From features to neural networks

CS 690N, Spring 2018

Advanced Natural Language Processing

http://people.cs.umass.edu/~brenocon/anlp2018/

Brendan O'Connor

College of Information and Computer Sciences

University of Massachusetts Amherst

MaxEnt / Log-Linear models

- **x**: input (all previous words)
- **y**: output (next word)
- **f(x,y)** => R^d feature function [[domain knowledge here!]]
- **v**: R^d parameter vector (weights)

$$p(y|x;v) = \frac{\exp\left(v \cdot f(x,y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x,y')\right)}$$

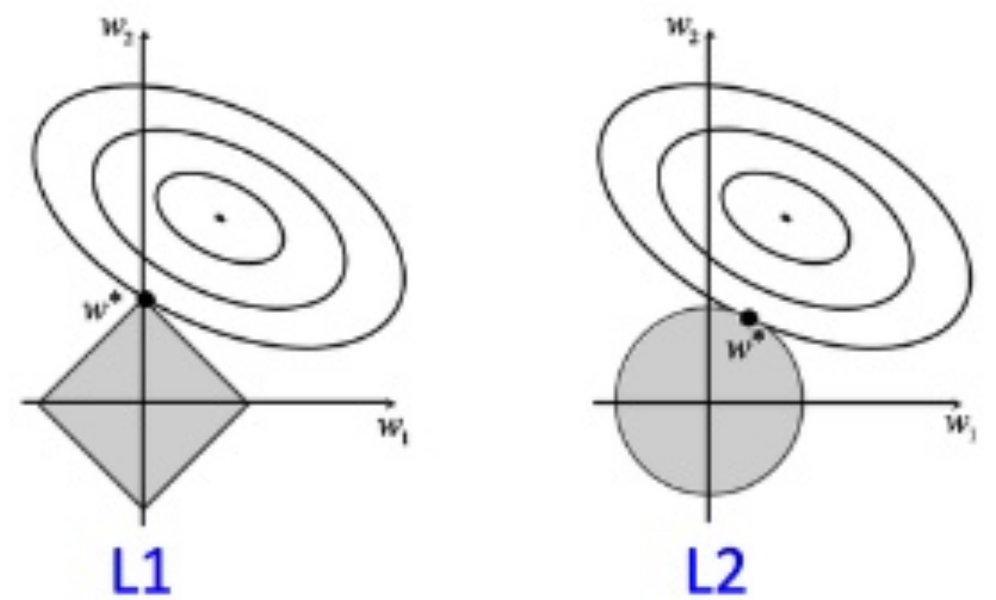
Application to history-based LM:

$$\begin{aligned} P(w_1..w_T) &= \prod_t P(w_t \mid w_1..w_{t-1}) \\ &= \prod_t \frac{\exp(v \cdot f(w_1..w_{t-1}, w_t))}{\sum_{w \in \mathcal{V}} \exp(v \cdot f(w_1..w_{t-1}, w))} \end{aligned}$$

Feature selection

- Offline feature selection
  - Count cutoffs: computational, not performance benefits
  - Predictive value: mutual info. / info. gain / chi-square
- Jointly learning for feature selection via L1 regularization: encourages θ sparsity

$$\min_{\theta} \; -\log p_\theta(y|x) + \lambda \sum_j |\theta_j|$$



- L1 optimization: convex but nonsmooth; requires subgradient methods (e.g. OWL-QN: variant of LBFGS. Available in *LibLBFGS*)

Too many features

- Millions to billions of features: performance often keeps improving!
- Engineering issue: feature name=>number mapping
- Feature selection ... mixed results
    - Count cutoffs: great computational benefits; typically not for performance
    - Features seen only once at training time typically help (!), or even features not seen at training time
    - Predictive value: mutual info. / info. gain / chi-square
    - L1 regularization: encourages θ sparsity, but not always better than L2
        - [structured sparsity more interesting: Yogatama, Martins tutorial]
    - Personal opinion: feature-based models just want a high diversity of weak signals

Feature hashing

- Feature hashing: make e.g. N(u,v,w) mapping random with collisions (!) *(Weinberger et al. 2009)*
  - Accuracy loss low since collisions are rare (since features are sparse). Works well, great for large-scale data (memory usage constant!)
  - Practically: use a fast string hashing function (e.g. murmurhash or Python's internal one)
- This is a type of *randomized projection* Ax. Typically not better than the original representation.
  - Instead of randomized embeddings, better generalization from learning them

Dense linear representations

- Feature hashing as dense representation

$$\mathrm{x} \xrightarrow{\text{A (fixed)}} \mathrm{z} \xrightarrow{\text{B (learned)}} \mathrm{y}$$

$P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}})$

A (learned) B (learned)

Dense linear representations

- Feature hashing as dense representation

$$x \xrightarrow{\text{A (fixed)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}})$$

- Saul and Pereira 1997 as dense representation

$$x \xrightarrow{\text{A (learned)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) = A_{w_{prev}} \cdot B_{w_{next}}$$

Dense linear representations

- Feature hashing as dense representation

$$x \xrightarrow{\text{A (fixed)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}})$$

- Saul and Pereira 1997 as dense representation

$$x \xrightarrow{\text{A (learned)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) = A_{w_{prev}} \cdot B_{w_{next}}$$

- Mnih and Hinton 2007: log-bilinear model
[related: *word2vec*, Mikolov et al.]

Dense linear representations

- Feature hashing as dense representation

$$x \xrightarrow{\text{A (fixed)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}})$$

- Saul and Pereira 1997 as dense representation

$$x \xrightarrow{\text{A (learned)}} z \xrightarrow{\text{B (learned)}} y$$

$$P(w_{next} \mid w_{prev}) = A_{w_{prev}} \cdot B_{w_{next}}$$

- Mnih and Hinton 2007: log-bilinear model [related: *word2vec*, Mikolov et al.]

$$P(w_{next} \mid w_{prev}) \propto \exp(A_{w_{prev}} \cdot B_{w_{next}})$$

  - Learn with gradient descent
  - Unlike S+P: A,B don't have to be on simplex
  - (this is simplified from their version)

Neural networks

- Idea: learn distributed representations of concepts
  - Nonlinear functions seem to help
- Multilayer perceptron: http://playground.tensorflow.org/

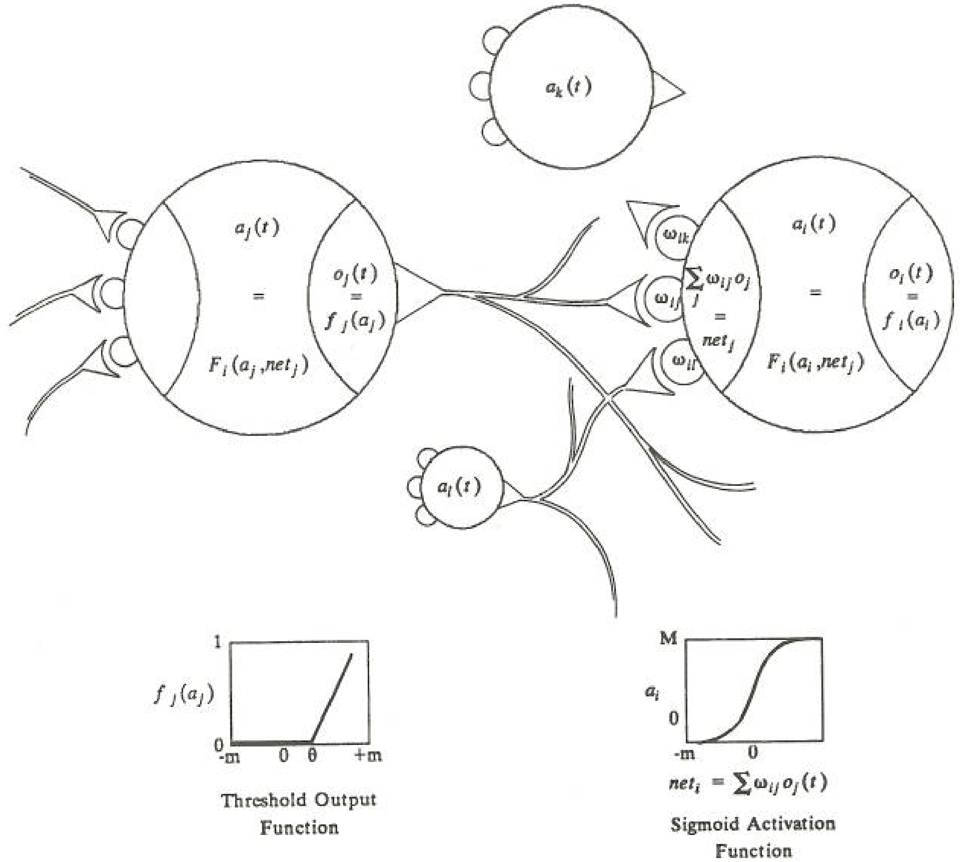
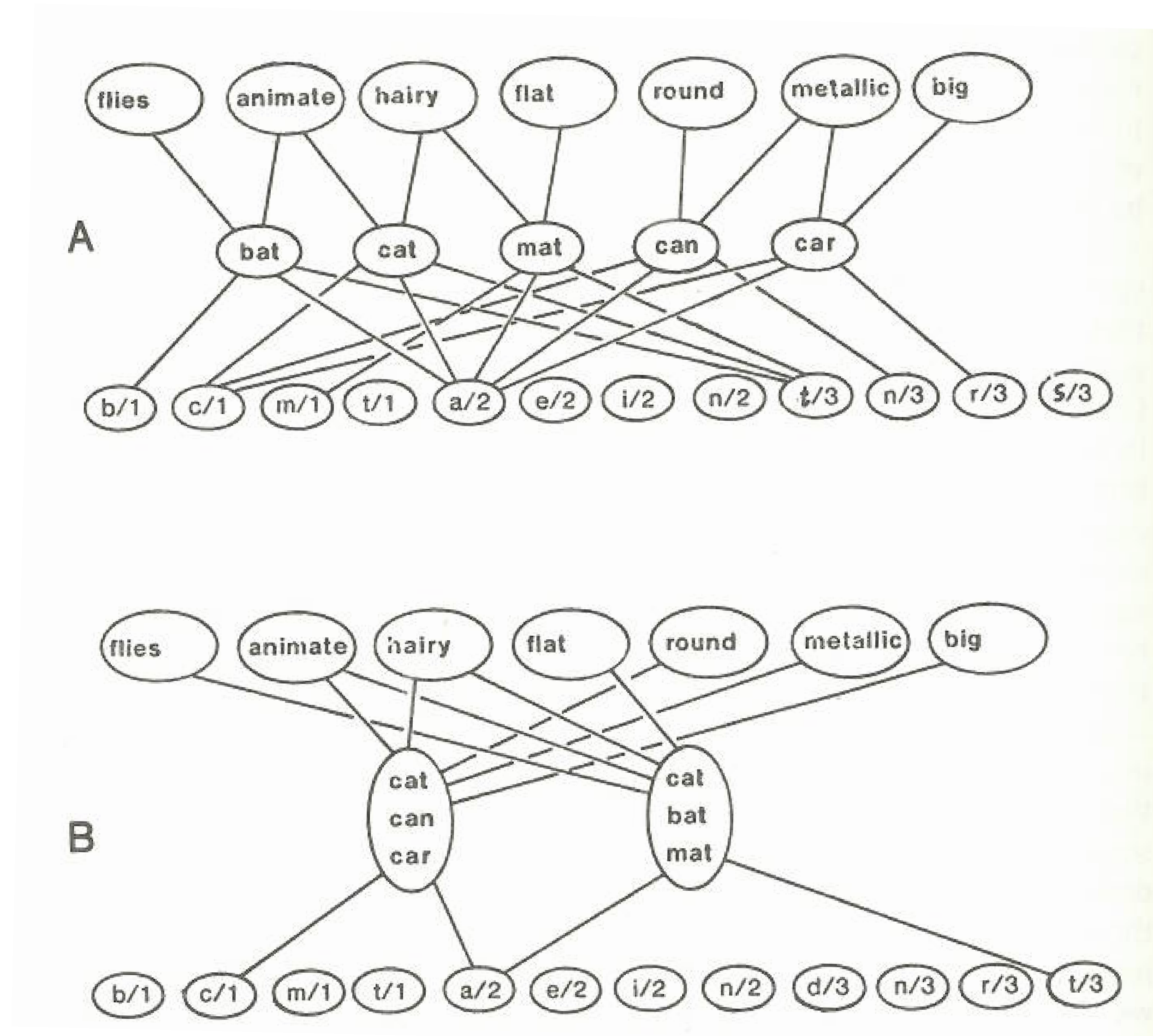


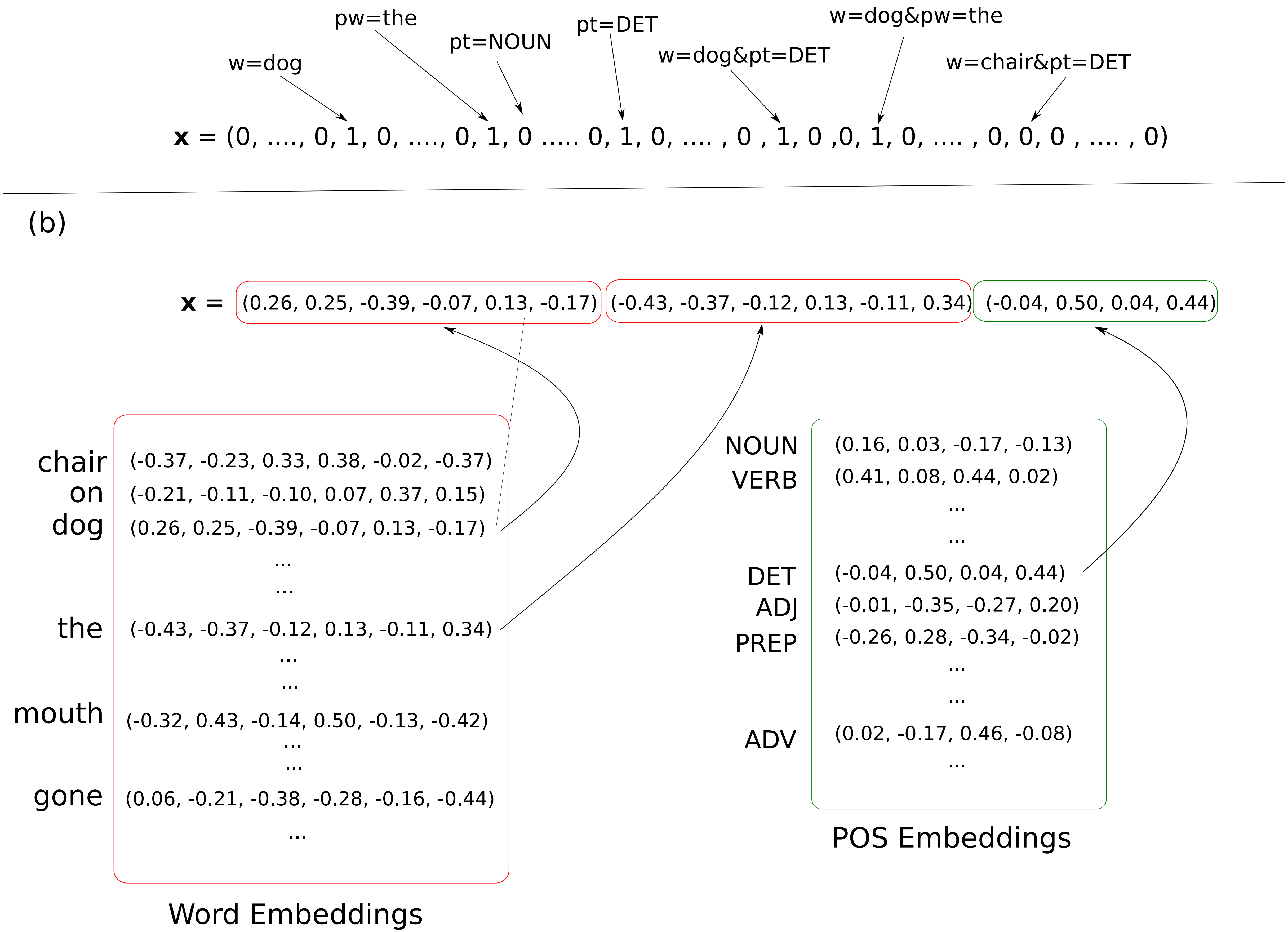
FIGURE 1. The basic components of a parallel distributed processing system.



[Diagrams from: Rumelhart and McClelland (ed.) 1986, *Parallel Distributed Processing*]

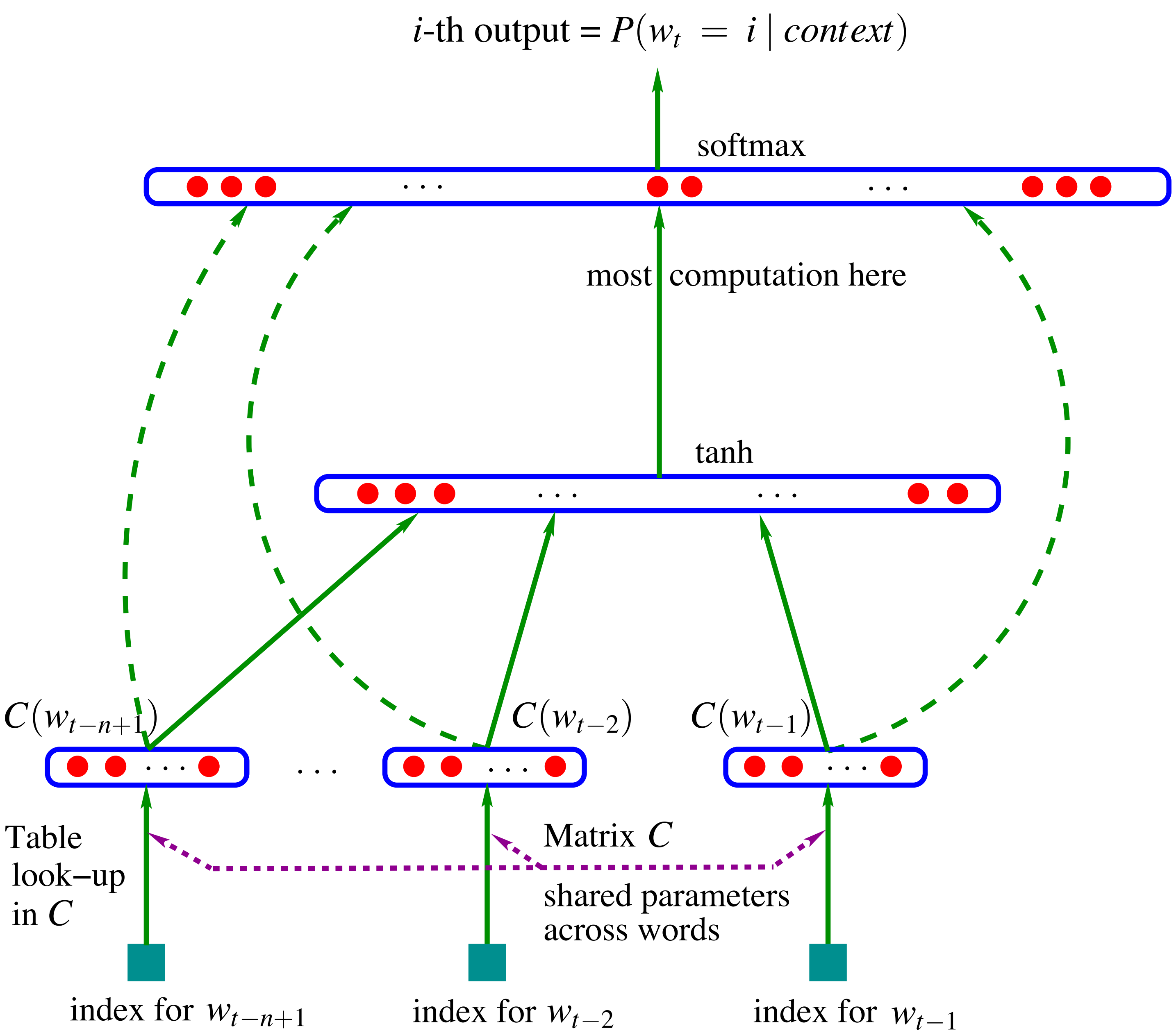
Neural networks in NLP

- Text representation: real-valued vectors
  - Word embeddings ... {character, phrase, part-of-speech tag, entity, relation} embeddings ...
- Probability model (e.g. p(y|x))
  - Output: logistic/softmax (like log-linear), but
  - "Squash network" nonlinear combination of the input. e.g. multilayer perceptron / feedforward network
- *Learn* both word embeddings and how to combine them as parameters.
  - Hopefully learn interesting high-level or fine-grained features of language, and how they interact


w=dog
pw=the
pt=NOUN
pt=DET
w=dog&pt=DET
w=dog&pw=the
w=chair&pt=DET
x = (0, ..., 0, 1, 0, ..., 0, 1, 0 ..... 0, 1, 0, ... , 0 , 1, 0 ,0, 1, 0, ... , 0, 0, 0 , ... , 0)
(b)
x = (0.26, 0.25, -0.39, -0.07, 0.13, -0.17) (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34) (-0.04, 0.50, 0.04, 0.44)
chair (-0.37, -0.23, 0.33, 0.38, -0.02, -0.37)
on (-0.21, -0.11, -0.10, 0.07, 0.37, 0.15)
dog (0.26, 0.25, -0.39, -0.07, 0.13, -0.17)
...
...
the (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34)
...
...
mouth (-0.32, 0.43, -0.14, 0.50, -0.13, -0.42)
...
...
gone (0.06, -0.21, -0.38, -0.28, -0.16, -0.44)
...
Word Embeddings
NOUN (0.16, 0.03, -0.17, -0.13)
VERB (0.41, 0.08, 0.44, 0.02)
...
...
DET (-0.04, 0.50, 0.04, 0.44)
ADJ (-0.01, -0.35, -0.27, 0.20)
PREP (-0.26, 0.28, -0.34, -0.02)
...
...
ADV (0.02, -0.17, 0.46, -0.08)
...
POS Embeddings

Bengio et al. 2003: N-gram multilayer perceptron

$$f(w_t, \cdots, w_{t-n+1}) = \hat{P}(w_t | w_1^{t-1})$$



Learn: C, W, U, H, d (chain rule)

$C(i) \in \mathbb{R}^m$ Word embedding parameters

$$x = (C(w_{t-1}), C(w_{t-2}), \cdots, C(w_{t-n+1}))$$

Lookup layer with concatenation: (kinda) hidden layer size *(n-1)m*

another hidden layer, size *h*

$$y = b + Wx + U \tanh(d + Hx)$$

Vocab output: log-probs size *V*

$$\hat{P}(w_t | w_{t-1}, \cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$

Output layer (softmax / log-linear)

- stopped here 2/6

Why?

- Curse of dimensionality: bottleneck information into K~30 hidden dimensions (K<<V)
- NNs can learn complicated functions
  - ... we don't really have a good grip on what's learnable beyond universal function approximation
  - ... but seems better than linear dim reduction (e.g. S+P). Non-planar regions in embedding space?
- Multilayer structures
  - Maybe: "deep" models learn more abstract concepts (clearly in vision; less clear for NLP, though can help)
  - Definitely: hierarchical and sequential NNs to match hierarchical/memory-ful structure in language (recursive/ recurrent NNs)

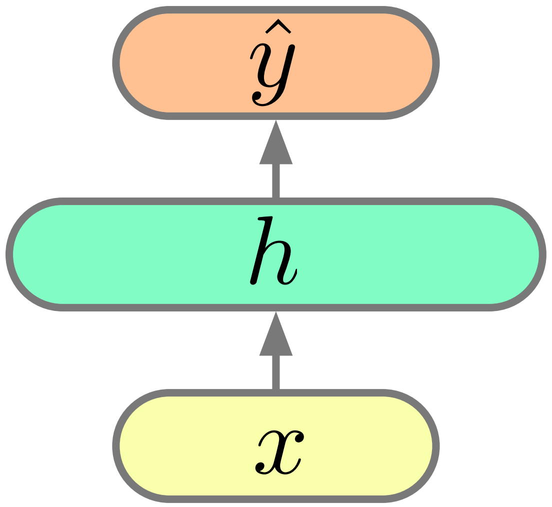
Word/feature embeddings

- "Lookup layer": from discrete input features (words, ngrams, etc.) to continuous vectors
  - Any binary feature that was directly used in log-linear models, give it a vector
  - Character n-grams, part-of-speech tags, etc.
- As model parameters: learn them like everything else
- Or, as external information: use pretrained embeddings
  - Common in practice: use a faster-to-train model on very large, perhaps different, dataset
    [e.g. *word2vec*, *glove* pretrained word vectors]
- Shared representations for
  domain adaptation and multitask learning

Neural Language Models

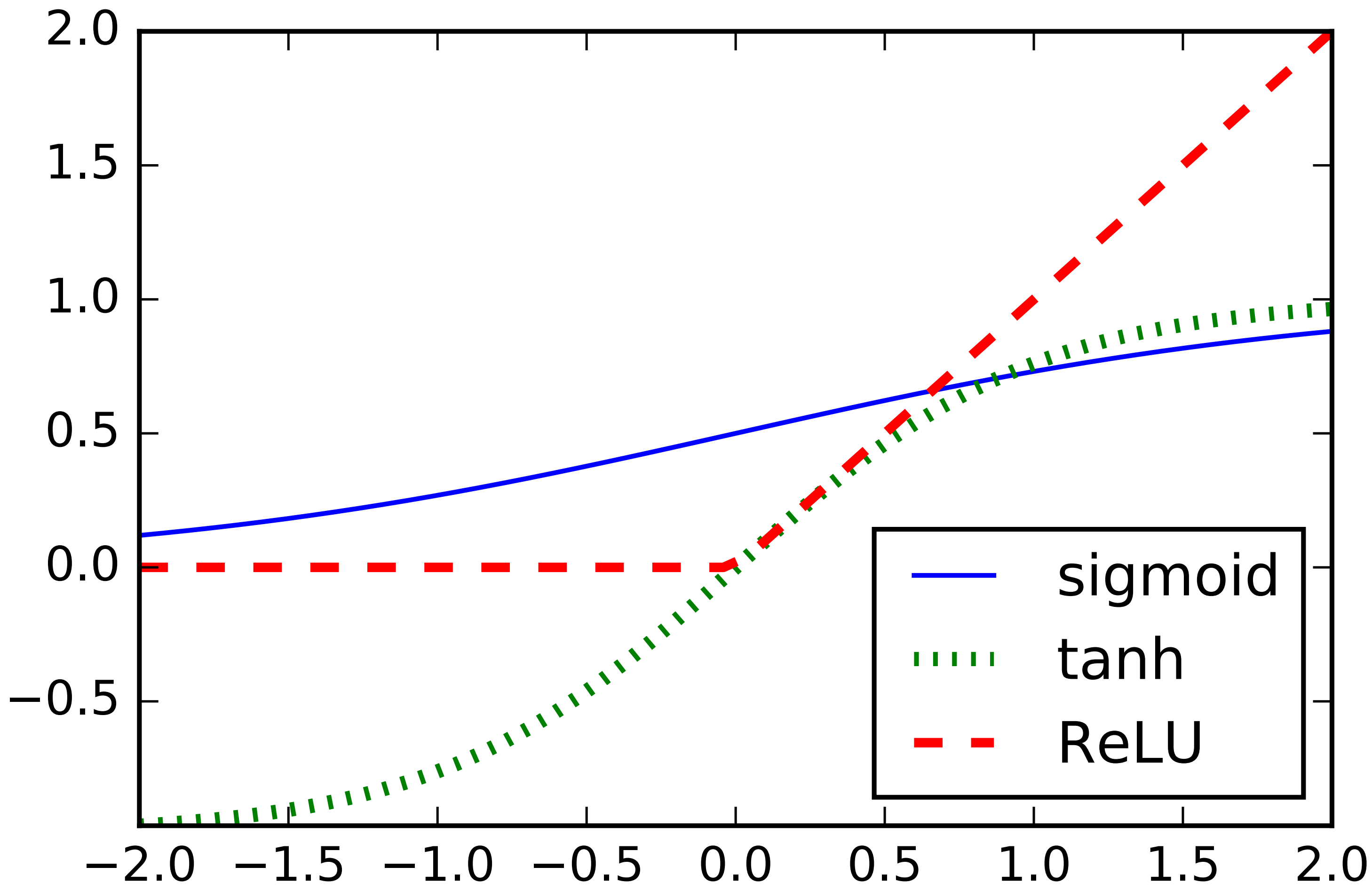
Feed forward network

$$
\begin{aligned}
h &= g(Vx + c) \\
\hat{y} &= Wh + b
\end{aligned}
$$



[Slide: Phil Blunsom]

Nonlinear activation functions



$$\text{sigmoid}(x) = \frac{e^x}{1 + e^x}$$

$$\tanh(x) = 2 \times \text{sgm}(x) - 1$$

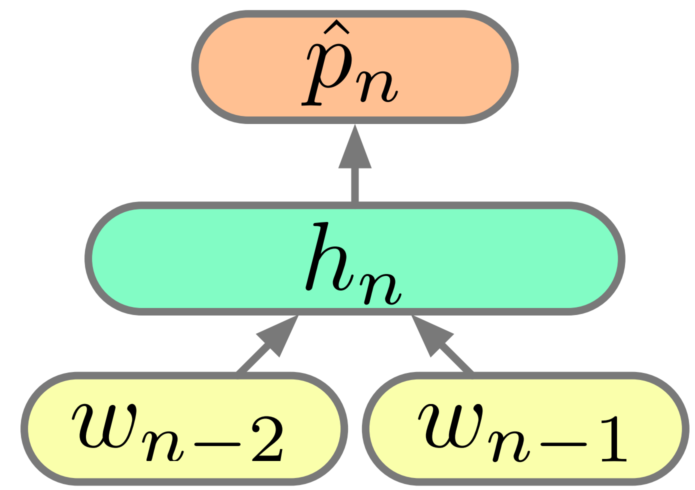
$$(x)_+ = \max(0, x)$$

a.k.a. "ReLU"

Trigram NN language model

Word embeddings

$$
\begin{aligned}
h_n &= g(V[w_{n-1}; w_{n-2}] + c) \\
\hat{p}_n &= \text{softmax}(W h_n + b) \\
\text{softmax}(u)_i &= \frac{\exp u_i}{\sum_j \exp u_j}
\end{aligned}
$$

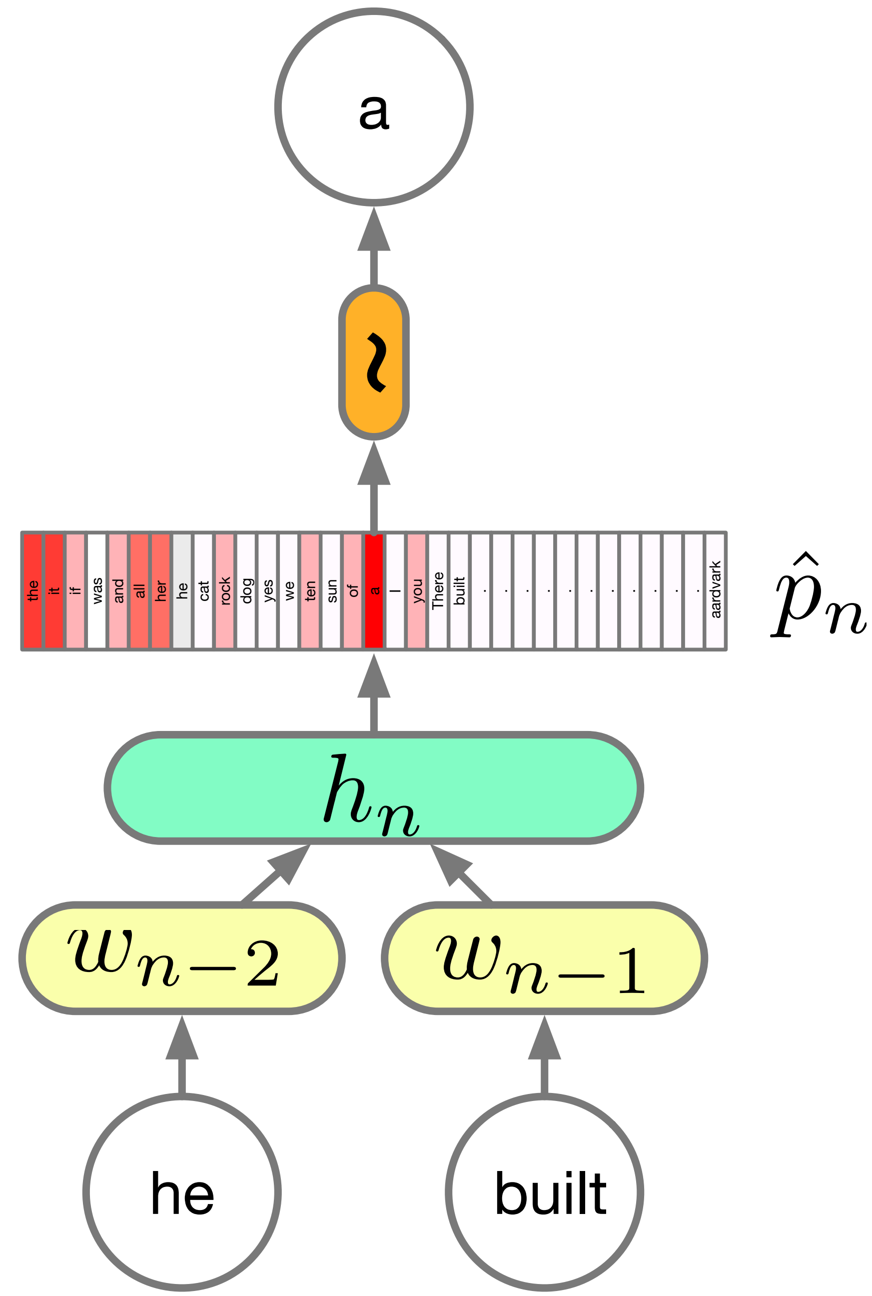


- w_i are one hot vetors and $\hat{p}_i$ are distributions,
- $|w_i| = |\hat{p}_i| = V$ (words in the vocabulary),
- V is usually very large $> 1e5$.

[Slide: Phil Blunsom]

Neural Language Models: Sampling

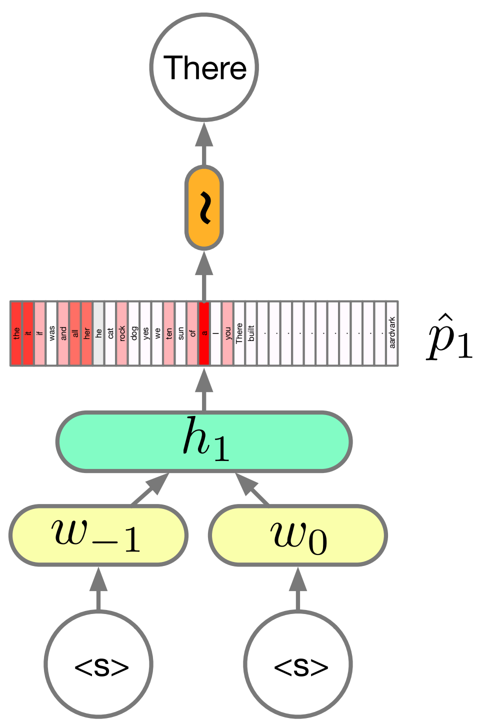
$$w_n | w_{n-1}, w_{n-2} \quad \sim \quad \hat{p}_n$$



[Slide: Phil Blunsom]

Neural Language Models: Sampling

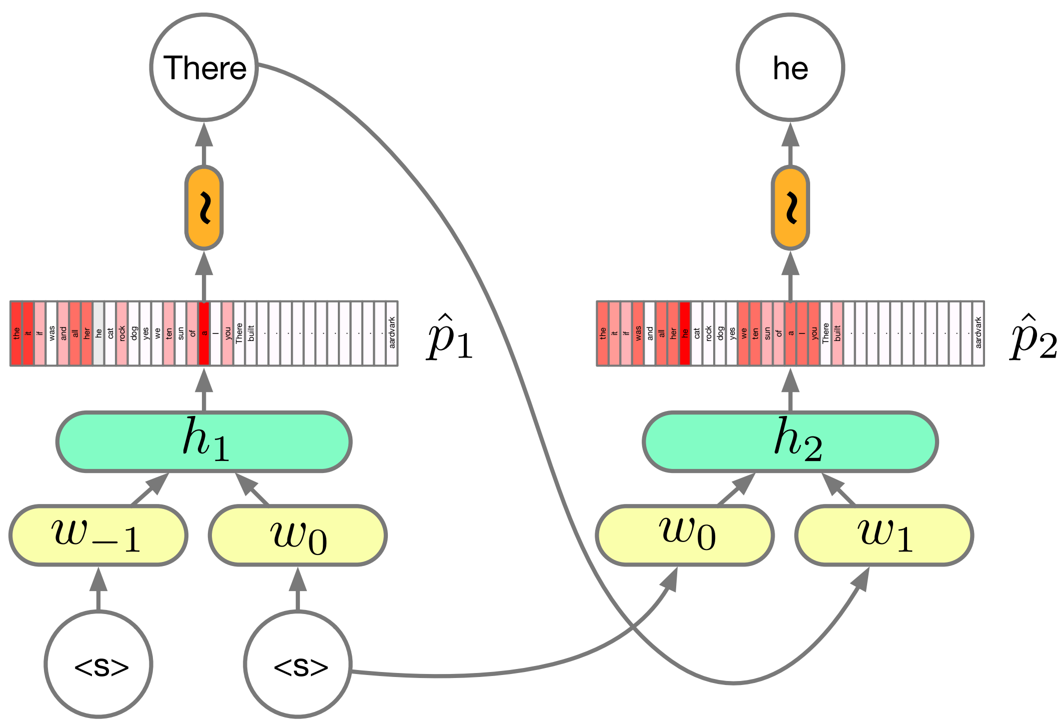
$$w_n | w_{n-1}, w_{n-2} \quad \sim \quad \hat{p}_n$$



[Slide: Phil Blunsom]

Neural Language Models: Sampling

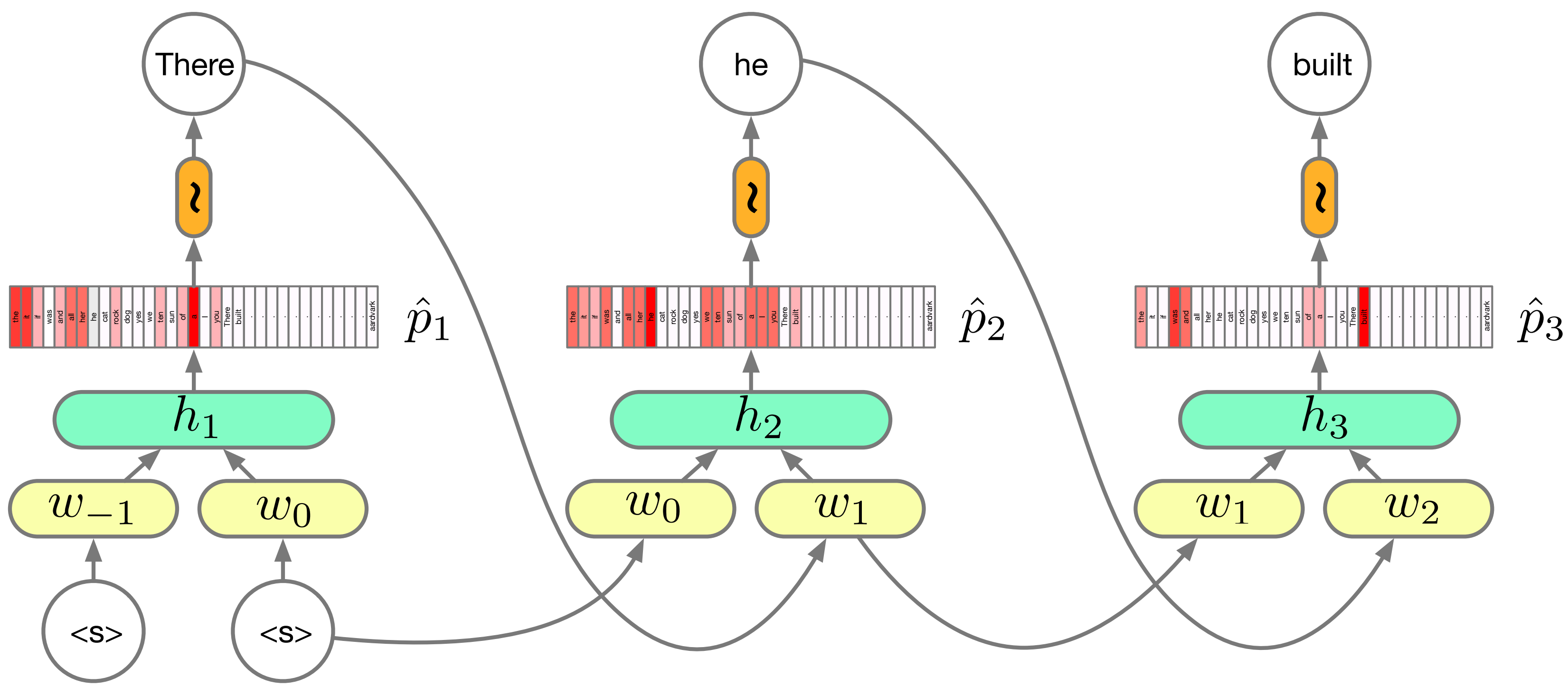
$$w_n | w_{n-1}, w_{n-2} \quad \sim \quad \hat{p}_n$$



[Slide: Phil Blunsom]

Neural Language Models: Sampling

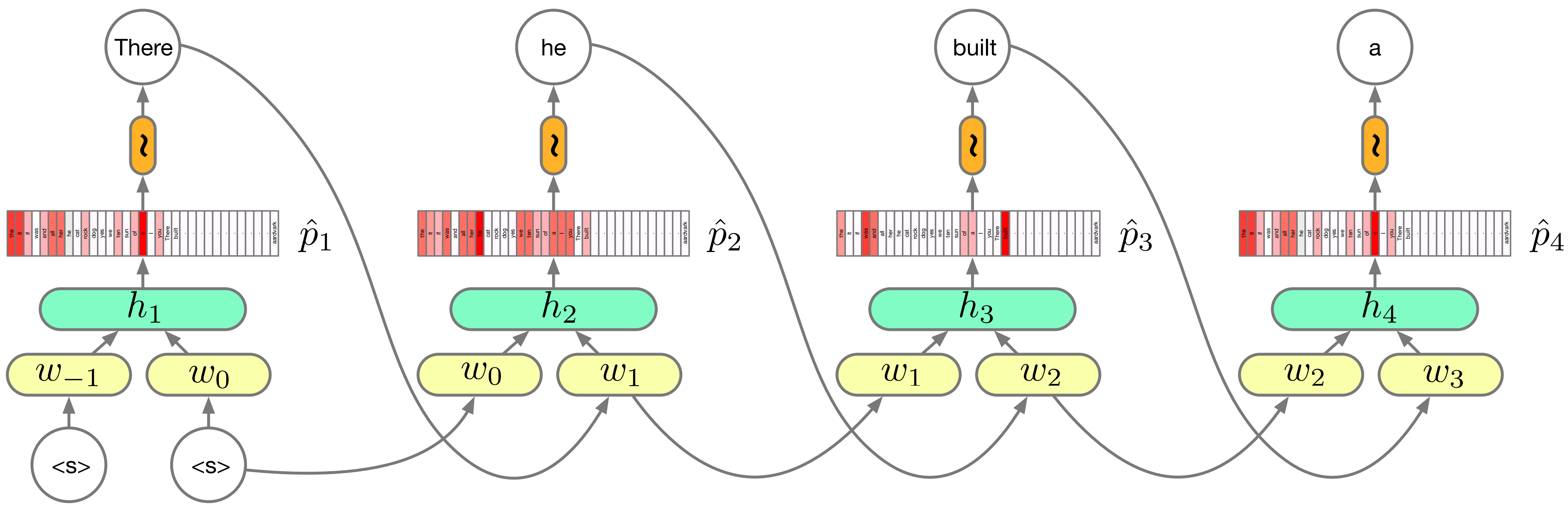
$$w_n | w_{n-1}, w_{n-2} \quad \sim \quad \hat{p}_n$$



[Slide: Phil Blunsom]

Neural Language Models: Sampling

$$w_n \big| w_{n-1}, w_{n-2} \quad \sim \quad \hat{p}_n$$



[Slide: Phil Blunsom]

Neural Language Models: Training

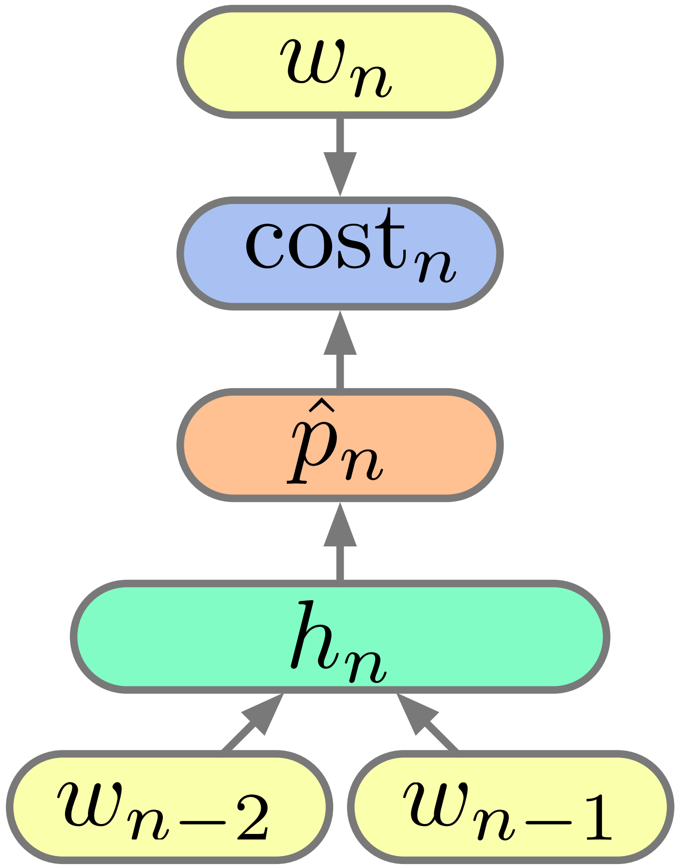
The usual training objective is the cross entropy of the data given the model (MLE):

$$\mathcal{F} = -\frac{1}{N}\sum_n \text{cost}_n(w_n, \hat{p}_n)$$

The cost function is simply the model's estimated log-probability of w_n:

$$\text{cost}(a, b) = a^T \log b$$

(assuming w_i is a one hot encoding of the word)



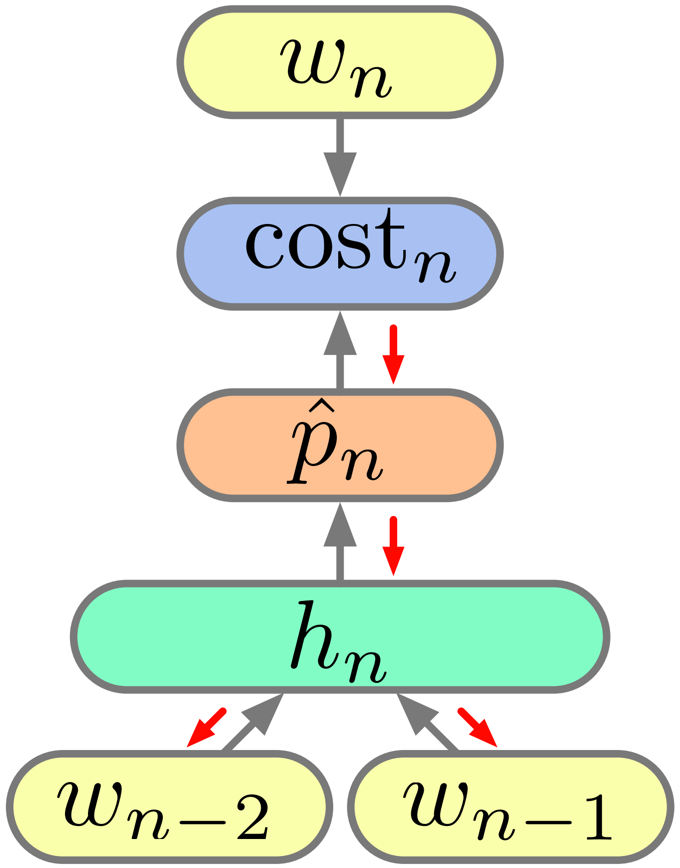
[Slide: Phil Blunsom]

Neural Language Models: Training

Calculating the gradients is straightforward with back propagation:

$$\frac{\partial \mathcal{F}}{\partial W} = -\frac{1}{N}\sum_n \frac{\partial \mathrm{cost}_n}{\partial \hat{p}_n}\frac{\partial \hat{p}_n}{\partial W}$$

$$\frac{\partial \mathcal{F}}{\partial V} = -\frac{1}{N}\sum_n \frac{\partial \mathrm{cost}_n}{\partial \hat{p}_n}\frac{\partial \hat{p}_n}{\partial h_n}\frac{\partial h_n}{\partial V}$$

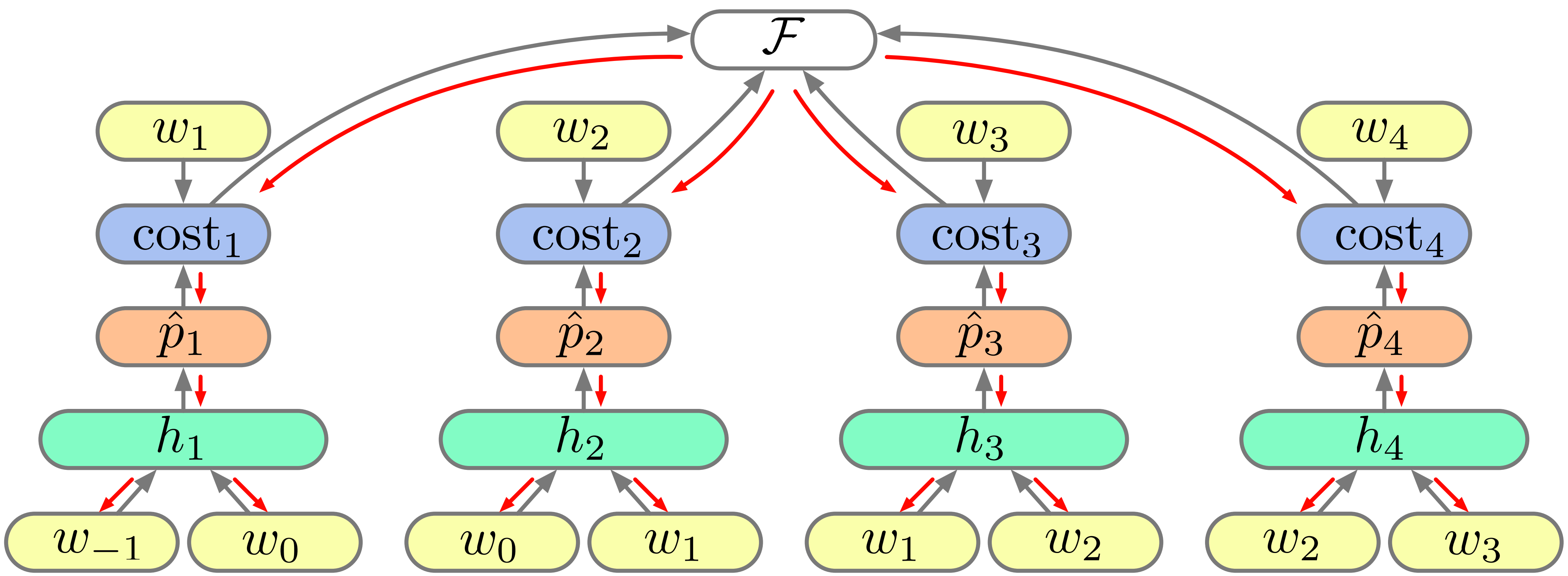


[Slide: Phil Blunsom]

Neural Language Models: Training

Calculating the gradients is straightforward with back propagation:

$$\frac{\partial \mathcal{F}}{\partial W} = -\frac{1}{4}\sum_{n=1}^{4} \frac{\partial \text{cost}_n}{\partial \hat{p}_n}\frac{\partial \hat{p}_n}{\partial W} \quad , \quad \frac{\partial \mathcal{F}}{\partial V} = -\frac{1}{4}\sum_{n=1}^{4} \frac{\partial \text{cost}_n}{\partial \hat{p}_n}\frac{\partial \hat{p}_n}{\partial h_n}\frac{\partial h_n}{\partial V}$$



Note that calculating the gradients for each time step n is independent of all other timesteps, as such they are calculated in parallel and summed.

[Slide: Phil Blunsom]

Comparison with Count Based N-Gram LMs

Good

- Better generalisation on unseen n-grams, poorer on seen n-grams. Solution: direct (linear) ngram features.
- Simple NLMs are often an order magnitude smaller in memory footprint than their vanilla n-gram cousins (though not if you use the linear features suggested above!).

Bad

- The number of parameters in the model scales with the n-gram size and thus the length of the history captured.
- The n-gram history is finite and thus there is a limit on the longest dependencies that an be captured.
- Mostly trained with Maximum Likelihood based objectives which do not encode the expected frequencies of words a priori.

[Slide: Phil Blunsom]

Training NNs

- Dropout (preferred regularization method)
- Minibatching
- Parallelization (GPUs)

- Local optima?

Local models

$$w_t \mid w_{t-2}, w_{t-1}$$

Long-history models

$$w_t \mid w_1, \ldots w_{t-1}$$

Fully observed
direct word models

Latent-class
direct word models

...... Log-linear models ......

Markovian neural LM

Recurrent neural LM