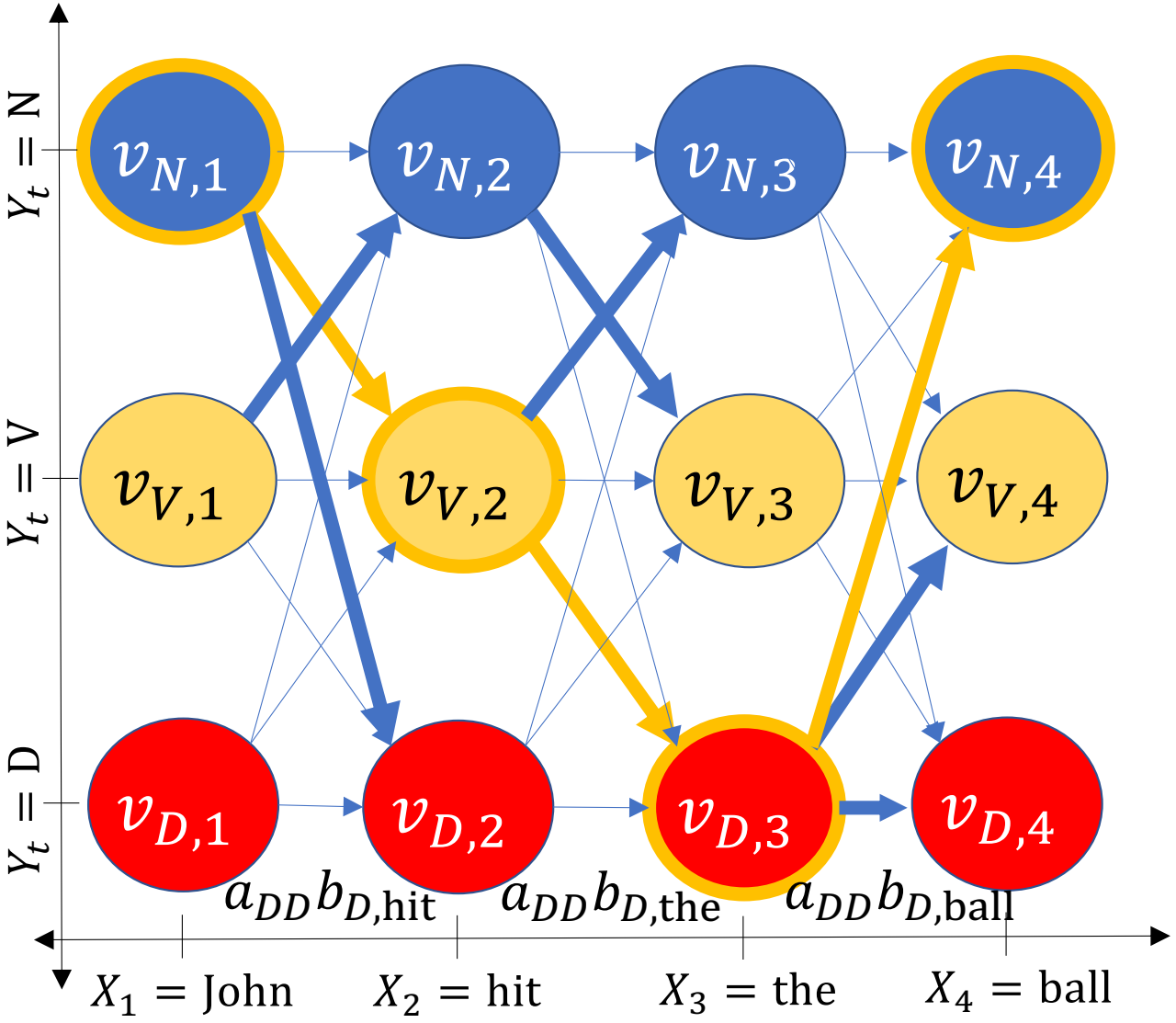
Shown: possible backpointers. Actual backpointers depend on model parameters!



Backtrace

Find the node with the highest value of $v_{j,T}$ at the end, and follow the backpointers!

Shown: possible backtrace. Actual backtrace depends on model parameters!



Viterbi algorithm key formulas

Initial Node Probability:

$$v_{j,1} = \pi_j b_{j,x_1}$$

Edge Probability:

$$a_{ij} b_{j,x_t}$$

Node Probability:

$$v_{j,t} = \max_i v_{i,t-1} a_{ij} b_{j,x_t}$$

Backpointer:

$$i^*_{j,t} = \operatorname{argmax}_i v_{i,t-1} a_{ij} b_{j,x_t}$$

Viterbi algorithm key formulas

Initial Node Probability:

$$\log v_{j,1} = \log \pi_j + \log b_{j,x_1}$$

Edge Probability:

$$\log a_{ij} + \log b_{j,x_t}$$

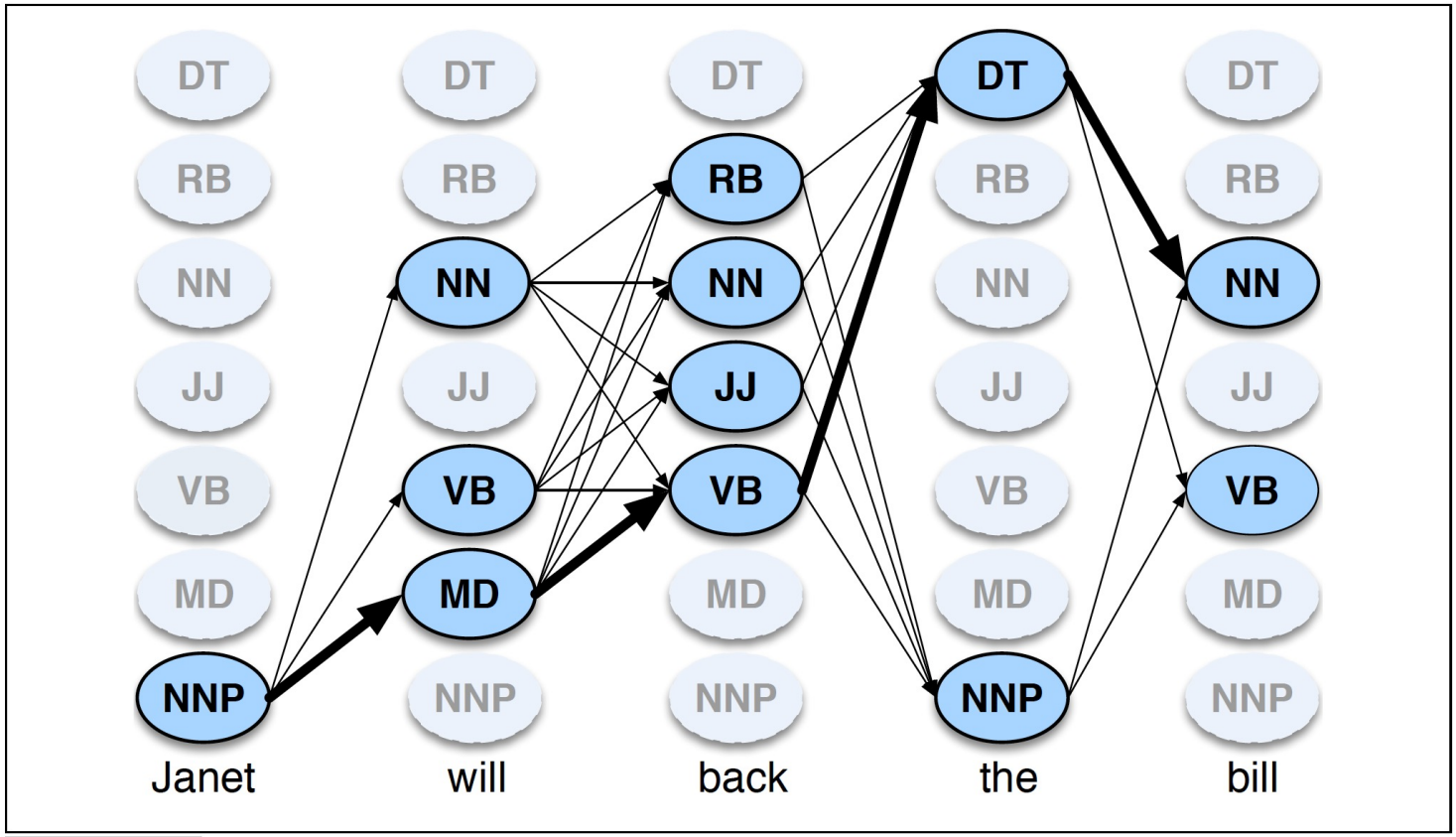
Node Probability:

$$\ln v_{j,t} = \max_i \left(\ln v_{i,t-1} + \ln a_{ij} + \ln b_{j,x_t} \right)$$

Backpointer:

$$i^*_{j,t} = \operatorname{argmax}_i \left(\ln v_{i,t-1} + \ln a_{ij} + \ln b_{j,x_t} \right)$$

Example from Jurafsky & Martin



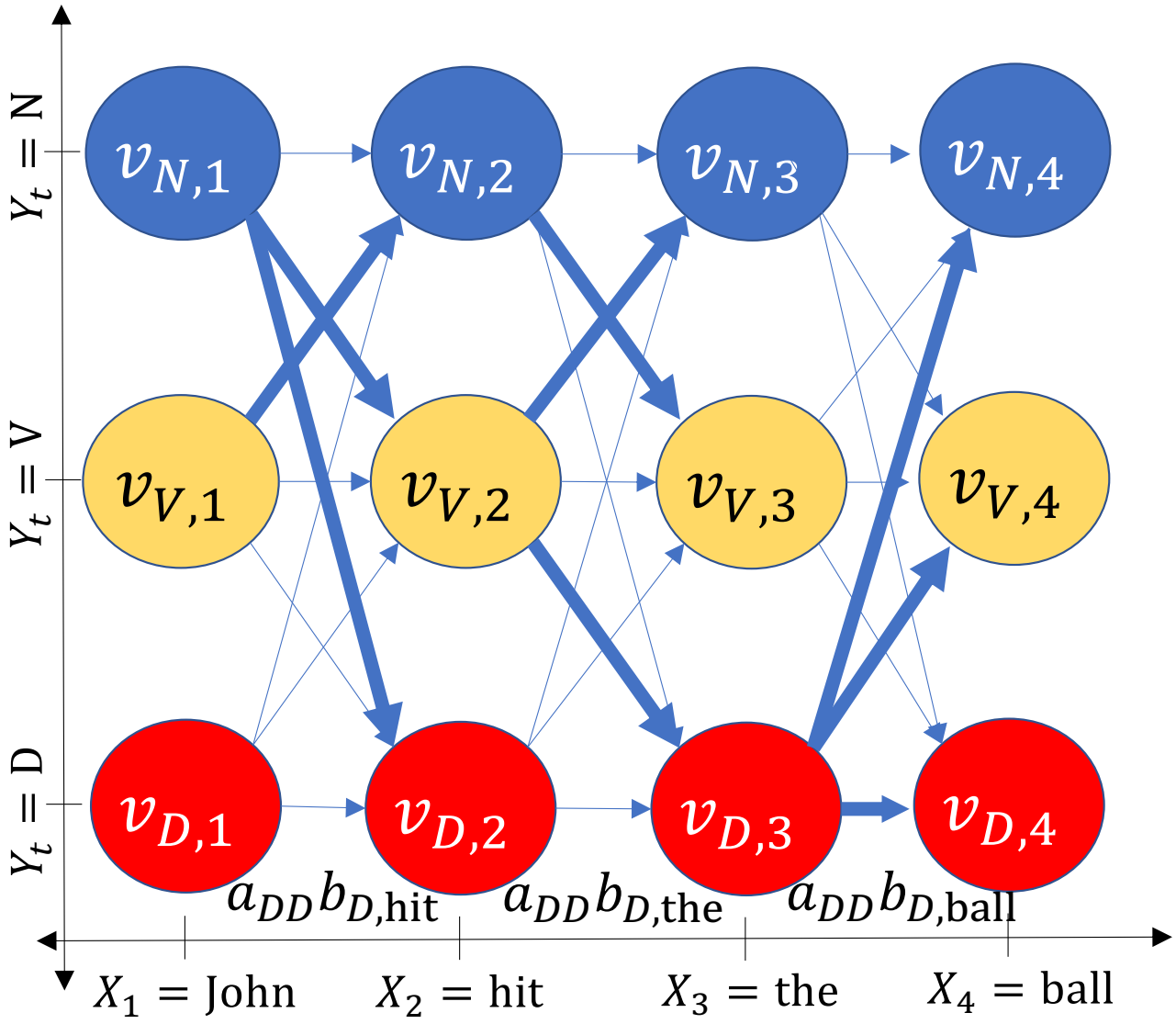
Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

From HMM to Neural Net

Viterbi algorithm:

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$



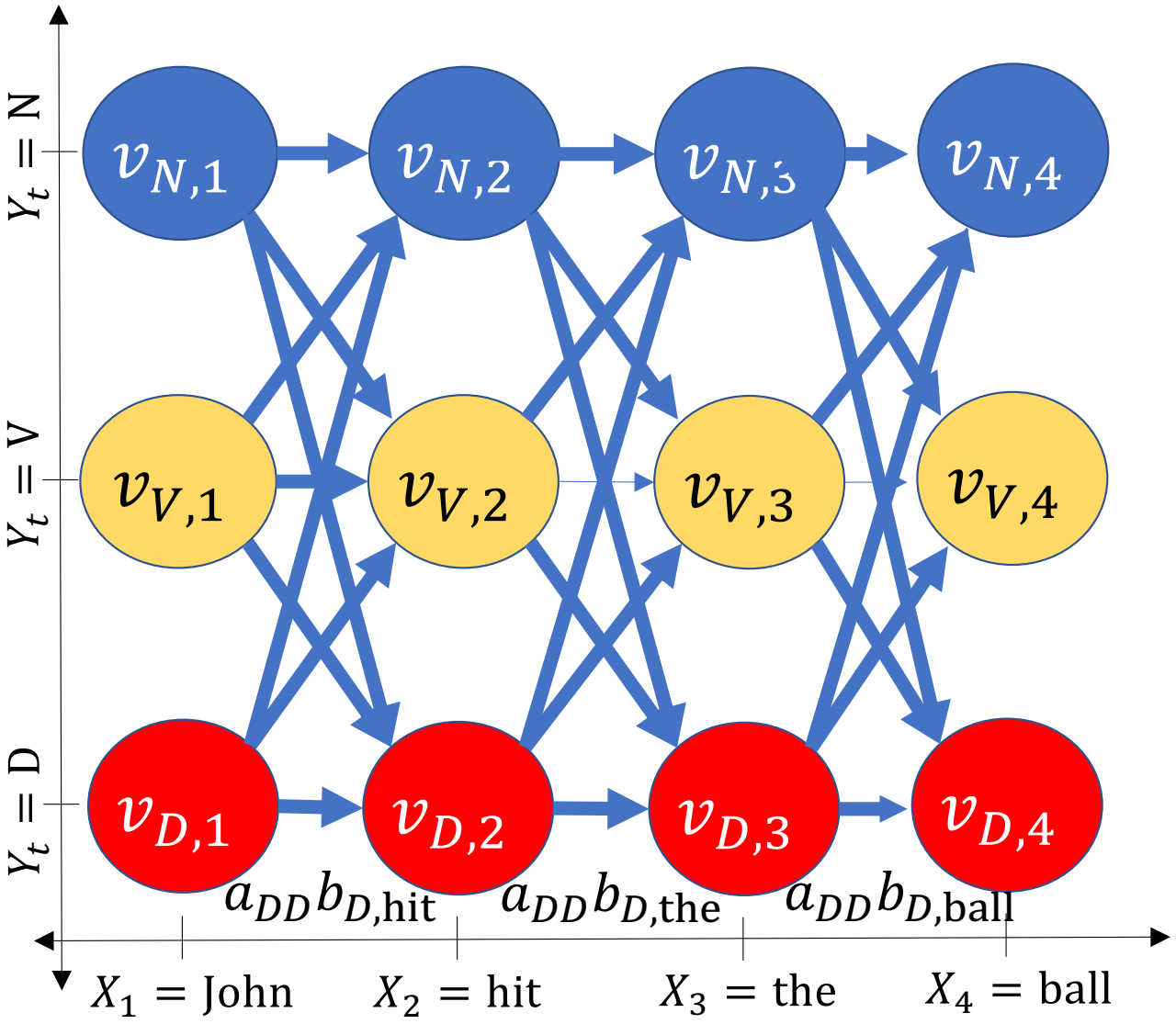
From HMM to Neural Net

Viterbi algorithm:

$$v_{j,t} = \max_i v_{i,t-1} a_{i,j} b_{j,x_t}$$

Suppose we want $v_{j,t}$ to add over all previous paths, instead of being the maximum. We get this using the belief propagation equation:

$$v_{j,t} = \sum_i v_{i,t-1} a_{i,j} b_{j,x_t}$$



From HMM to Neural Net

Here's a weird way to write the belief propagation equation ($v_{j,t} = \sum_i v_{i,t-1} a_{i,j} b_{j,x_t}$):

$$v_{j,t} = e^{\ln b_{j,x_t} + \ln \sum_i v_{i,t-1} a_{i,j}}$$

Suppose that we define an input one-hot vector, $\vec{z}_t$, such that

$$z_{k,t} = \begin{cases} 1 & k = x_t \\ 0 & \text{otherwise} \end{cases}$$

…and suppose we define a matrix B as:

$$B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,|V|} \\ \vdots & \ddots & \vdots \\ b_{N,1} & \cdots & b_{N,|V|} \end{bmatrix}$$

…then b_{j,x_t} is the j^{th} element of the vector $B\vec{z}_t$.

From HMM to Neural Net

$$v_{j,t} = e^{\ln b_{j,x_t} + \ln \sum_i v_{i,t-1} a_{i,j}}$$

Similarly, define the vector $\vec{v}_t$ and the matrix A to be:

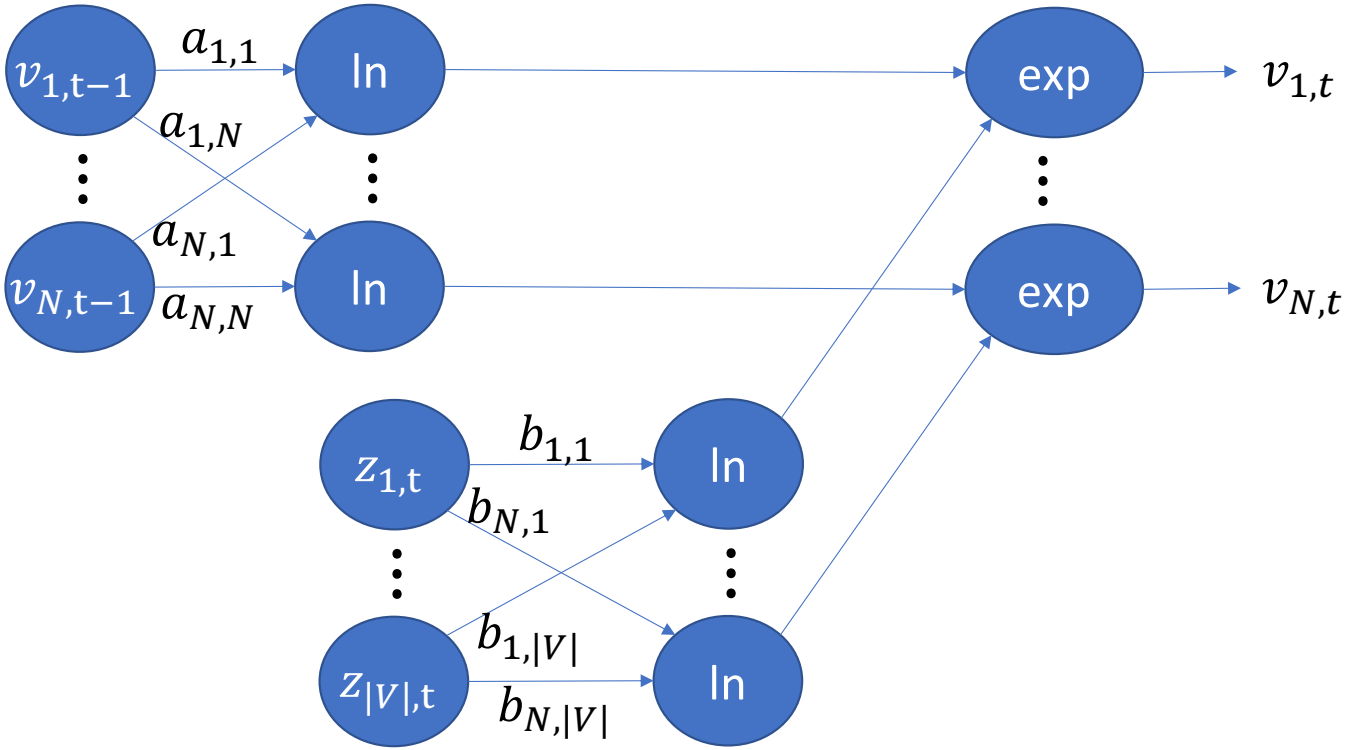
$$\vec{v}_t = \begin{bmatrix} v_{1,t} \\ \vdots \\ v_{N,t} \end{bmatrix}, \qquad A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{bmatrix}$$

…then $\sum_i v_{i,t-1} a_{i,j}$ is the j^{th} element of the vector $A^T \vec{v}_{t-1}$.

From HMM to Neural Net

With those definitions, the belief propagation equation can be implemented using the following neural network:

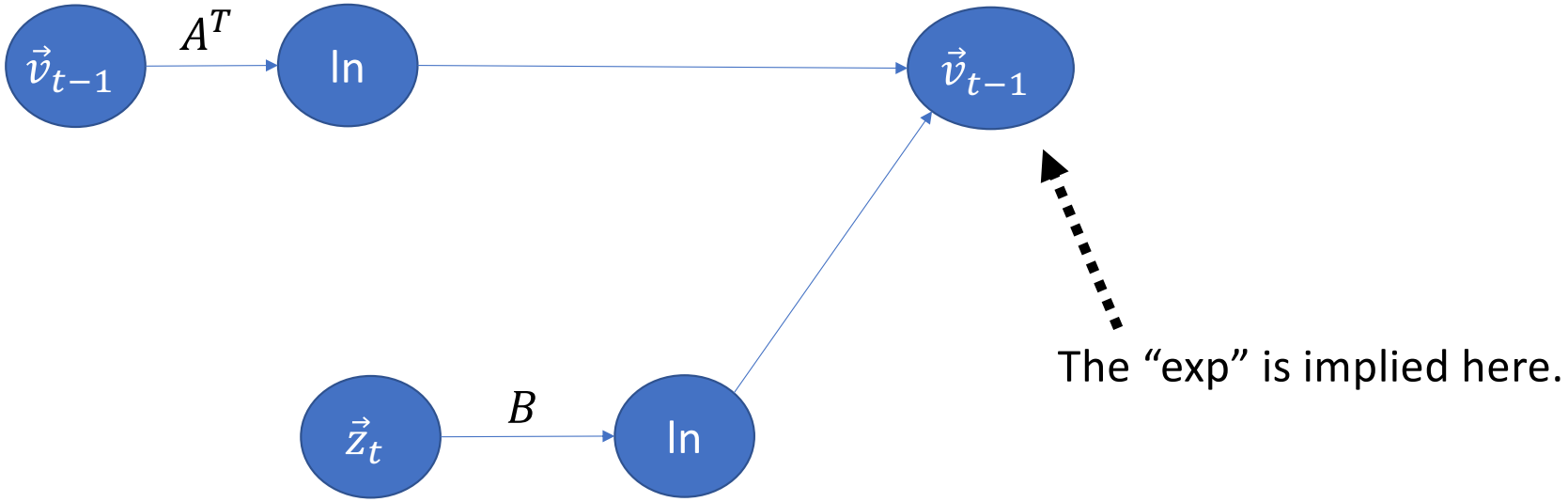
$$v_{j,t} = e^{\ln b_{j,x_t} + \ln \sum_i v_{i,t-1} a_{i,j}}$$



From HMM to Neural Net

Using scalar nonlinearities applied after matrix transforms, we can write it as:

$$\vec{v}_t = e^{\ln B\vec{z}_t + \ln A^T \vec{v}_{t-1}}$$



From HMM to Neural Net: Implementation Hack

If we want to implement belief propagation exactly, then we need to use these nonlinearities:

$$\vec{v}_t = e^{\ln B\vec{z}_t + \ln A^T \vec{v}_{t-1}}$$

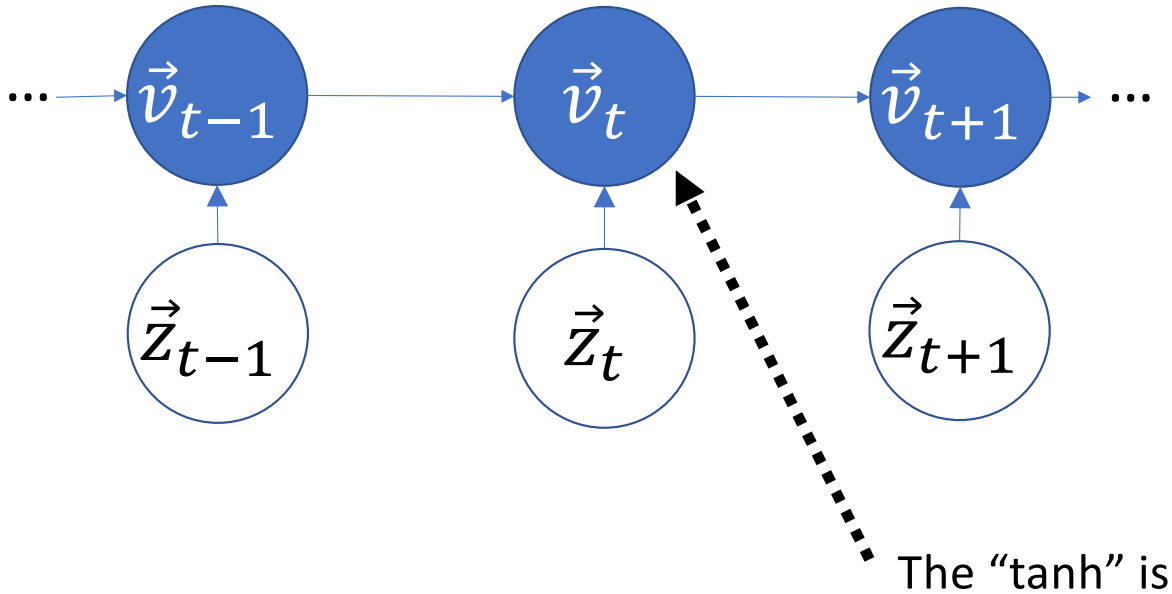
In a neural network, though, the parameters will be learned from data, in order to minimize some loss function. The exact form of the nonlinearity is therefore not too important, so we usually choose some nonlinearity that's easily available in pytorch. For example, a common one in recurrent neural networks is:

$$\vec{v}_t = \tanh(B\vec{z}_t + A^T \vec{v}_{t-1}) = \frac{1 - e^{-2\left(B\vec{z}_t + A^T \vec{v}_{t-1}\right)}}{1 + e^{-2(B\vec{z}_t + A^T \vec{v}_{t-1})}}$$

From HMM to Neural Net: Implementation Hack

Using the implementation hack, the network simplifies to:

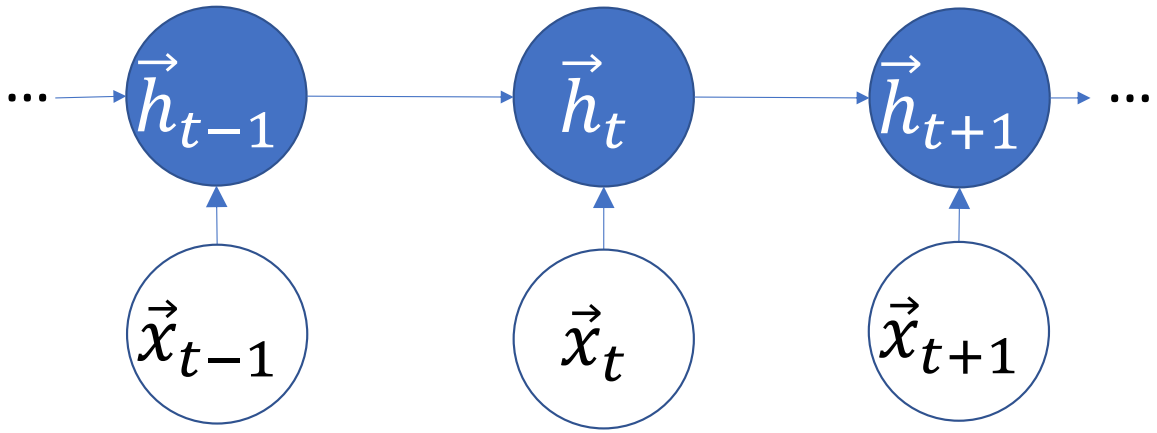
$$\vec{v}_t = \tanh(B\vec{z}_t + A^T\vec{v}_{t-1})$$



Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Recurrent neural network (RNN)

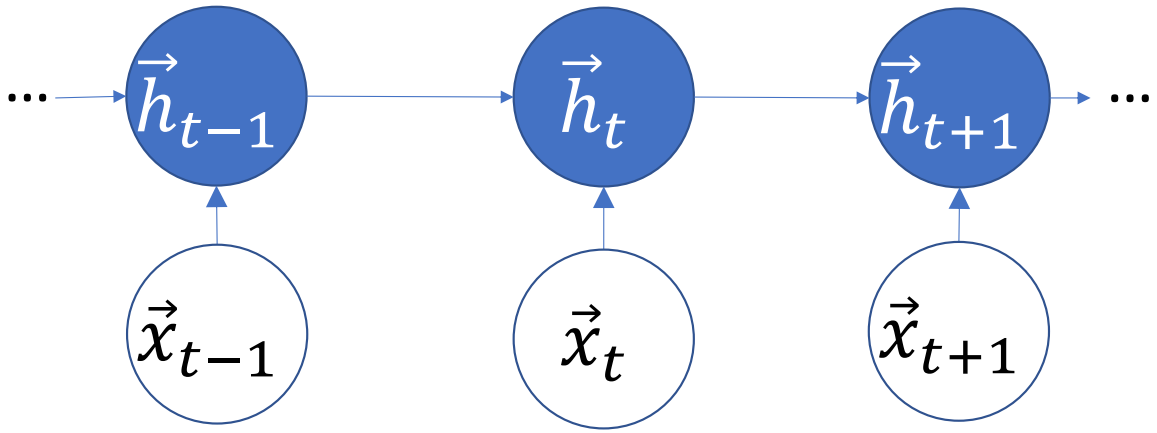


A recurrent neural network (RNN) is a network in which the hidden nodes at time t depend on the input at time t, and on the hidden nodes at time t-1:

$$\vec{h}_t = g\left(A^T \vec{h}_{t-1} + B\vec{x}_t\right)$$

...where A and B are weight matrices, and g() is some kind of scalar nonlinearity.

Recurrent neural network (RNN)

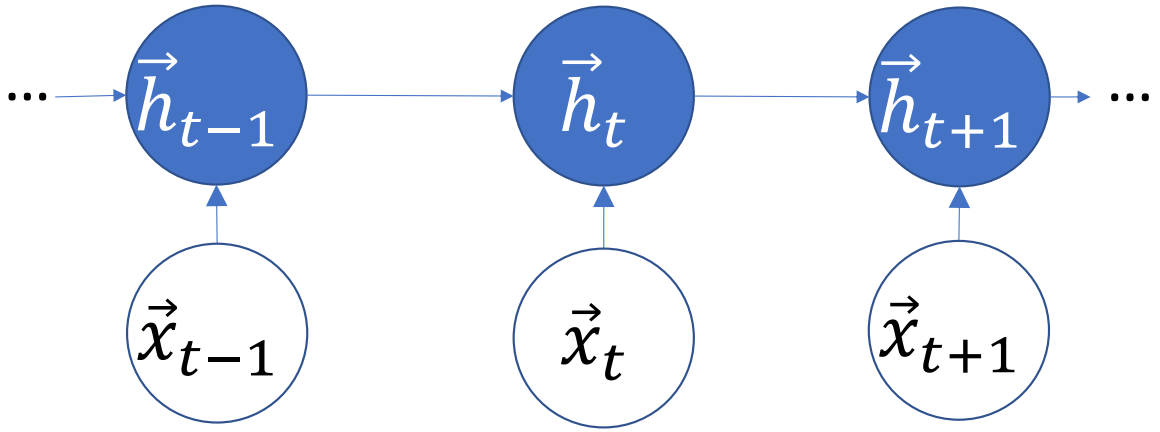


For example, suppose that we have the sentence

"John hit the ball"

... and we want to find each word's part of speech.

Recurrent neural network (RNN)



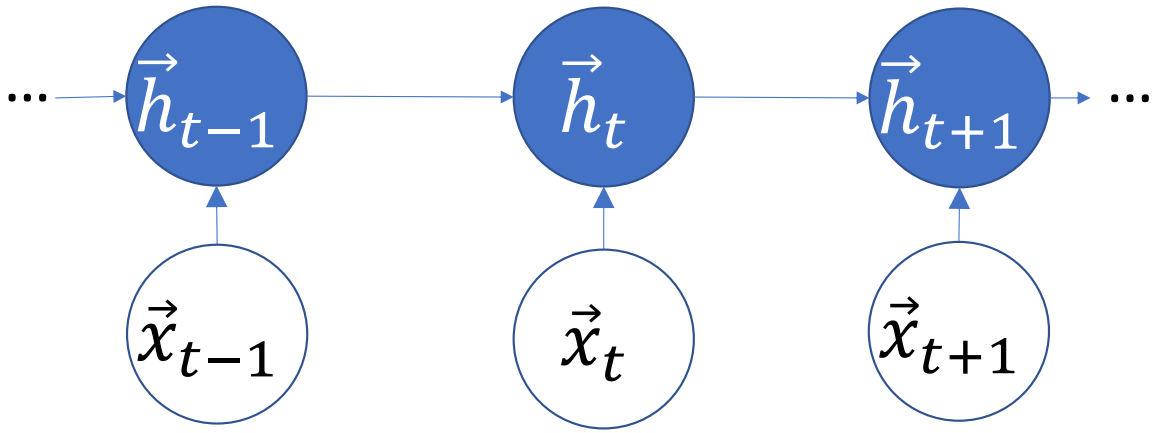
Let's define

$$\vec{x}_t = \begin{bmatrix} 1 \text{ if } X_t = \text{ball} \\ 1 \text{ if } X_t = \text{hit} \\ 1 \text{ if } X_t = \text{John} \\ 1 \text{ if } X_t = \text{the} \end{bmatrix}$$

...so the observation sequence is...

$$\vec{x}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \vec{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \vec{x}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \vec{x}_4 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Recurrent neural network (RNN)



Let's define

$$\vec{h}_t = g\left(A^T \vec{h}_{t-1} + B\vec{x}_t\right) \approx \begin{bmatrix} P(Y_t = \text{Det}|X_1, \ldots, X_t) \\ P(Y_t = \text{Noun}|X_1, \ldots, X_t) \\ P(Y_t = \text{Verb}|X_1, \ldots, X_t) \end{bmatrix}$$

If we define $\vec{h}_0 = [0,0,0]^T$, and with a reasonable set of weight matrices and a softmax nonlinearity (instead of tanh), one simulation result gave this result:

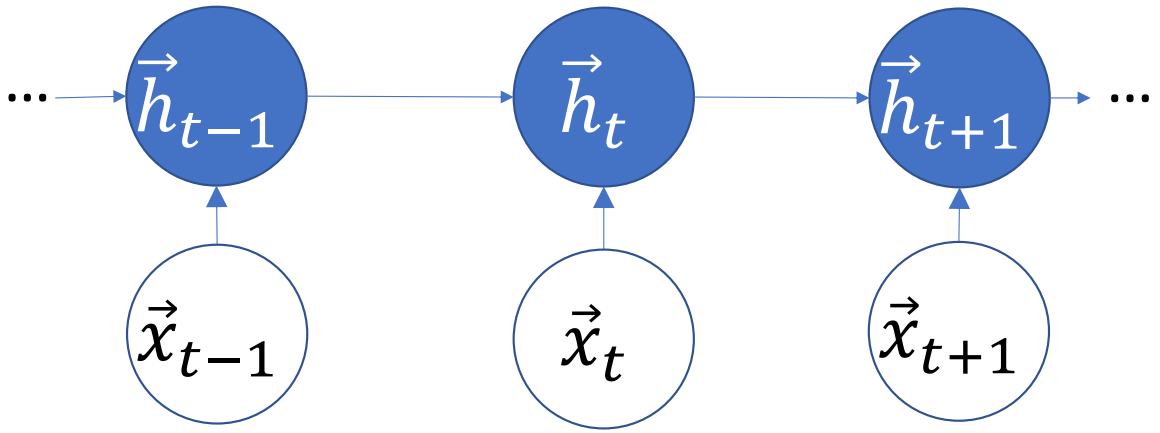
$$\vec{h}_1 = \begin{bmatrix} 0.28 \\ 0.44 \\ 0.28 \end{bmatrix}, \vec{h}_2 = \begin{bmatrix} 0.20 \\ 0.20 \\ 0.60 \end{bmatrix}, \vec{h}_3 = \begin{bmatrix} 0.63 \\ 0.18 \\ 0.18 \end{bmatrix}, \vec{h}_4 = \begin{bmatrix} 0.24 \\ 0.53 \\ 0.24 \end{bmatrix}$$

Thus "John hit the ball" has the following most-likely POS: "Noun Verb Det Noun."

Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Training an RNN



In order to match the convention used in Wikipedia, let's rename the weight matrices as

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right)$$

An RNN is trained using gradient descent, just like any other neural network!

$$u_{j,i} \leftarrow u_{j,i} - \eta \frac{\partial \mathfrak{L}}{\partial u_{j,i}}$$

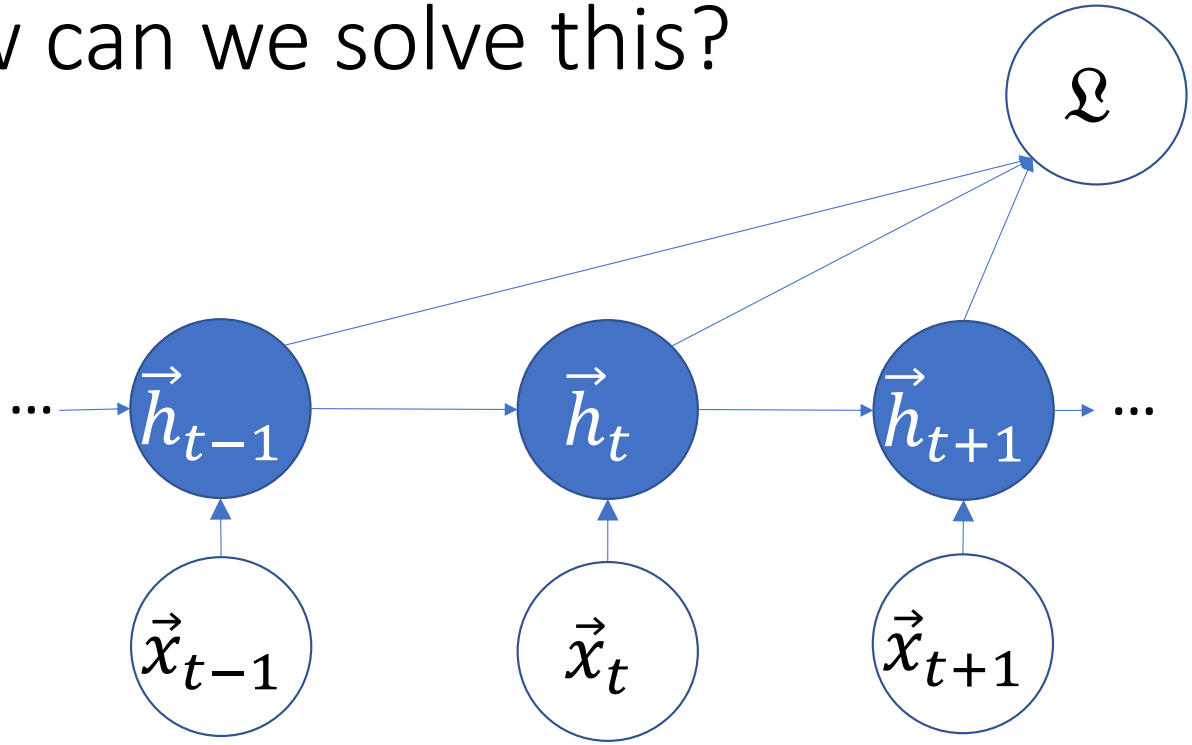
$$w_{j,k} \leftarrow w_{j,k} - \eta \frac{\partial \mathfrak{L}}{\partial w_{j,k}}$$

…where $\mathfrak{L}$ is the loss function, and η is a step size.

Training an RNN: How can we solve this?

The big difference is that now the loss function depends on U and W in many different ways:

- The loss function depends on each of the state vectors $\vec{h}_t$
- Each of the state vectors depends on U and W
- Each of the state vectors ALSO depends on the previous state vector, $\vec{h}_{t-1}$ …
- … which ALSO depends on U and W, and on $\vec{h}_{t-2}$ …
- AUGH!

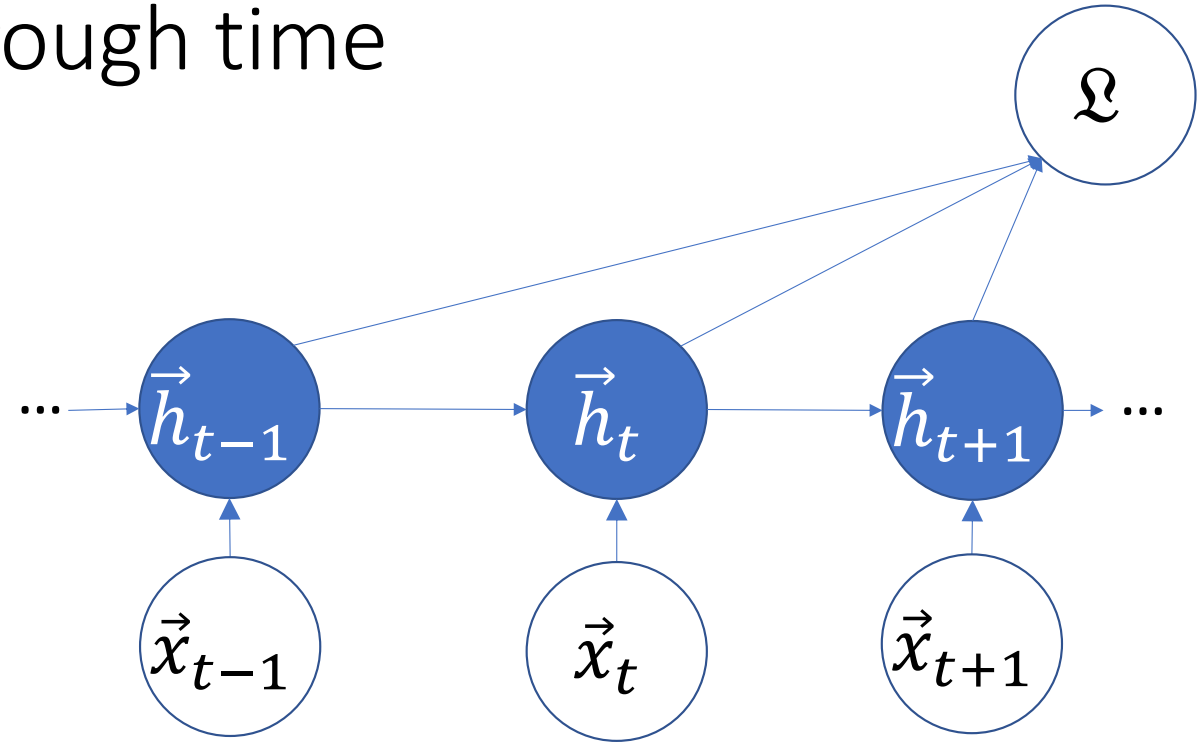


Back-propagation through time

The solution is something called back-propagation through time:

$$\frac{d\mathfrak{L}}{dh_{i,t}} = \frac{\partial \mathfrak{L}}{\partial h_{i,t}} + \frac{d\mathfrak{L}}{dh_{j,t+1}} \frac{\partial h_{j,t+1}}{\partial h_{i,t}}$$

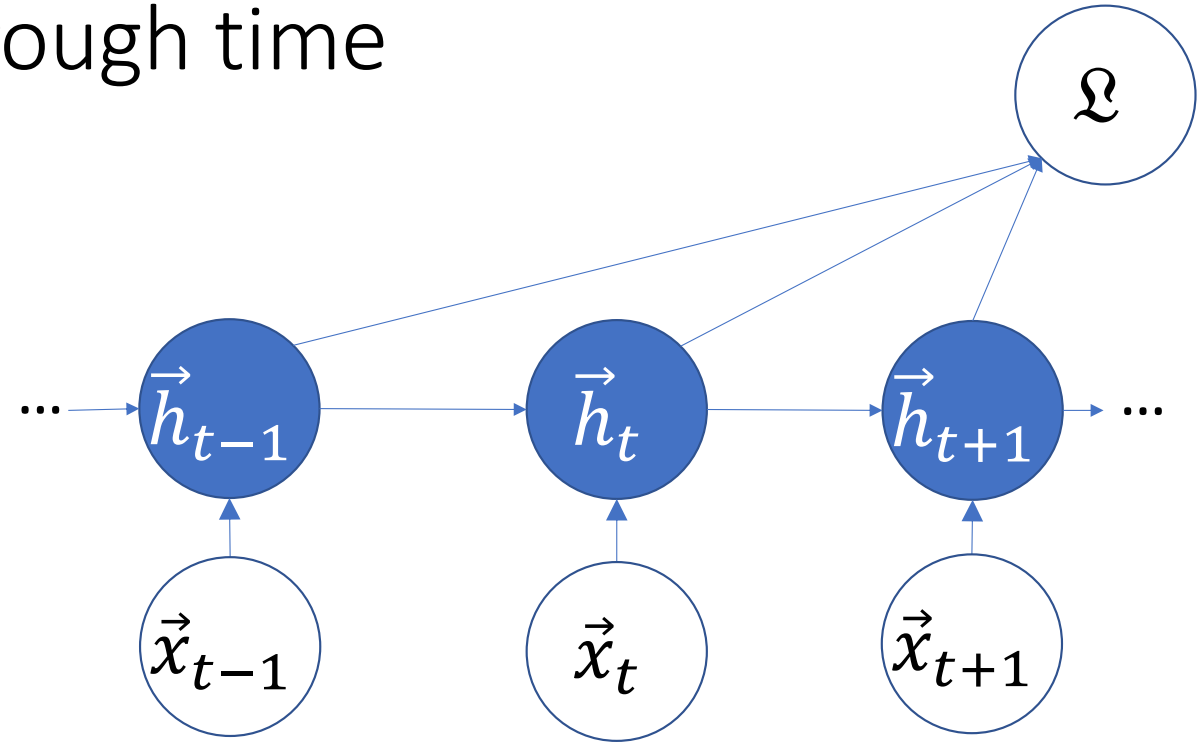
- The first term measures losses caused directly by $h_{i,t}$, for example, if $h_{i,t}$ is wrong.
- The second term measures losses caused indirectly, for example, because $h_{i,t}$ caused $h_{j,t+1}$ to be wrong.



Back-propagation through time

Once we've back-propagated through time, then we add up all the different ways in which the weight matrix affects the output:

$$\frac{d\mathfrak{L}}{du_{j,i}} = \sum_{t=1}^{T} \frac{d\mathfrak{L}}{dh_{i,t}} \frac{\partial h_{i,t}}{\partial u_{j,i}}$$

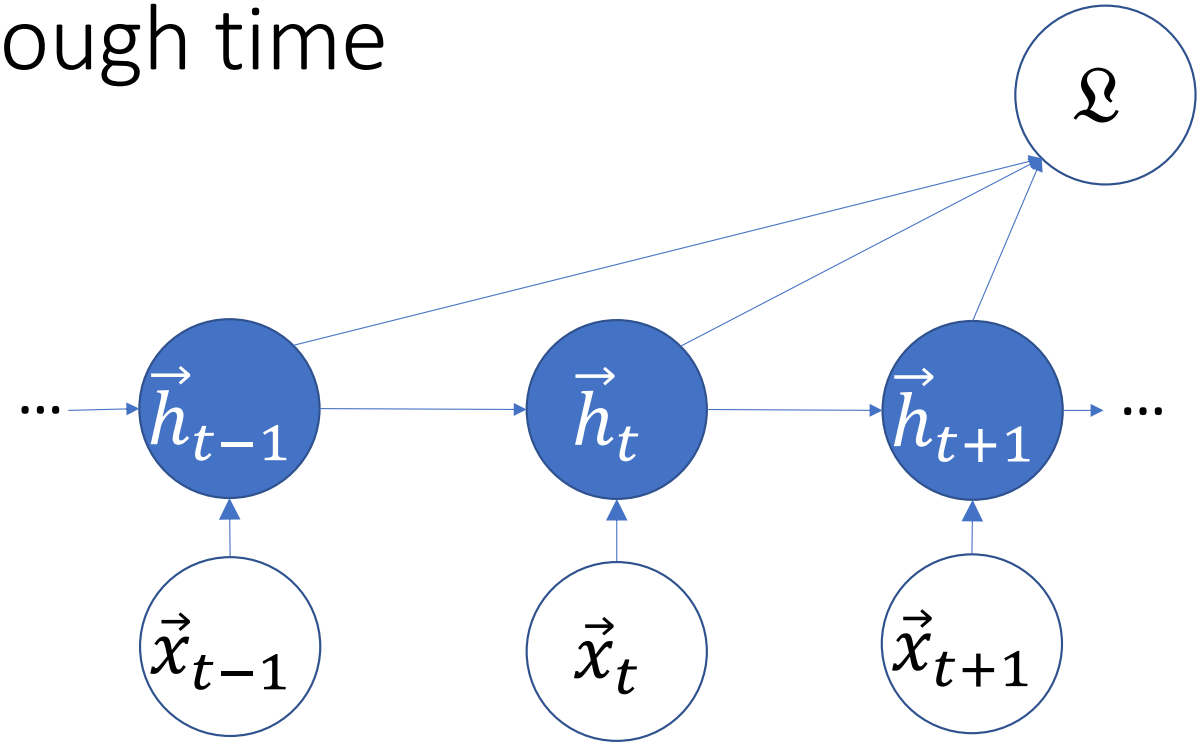


Back-propagation through time

Notice that this is just like training a very deep network!

- Back-propagation through time: back-propagate from time step $t+1$ to time step t
- Back-propagation in a very deep network: back-propagate from layer $l+1$ to layer l

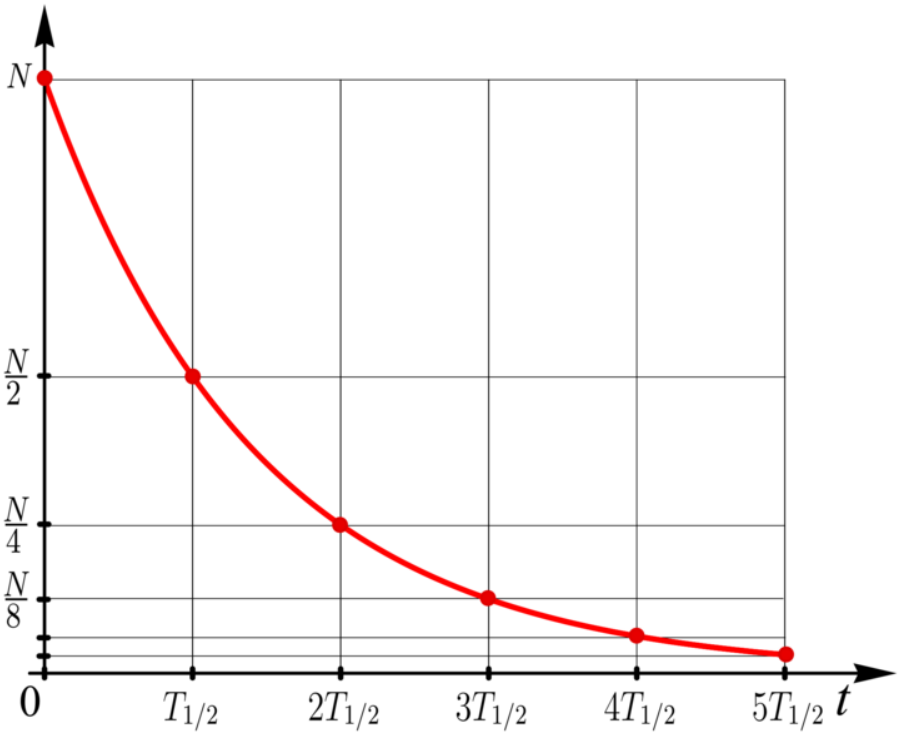
Toolkits like PyTorch use the same code in both cases.



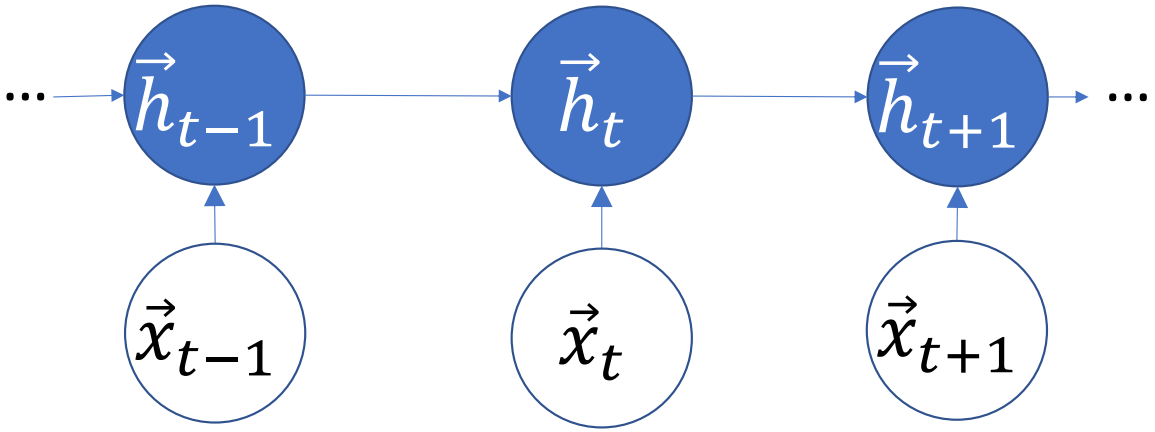
Outline

- Syntax and semantics
- Part of speech tagging
- An HMM for POS tagging
- The Viterbi algorithm for POS tagging
- From HMM to Neural Net
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Exponential forgetting



Exponential-decay.png. CC-SA-4.0, Svjo, 2017



Regular RNNs have a problem: they forget what they know!

For example, suppose that the feedback matrix is $\mathrm{U} = \left(\frac{1}{2}\right)$, so that $\vec{h}_t = \left(\frac{1}{2}\right)\vec{h}_{t-1}$.

Then the state vector decays as $\left(\frac{1}{2}\right)^t$!

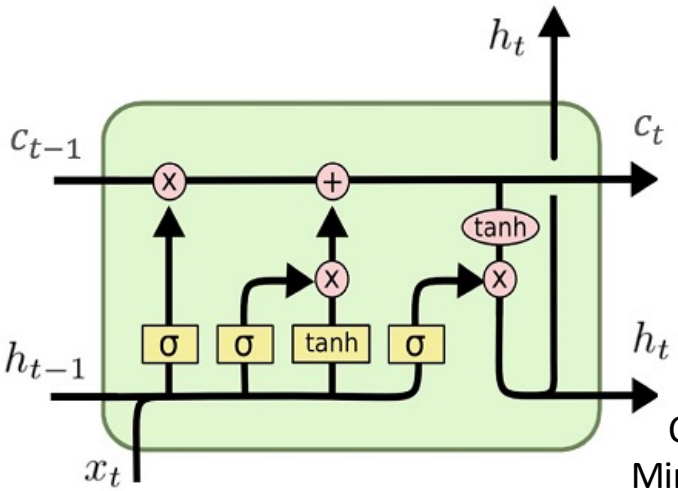
Long-Short Term Memory (LSTM)

A Long-Short Term Memory network (LSTM) solves the exponential forgetting problem using something called a gate.

Remember that a normal RNN computes

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right)$$

…so if $\mathrm{U} = \left(\frac{1}{2}\right)$, and if $g(\cdot)$ is linear, then $\vec{h}_t = \left(\frac{1}{2}\right)^t$.



CC-SA-4.0, MingxianLin, 2018

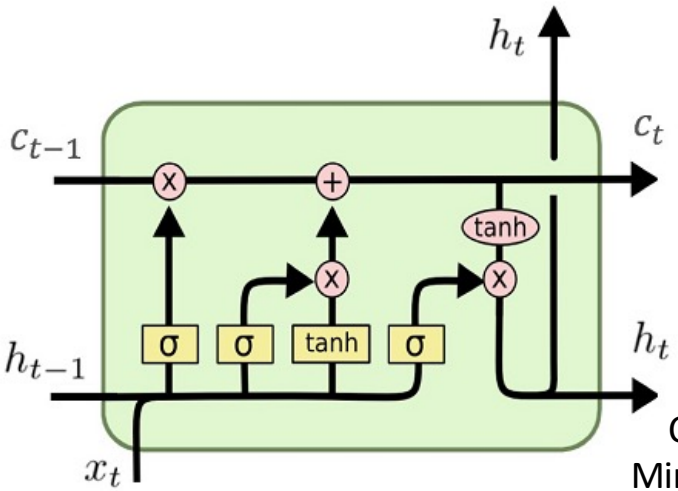
LSTM
(Long-Short Term Memory)

Long-Short Term Memory (LSTM)

An LSTM computes

$$\vec{c}_t = f_t \vec{c}_{t-1} + i_t \vec{x}_t$$

This is just like a regular RNN, except that now, f_t and i_t are not constant. They are adjusted, depending on what the LSTM sees in the input.



LSTM
(Long−Short Term Memory)

Long-Short Term Memory (LSTM)

An LSTM computes

$$\vec{c}_t = f_t \vec{c}_{t-1} + i_t \vec{x}_t$$

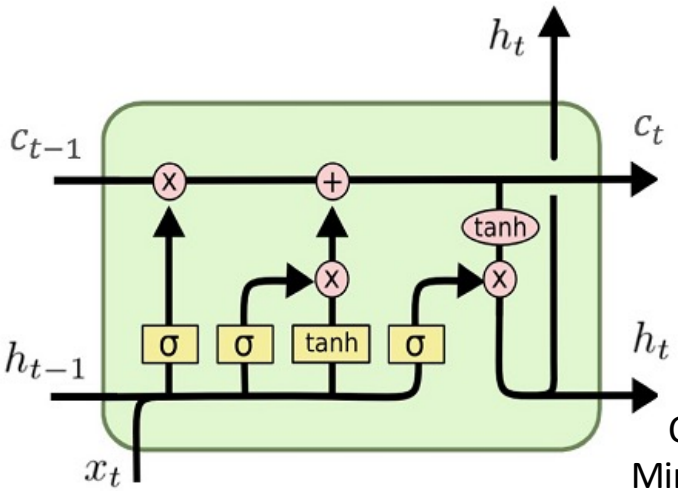
f_t and i_t are called the "forget gate" and the "input gate," respectively. They are computed as

$$f_t = \sigma\left(U_f \vec{h}_{t-1} + W_f \vec{x}_t\right)$$

$$i_t = \sigma\left(U_i \vec{h}_{t-1} + W_i \vec{x}_t\right)$$

…where $\sigma(\cdot)$ is the logistic sigmoid function. Remember that $0 < \sigma(\cdot) < 1$. So:

- If the LSTM wants to remember what it knows, then it will choose $f_t \approx 1$.
- If the LSTM wants to forget what it knows, then it will choose $f_t \approx 0$.



CC-SA-4.0, MingxianLin, 2018

LSTM
(Long-Short Term Memory)

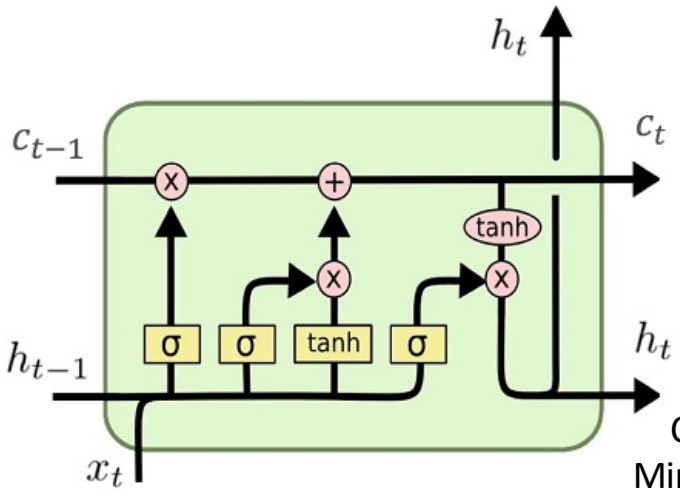
Long-Short Term Memory (LSTM)

$$f_t = \sigma\left(U_f \vec{h}_{t-1} + W_f \vec{x}_t\right)$$
$$i_t = \sigma\left(U_i \vec{h}_{t-1} + W_i \vec{x}_t\right)$$

In order to decide whether to remember what it knows, the LSTM compares $U_f \vec{h}_{t-1}$ to $W_f \vec{x}_t$.

Before it does that, it decides whether it needs to make such a comparison: $\vec{h}_{t-1}$ is equal to the previous time step's memory cell, multiplied by an "output gate" o_{t-1}:

$$\vec{h}_t = o_t \vec{c}_t$$
$$o_t = \sigma\left(U_o \vec{h}_{t-1} + W_o \vec{x}_t\right)$$



LSTM
(Long-Short Term Memory)

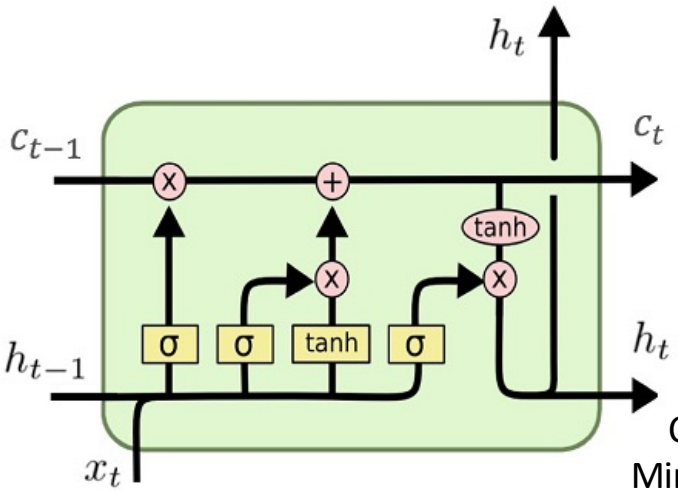
Long-Short Term Memory (LSTM)

An LSTM replaces the one equation of a normal RNN:

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right)$$

…with these five equations:

- **Forget Gate:** $f_t = \sigma\left(U_f\vec{h}_{t-1} + W_f\vec{x}_t\right)$
- **Input Gate:** $i_t = \sigma\left(U_i\vec{h}_{t-1} + W_i\vec{x}_t\right)$
- **Output Gate:** $o_t = \sigma\left(U_o\vec{h}_{t-1} + W_o\vec{x}_t\right)$
- **Cell:** $\vec{c}_t = f_t\vec{c}_{t-1} + i_t\vec{x}_t$
- **Output:** $\vec{h}_t = o_t\vec{c}_t$

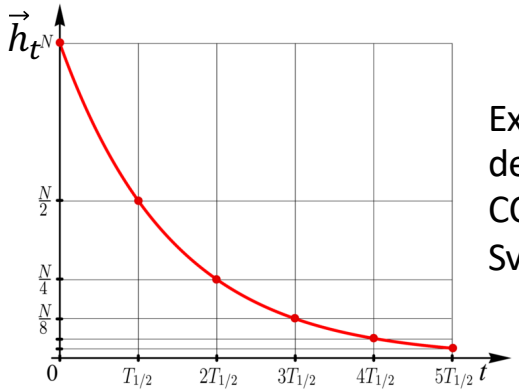


CC-SA-4.0, MingxianLin, 2018

LSTM
(Long-Short Term Memory)

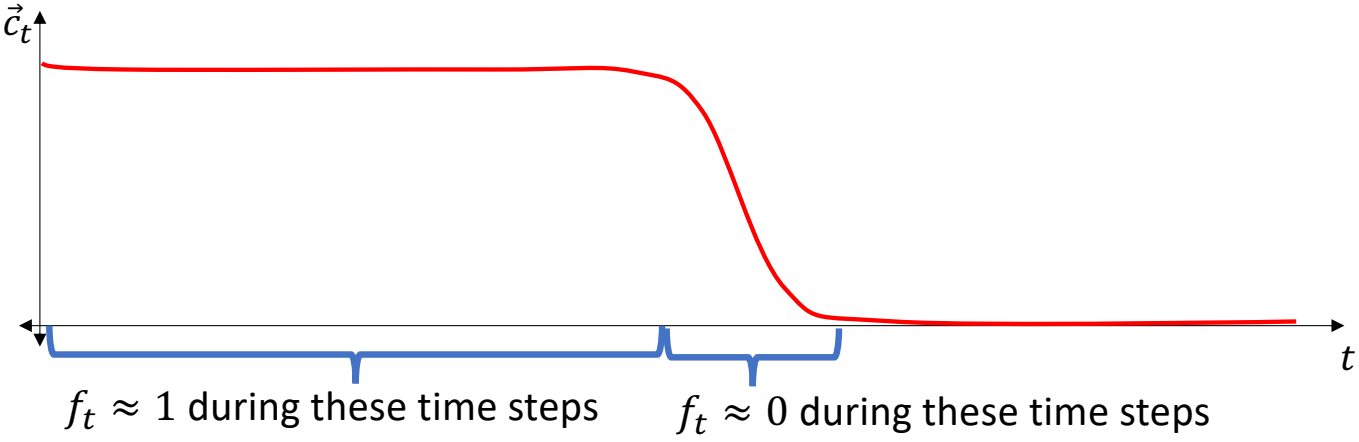
LSTM: Remember when you want to remember, forget when you want to forget

Remember that an RNN tends to forget exponentially, like this:



Exponential-decay.png. CC-SA-4.0, Svjo, 2017

An LSTM forgets more like this:



Outline

- Syntax and semantics
  - A grammar specifies which word sequences are valid sentences
  - Finite-depth CFG = Regular grammar = HMM
- Part of speech tagging
  - Open-class words: nouns, verbs, adverbs, adjectives, interjections
  - Closed-class words: prepositions, pronouns, conjunctions, determiners
- An HMM for POS tagging
  - State variable is the part of speech
  - Observation is the word
- The Viterbi algorithm
  - $\ln v_{j,t} = \max_i \left(\ln v_{i,t-1} + \ln a_{ij} + \ln b_{j,x_t} \right)$

Content

- Recurrent neural networks

$$\vec{h}_t = g\left(U\vec{h}_{t-1}, W\vec{x}_t\right)$$

- Training a recurrent neural network: Back-propagation through time (BPTT)

$$\frac{d\mathfrak{L}}{dh_{i,t}} = \frac{\partial\mathfrak{L}}{\partial h_{i,t}} + \frac{d\mathfrak{L}}{dh_{j,t+1}}\frac{\partial h_{j,t+1}}{\partial h_{i,t}}$$

- Avoid catastrophic forgetting: use Long short-term memory (LSTM)

$$\vec{c}_t = f_t\vec{c}_{t-1} + i_t\vec{x}_t$$