Neural sequence models

COMP90042 Lecture 12

Language models

- Assign a probability to a sequence of words
- Framed as "sliding a window" over the sentence, predicting each word from finite context to left

  E.g., n = 3, a trigram model

$$P(w_1, w_2, .. w_m) = \prod_{i=1}^{m} P(w_i | w_{i-2} w_{i-1})$$

- Training (estimation) from frequency counts
  - Difficulty with rare events → smoothing

LMs as classifiers

LMs can be considered simple classifiers, e.g. trigram model

$$P(w_i |\, w_{i-2} = \text{“}cow\text{”}, \mathrm{w}_{\mathrm{i}-1} = \text{“}eats\text{”})$$

classifies the likely next word in a sequence.

Has a parameter for every

$$w_{i-2}, \mathrm{w}_{\mathrm{i}-1}, w_i$$

Can think of this as a specific type of classifier — one with a simple parameterisation.

POS tagging as sequence classification

POS tagging can also be framed as classification:

$$P(t_i |\, w_{i-1} = \text{“}cow\text{”}, \mathrm{w_i} = \text{“}eats\text{”})$$

classifies the likely POS tag for “eats”.

Could use same parameterisation, with parameter for every

$$w_{i-1}, \mathrm{w_i}, t_i$$

- Why not use a fancier classifier? (Neural net)
- Can we make better use of context? (Recurrence)

Outline

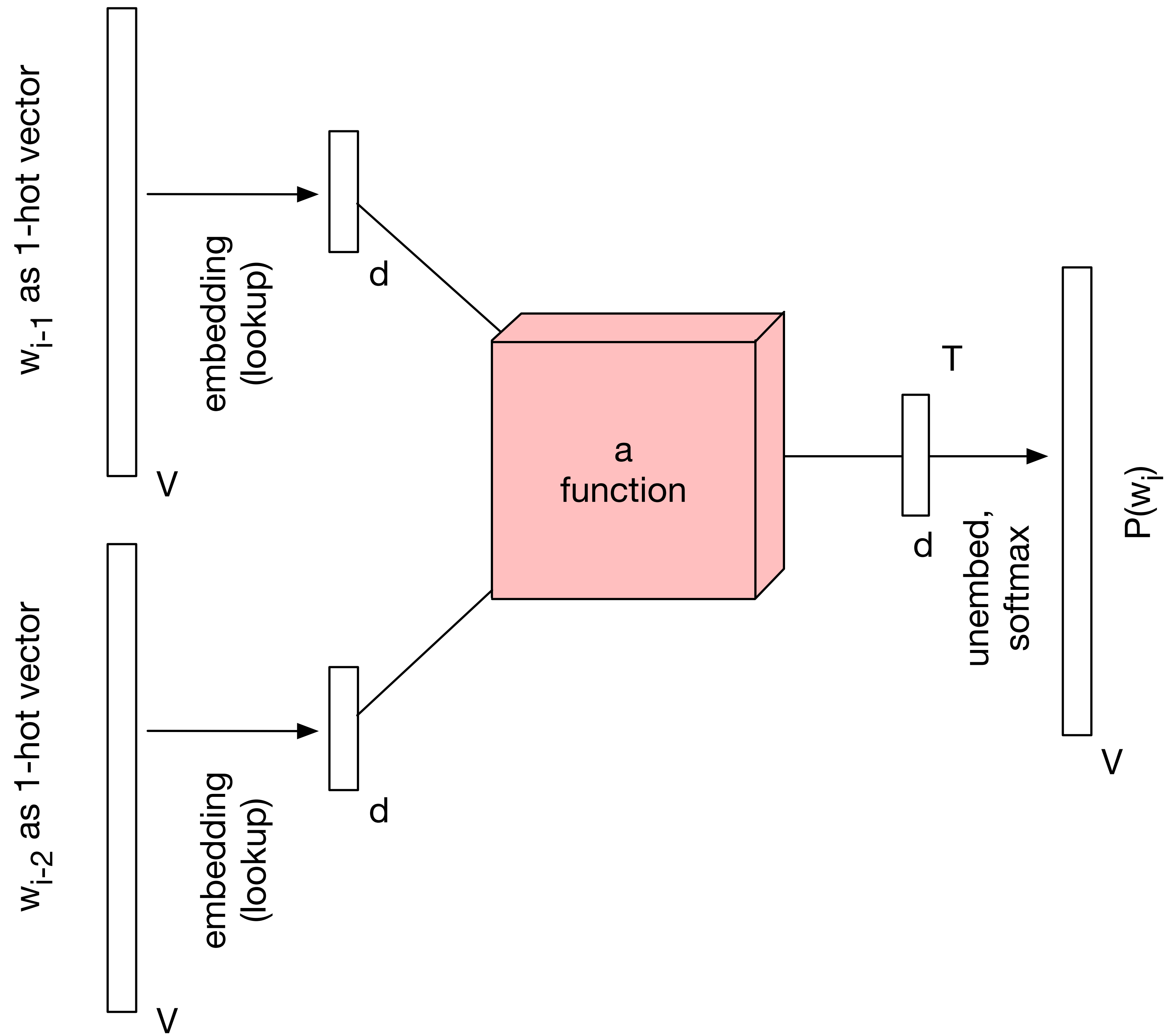
Neural network fundamentals

"Feed-forward" & recurrent neural language models

Feed forward neural net LMs

- Use neural network "classifier" to model $P(w_i|w_{i-2}w_{i-1})$
  - input features = the previous two words
  - output class = the next word
- How to handle massive space of V words? **Embeddings**!
  - embed input context words
  - transform in "hidden" space
  - "un-embed" back to vocab space
- Neural network used to define transformations

Feed forward neural net LM



Why bother?

- Ngram LMs
  - cheap to train (just compute counts)
  - but too many parameters, problems with sparsity and scaling to larger contexts
  - don't adequately capture properties of words (grammatical and semantic similarity), e.g., film vs movie
- NNLMs more robust
  - force words through low-dimensional embeddings
  - automatically capture word properties, leading to more robust estimates
  - flexible: minor change to adapt to other tasks (tagging)

Neural networks

"Deep" neural networks provide mechanism for learning richer models.

Based on **vector** *embeddings* and compositional functions over these vectors.

- Word embeddings capture grammatical and semantic similarity "cows" ~ "sheep", "eats" ~ "chews" etc.
- Vector composition can allow for combinations of features to be learned (e.g., humans consume meat)
- Limit size of vector representation to keep model capacity under control.

Components of NN classifier

- NN = Neural Network
  - a.k.a. artificial NN, deep learning, multilayer perceptron
- Composed of simple functions of vector-valued inputs

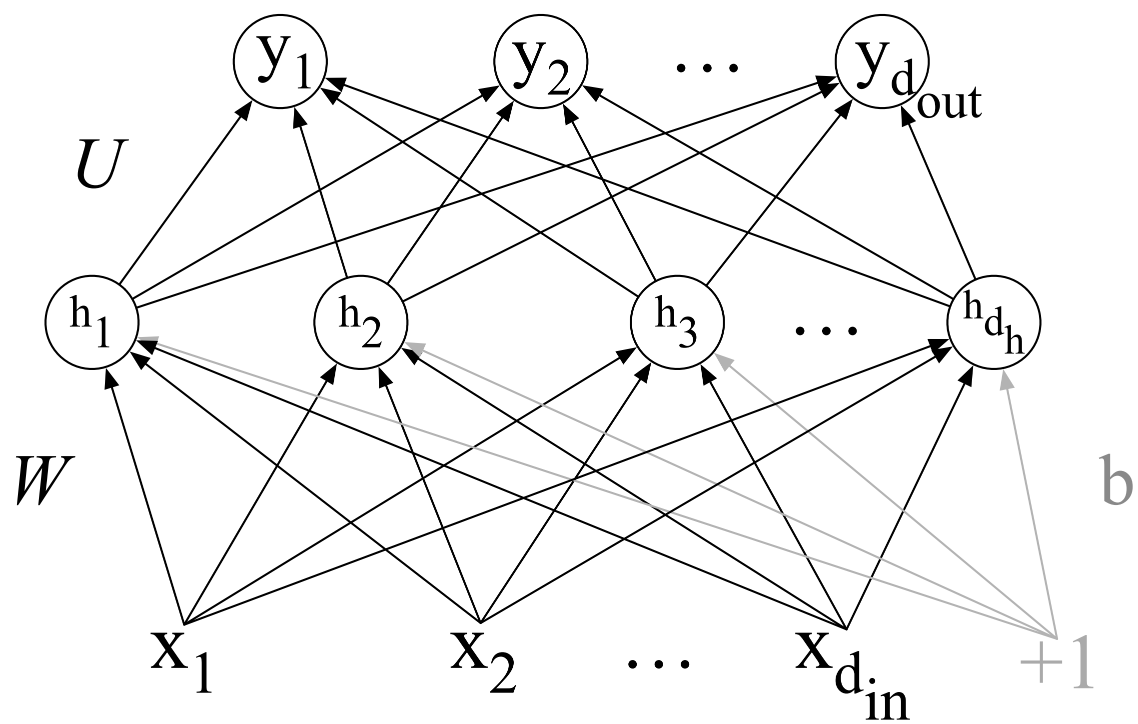


Fig JM3 Ch 8

NN Units

- Each "unit" is a function
  - given input x, computes real-value (scalar) h

$$h = \tanh\left(\sum_j w_j x_j + b\right)$$

  - scales input (with weights, w) and adds offset (bias, b)
  - applies non-linear function, such as logistic sigmoid, hyperbolic sigmoid (tanh), or rectified linear unit

Neural network components

- Typically have several hidden units, i.e.,

$$h_i = \tanh\left(\sum_j w_{ij} x_j + b_i\right)$$

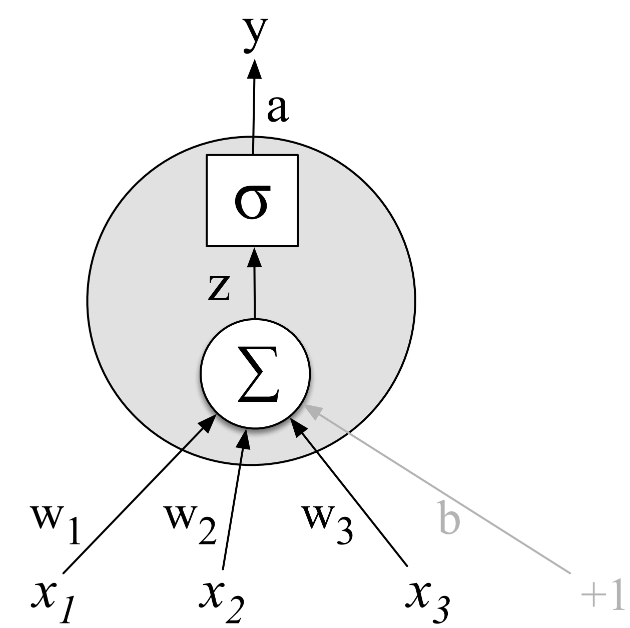
  - each with own weights (w_i) and bias term (b_i)
  - can be expressed using matrix & vector operators

$$\vec{h} = \tanh\left(W\vec{x} + \vec{b}\right)$$

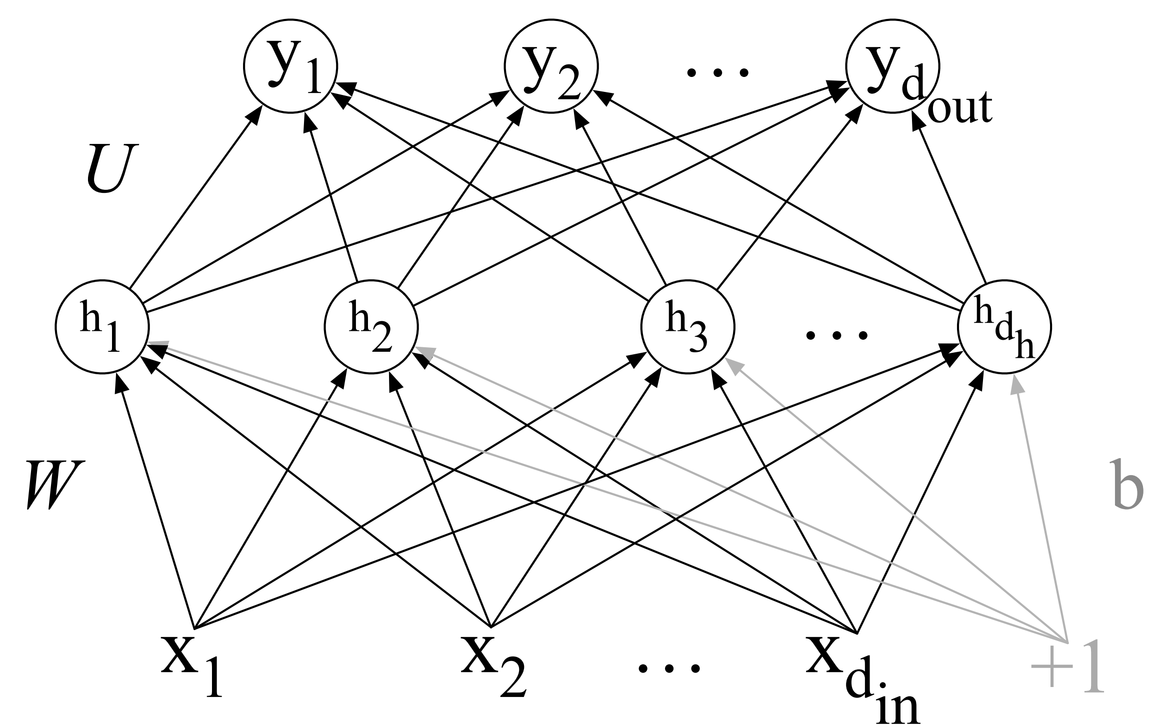
  - where W is a matrix comprising the unit weight vectors, and b is a vector of all the bias terms
  - non-linear function applied element-wise

ANN in pictures

- Pictorial representation of a single unit, for computing y from x



- Typical networks have several units, and additional layers
- E.g., output layer, for classification target



Figs JM3 Ch 8

Coupling the Output layer

- To make this into a classifier, need to produce a classification output
  - probabilities for the next word (of size |V|)
- Add another layer, which takes *h* as input, and maps into |V| sized vector
- Softmax ensures probabilities >0 & sum to 1

$$\left[\frac{\exp(v_1)}{\sum_i \exp(v_i)}, \quad \frac{\exp(v_2)}{\sum_i \exp(v_i)}, \quad \ldots \quad \frac{\exp(v_m)}{\sum_i \exp(v_i)}\right]$$

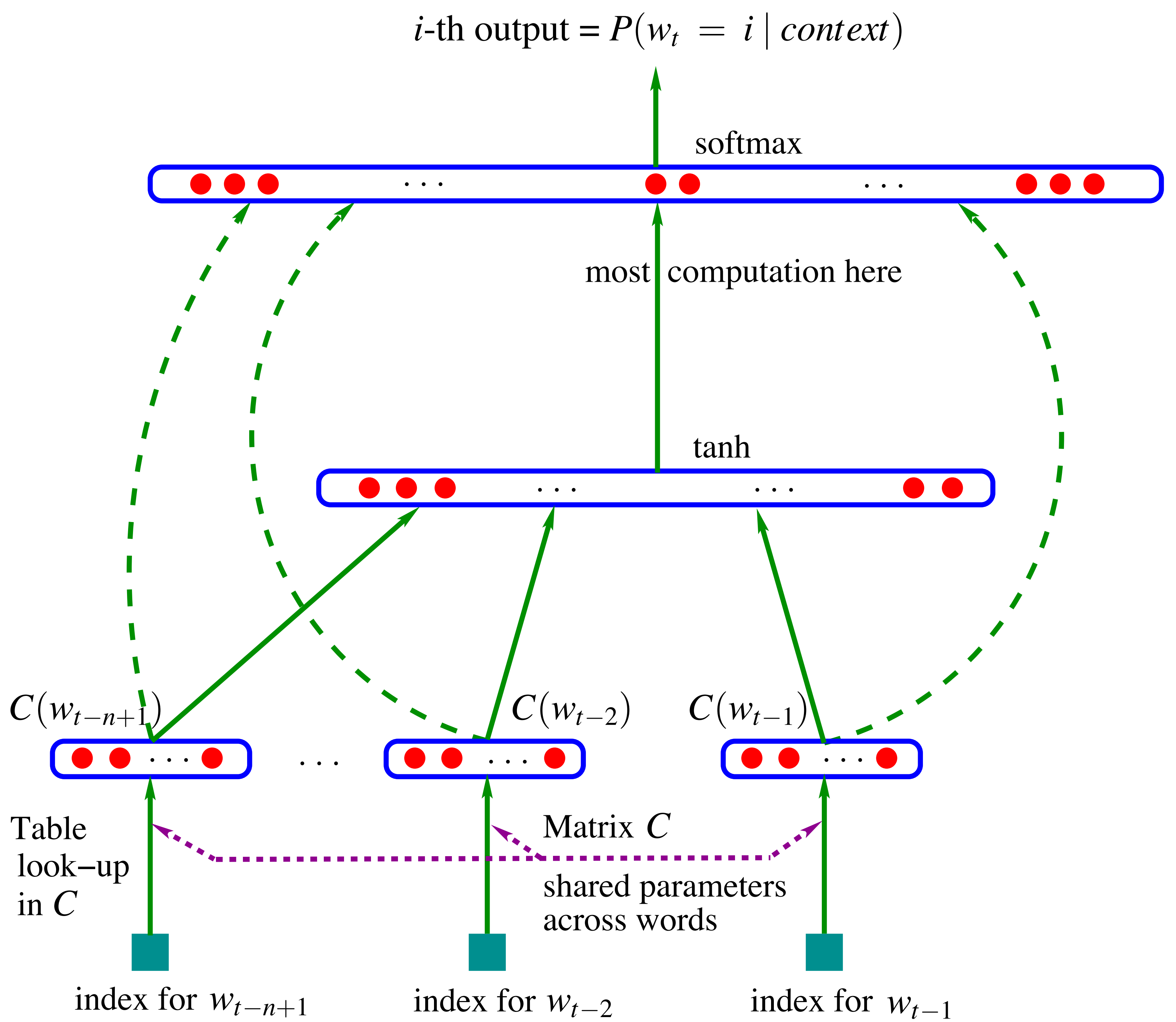
Deep structures

- Can stack several hidden layers; e.g.,
  1. map from 1-hot words, *w*, to word embeddings, ***e*** (lookup)
  2. transform ***e*** to hidden state $\boldsymbol{h}_1$ (with non-linearity)
  3. transform $\boldsymbol{h}_1$ to hidden state $\boldsymbol{h}_2$ (with non-linearity)
  4. *… repeat …*
  5. transform $\boldsymbol{h}_n$, to output classification space ***y*** (with softmax)
- Each layer typically fully-connected to next lower layer, i.e., each unit is connected to all input elements

Learning from Data

- How to learn the parameters from data?
  - parameters = sets of weights, bias, embeddings
- Consider how well the model "fits" the training data, in terms of the probability it assigns to the correct output
  - *e.g.*, $L = \prod_{i=1}^{m} P(w_i | w_{i-2} w_{i-1})$
  - want to *maximise* total probability, L
  - equivalently *minimise* -log L with respect to parameters
- Trained using gradient descent
  - tools like *tensorflow, pytorch, dynet* use autodiff to compute gradients automatically

FF-NN-LM

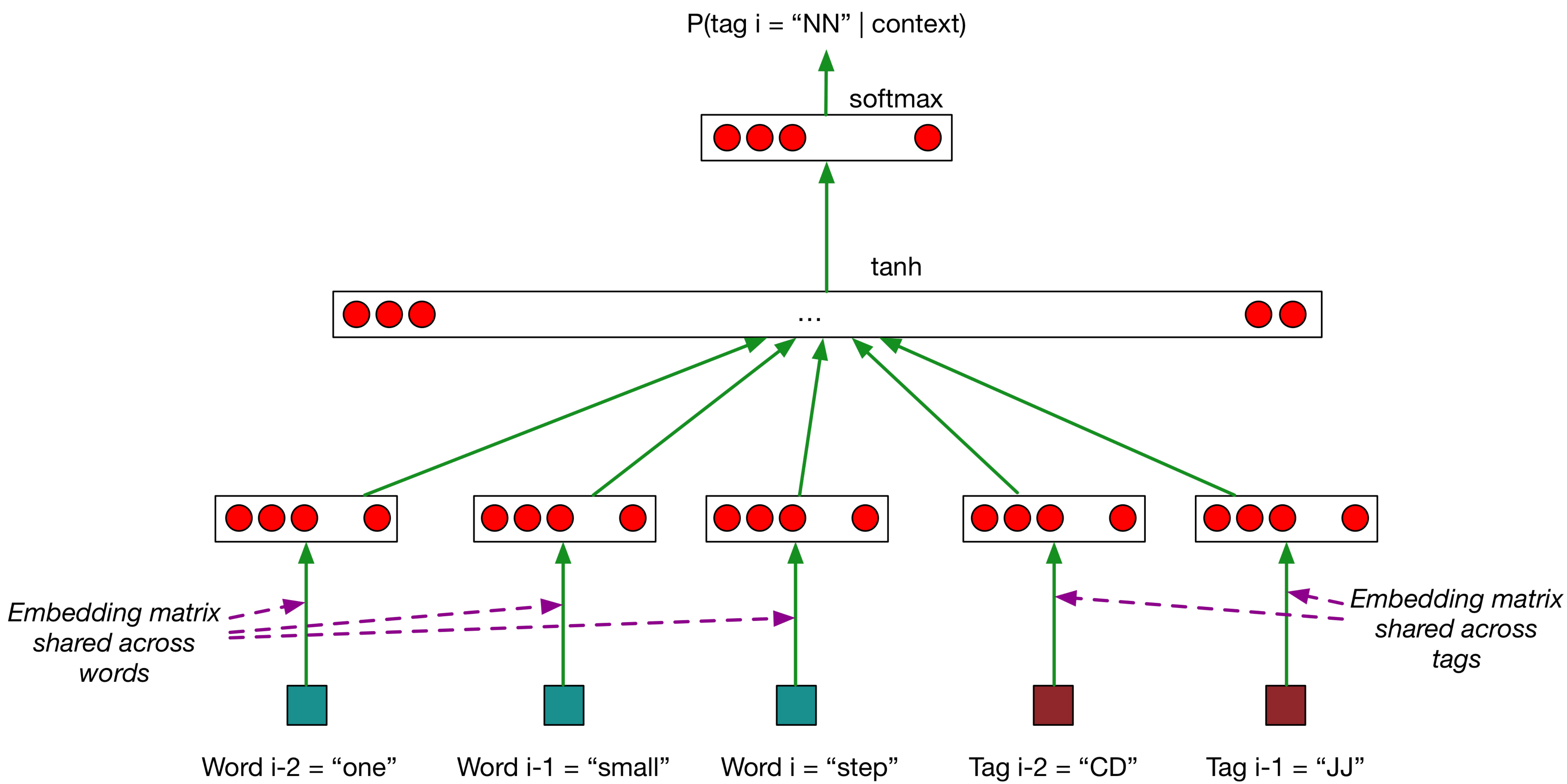


Bengio et al, 2003

FF-NN for Tagging

- MEMM tagger takes as input:
  - recent words w_{i-2}, w_{i-1}, w_i
  - recent tags t_{i-2}, t_{i-1}
- And outputs: current tag t_i
- Frame as neural network with
  - 5 inputs: 3 x word embeddings and 2 x tag embeddings
  - 1 output: vector of size |T|, using softmax
- Train to minimise
  $-\sum_i \log P(\, t_i | w_{i-2}, w_{i-1}, w_i, t_{i-2}, t_{i-1})$

FF-NN for tagging



Recurrent NNLMS

- What if we structure the network differently, e.g., according to sequence with Recurrent Neural Networks (RNNs)

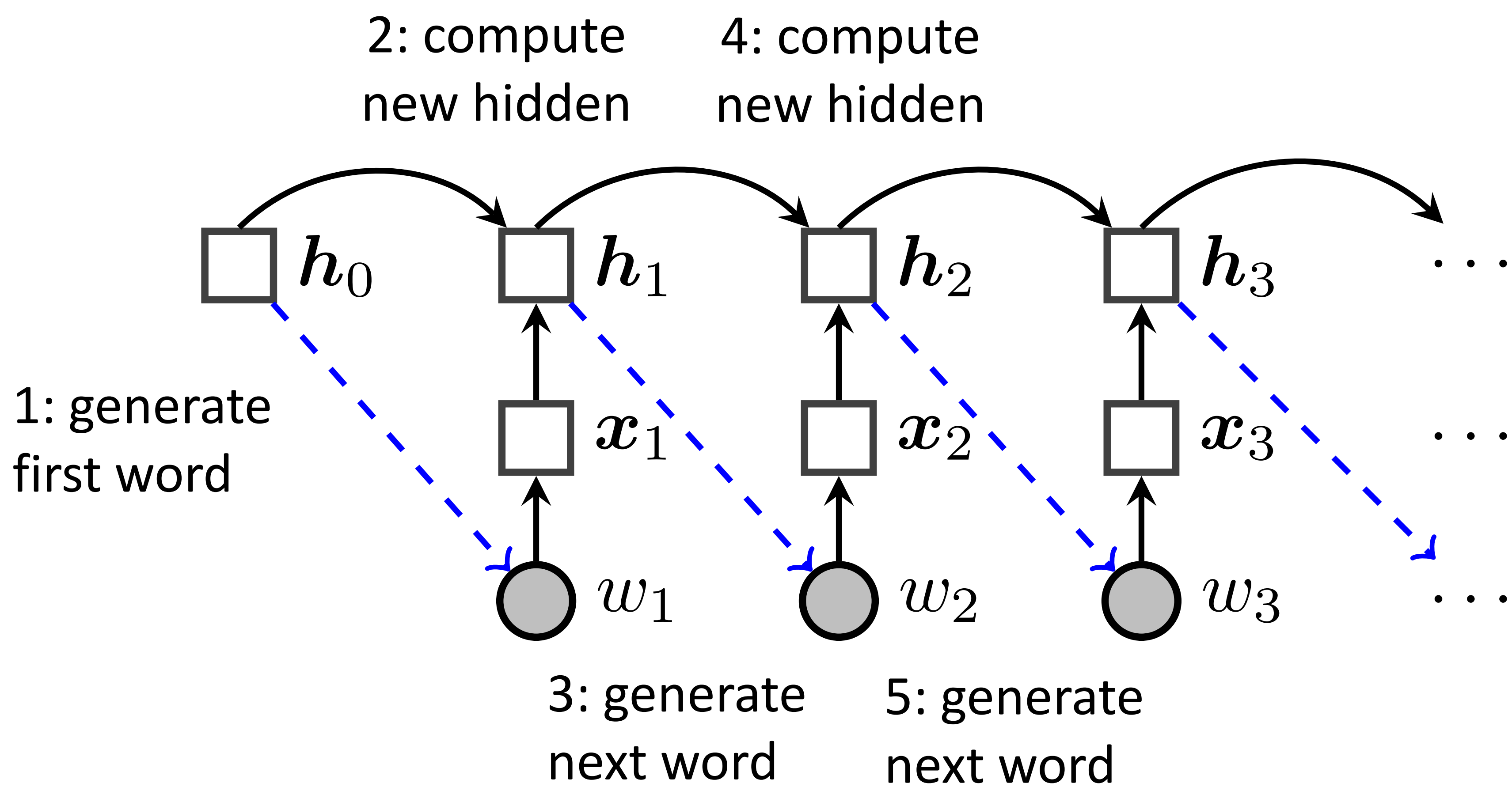


Figure: E18, p 154

Recurrent NNLMS

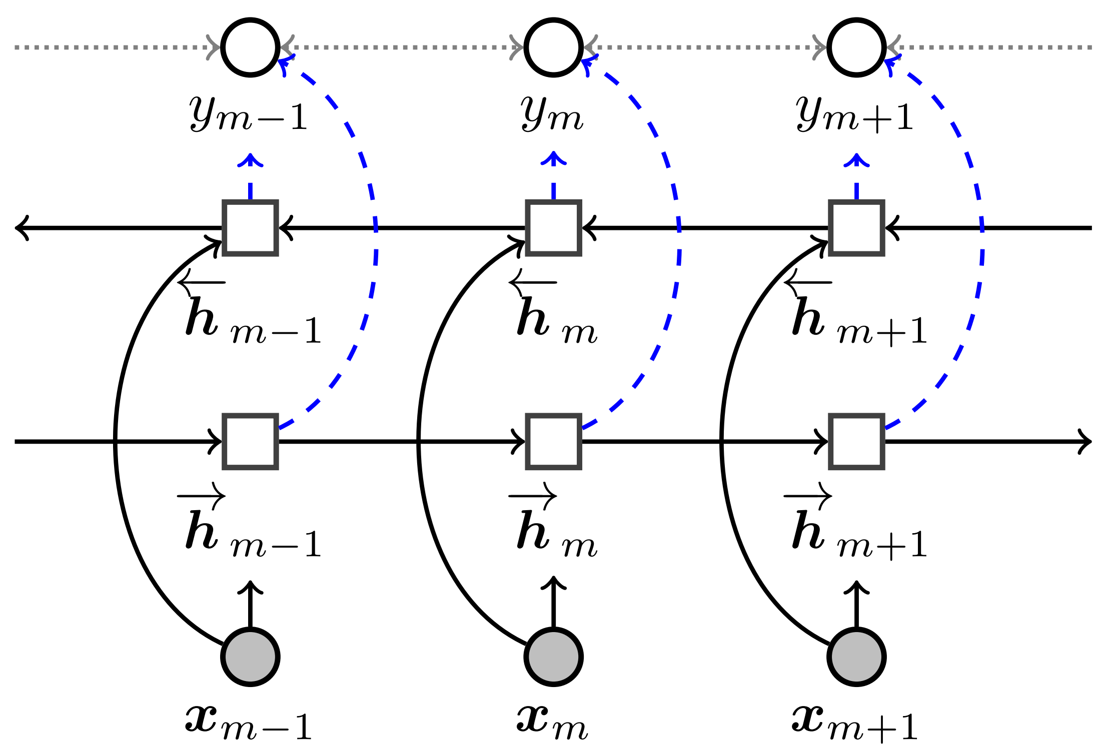
- Start with
  - initial hidden state $\boldsymbol{h}_0$
- For each word, w_i, in order $i=1..m$
  - embed word to produce vector, $\boldsymbol{e}_i$
  - compute hidden $\boldsymbol{h}_i = \tanh(W\ \boldsymbol{e}_i + V\ \boldsymbol{h}_{i-1} + \boldsymbol{b})$
  - compute output $\mathrm{P}(\boldsymbol{w}_{i+1}) = \mathrm{softmax}(U\ \boldsymbol{h}_i + \boldsymbol{c})$
- Train such to minimise $\sum_i -\log \mathrm{P}(\boldsymbol{w}_i)$
  - to learn parameters W, V, U, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{h}_0$
- Adapt to tagging, e.g., using two RNNs, one for words and one for tags; and tags as outputs

RNNs

- Can results in very “deep” networks,
  - great for capturing long fragments: in theory, no limit on context
  - difficult to train due to gradient explosion or vanishing
- Variant RNNs designed to behave better:
  Gated Recurrent Units (GRU),
  Long Short-Term Memory (LSTM)
- High computational cost with many classes (e.g., #vocab)
  - negative sampling or hierarchical softmax over outputs

Bidirectional RNNS

- Tagging can be benefit from context to left and right
  - easy: use two RNNs, left-to-right and right-to-left



- Often used as word encoding in other tasks, e.g., POS, translation, summarisation, sentence classification

Final words

- NNet models
  - Robust to word variation, typos, etc
  - Excellent generalization, especially RNNs
  - Flexible — forms the basis for many other models
- Cons
  - Much slower than counts… but GPU acceleration
  - Lots of classes (e.g., vocabulary)
  - Not good for rare words… but pre-training on big corpora
  - Data hungry, not so good on tiny data sets

Required Reading

- E18, 6.3 (skip 6.3.1), 7.6