Neural Networks Language Models

Philipp Koehn

16 April 2015

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

→ 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

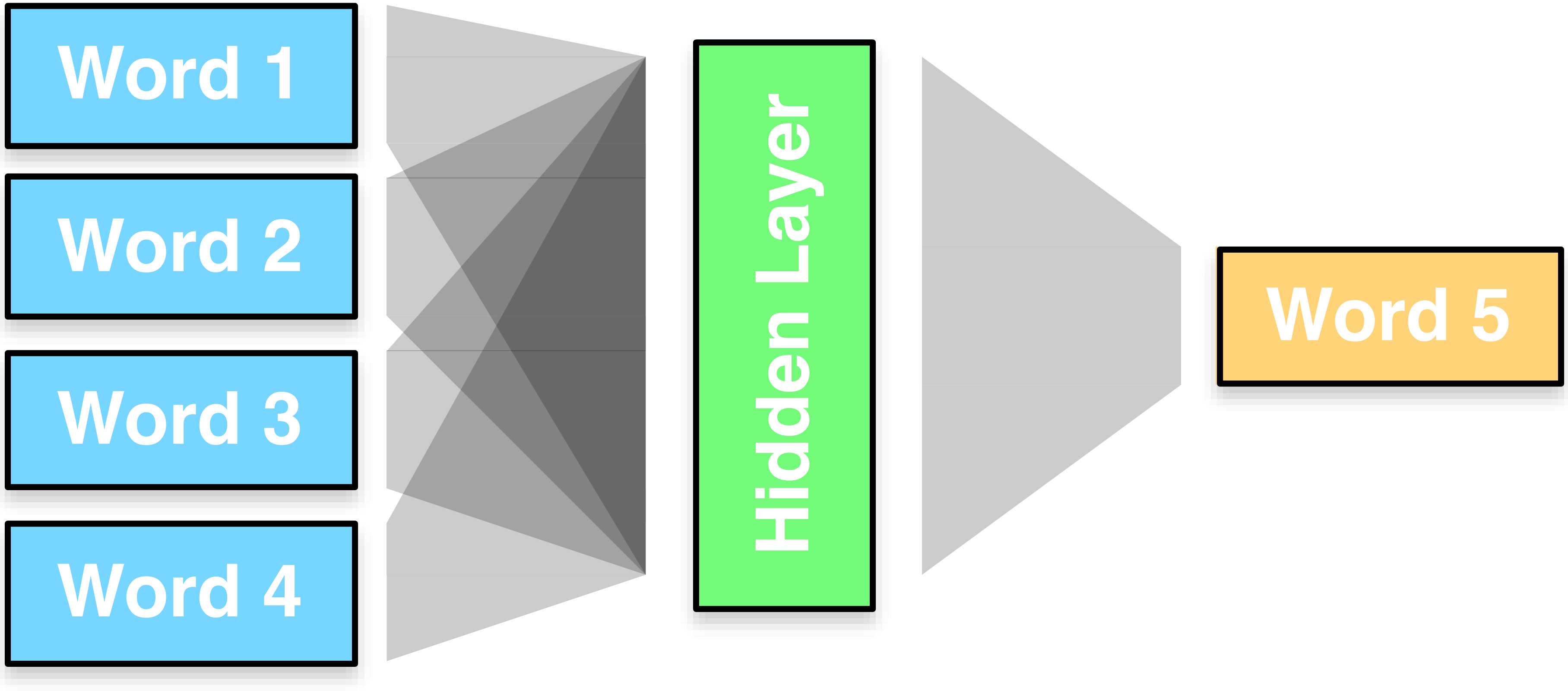
- Skip-n gram models that may back off to $p(w_i|w_{i-2})$

- Class-based models $p(C(w_i)|C(w_{i-4}), C(w_{i-3}), C(w_{i-2}), C(w_{i-1}))$

⇒ 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

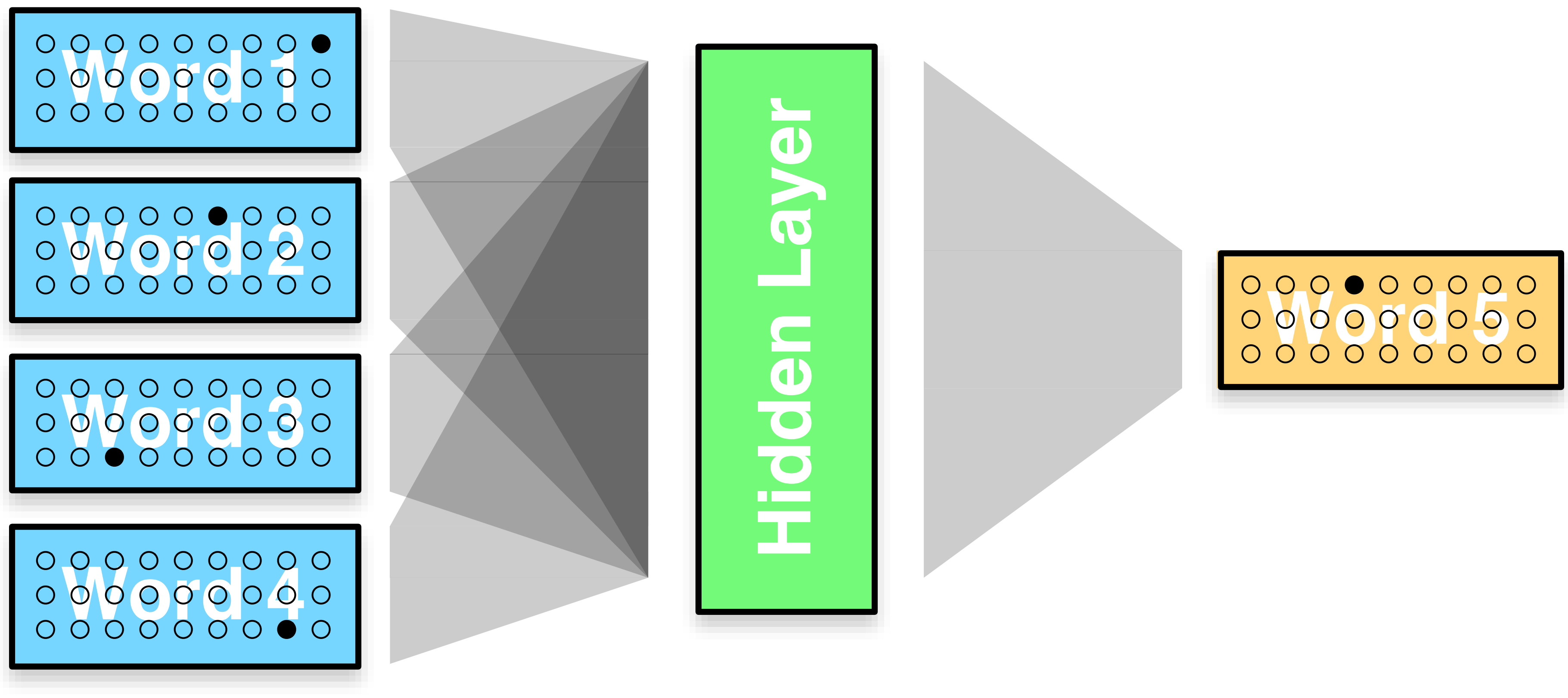
Word Classes for Two-Hot Representations

- 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

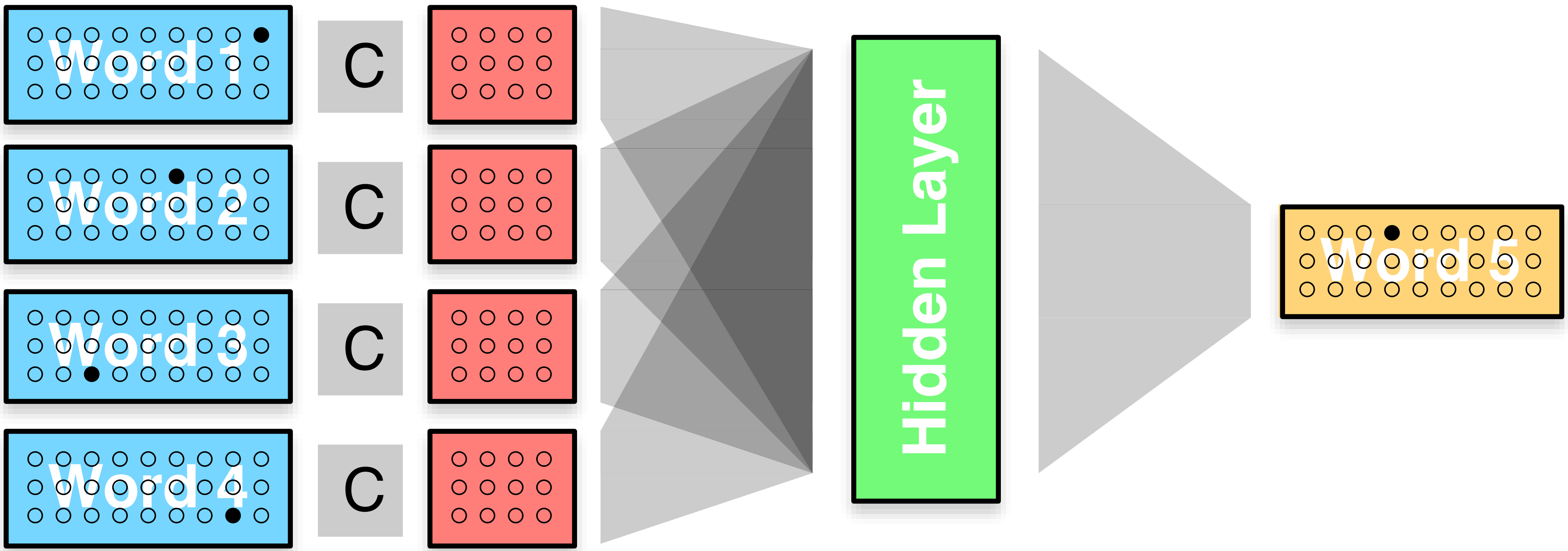
  $\rightarrow$ rare words share class with many other words
- Anything goes: assign words randomly to classes

Second Sketch



word embeddings

Add a Hidden Layer



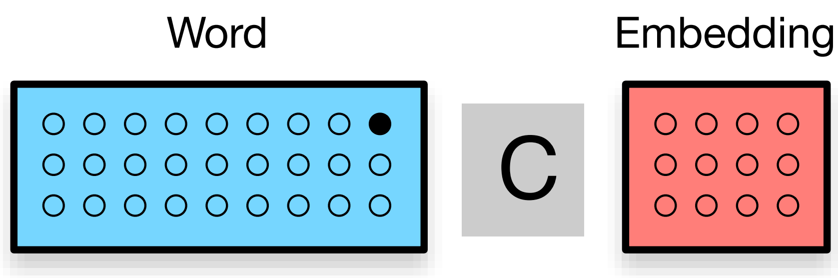
- Map each word first into a lower-dimensional real-valued space

- Shared weight matrix C

Details (Bengio et al., 2003)

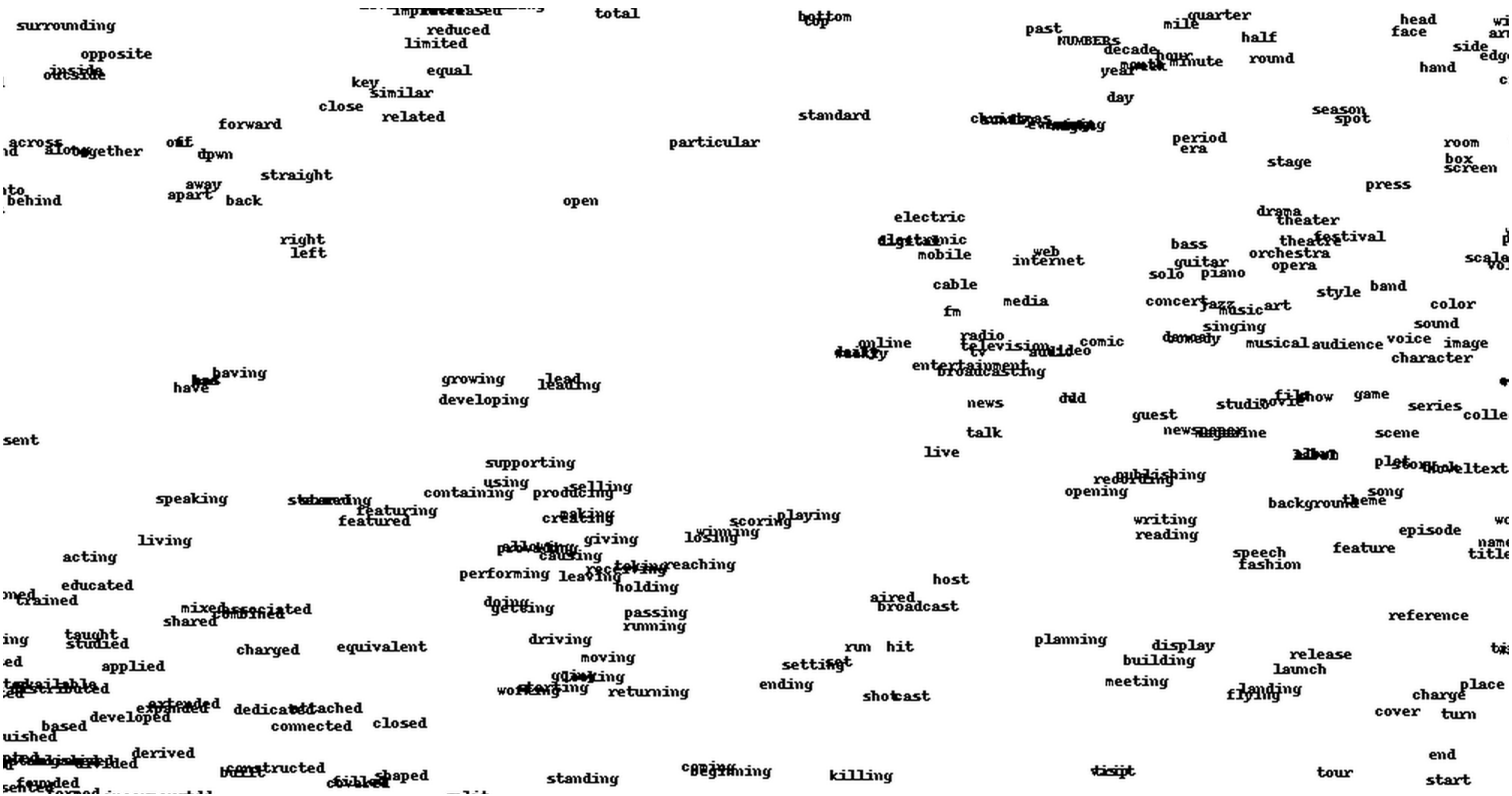
- Add direct connections from embedding layer to output layer
- Activation functions
  - input→embedding: none
  - embedding→hidden: tanh
  - hidden→output: softmax
- Training
  - loop through the entire corpus
  - update between predicted probabilities and 1-hot vector for output word

Word Embeddings

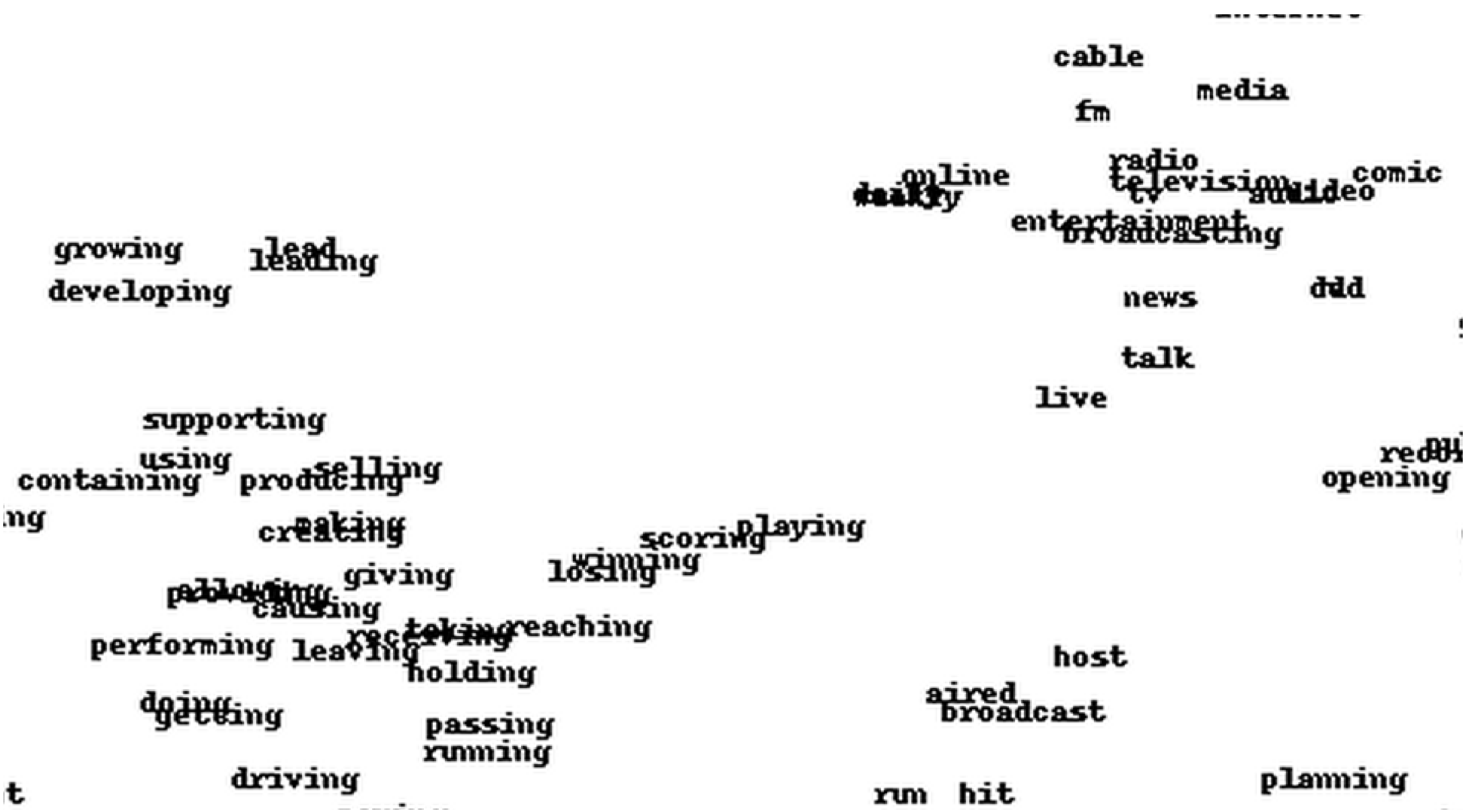


- By-product: embedding of word into continuous space
- Similar contexts $\rightarrow$ similar embedding
- Recall: distributional semantics

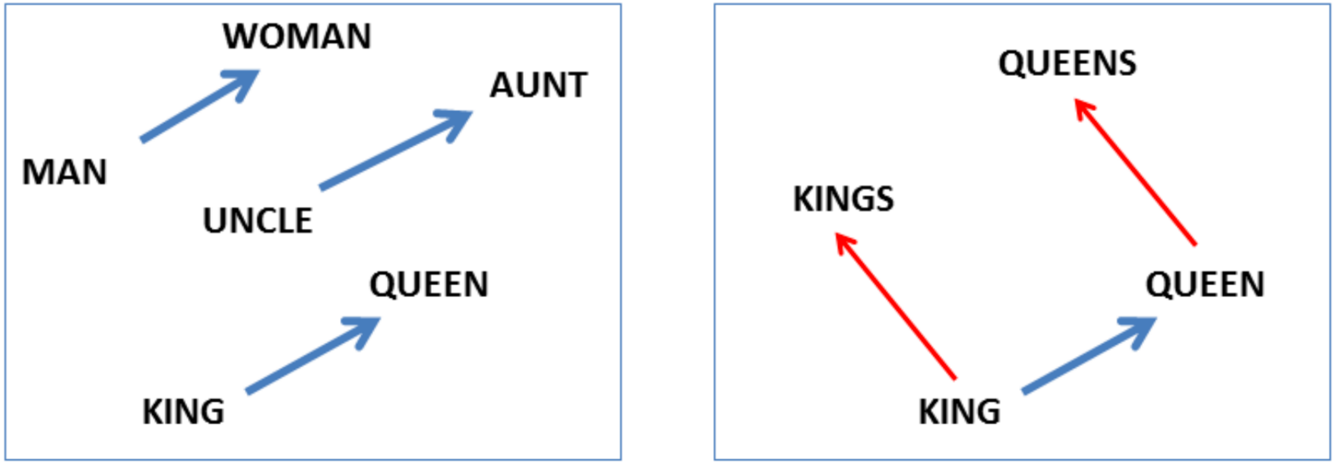
Word Embeddings



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

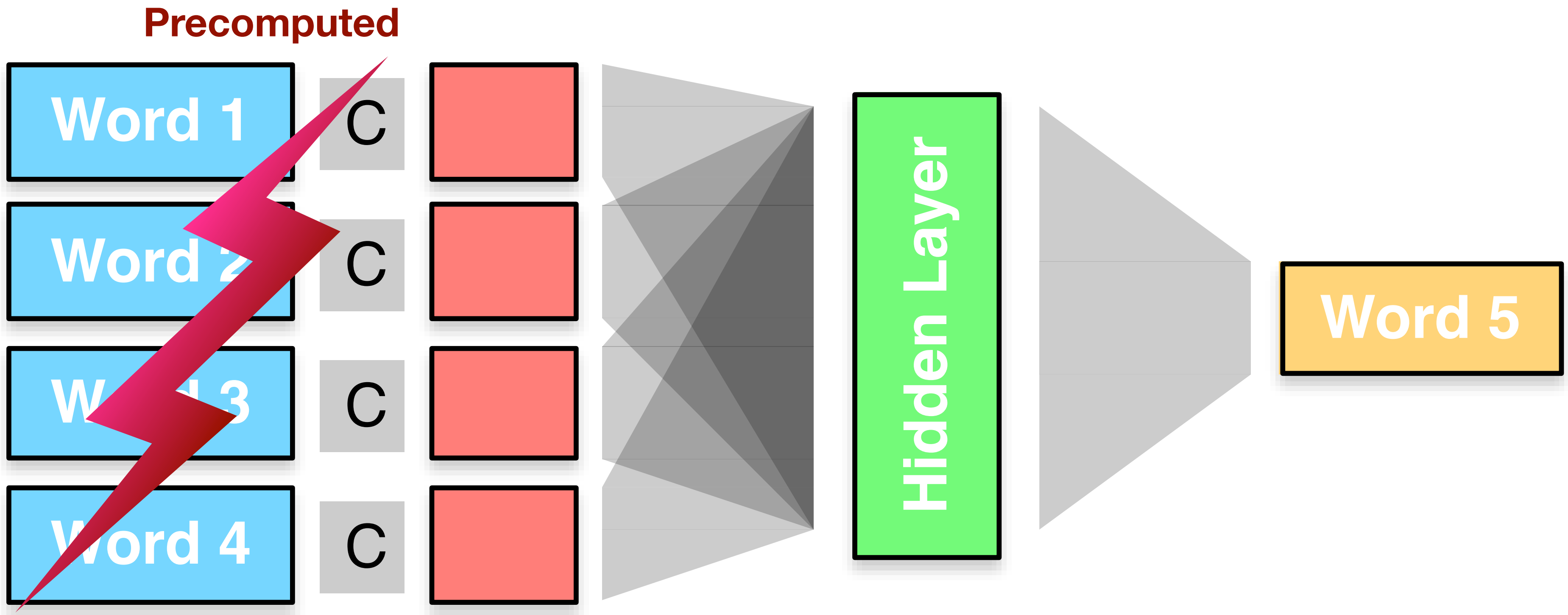
- Semantic regularities
  - clothing is to shirt as dish is to bowl
  - evaluated on human judgment data of semantic similarities

integration into machine translation systems

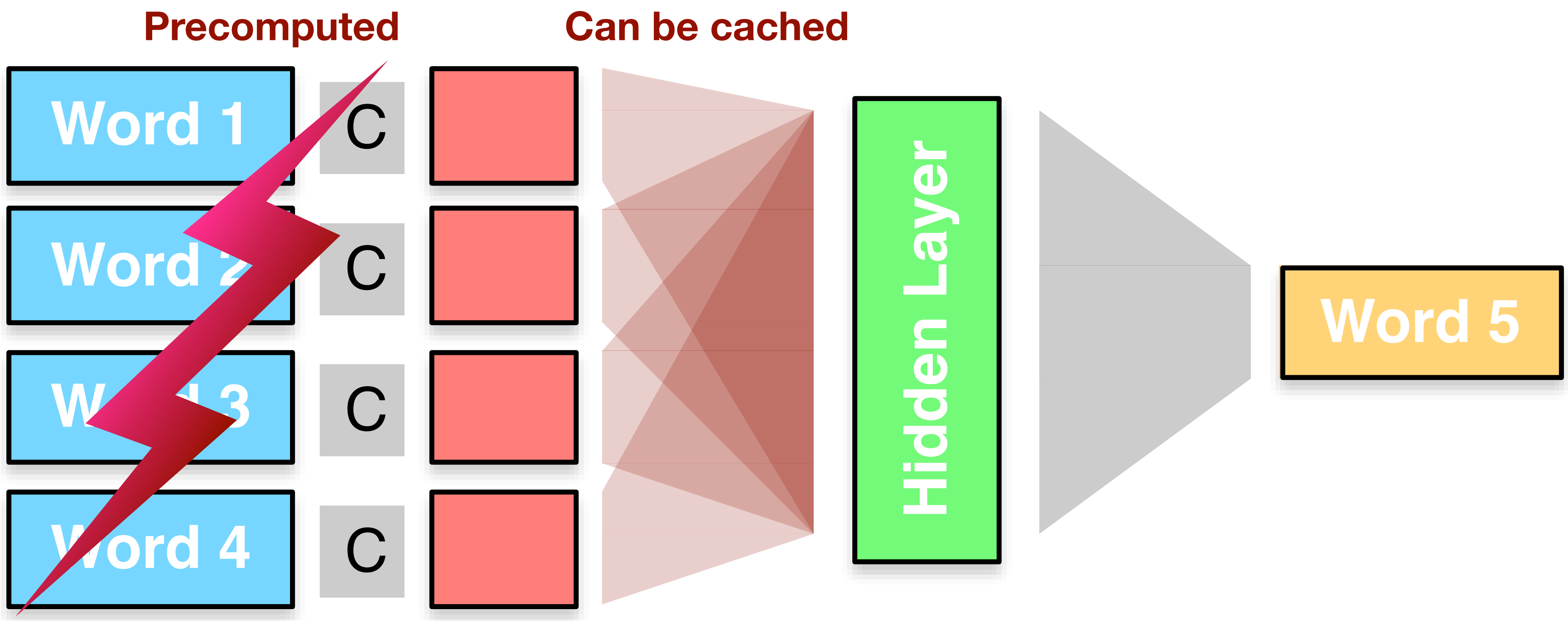
Reranking

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

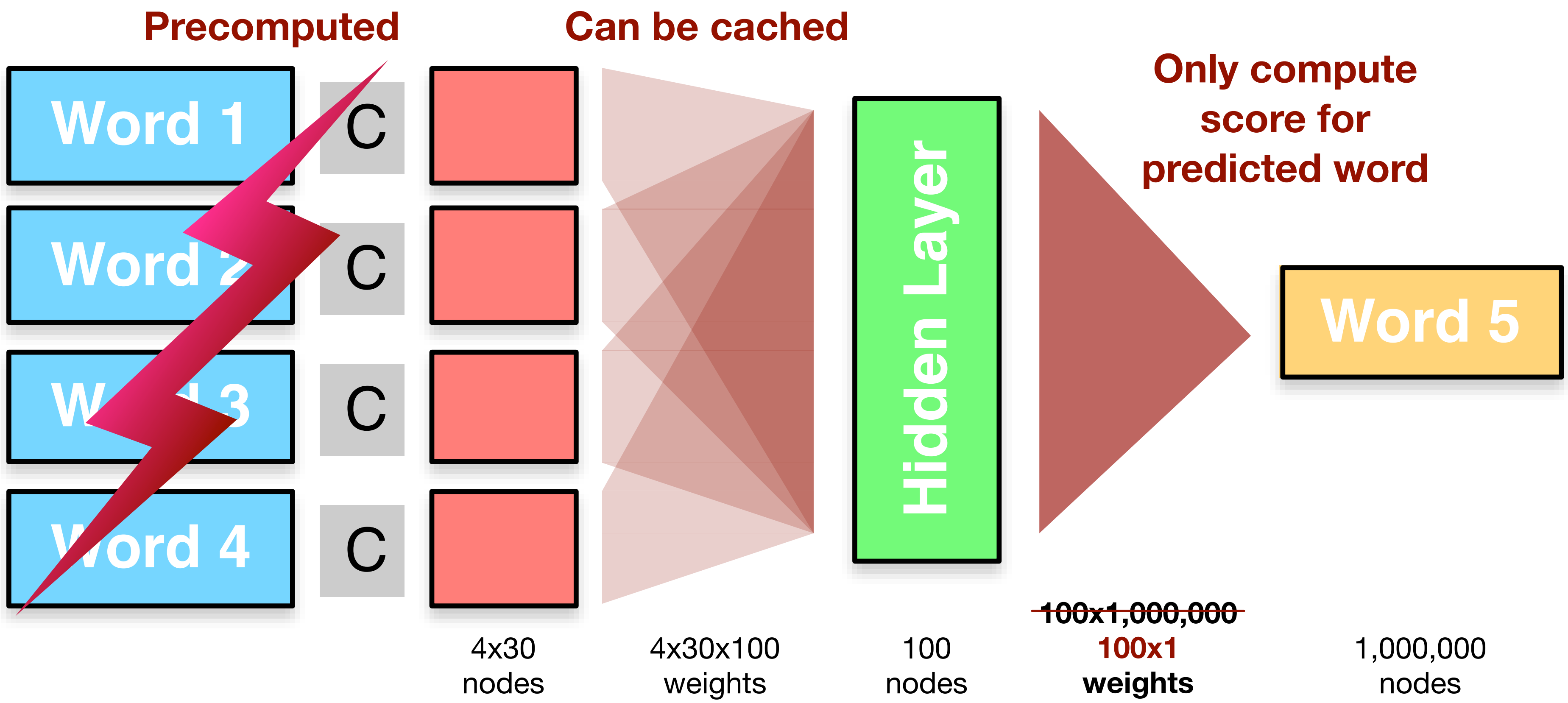
Computations During Inference



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)

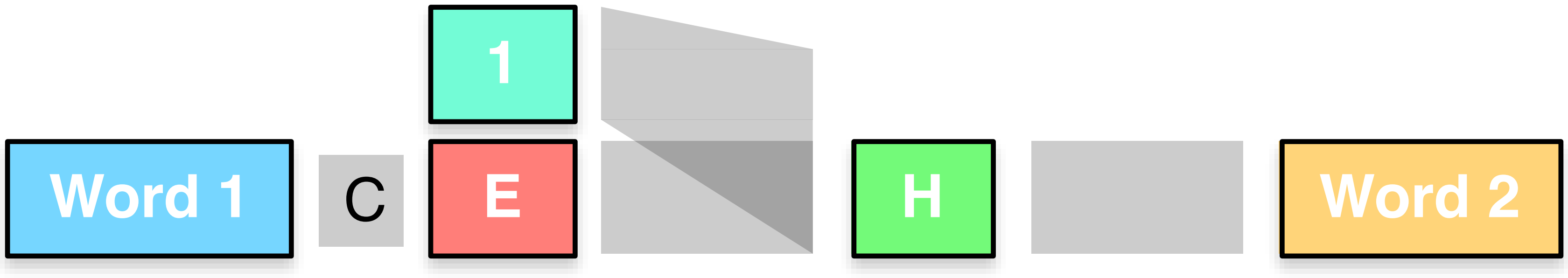
- How can we get away with it?
  - we do not care — a score is a score (Auli and Gao, 2014)
  - training regime that normalizes (Vaswani et al, 2013)
  - integrate normalization into objective function (Devlin et al., 2014)

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

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

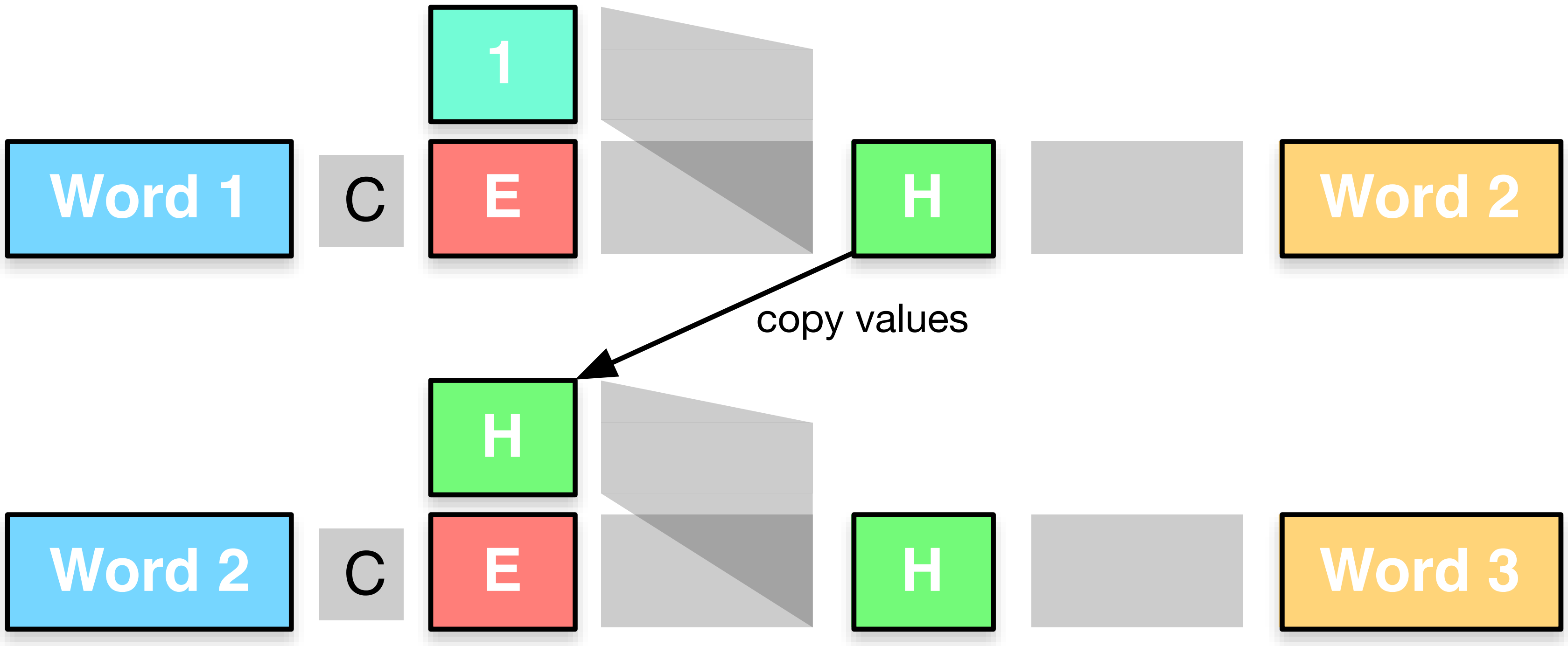
recurrent neural networks

Recurrent Neural Networks

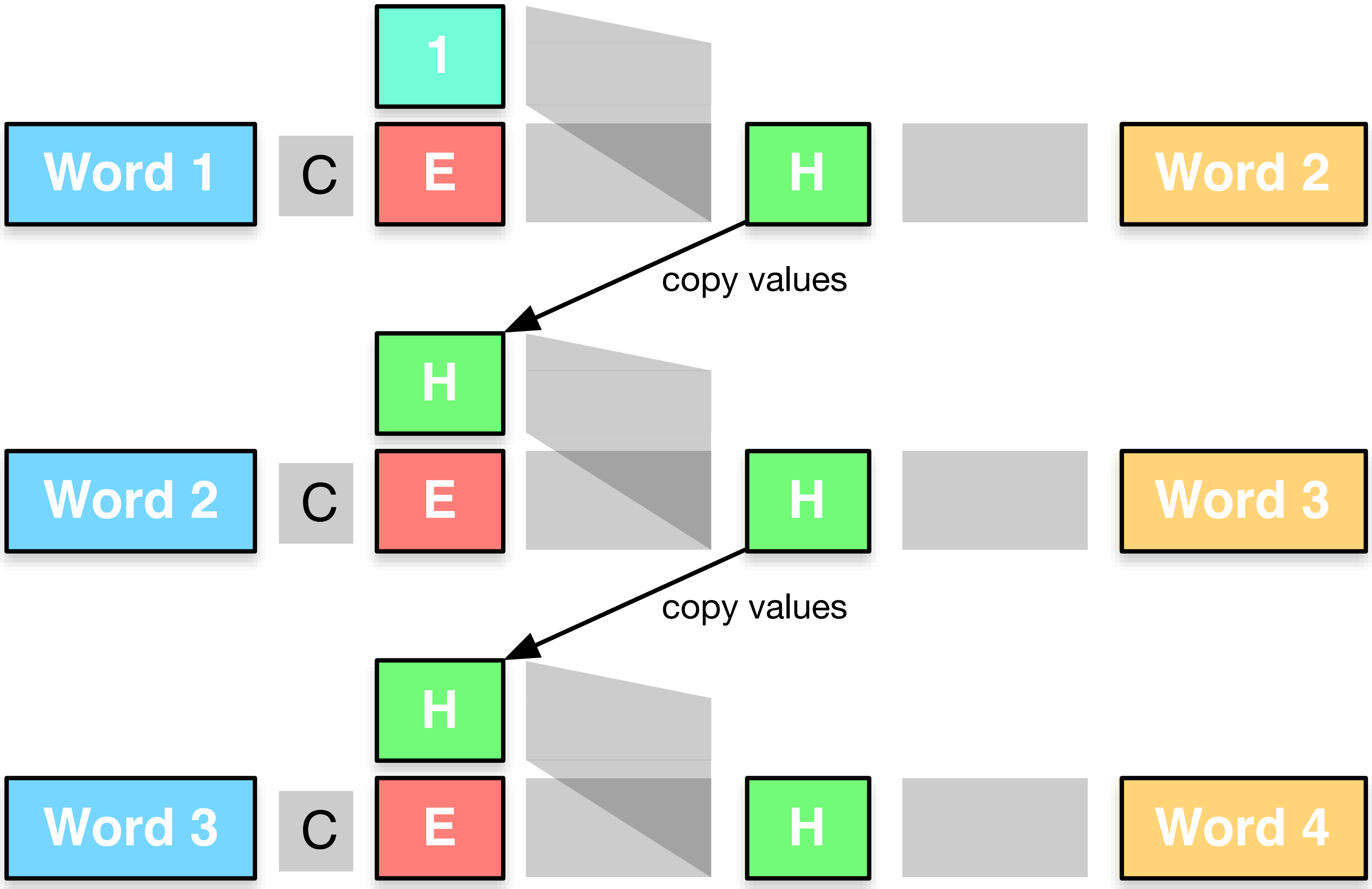


- Start: predict second word from first
- Mystery layer with nodes all with value 1

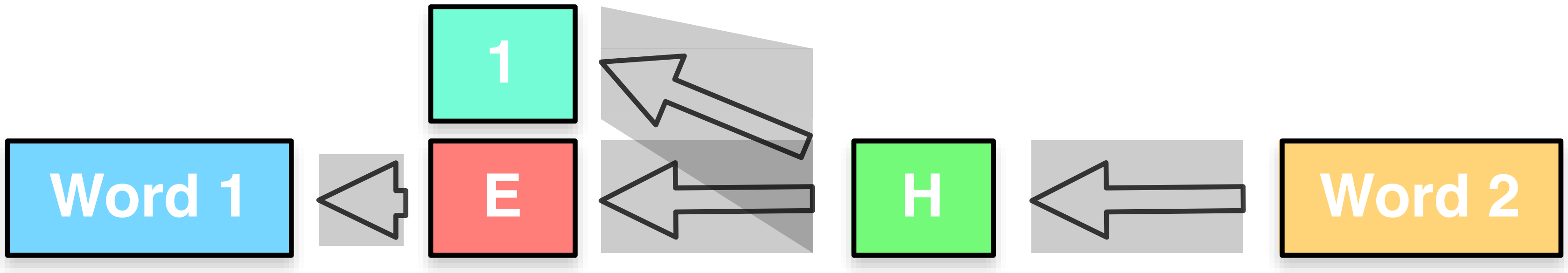
Recurrent Neural Networks



Recurrent Neural Networks

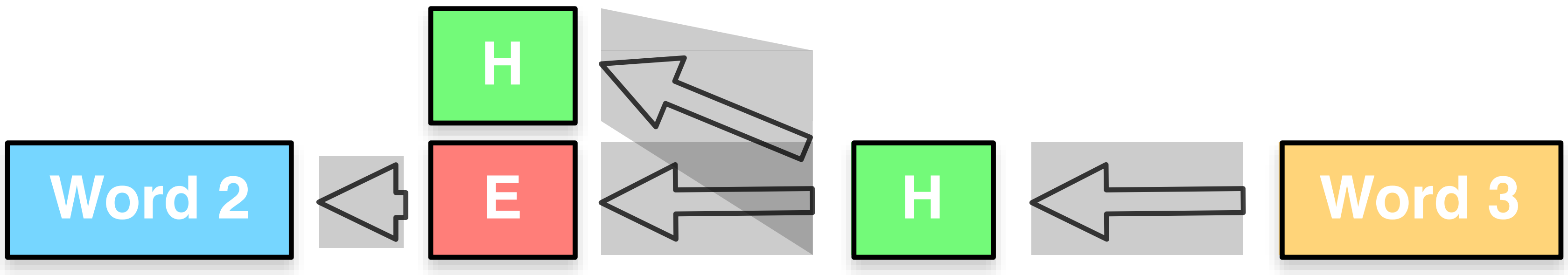


Training



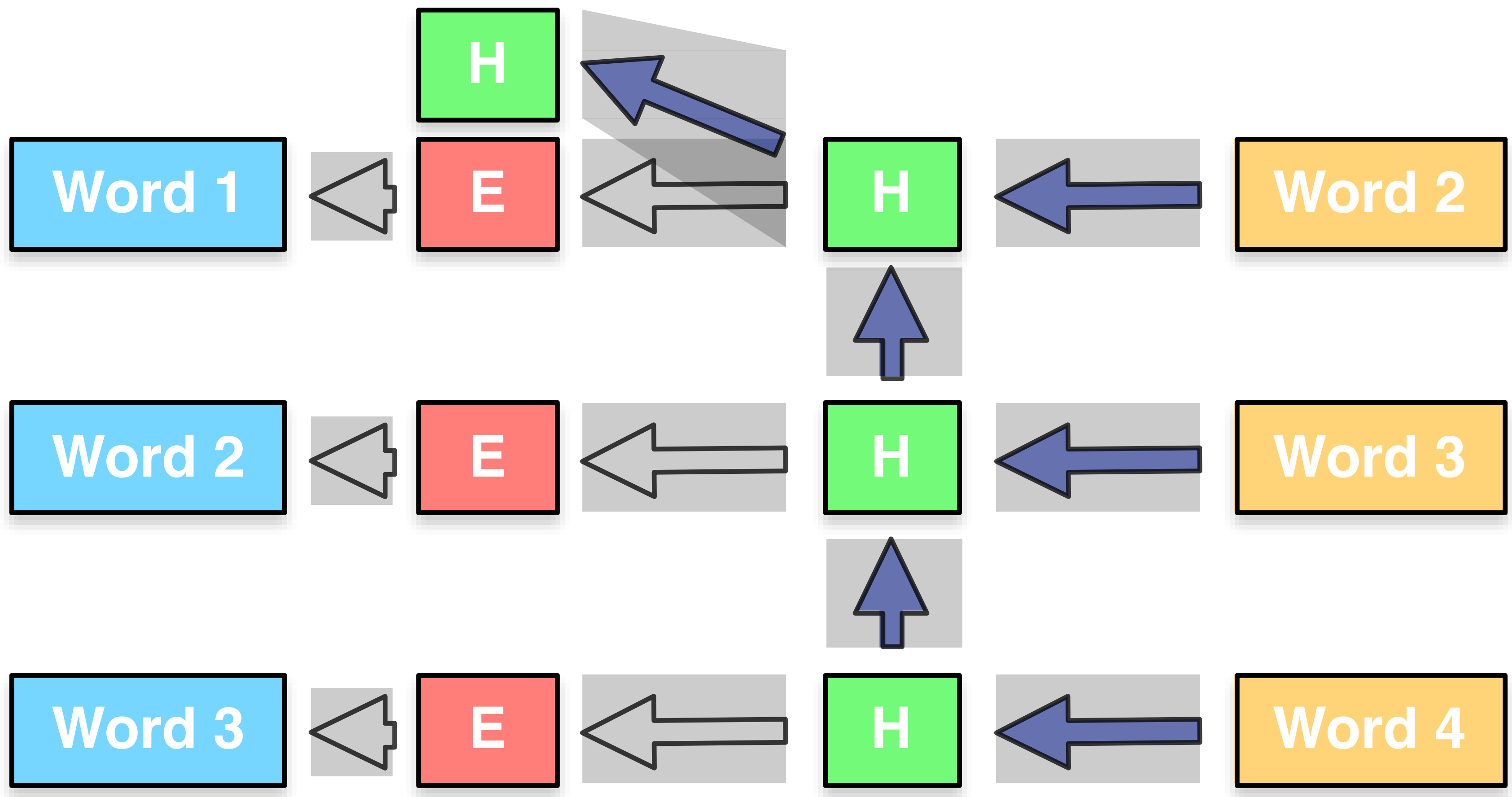
- Process first training example
- Update weights with back-propagation

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

$\Rightarrow$ very bad for dynamic programming

long short term memory

Vanishing and Exploding Gradients

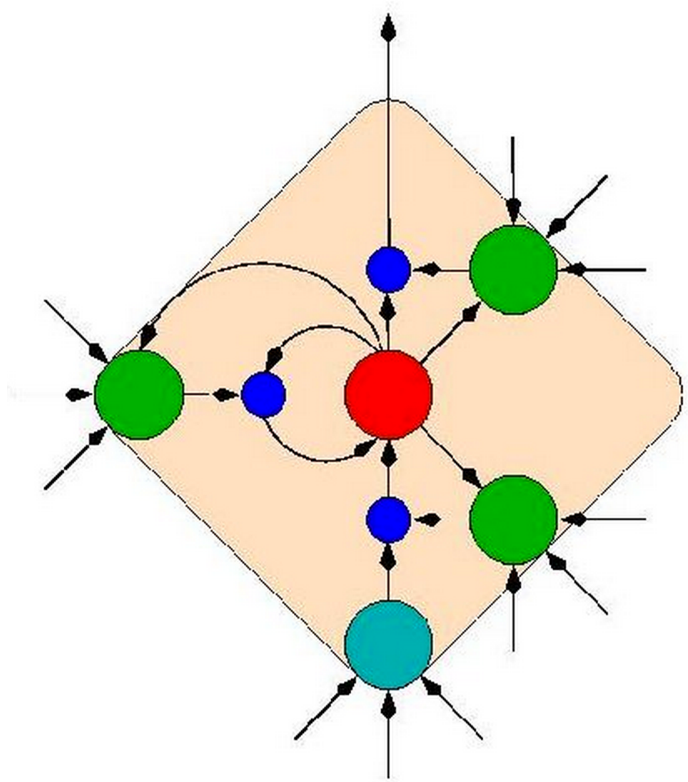
- Error is propagated to previous steps
- Updates consider
  - prediction at that time step
  - impact on future time steps
- Exploding gradient: propagated error dominates weight update
- Vanishing gradient: propagated error disappears

⇒ We want the proper balance

Long Short Term Memory (LSTM)

- Redesign of the neural network node to keep balance

- Rather complex



- ... but reportedly simple to train

Node in a Recurrent Neural Network

- Given
  - input word embedding $\vec{x}$
  - previous hidden layer values $\vec{h}^{(t-1)}$
  - weight matrices W and U
- Sum $s_i = \sum_j w_{ij} x_j + \sum_j u_{ij} h_j^{(t-1)}$
- Activation $y_i = \text{sigmoid}(s_i)$

Node ("Cell") in LSMT

- Now three gates: input, output, forget
  each with their own weight matrices: W_I, U_I, W_O, U_O, W_F, U_F
- Input and forget gates lead to activations as before
  $y_i^I = \text{sigmoid}(\sum_j w_{ij}^I x_j + \sum_j u_{ij}^I h_j^{(t-1)})$
  $y_i^F = \text{sigmoid}(\sum_j w_{ij}^F x_j + \sum_j u_{ij}^F h_j^{(t-1)})$
- Compute a candidate value for the "state" of the node (weight matrices W_C, U_C)
  $\tilde{C}_i^{(t)} = \tanh(\sum_j w_{ij}^C x_j + \sum_j u_{ij}^C h_j^{(t-1)})$
- Input and forget activations balance candidate state and previous state
  $C_i^{(t)} = y_i^I \; \tilde{C}_i^{(t)} + y_i^F \; C^{(t-1)}$
- Output gate also considers state (additional weight matrix V)
  $y_i^O = \text{sigmoid}(\sum_j w_{ij}^O x_j + \sum_j u_{ij}^O h_j^{(t-1)}) + \sum_j v_{ij} C_j^{(t)})$
- Output
  $h^{(t)} = y_i^O \; \tanh(C^{(t)})$