Neural Networks Language Models

Philipp Koehn

14 April 2016

N-Gram Backoff Language Model

- Previously, we approximated

$$p(W) = p(w_1, w_2, ..., w_n)$$

- ... by applying the chain rule

$$p(W) = \sum_i p(w_i|w_1, ..., w_{i-1})$$

- ... and limiting the history (Markov order)

$$p(w_i|w_1, ..., w_{i-1}) \simeq p(w_i|w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$$

- Each $p(w_i|w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$ may not have enough statistics to estimate

$\rightarrow$ we back off to $p(w_i|w_{i-3}, w_{i-2}, w_{i-1})$, $p(w_i|w_{i-2}, w_{i-1})$, etc., all the way to $p(w_i)$

  – exact details of backing off get complicated — "interpolated Kneser-Ney"

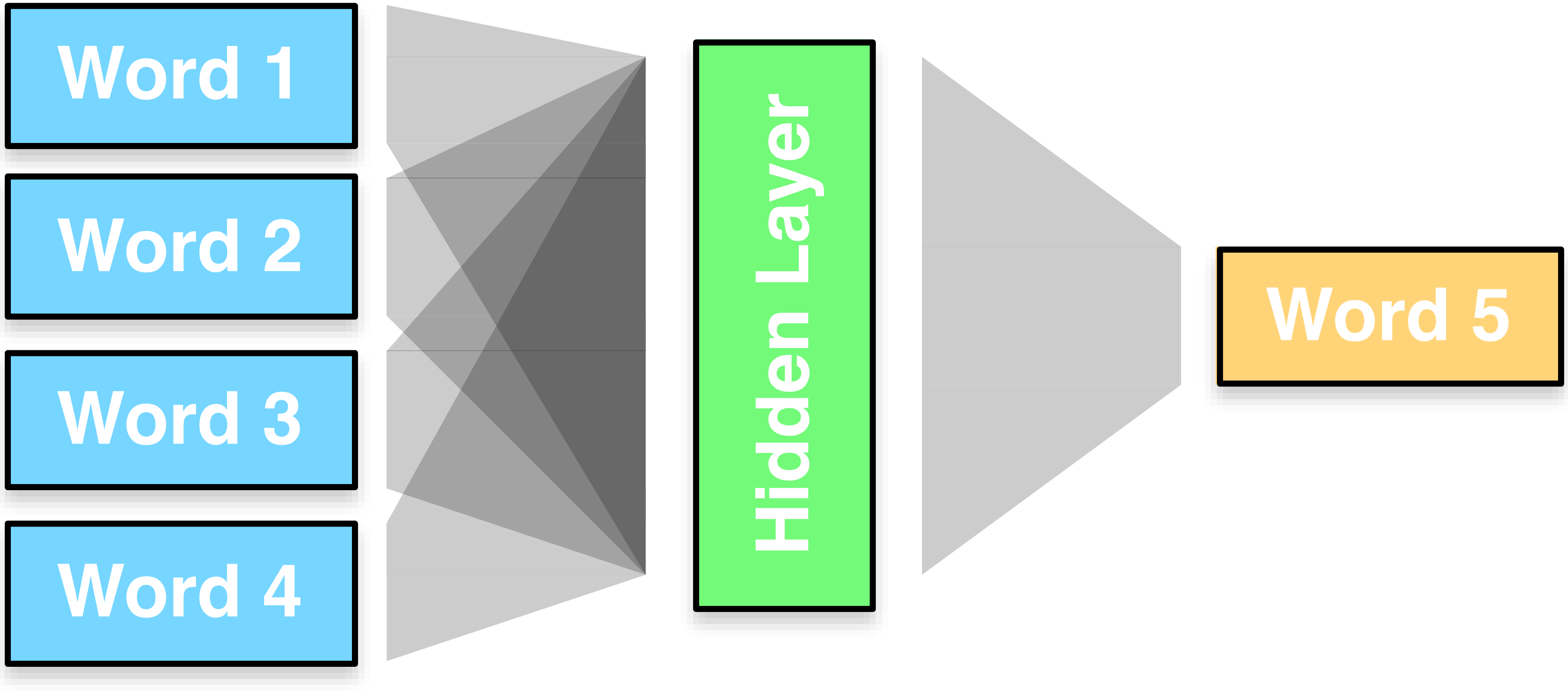
Refinements

- A whole family of back-off schemes
- Skip-n gram models that may back off to $p(w_i|w_{i-2})$
- Class-based models $p(C(w_i)|C(w_{i-4}), C(w_{i-3}), C(w_{i-2}), C(w_{i-1}))$

$\Rightarrow$ We are wrestling here with

- using as much relevant evidence as possible
- pooling evidence between words

First Sketch



Representing Words

- Words are represented with a one-hot vector, e.g.,
  - dog = (0,0,0,0,1,0,0,0,0,....)
  - cat = (0,0,0,0,0,0,0,1,0,....)
  - eat = (0,1,0,0,0,0,0,0,0,....)

- That's a large vector!

- Remedies
  - limit to, say, 20,000 most frequent words, rest are OTHER
  - place words in $\sqrt{n}$ classes, so each word is represented by
    * 1 class label
    * 1 word in class label

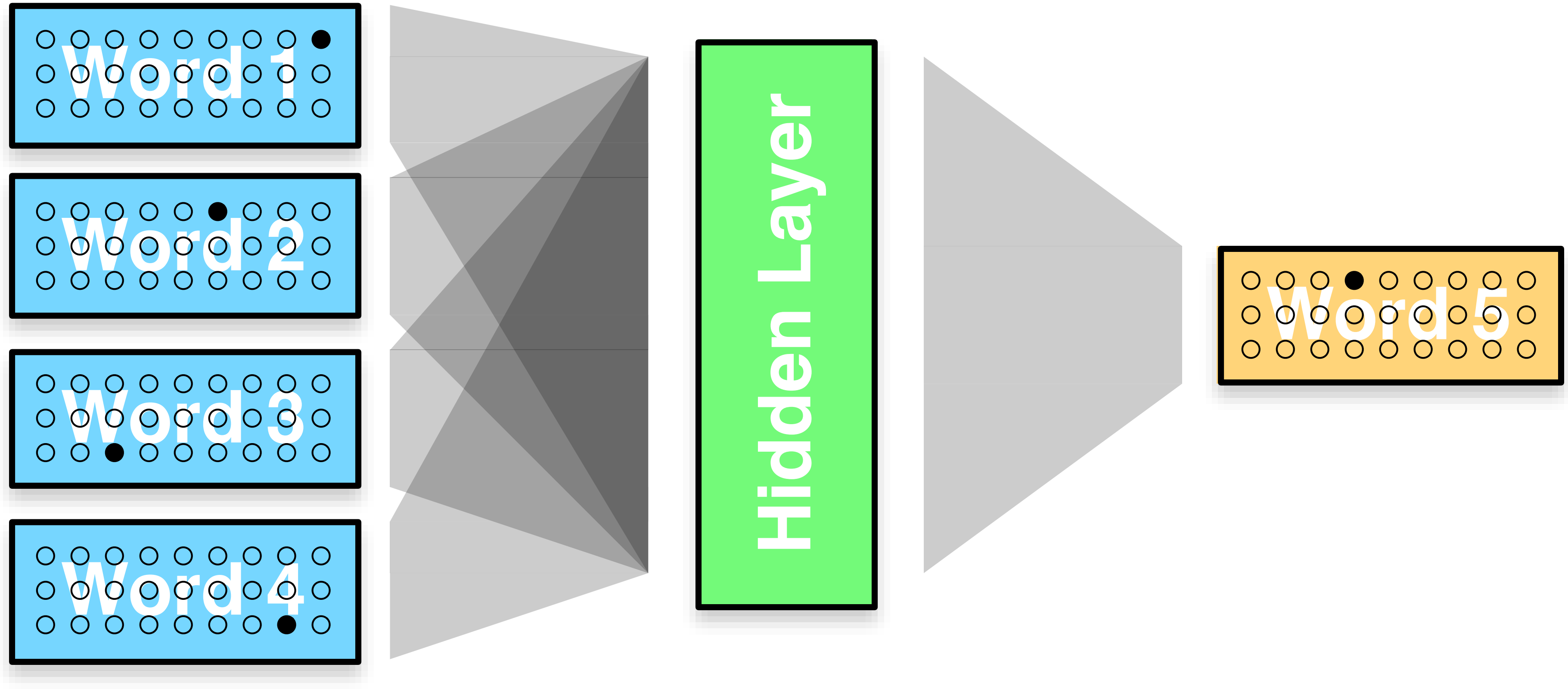
Word Classes for Two-Hot Representations

- WordNet classes
- Brown clusters
- Frequency binning
  - sort words by frequency
  - place them in order into classes
  - each class has same token count

  $\rightarrow$ very frequent words have their own class

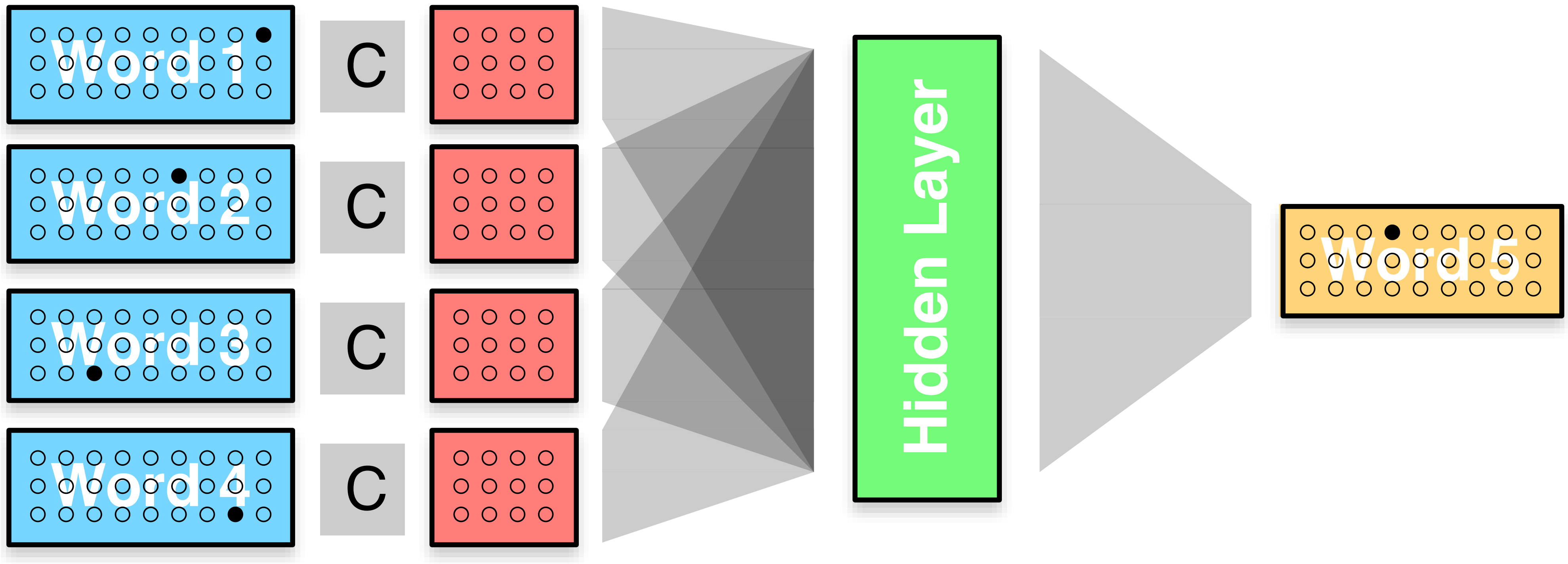
  $\rightarrow$ rare words share class with many other words
- Anything goes: assign words randomly to classes

Second Sketch



word embeddings

Add a Hidden Layer

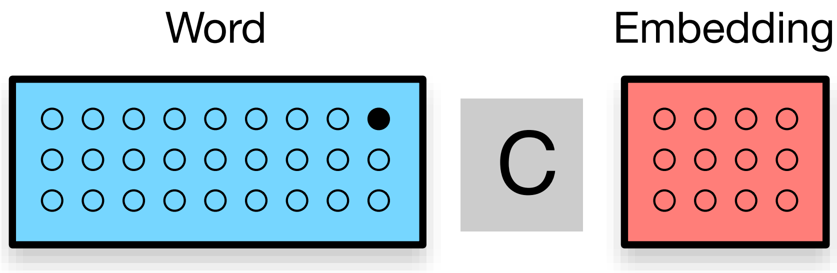


- Map each word first into a lower-dimensional real-valued space
- Shared weight matrix C

Details (Bengio et al., 2003)

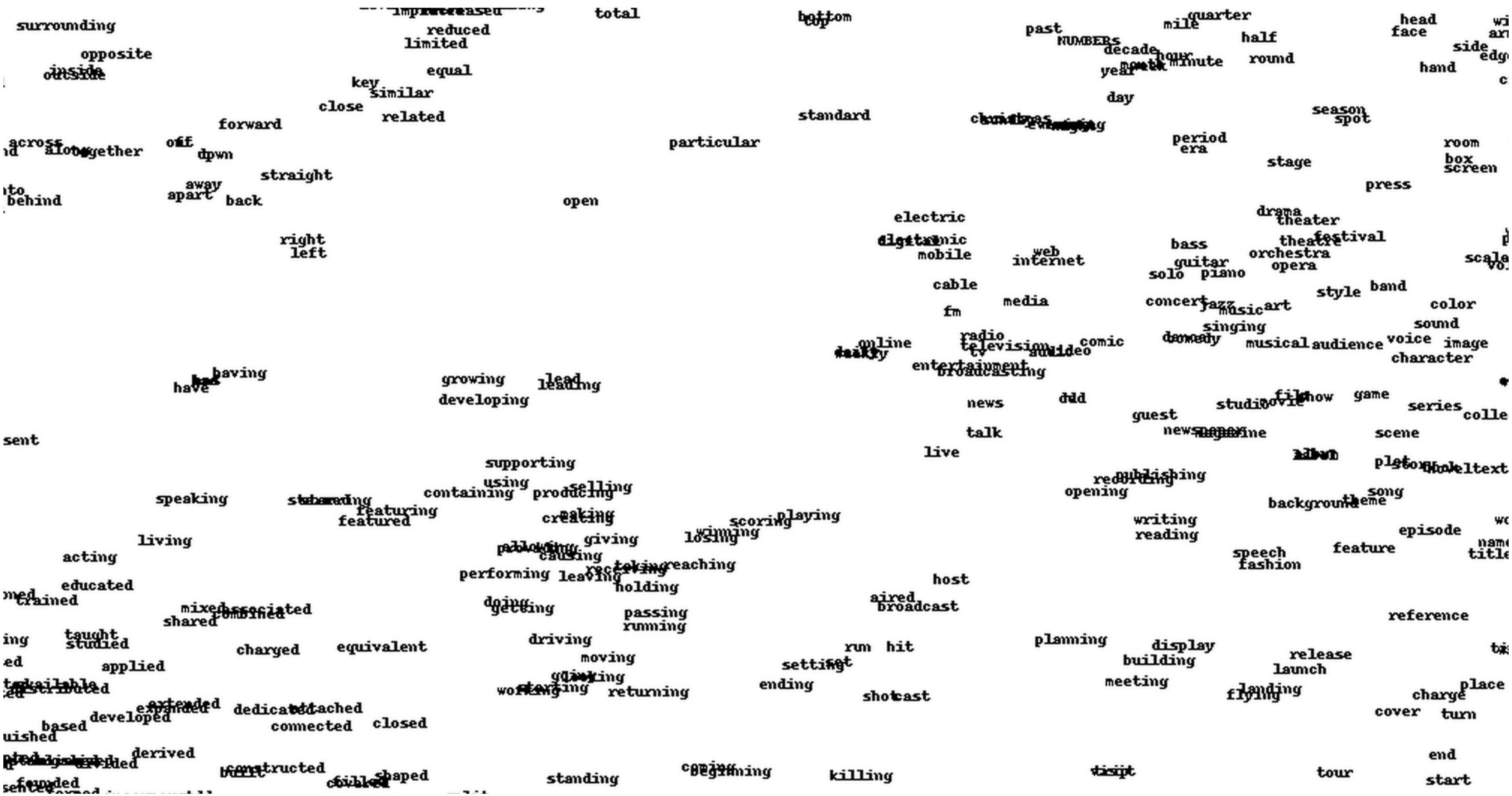
- Add direct connections from embedding layer to output layer
- Activation functions
  - input→embedding: none
  - embedding→hidden: tanh
  - hidden→output: softmax
- Training
  - loop through the entire corpus
  - update between predicted probabilities and 1-hot vector for output word

Word Embeddings

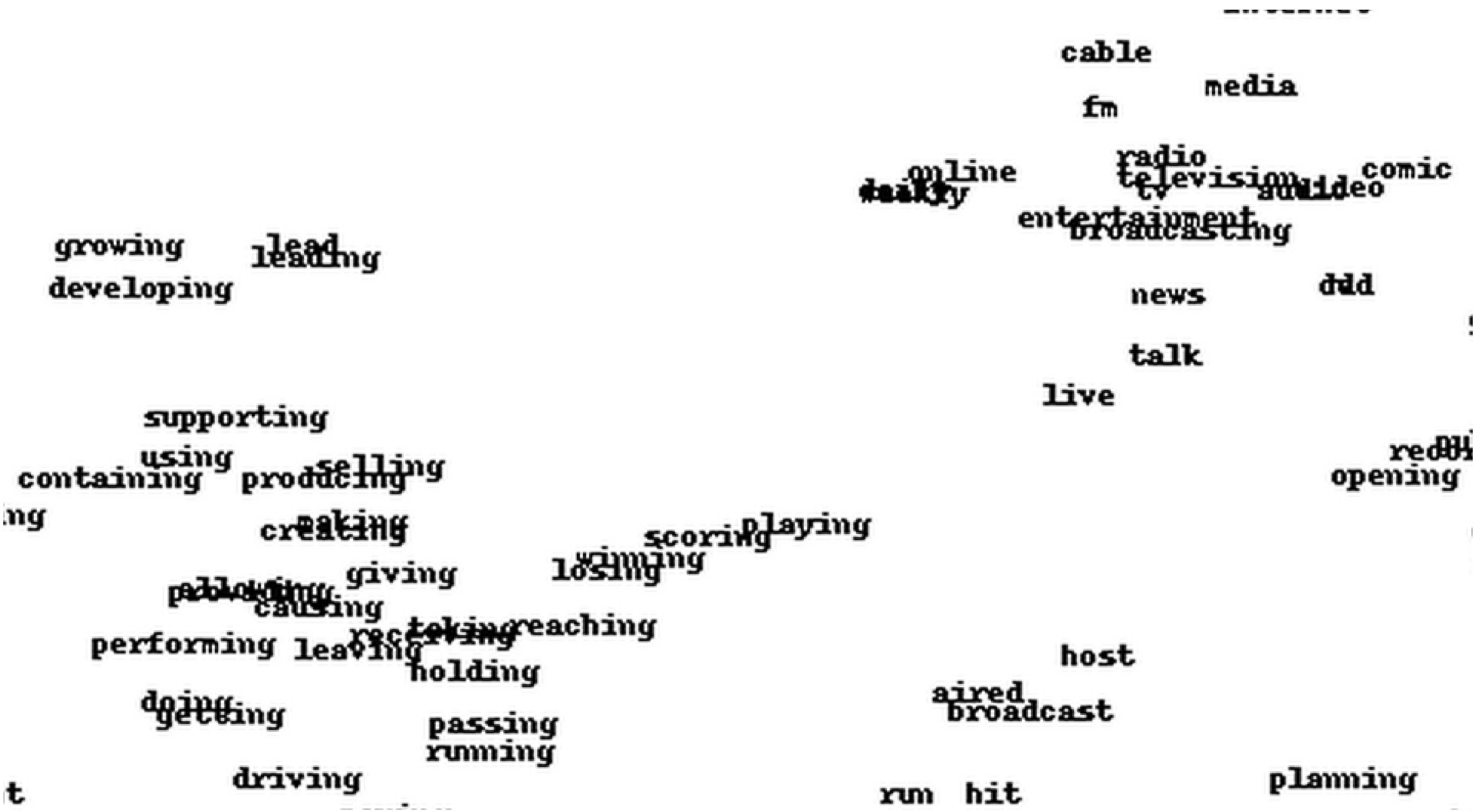


- By-product: embedding of word into continuous space
- Similar contexts $\rightarrow$ similar embedding
- Recall: distributional semantics

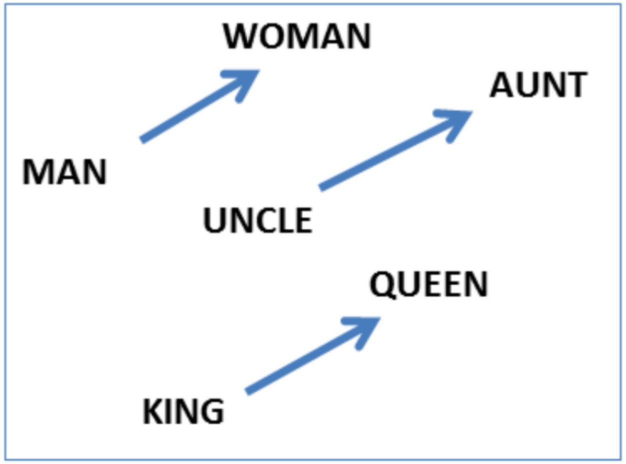
Word Embeddings

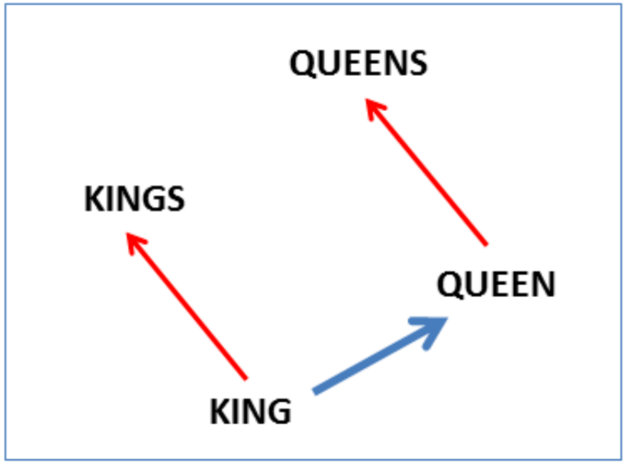


Word Embeddings



Are Word Embeddings Magic?





- Morphosyntactic regularities (Mikolov et al., 2013)
  - adjectives base form vs. comparative, e.g., good, better
  - nouns singular vs. plural, e.g., year, years
  - verbs present tense vs. past tense, e.g., see, saw

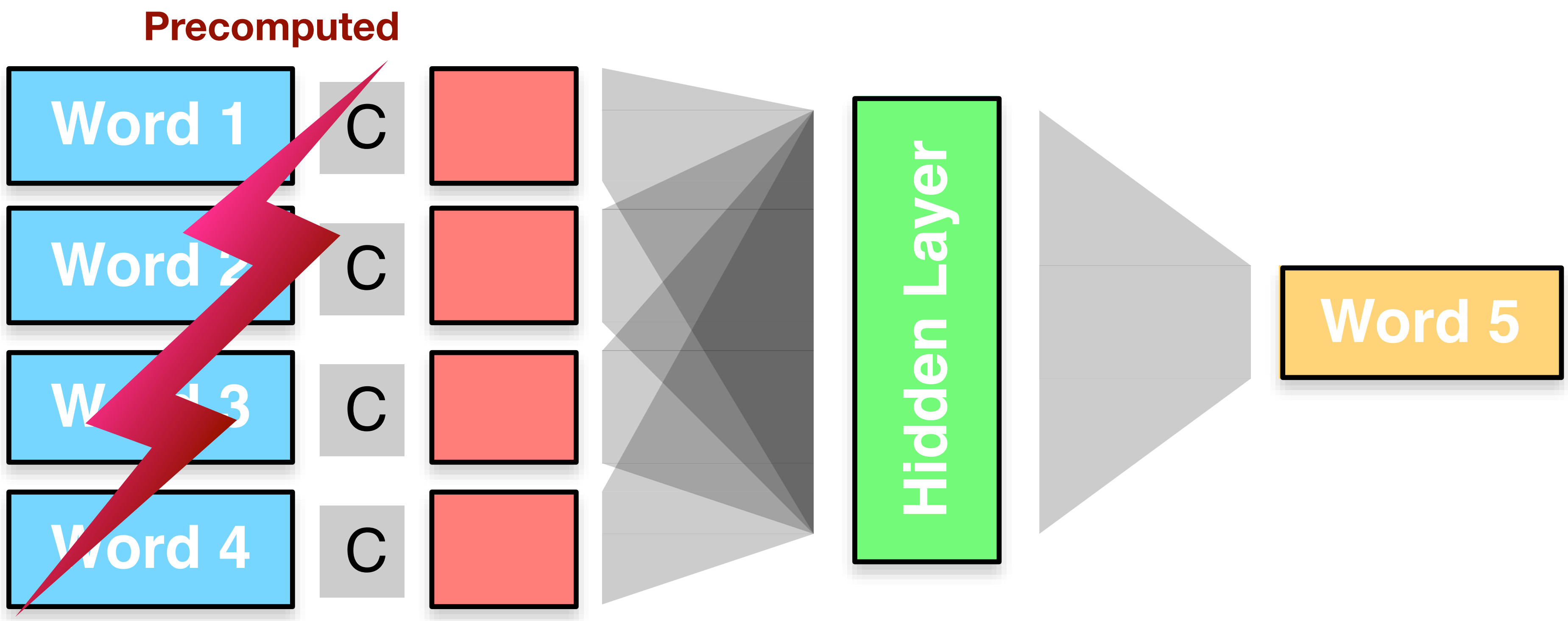
- Semantic regularities
  - clothing is to shirt as dish is to bowl
  - evaluated on human judgment data of semantic similarities

integration into machine translation systems

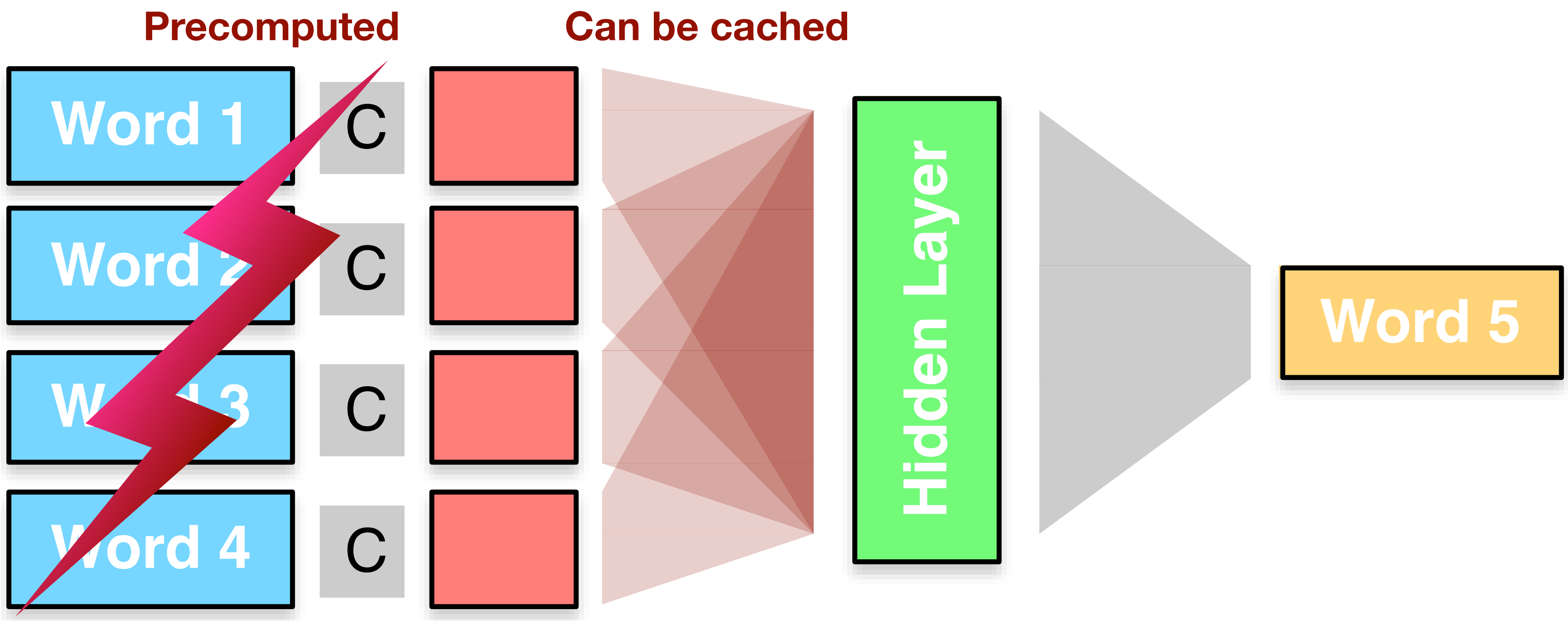
Reranking

- First decode without neural network language model (NNLM)
- Generate
  - n-best list
  - lattice
- Score candidates with NNLM
- Rerank (requires training of weight for NNLM)

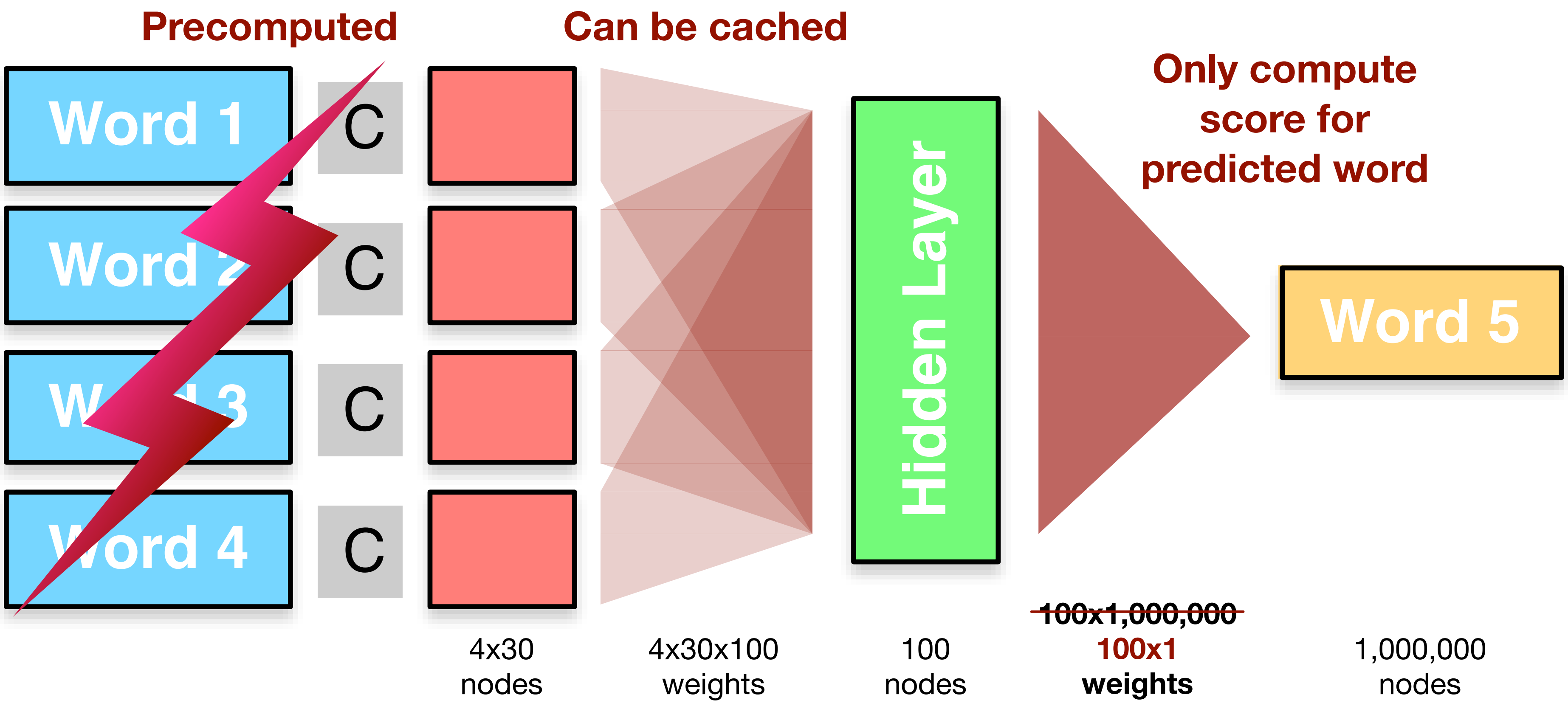
Computations During Inference



Computations During Inference



Computations During Inference



Only Compute Score for Predicted Word?

- Proper probabilities require normalization
  - compute scores for all possible words
  - add them up
  - normalize (softmax)

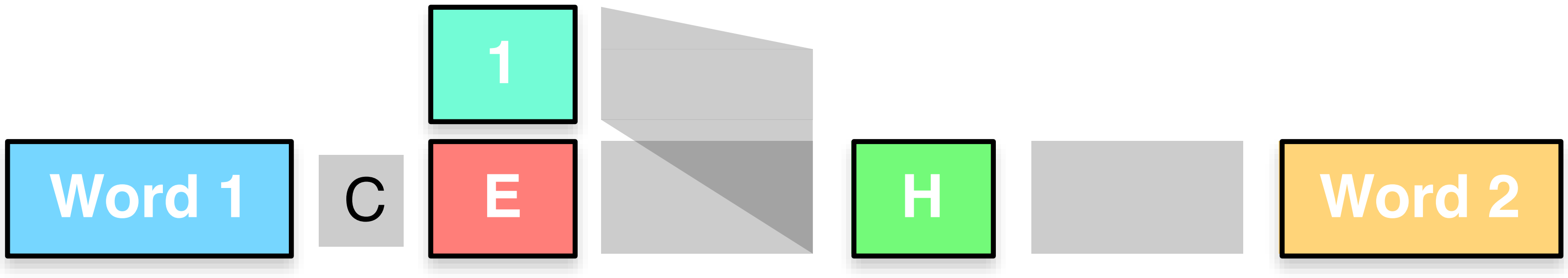
- How can we get away with it?
  - we do not care — a score is a score (Auli and Gao, 2014)
  - training regime that normalizes (Vaswani et al, 2013)
  - integrate normalization into objective function (Devlin et al., 2014)

- Class-based word representations may help
  - first predict class, normalize
  - then predict word, normalize

  → compute $2\sqrt{n}$ instead of n output node values

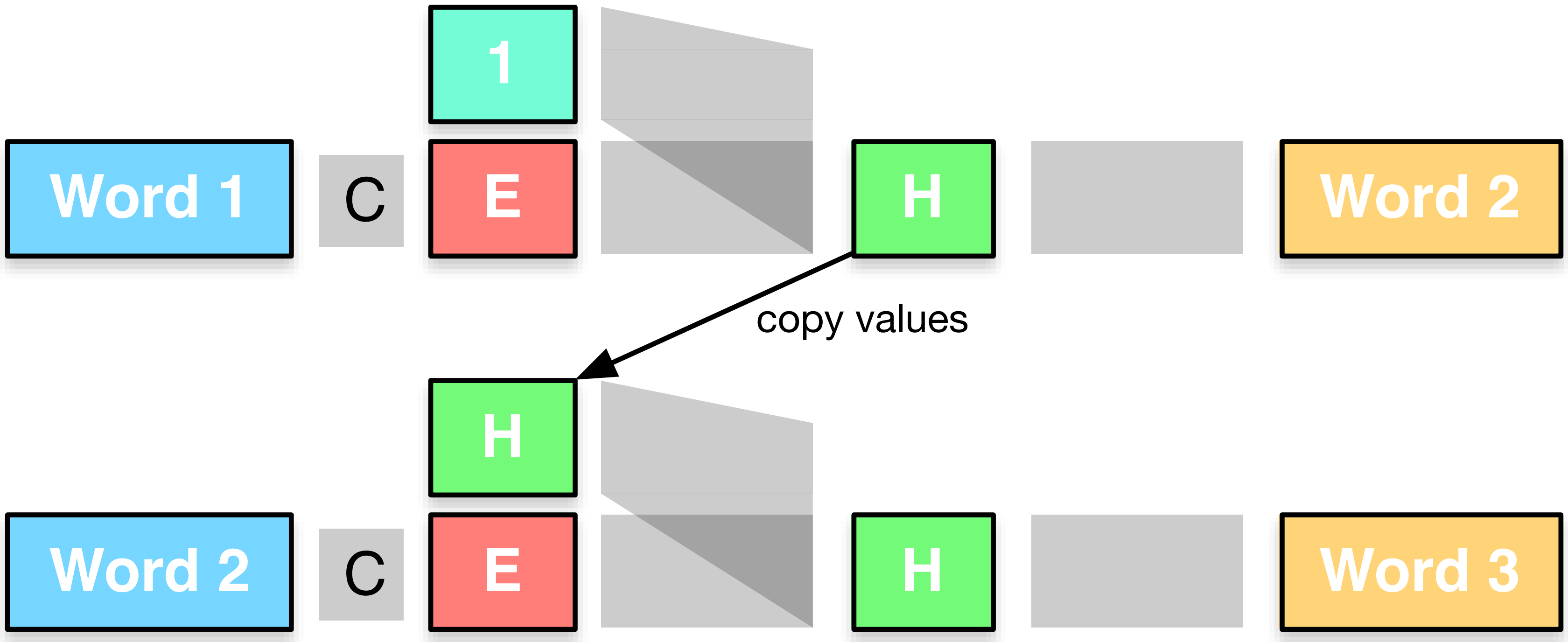
recurrent neural networks

Recurrent Neural Networks

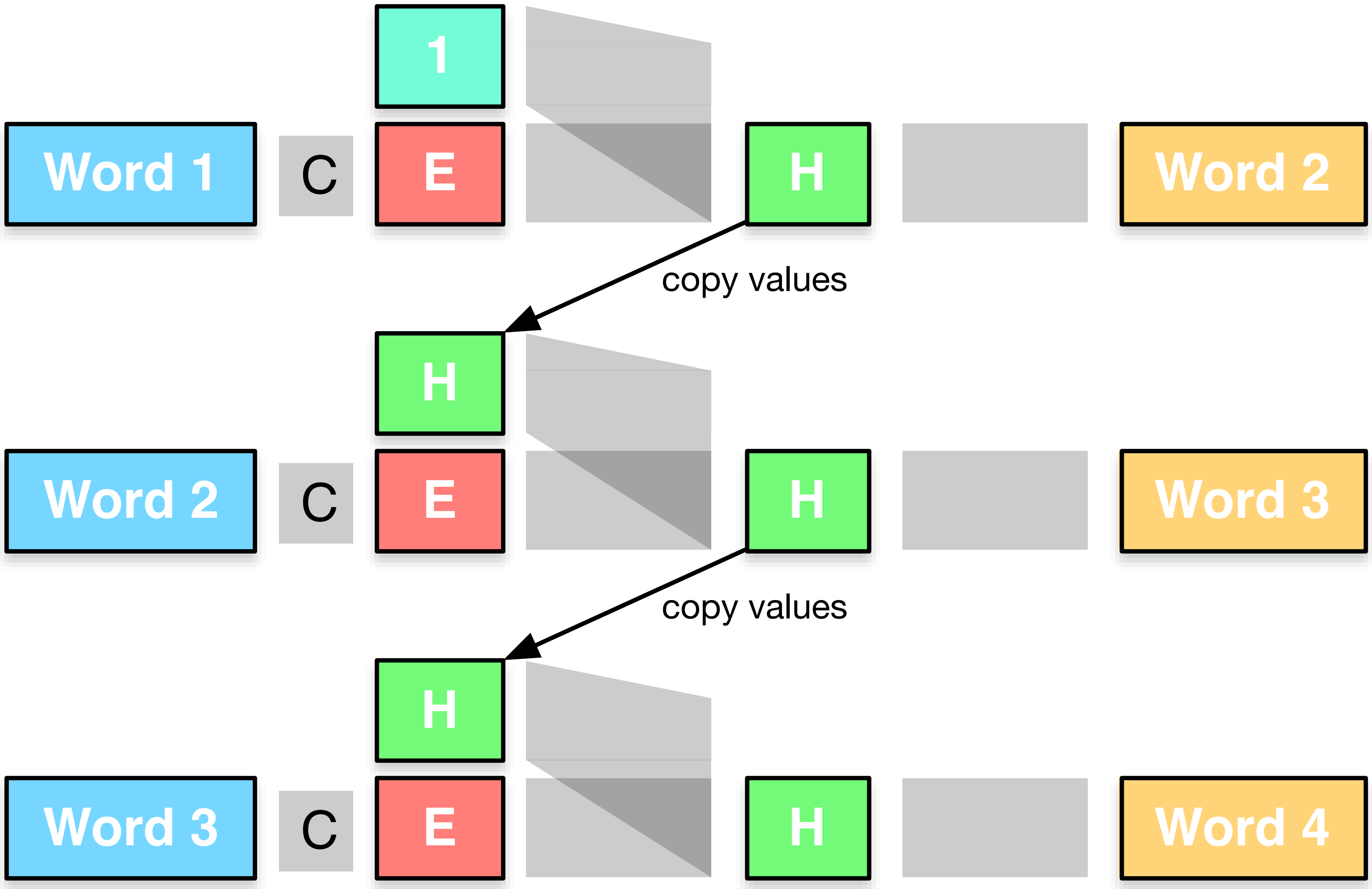


- Start: predict second word from first
- Mystery layer with nodes all with value 1

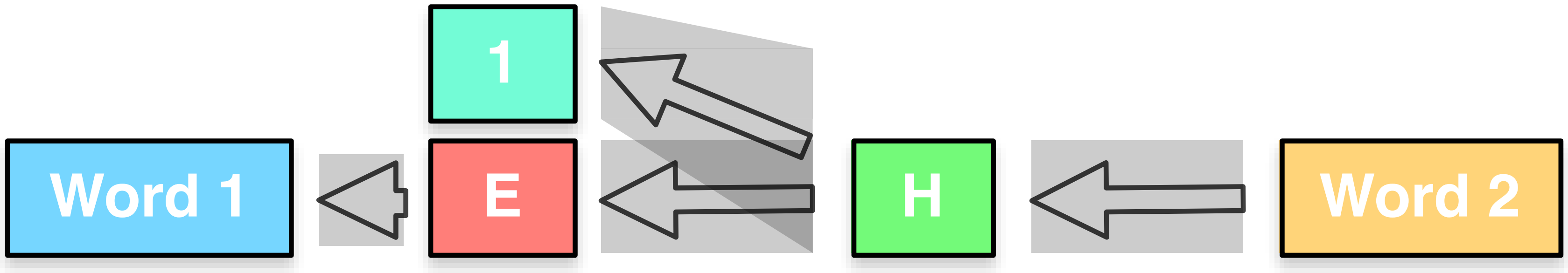
Recurrent Neural Networks



Recurrent Neural Networks

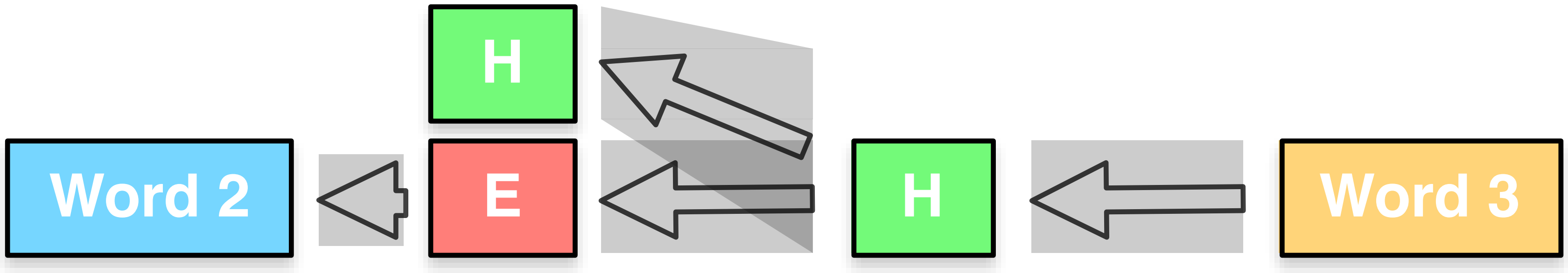


Training



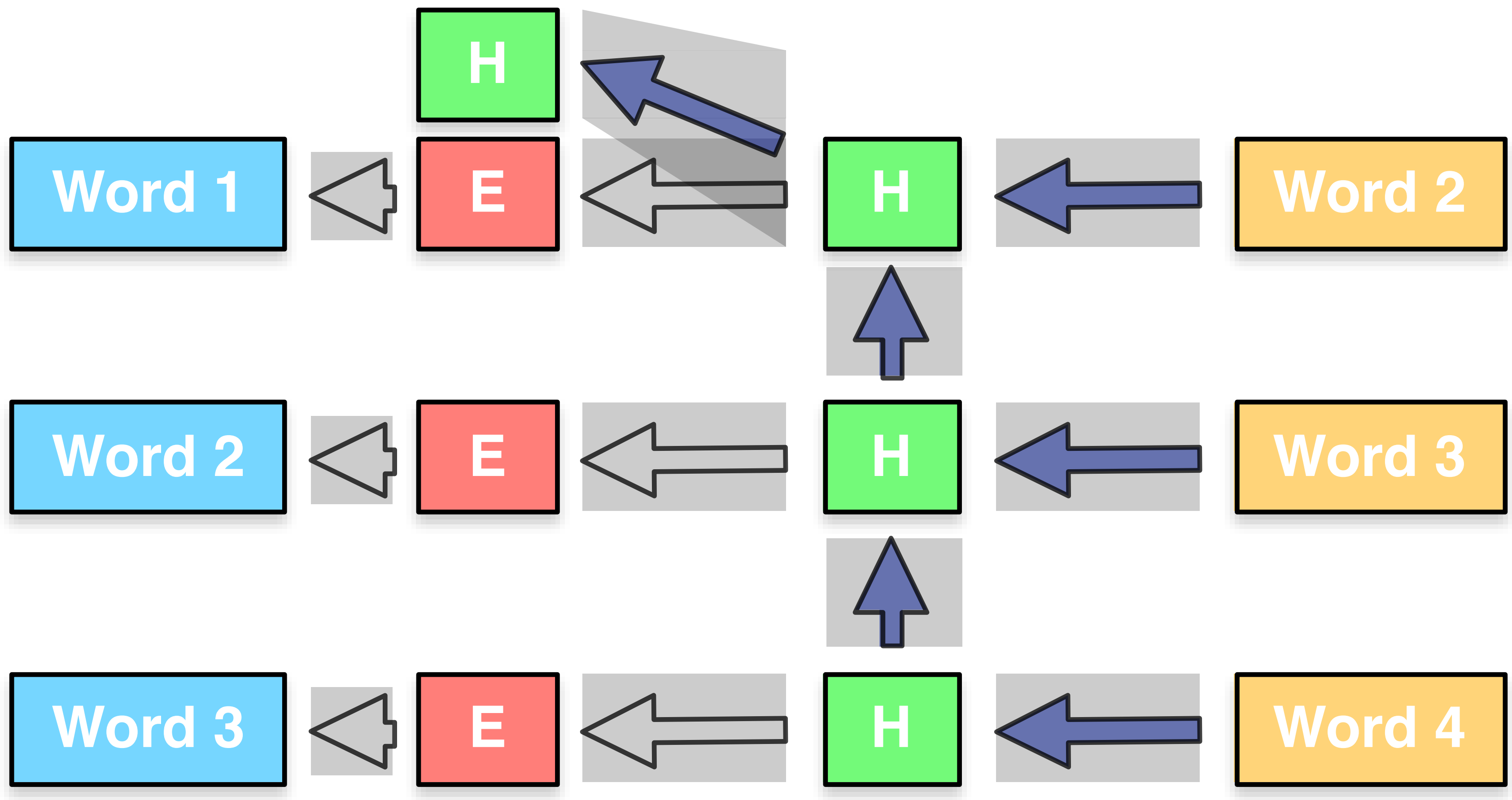
- Process first training example
- Update weights with back-propagation

Training



- Process second training example
- Update weights with back-propagation
- And so on...
- But: no feedback to previous history

Back-Propagation Through Time



- After processing a few training examples,
  update through the unfolded recurrent neural network

Back-Propagation Through Time

- Carry out back-propagation though time (BPTT) after each training example
  - 5 time steps seems to be sufficient
  - network learns to store information for more than 5 time steps
- Or: update in mini-batches
  - process 10-20 training examples
  - update backwards through all examples
  - removes need for multiple steps for each training example

Integration into Decoder

- Recurrent neural networks depend on entire history

⇒ very bad for dynamic programming

long short term memory

Vanishing and Exploding Gradients

- Error is propagated to previous steps

- Updates consider
  - prediction at that time step
  - impact on future time steps

- Exploding gradient: propagated error dominates weight update

- Vanishing gradient: propagated error disappears

⇒ We want the proper balance

Other Problems with RNNs

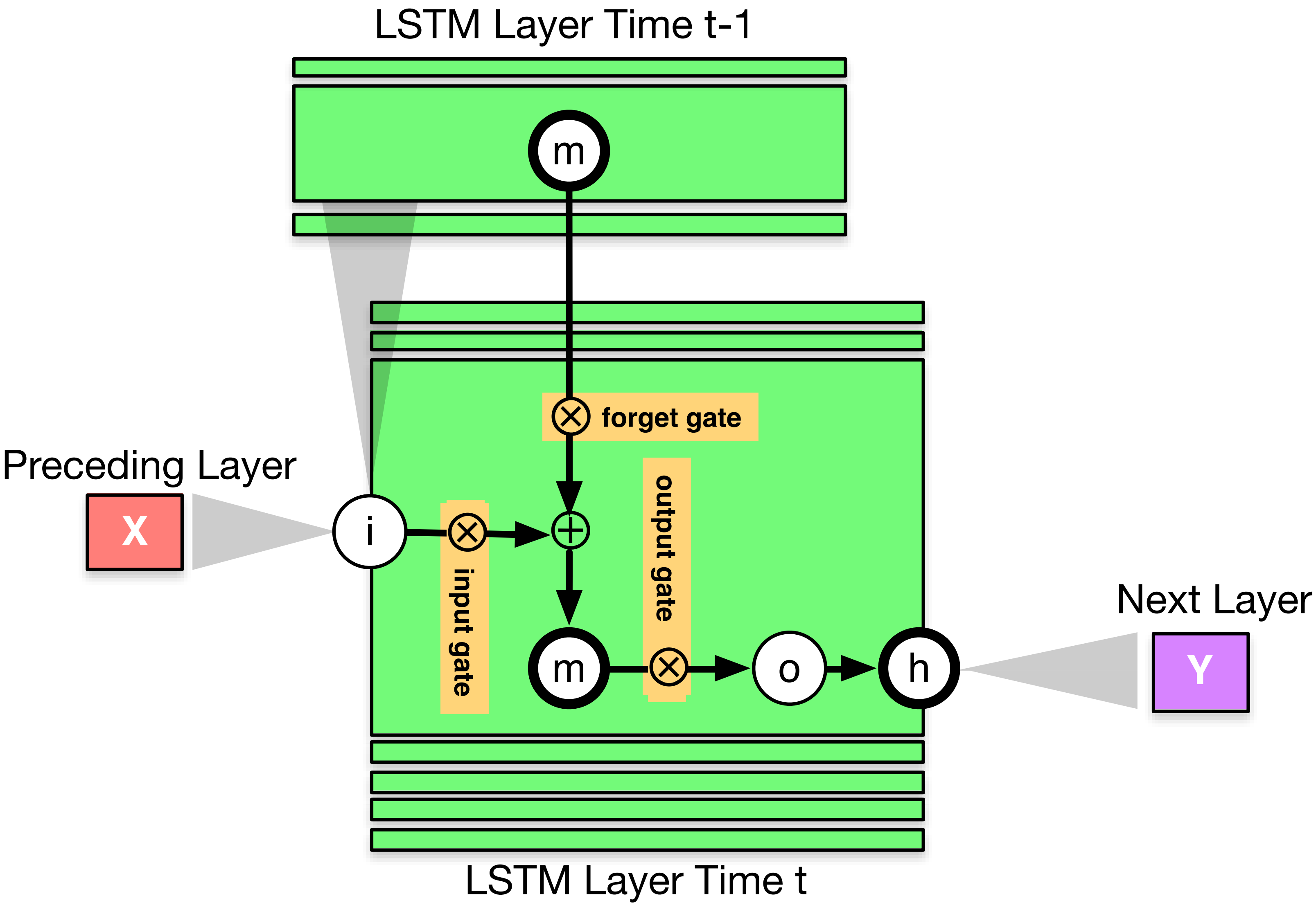
- Hidden layer plays double duty
  - memory of the network
  - continuous space representation used to predict output words

- No clear mechanism to distinguish:
  - sometimes only recent context important
  - sometimes much earlier context important

Long Short Term Memory (LSTM)

- Design quite elaborate, although not very complicated to use
- Basic building block: **LSTM cell**
  - similar to a node in a hidden layer
  - but: has a explicit memory state
- Output and memory state change depends on gates
  - **input gate**: how much new input changes memory state
  - **forget gate**: how much of prior memory state is retained
  - **output gate**: how strongly memory state is passed on to next layer.
- Gates can be not just be open (1) and closed (0), but slightly ajar (e.g., 0.2)

LSTM Cell


LSTM Layer Time t-1
m
forget gate
Preceding Layer
X
i
input gate
output gate
m
o
h
Next Layer
Y
LSTM Layer Time t

LSTM Cell (Math)

- Memory and output values at time step t

$$\text{memory}^t = \text{gate}_{\text{input}} \times \text{input}^t + \text{gate}_{\text{forget}} \times \text{memory}^{t-1}$$

$$\text{output}^t = \text{gate}_{\text{output}} \times \text{memory}^t$$

- Hidden node value h^t passed on to next layer applies activation function f

$$h^t = f(\text{output}^t)$$

- Input computed as input to recurrent neural network node
  - given node values for prior layer $\vec{x}^t = (x_1^t, ..., x_X^t)$
  - given values for hidden layer from previous time step $\vec{h}^{t-1} = (h_1^{t-1}, ..., h_H^{t-1})$
  - input value is combination of matrix multiplication with weights w^x and w^h and activation function g

$$\text{input}^t = g\left(\sum_{i=1}^{X} w_i^x x_i^t + \sum_{i=1}^{H} w_i^h h_i^{t-1}\right)$$

Values for Gates

- Gates are very important
- How do we compute their value?

  $\rightarrow$ with a neural network layer!

- For each gate $a \in (\text{input}, \text{forget}, \text{output})$
  - weight matrix W^{xa} to consider node values in previous layer $\vec{x}^t$
  - weight matrix W^{ha} to consider hidden layer $\vec{h}^{t-1}$ at previous time step
  - weight matrix W^{ma} to consider memory at previous time step $\vec{\text{memory}}^{t-1}$
  - activation function h

$$\text{gate}_a = h\left(\sum_{i=1}^{X} w_i^{xa} x_i^t + \sum_{i=1}^{H} w_i^{ha} h_i^{t-1} + \sum_{i=1}^{H} w_i^{ma} \text{memory}_i^{t-1}\right)$$

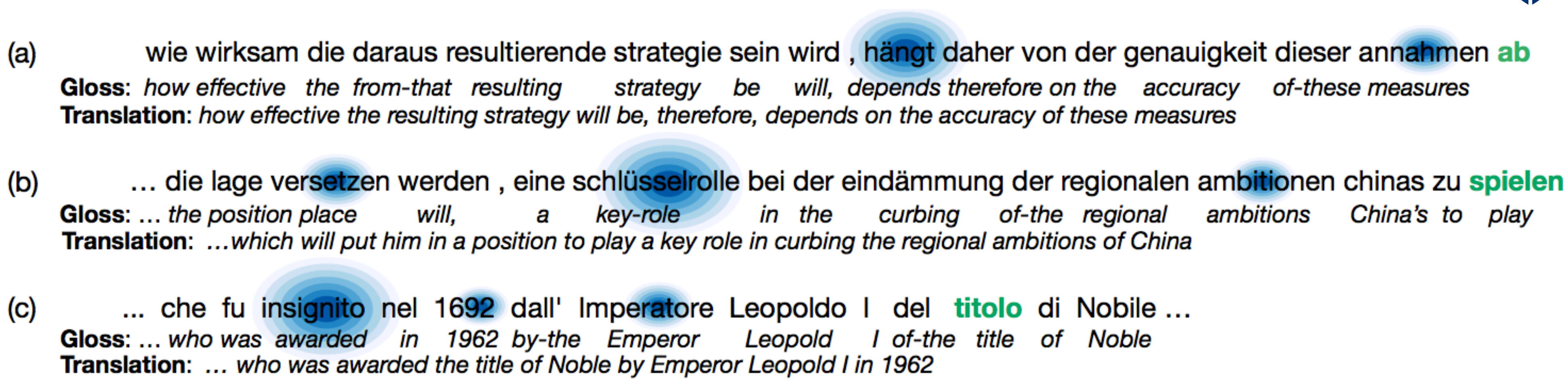
Training

- LSTM are trained the same way as recurrent neural networks
- Back-propagation through time
- This looks all very complex, but:
  - all the operations are still based on
    - matrix multiplications
    - differentiable activation functions

$\rightarrow$ we can compute gradients for objective function with respect to all parameters

$\rightarrow$ we can compute update functions

What is the Point?



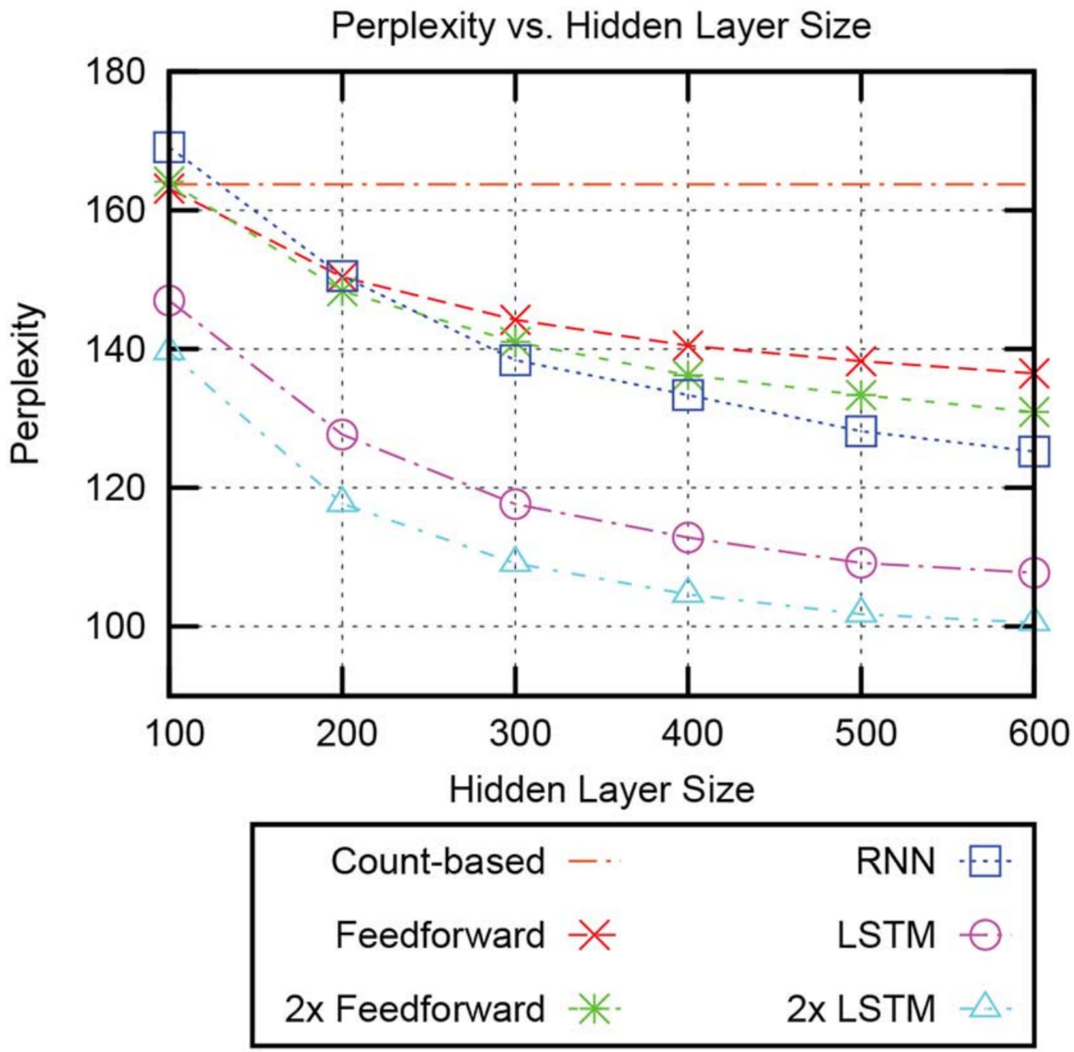
(from Tran, Bisazza, Monz, 2016)

- Each node has memory memory_i independent from current output h_i
- Memory may be carried through unchanged ($\text{gate}^i_{\text{input}} = 0$, $\text{gate}^i_{\text{memory}} = 1$)

⇒ can remember important features over long time span
(capture long distance dependencies)

- Simpler version of this idea by Cho et al. (2014): Gated Recurrent Unit (GRU)

Language Model Comparison: RNN vs LSTM



(from Sundermeyer, Ney, Schlüter, IEEE TASLP 2015)