10-418 / 10-618 Machine Learning for Structured Data

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Learning to Search
+
Recurrent Neural Networks

Matt Gormley
Lecture 4
Sep. 9, 2019

Reminders

- **Homework 1: DAgger for seq2seq**
  - **Out: Mon, Sep. 09 (+/- 2 days)**
  - **Due: Mon, Sep. 23 at 11:59pm**

LEARNING TO SEARCH

Learning to Search

Whiteboard:

- Problem Setting
- Ex: POS Tagging
- Other Solutions:
  - Completely Independent Predictions
  - Sharing Parameters / Multi-task Learning
  - Graphical Models
- Today's Solution: Structured Prediction to Search
  - Search spaces
  - Cost functions
  - Policies

FEATURES FOR POS TAGGING

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag P as the tag for "like"

Weight of this feature is like log of an emission probability in an HMM

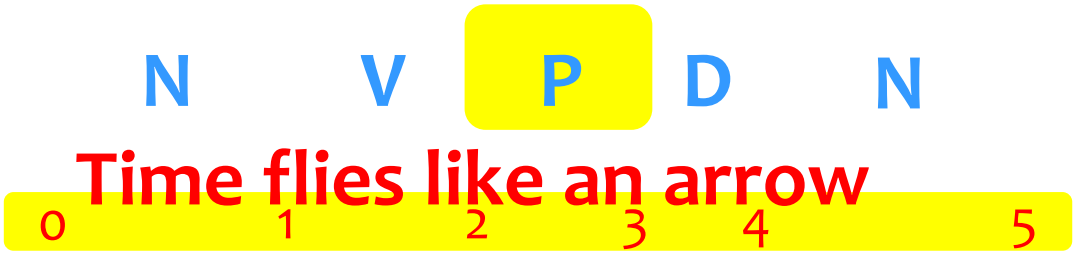
Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P

Features for tagging …



- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag P as the tag for "like"
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P

Weight of this feature is like log of a transition probability in an HMM

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”

Features for tagging …

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”
- Count of tag bigram V P where P is the tag for “like”

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”
- Count of tag bigram V P where P is the tag for “like”
- Count of tag bigram V P where both words are lowercase

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag trigram N V P?
  - A bigram tagger can only consider within-bigram features: only look at 2 adjacent blue tags (plus arbitrary red context).
  - So here we need a trigram tagger, which is slower.
  - The forward-backward states would remember *two* previous tags.

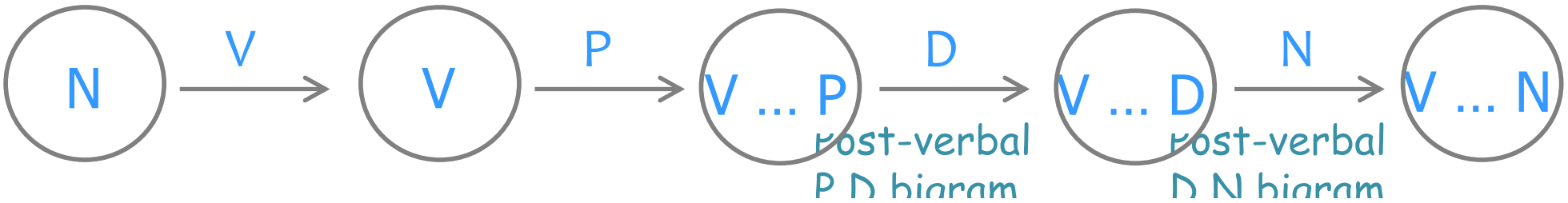
We take this arc once per N V P triple, so its weight is the total weight of the features that fire on that triple.

Features for tagging …

N V P D N

Time flies like an arrow

- Count of tag trigram N V P?
  - A bigram tagger can only consider within-bigram features: only look at 2 adjacent blue tags (plus arbitrary red context).
  - So here we need a trigram tagger, which is slower.
- Count of "post-verbal" nouns? ("discontinuous bigram" V N)
  - An n-gram tagger can only look at a narrow window.
  - Here we need a *fancier* model (finite state machine) whose states remember whether there was a verb in the left context.



How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).

For position i in a tagging, these might include:

- Full name of tag i
- First letter of tag i (will be "N" for both "NN" and "NNS")
- Full name of tag i-1 (possibly BOS); similarly tag i+1 (possibly EOS)
- Full name of word i
- Last 2 chars of word i (will be "ed" for most past-tense verbs)
- First 4 chars of word i (why would this help?)
- "Shape" of word i (lowercase/capitalized/all caps/numeric/…)
- Whether word i is part of a known city name listed in a "gazetteer"
- Whether word i appears in thesaurus entry e (one attribute per e)
- Whether i is in the middle third of the sentence

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

At each position of (x,y), exactly one of the many template7 features will fire:

N V P D N

Time flies like an arrow

At i=1, we see an instance of "template7=(BOS,N,-es)"
so we add one copy of that feature's weight to score(x,y)

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

At each position of (x,y), exactly one of the many template7 features will fire:

N V P D N

Time flies like an arrow

At i=2, we see an instance of "template7=(N,V,-ke)"
so we add one copy of that feature's weight to score(x,y)

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

At each position of (x,y), exactly one of the many template7 features will fire:

N V P D N

Time flies like an arrow

At i=3, we see an instance of "template7=(N,V,-an)"
so we add one copy of that feature's weight to score(x,y)

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

At each position of (x,y), exactly one of the many template7 features will fire:

At i=4, we see an instance of "template7=(P,D,-ow)"
so we add one copy of that feature's weight to score(x,y)

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

At each position of (x,y), exactly one of the many template7 features will fire:

N V P D N

Time flies like an arrow

At i=5, we see an instance of "template7=(D,N,-)"
so we add one copy of that feature's weight to score(x,y)

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

This template gives rise to *many* features, e.g.:

score(x,y) = …
+ θ["template7=(P,D,-ow)"] * count("template7=(P,D,-ow)")
+ θ["template7=(D,D,-xx)"] * count("template7=(D,D,-xx)")
+ …

With a handful of feature templates and a large vocabulary, you can easily end up with millions of features.

How might you come up with the features that you will use to score (x,y)?

1. Think of some attributes ("basic features") that you can compute at each position in (x,y).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be (tag(i-1), tag(i), suffix2(i+1)).

Note: Every template should mention at least some blue.

- Given an input x, a feature that only looks at red will contribute the same weight to score(x,y_1) and score(x,y_2).
- So it can't help you choose between outputs y_1, y_2.

LEARNING TO SEARCH

Learning to Search

***Whiteboard*:**

- Scoring functions for "Learning to Search"
- Learning to Search: a meta-algorithm
- Algorithm #1: Traditional Supervised Imitation Learning
- Algorithm #2: DAgger

DAgger Policy During Training

- DAgger assumes that we follow a **stochastic policy** that flips a weighted coin (with weight β_i at timestep i) to decide between the oracle policy and the model's policy

$$\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$$

- We require that (β_1, β_2, β_3, …) is chosen to be a sequence such that:

$$\overline{\beta}_N = \frac{1}{N}\sum_{i=1}^{N} \beta_i \to 0 \quad \text{as} \quad N \to \infty.$$

Q: What are examples of such sequences?

DAgger Theoretical Results

- The theory mirrors the intuition that **Exposure Bias is bad**
- The Supervised Approach to Imitation performs **not-so-well** even on the oracle (training time) distribution over states (i.e. quadratically number of mistakes grows **quadratically** in task horizon T and classification cost ϵ)
- DAgger yields an algorithm that performs **well** on the test-time distribution over states (i.e. number of mistakes grows **linearly** in task horizon T and classification cost ϵ)

$$J(\pi) = \sum_{t=1}^{T} \mathbb{E}_{s \sim d_\pi^t} [C_\pi(s)]$$

Algo #1: Supervised Approach to Imitation

Theorem 2.1. *(Ross and Bagnell, 2010) Let* $\mathbb{E}_{s \sim d_{\pi^*}} [\ell(s, \pi)] = \epsilon$, *then* $J(\pi) \leq J(\pi^*) + T^2 \epsilon$.

Algo #2: DAgger

Theorem 3.2. *For* DAGGER, *if* N *is* $\tilde{O}(uT)$ *there exists a policy* $\hat{\pi} \in \hat{\pi}_{1:N}$ *s.t.* $J(\hat{\pi}) \leq J(\pi^*) + uT\epsilon_N + O(1)$.

$$\epsilon_N = \min_{\pi \in \Pi} \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{s \sim d_{\pi_i}} [\ell(s, \pi)]$$

DAgger Theoretical Results

- The proof of the results for DAgger relies on a reduction to no-regret online learning

From Ross et al. (2011) "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning"...

sarial fashion over time. A no-regret algorithm is an algorithm that produces a sequence of policies $\pi_1, \pi_2, \ldots, \pi_N$ such that the average regret with respect to the best policy in hindsight goes to 0 as N goes to ∞:

$$\frac{1}{N}\sum_{i=1}^{N}\ell_i(\pi_i) - \min_{\pi\in\Pi}\frac{1}{N}\sum_{i=1}^{N}\ell_i(\pi) \leq \gamma_N \qquad (3)$$

for $\lim_{N\to\infty}\gamma_N = 0$. Many no-regret algorithms guarantee that γ_N is $\tilde{O}(\frac{1}{N})$ (e.g. when ℓ is strongly convex) (Hazan et al., 2006; Kakade and Shalev-Shwartz, 2008; Kakade and Tewari, 2009).

- The key idea is to choose the loss function to be that of the loss on the distribution over states given by the current policy chosen by the online learner

$$\ell_i(\pi) = \mathbb{E}_{s\sim d_{\pi_i}}[\ell(s,\pi)]$$

LEARNING TO SEARCH: EMPIRICAL RESULTS

Dagger for Mario Tux Cart

Video from Stéphane Ross (https://www.youtube.com/watch?v=V0onpNnWzSU)

Experiments: Vowpal Wabbit L2S

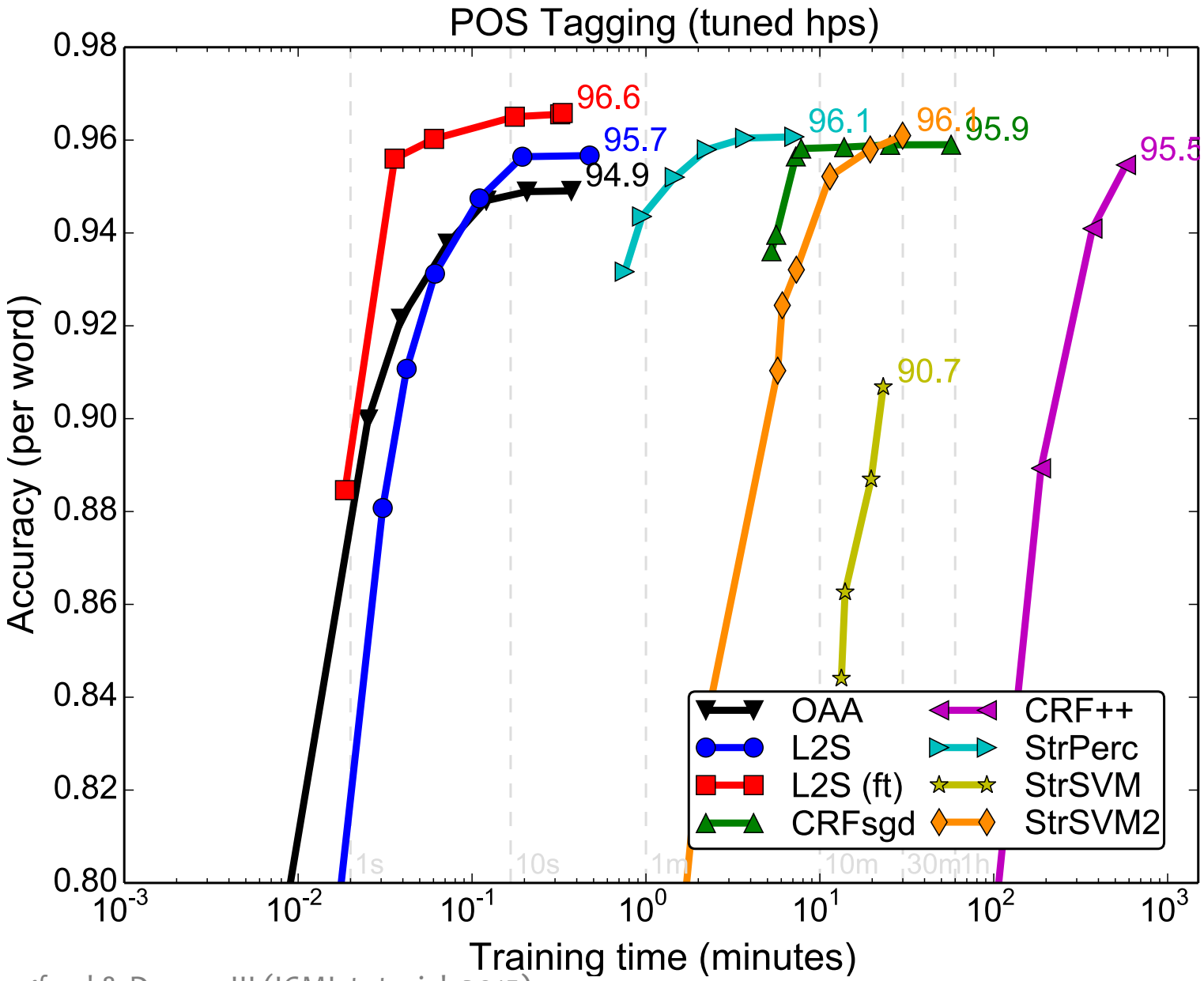


Figure from Langford & Daume III (ICML tutorial, 2015)

Experiments: Vowpal Wabbit L2S

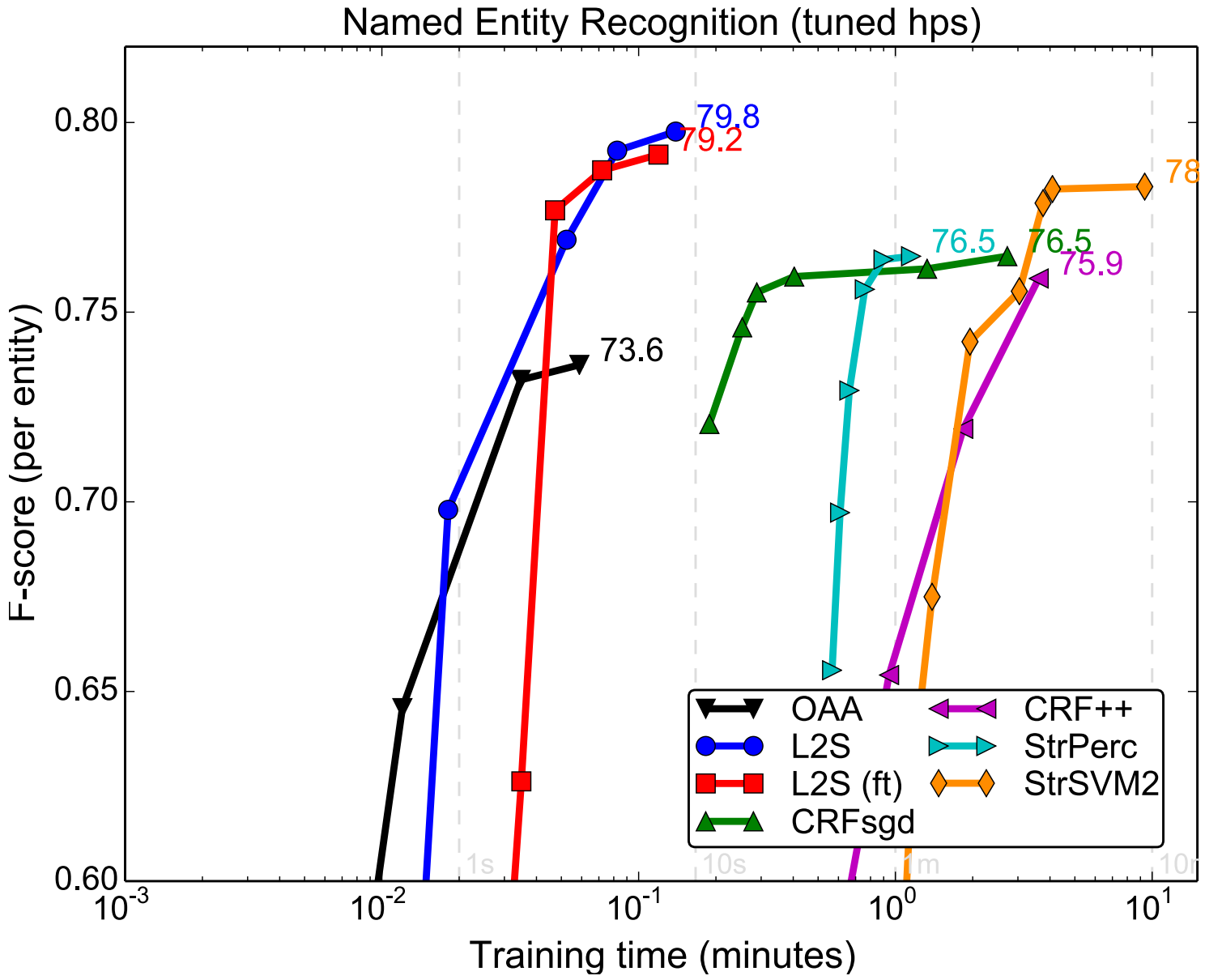


Figure from Langford & Daume III (ICML tutorial, 2015)

Experiments: Vowpal Wabbit L2S

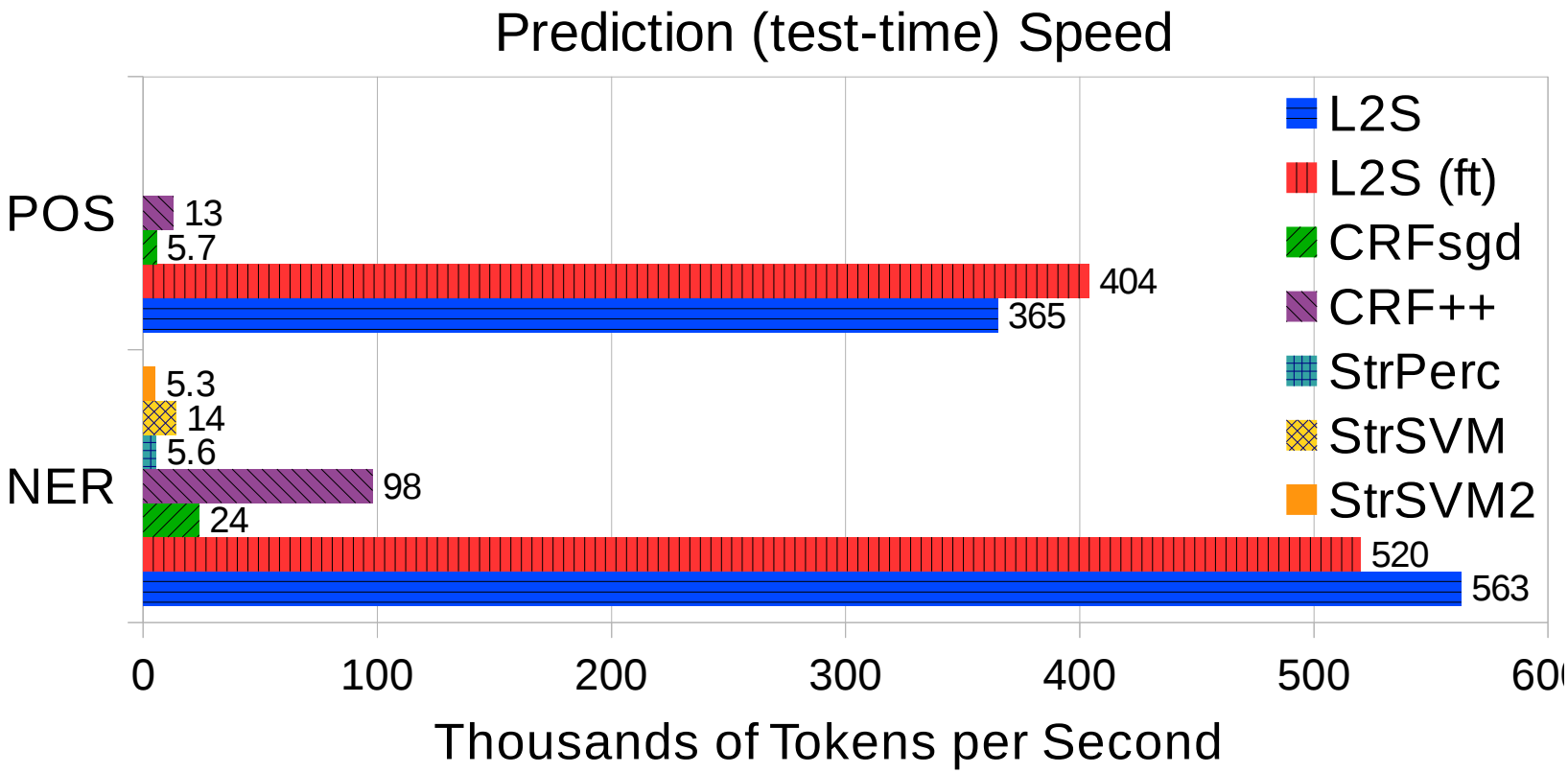


Figure from Langford & Daume III (ICML tutorial, 2015)

Learning 2 Search

Some key challenges:

- performance depends heavily on search order, but have to pick this by hand
- reference policy is critical, but what if it's too difficult to design one
- not always easy to make efficient on a GPU

Adapted from Langford & Daume III (ICML tutorial, 2015)

Learning Objectives

Structured Prediction as Search

You should be able to…

1. Reduce a structured prediction problem to a search problem
2. Implement Dagger, a learning to search algorithm
3. (If you already know RL…) Contrast imitation learning with reinforcement learning
4. Explain the reduction of structured prediction to no-regret online learning
5. Contrast various learning2search algorithms based on their properties

SEQ2SEQ: OVERVIEW

Why seq2seq?

- **~10 years ago:** state-of-the-art machine translation or speech recognition systems were complex pipelines
  - MT
    - unsupervised word-level alignment of sentence-parallel corpora (e.g. via GIZA++)
    - build phrase tables based on (noisily) aligned data (use prefix trees and on demand loading to reduce memory demands)
    - use factored representation of each token (word, POS tag, lemma, morphology)
    - learn a separate language model (e.g. SRILM) for target
    - combine language model with phrase-based decoder
    - tuning via minimum error rate training (MERT)
  - ASR
    - MFCC and PLP feature extraction
    - acoustic model based on Gaussian Mixture Models (GMMs)
    - model phones via Hidden Markov Models (HMMs)
    - learn a separate n-gram language model
    - learn a phonetic model (i.e. mapping words to phones)
    - combine language model, acoustic model, and phonetic model in a weighted finite-state transducer (WFST) framework (e.g. OpenFST)
    - decode from a confusion network (lattice)
- **Today:** just use a seq2seq model
  - *encoder*: reads the input one token at a time to build up its vector representation
  - *decoder*: starts with encoder vector as context, then decodes one token at a time – feeding its own outputs back in to maintain a vector representation of what was produced so far

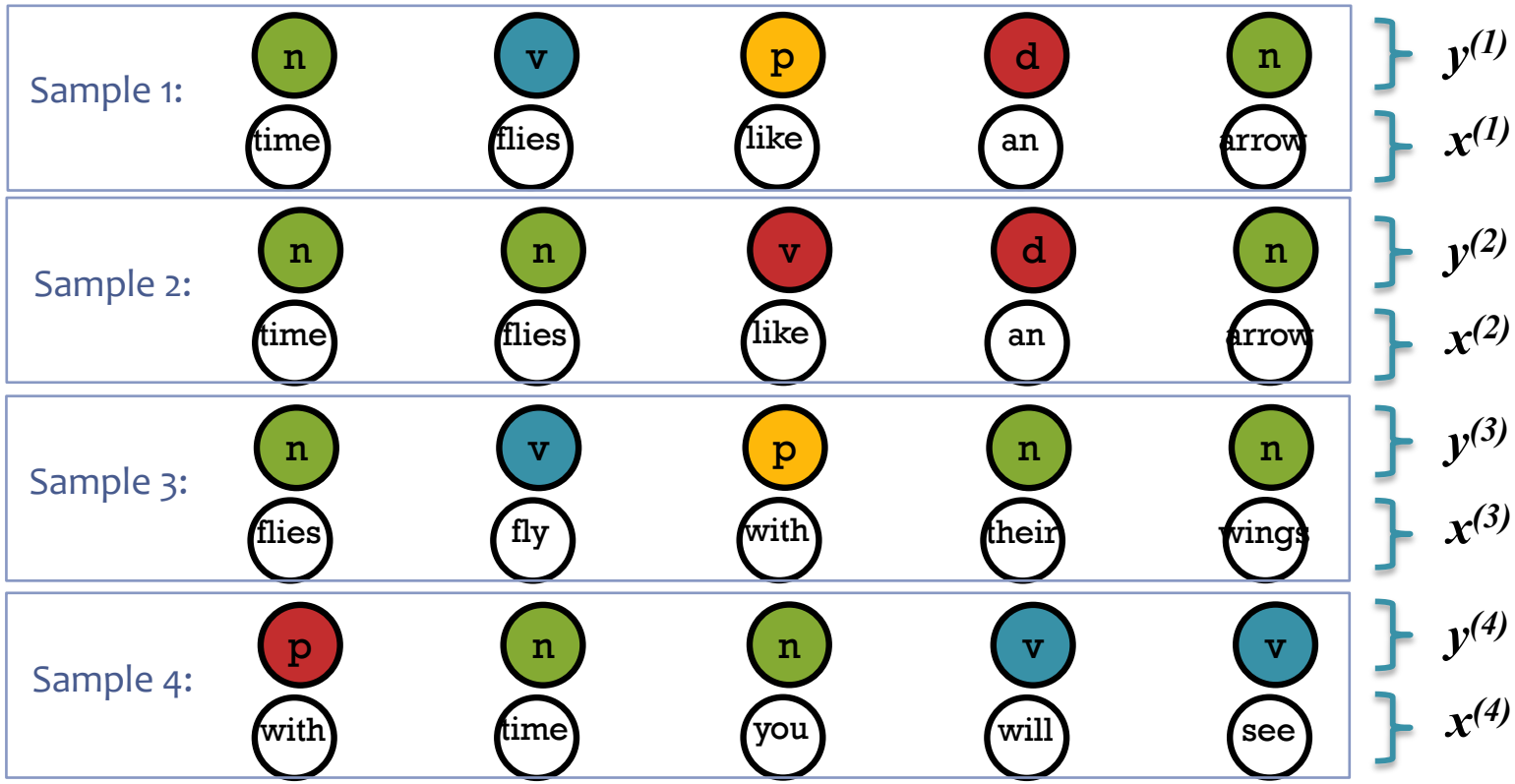
Outline

- Recurrent Neural Networks
  - Elman network
  - Backpropagation through time (BPTT)
  - Parameter tying
  - bidirectional RNN
  - Vanishing gradients
  - LSTM cell
  - Deep RNNs
  - Training tricks: mini-batching with masking, sorting into buckets of similar-length sequences, truncated BPTT
- RNN Language Models
  - Definition: language modeling
  - n-gram language model
  - RNNLM
- Sequence-to-sequence (seq2seq) models
  - encoder-decoder architectures
  - Example: biLSTM + RNNLM
  - Example: machine translation
  - Example: speech recognition
  - Example: image captioning
- Learning to Search for seq2seq
  - DAgger for seq2seq
  - Scheduled Sampling (a special case of DAgger)

RECURRENT NEURAL NETWORKS

Dataset for Supervised Part-of-Speech (POS) Tagging

Data: $\mathcal{D} = \{\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\}_{n=1}^{N}$



Dataset for Supervised Handwriting Recognition

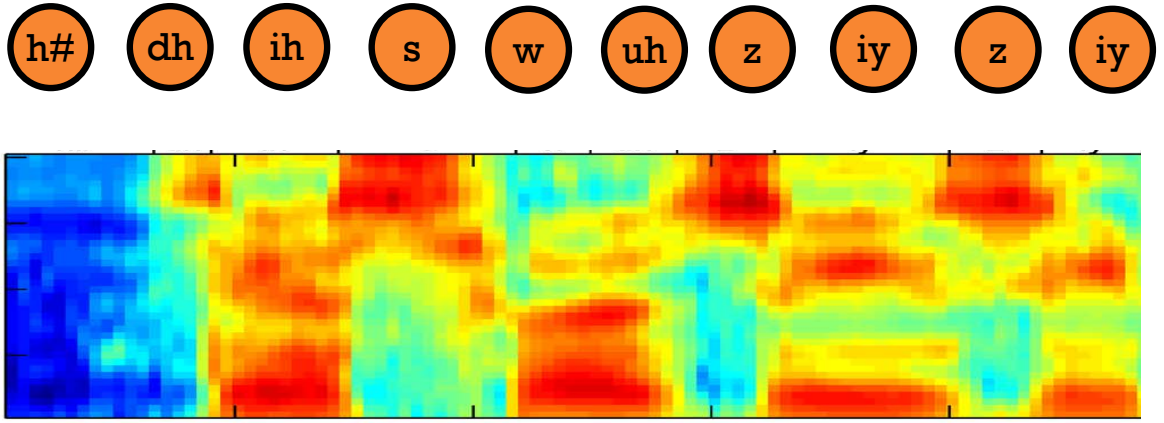
Data: $\mathcal{D} = \{\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\}_{n=1}^{N}$

Figures from (Chatzis & Demiris, 2013)

Dataset for Supervised Phoneme (Speech) Recognition

Data: $\mathcal{D} = \{\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\}_{n=1}^{N}$

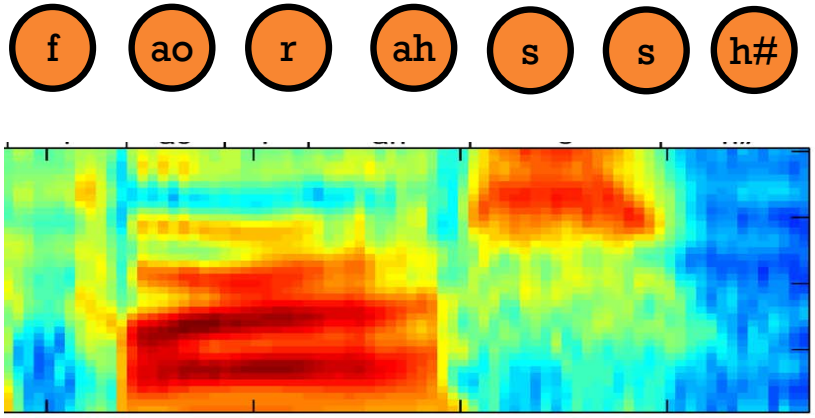
Sample 1:



$\boldsymbol{y}^{(1)}$

$\boldsymbol{x}^{(1)}$

Sample 2:



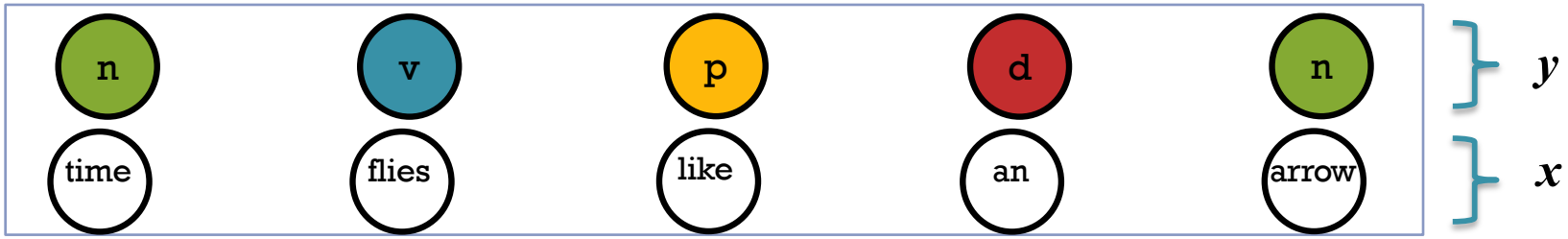
$\boldsymbol{y}^{(2)}$

$\boldsymbol{x}^{(2)}$

Figures from (Jansen & Niyogi, 2013)

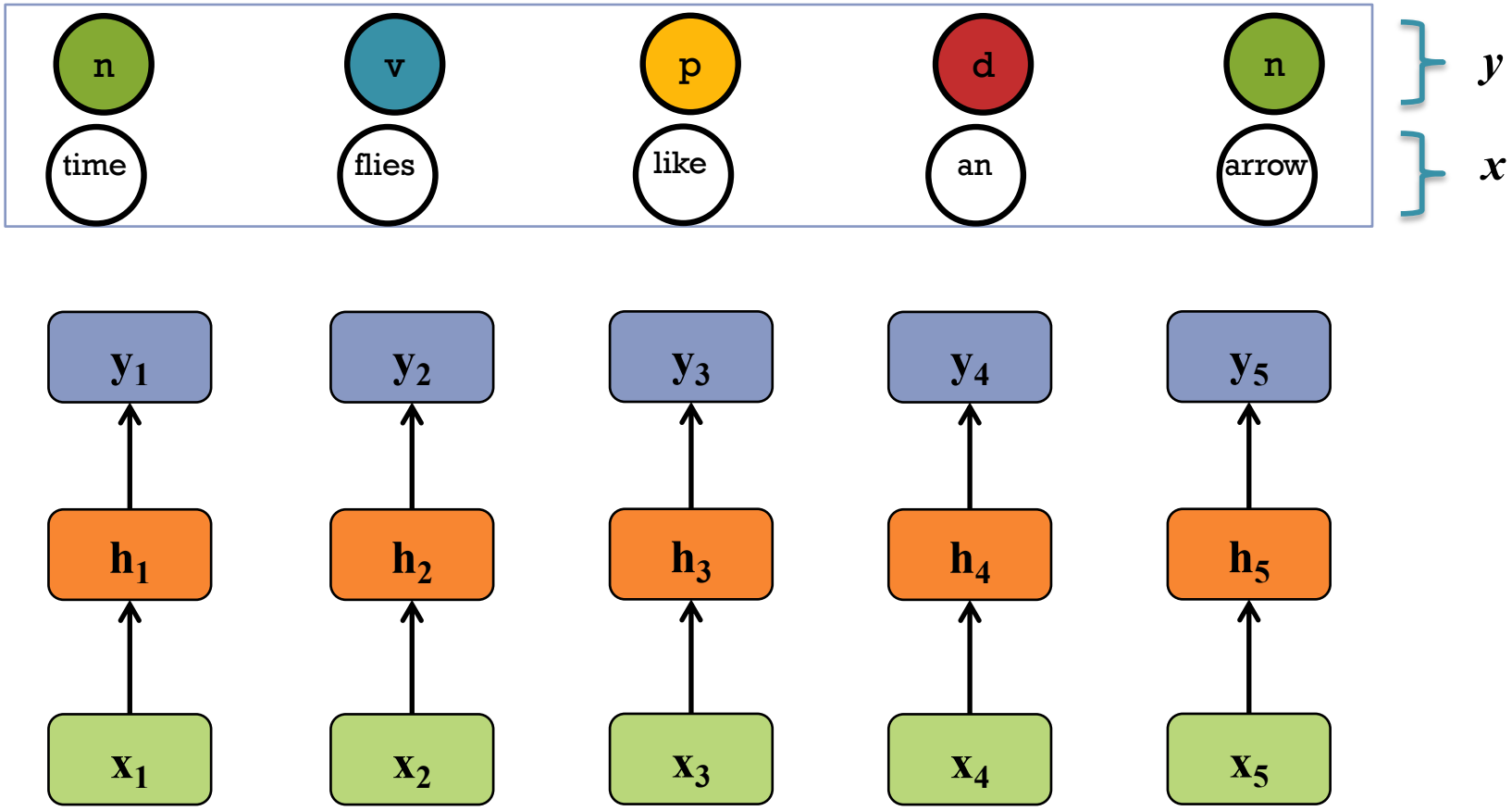
Time Series Data

Question 1: How could we apply the neural networks we've seen so far (which expect **fixed size input/output**) to a prediction task with **variable length input/output**?



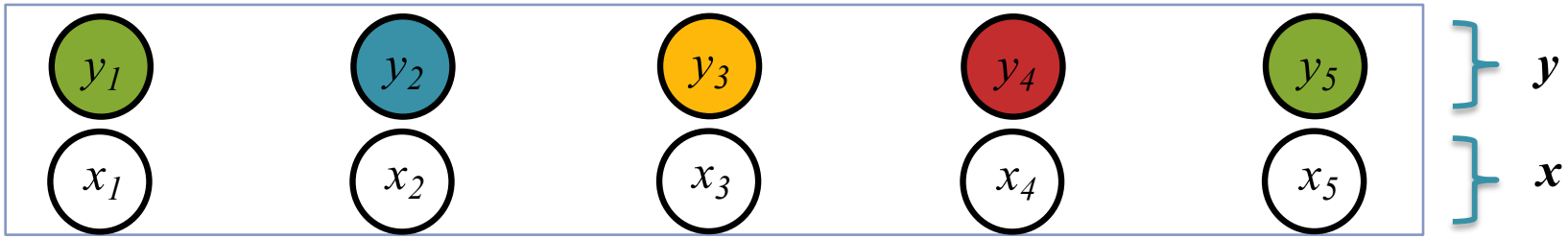
Time Series Data

Question 1: How could we apply the neural networks we've seen so far (which expect **fixed size input/output**) to a prediction task with **variable length input/output**?



Time Series Data

Question 2: How could we incorporate context (e.g. words to the left/right, or tags to the left/right) into our solution?



Multiple Choice:

Working left-to-right, use features of…

	x_{i-1}	x_i	x_{i+1}	y_{i-1}	y_i	y_{i+1}
A	✓					
B				✓		
C	✓			✓		
D	✓			✓	✓	✓
E	✓	✓		✓	✓	✓
F	✓	✓	✓	✓		
G	✓	✓	✓	✓	✓	
H	✓	✓	✓	✓	✓	✓

Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$

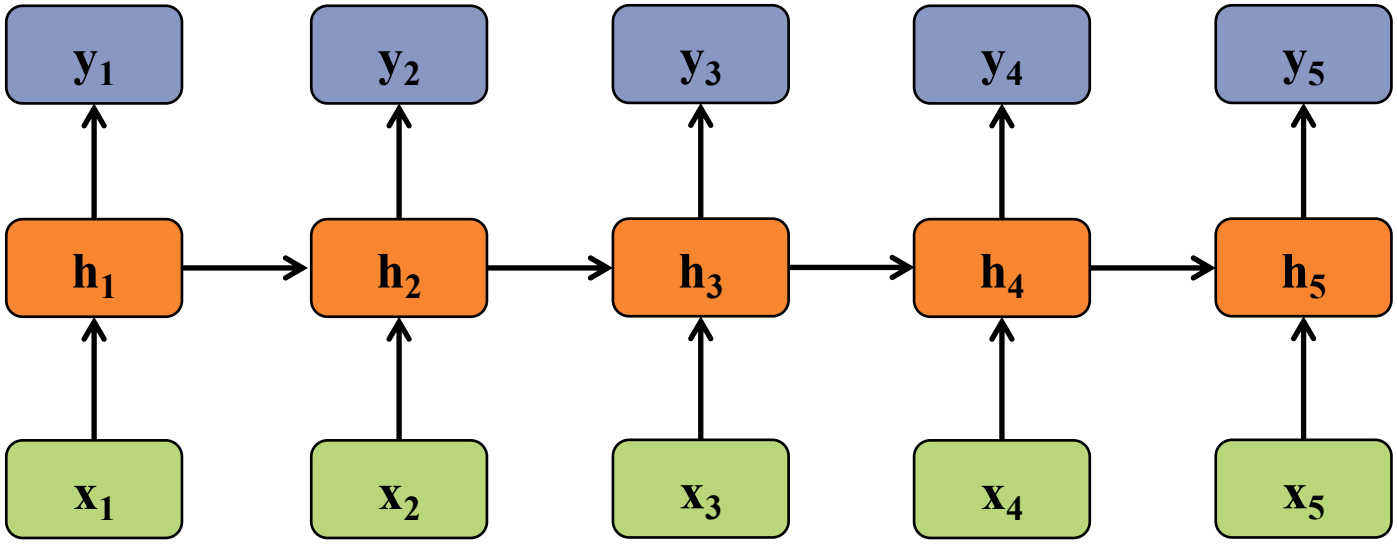
outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: $\mathcal{H}$

Definition of the RNN:

$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$y_t = W_{hy}h_t + b_y$$



Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$

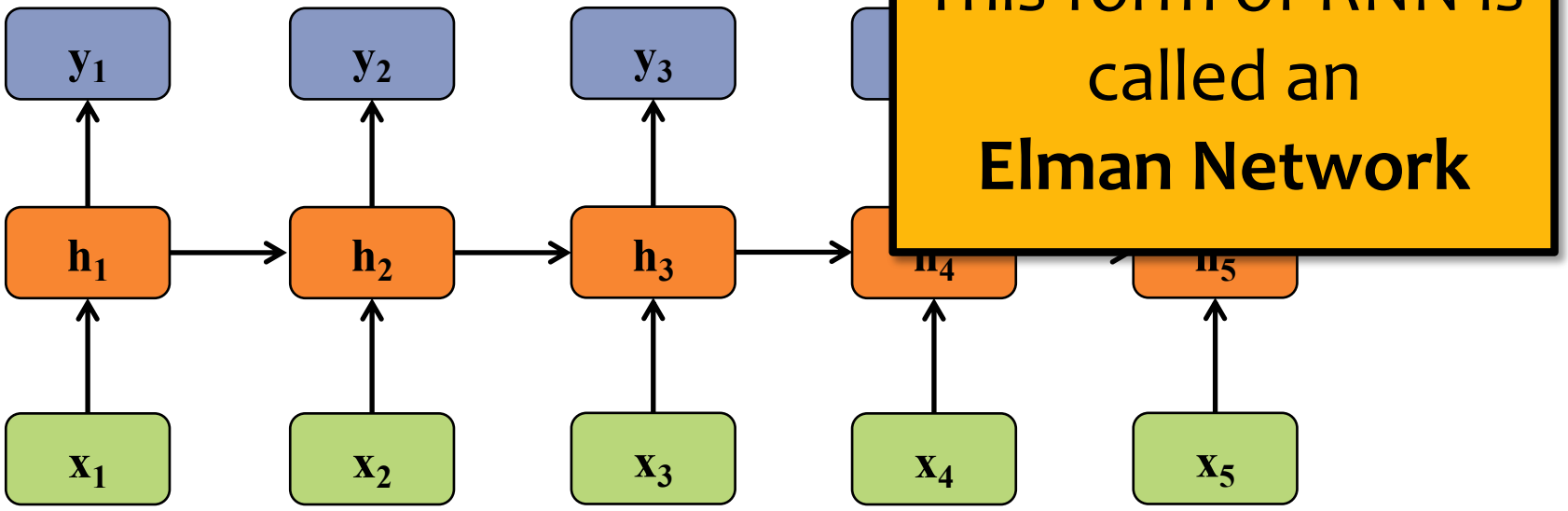
outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: $\mathcal{H}$

Definition of the RNN:

$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

$$y_t = W_{hy}h_t + b_y$$

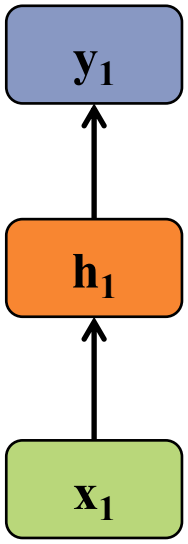


Recurrent Neural Networks (RNNs)

$$\text{inputs: } \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$$
$$\text{hidden units: } \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$$
$$\text{outputs: } \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$$
$$\text{nonlinearity: } \mathcal{H}$$

Definition of the RNN:

$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$
$$y_t = W_{hy}h_t + b_y$$



- If *T=1*, then we have a standard feed-forward **neural net with one hidden layer**
- All of the deep nets from last lecture required **fixed size inputs/outputs**

A Recipe for Machine Learning

1. Given training data:

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

2. Choose each of these:

- Decision function

$$\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

- Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

4. Train with SGD:

(take small steps opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

A Recipe for Machine Learning

Background

- Recurrent Neural Networks (RNNs) provide another form of **decision function**
- An RNN is just another differential function

2. Choose each of these:
   - Decision function

$$\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

4. Train with SGD:
(take small steps opposite the gradient)

- We'll just need a method of computing the gradient efficiently
- Let's use Backpropagation Through Time...

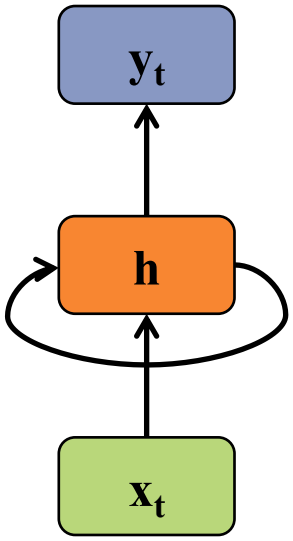
$$\eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Recurrent Neural Networks (RNNs)

$$\text{inputs: } \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$$
$$\text{hidden units: } \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$$
$$\text{outputs: } \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$$
$$\text{nonlinearity: } \mathcal{H}$$

Definition of the RNN:

$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$
$$y_t = W_{hy}h_t + b_y$$



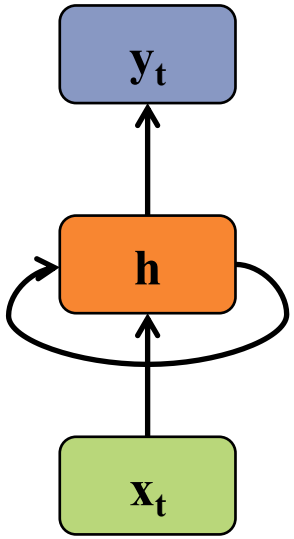
Recurrent Neural Networks (RNNs)

$$\text{inputs: } \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$$
$$\text{hidden units: } \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$$
$$\text{outputs: } \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$$
$$\text{nonlinearity: } \mathcal{H}$$

Definition of the RNN:

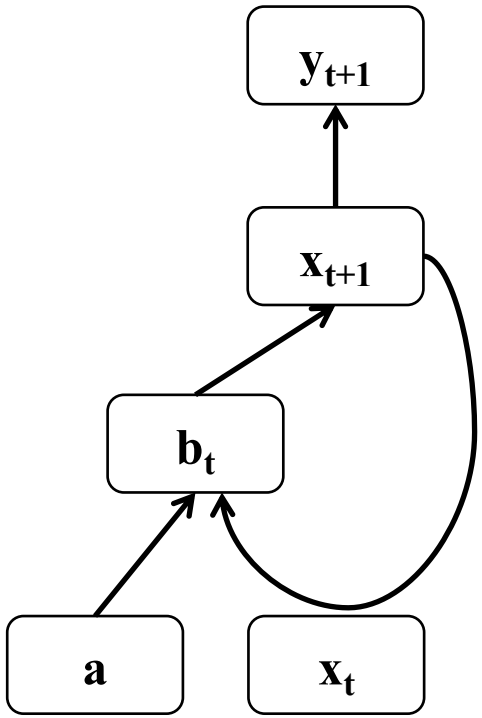
$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$
$$y_t = W_{hy}h_t + b_y$$

- By unrolling the RNN through time, we can **share parameters** and accommodate **arbitrary length** input/output pairs
- Applications: **time-series data** such as sentences, speech, stock-market, signal data, etc.



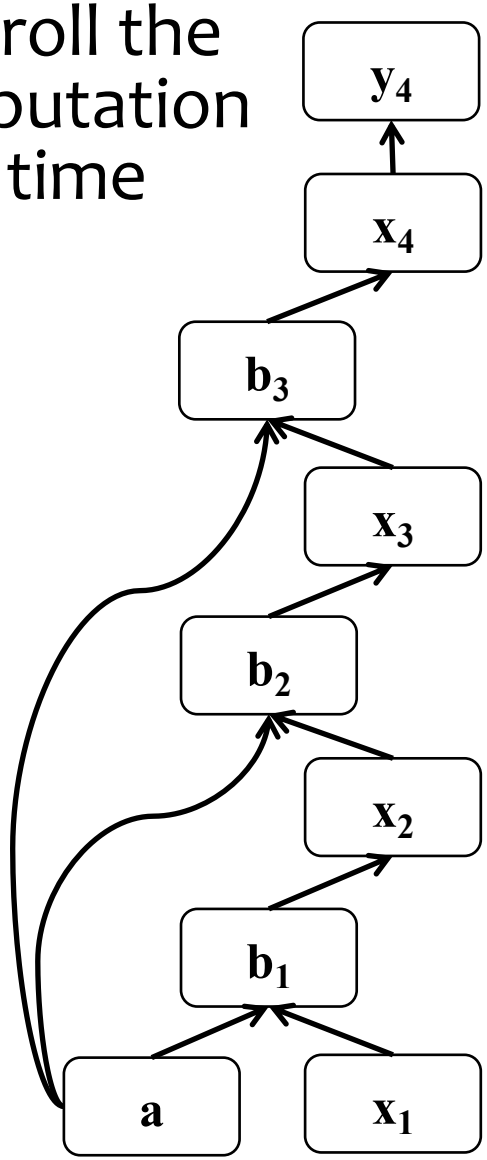
Background: Backprop through time

Recurrent neural network:



BPTT:

1. Unroll the computation over time



2. Run backprop through the resulting feed-forward network

(Robinson & Fallside, 1987)
(Werbos, 1988)
(Mozer, 1995)

Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

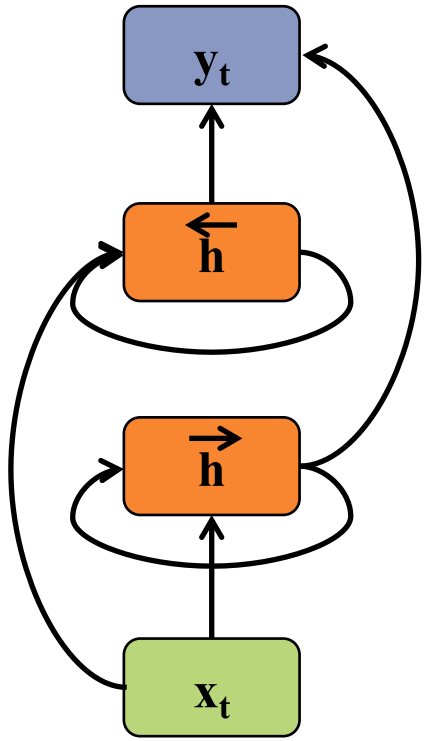
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

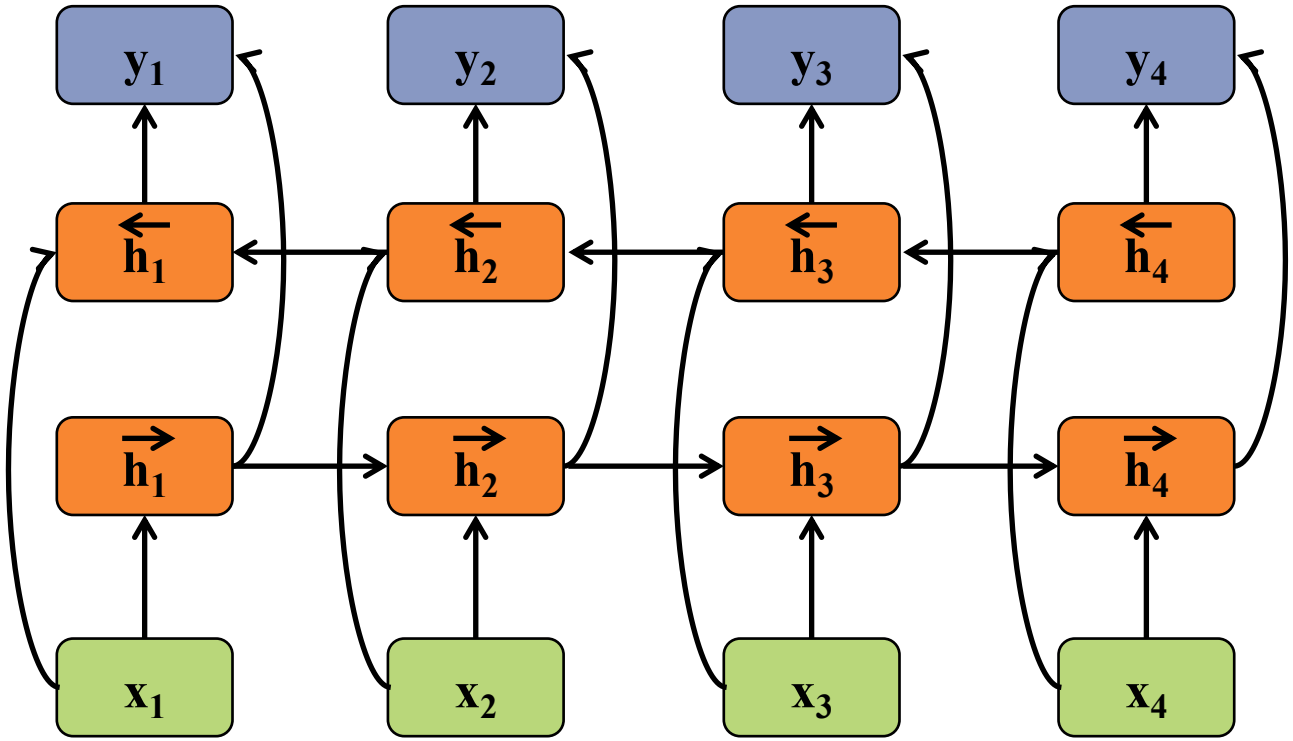
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

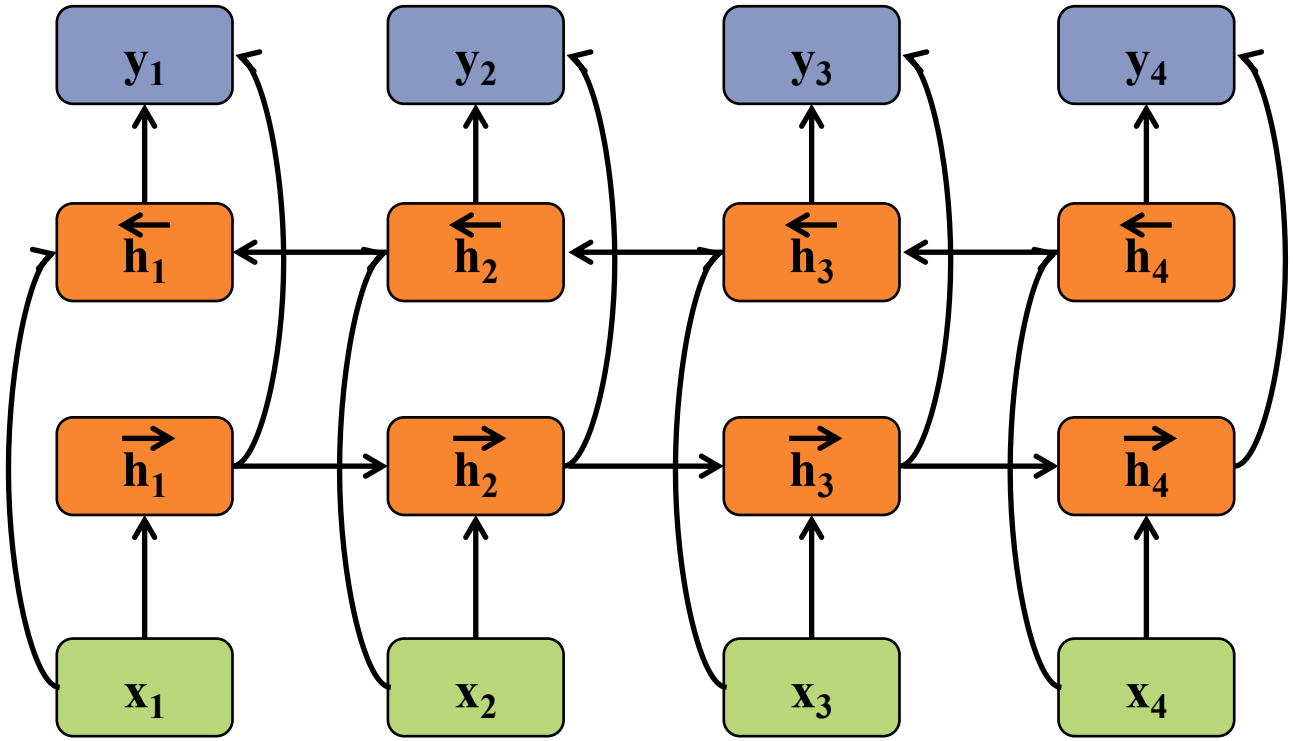
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Bidirectional RNN

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\overrightarrow{\mathbf{h}}$ and $\overleftarrow{\mathbf{h}}$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

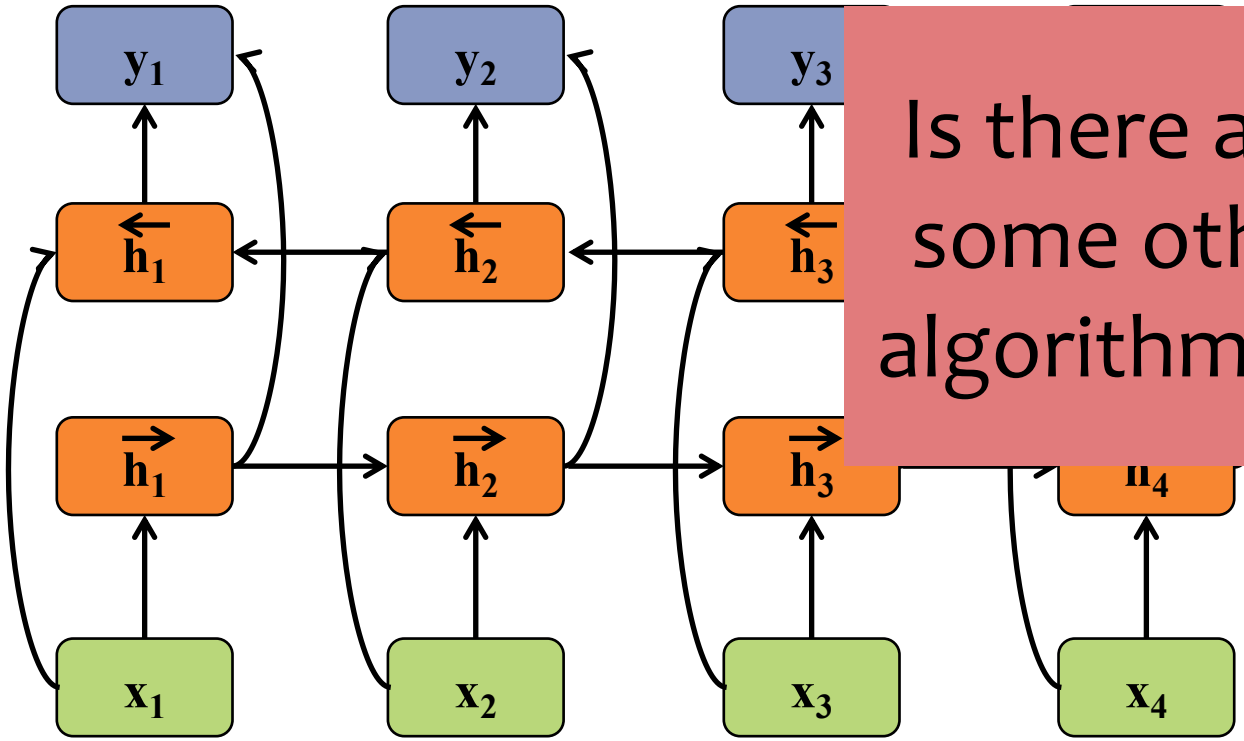
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$



Is there an analogy to some other recursive algorithm(s) we know?

Deep RNNs

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: $\mathcal{H}$

Recursive Definition:

$$h_t^n = \mathcal{H}\left(W_{h^{n-1}h^n} h_t^{n-1} + W_{h^n h^n} h_{t-1}^n + b_h^n\right)$$

$$y_t = W_{h^N y} h_t^N + b_y$$

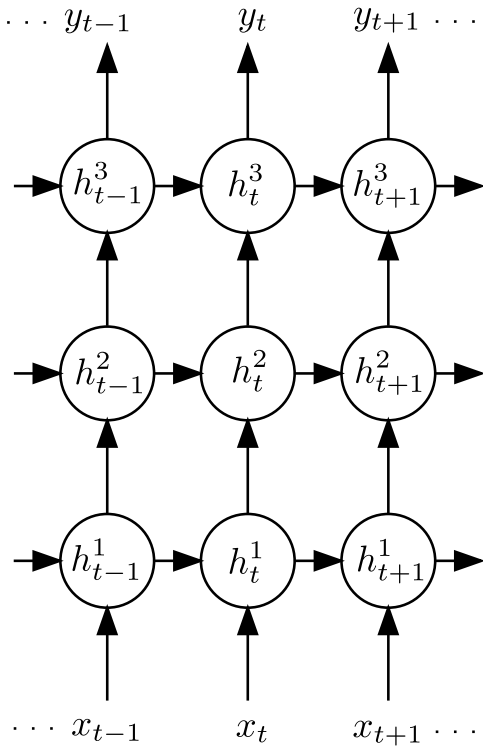


Figure from (Graves et al., 2013)

Deep Bidirectional RNNs

$$\begin{aligned}\text{inputs: } & \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \\ \text{outputs: } & \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \\ \text{nonlinearity: } & \mathcal{H}\end{aligned}$$

- Notice that the upper level hidden units have input from **two previous layers** (i.e. wider input)
- Likewise for the output layer
- What analogy can we draw to DNNs, DBNs, DBMs?

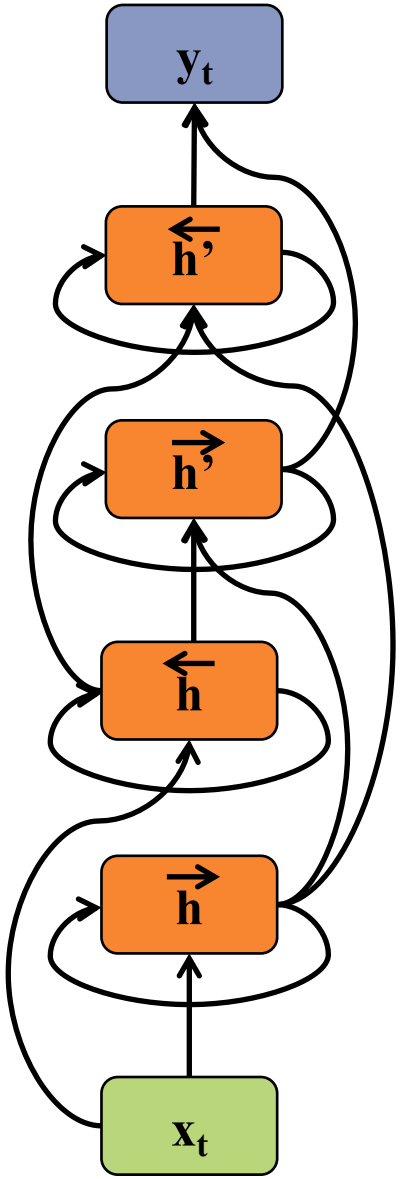


Figure from (Graves et al., 2013)