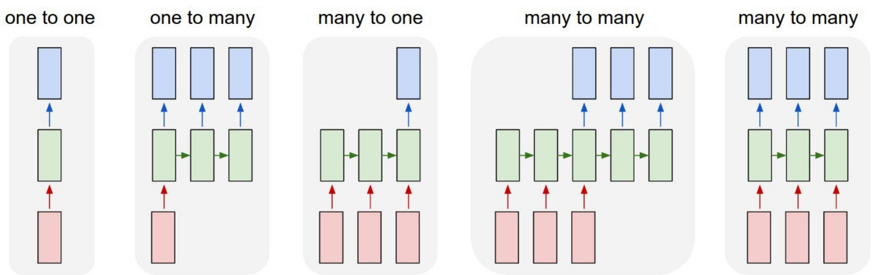
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

Recurrent neural networks

Recurrent neural networks

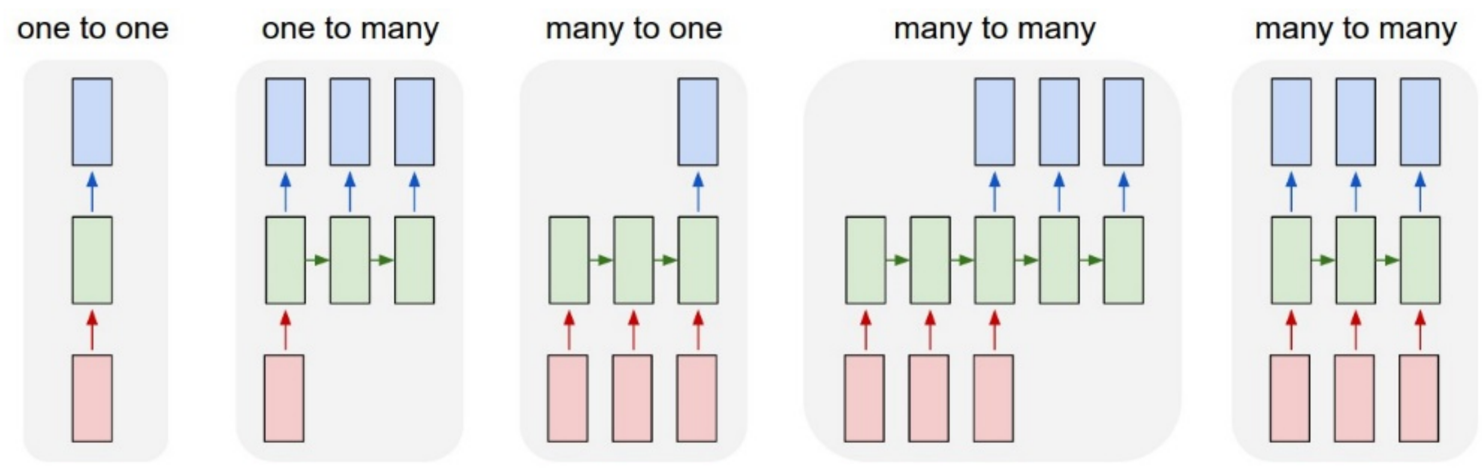
- Lots of information is sequential and requires a memory for successful processing
- Sequences as input, sequences as output



- Recurrent neural networks(RNNs) are called recurrent because they perform same task for every element of sequence, with output dependent on previous computations
- RNNs have memory that captures information about what has been computed so far
- RNNs can make use of information in arbitrarily long sequences – in practice they limited to looking back only few steps

Img Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Topologies of Recurrent Neural Network



(1) (2) (3) (4) (5)

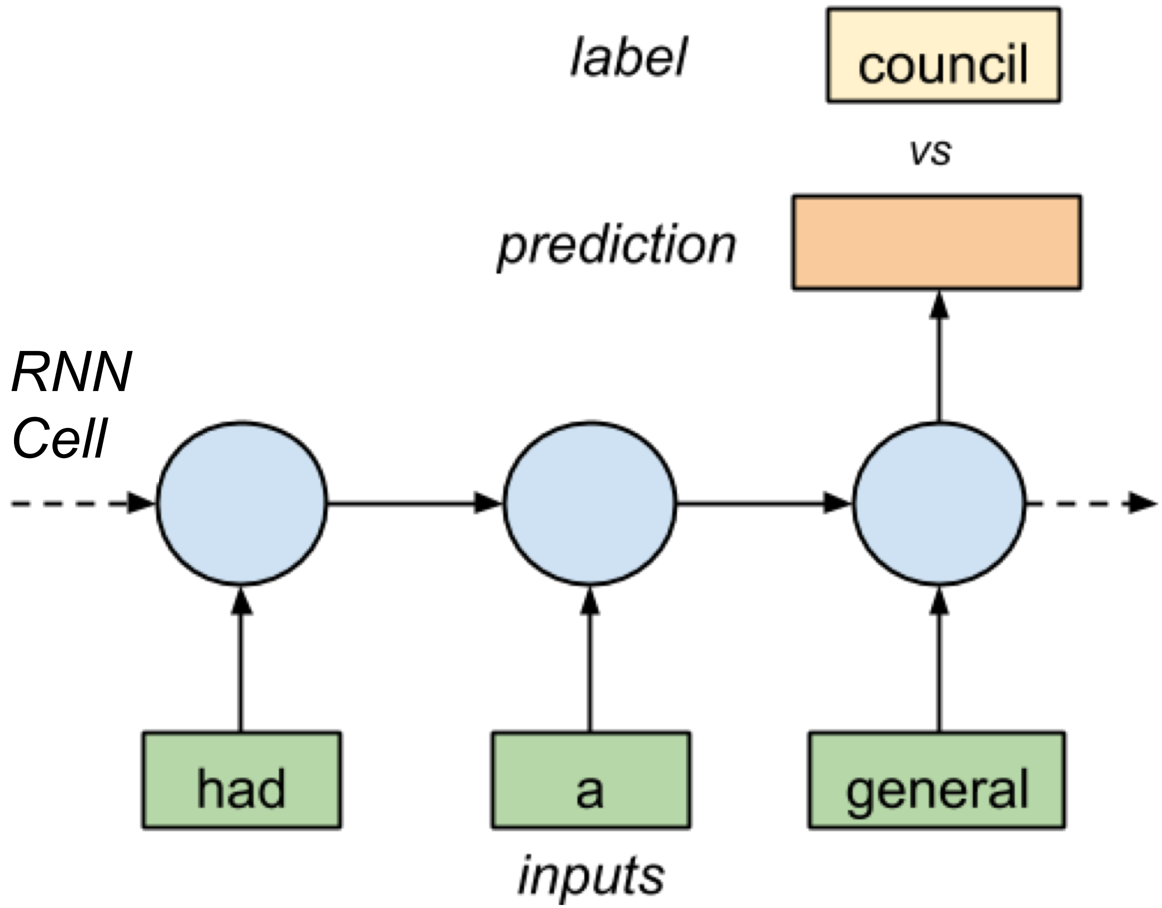
1) Common Neural Network (e.g. feed forward network)
2) Prediction of future states base on single observation
3) Sentiment classification
4) Machine translation
5) Simultaneous interpretation

Language Model

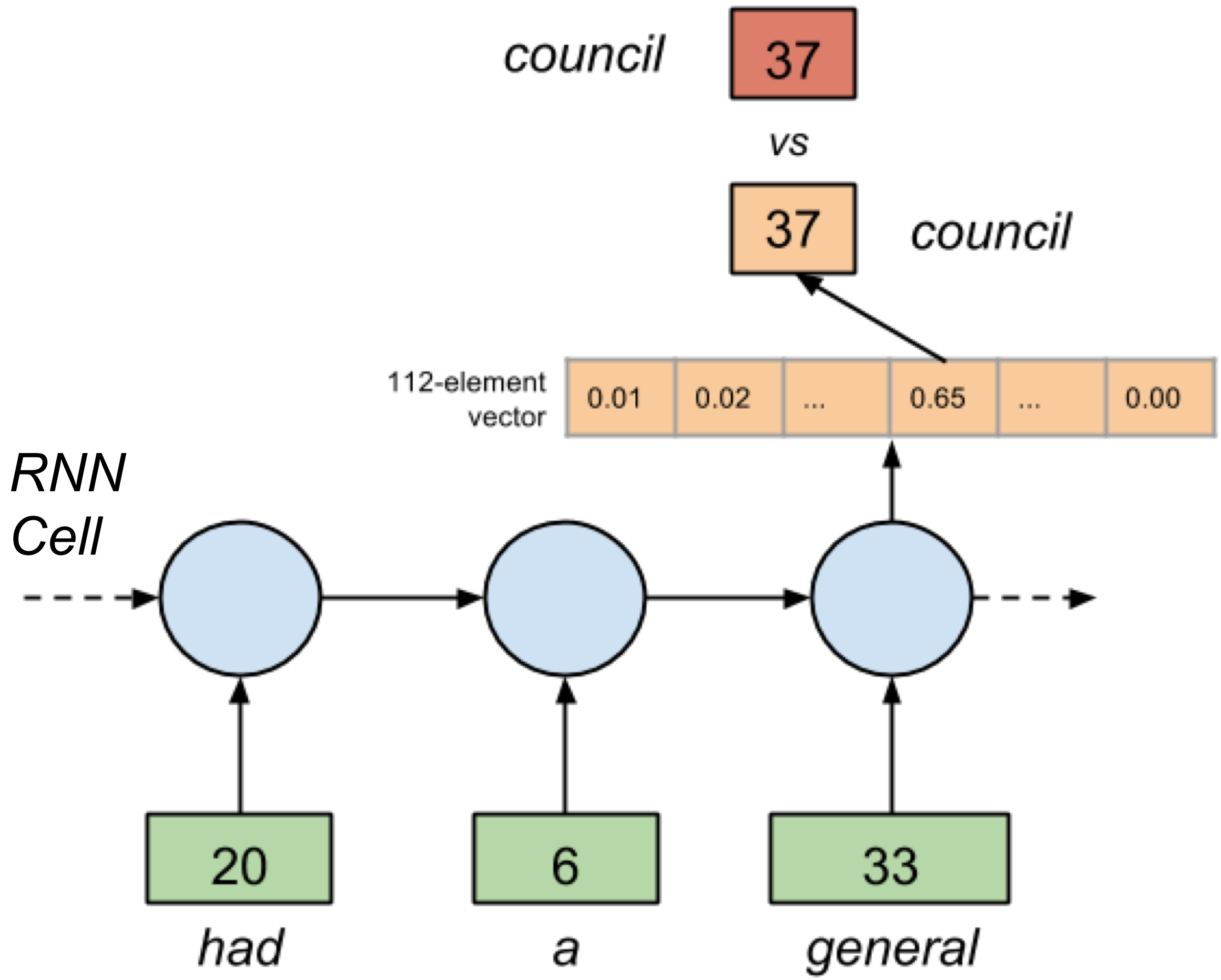
- Compute the probability of a sentence

- Useful in machine translation
  - Word ordering: p(the cat is small) > p(small the cat is)
  - Word choice: p(walking home after school) > p(walking house after school)

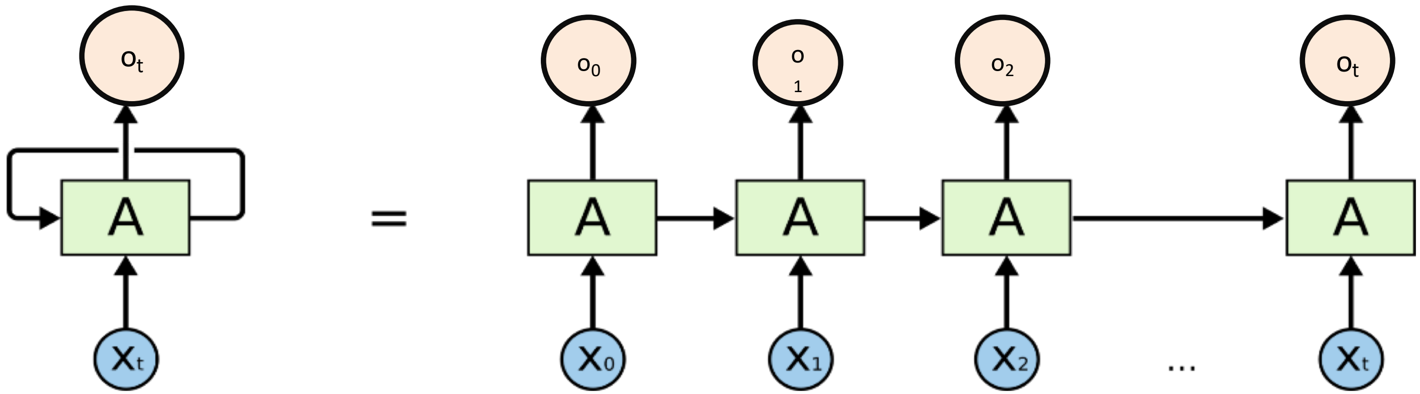
Recurrent Neural Network



Recurrent Neural Network



Recurrent Neural Network

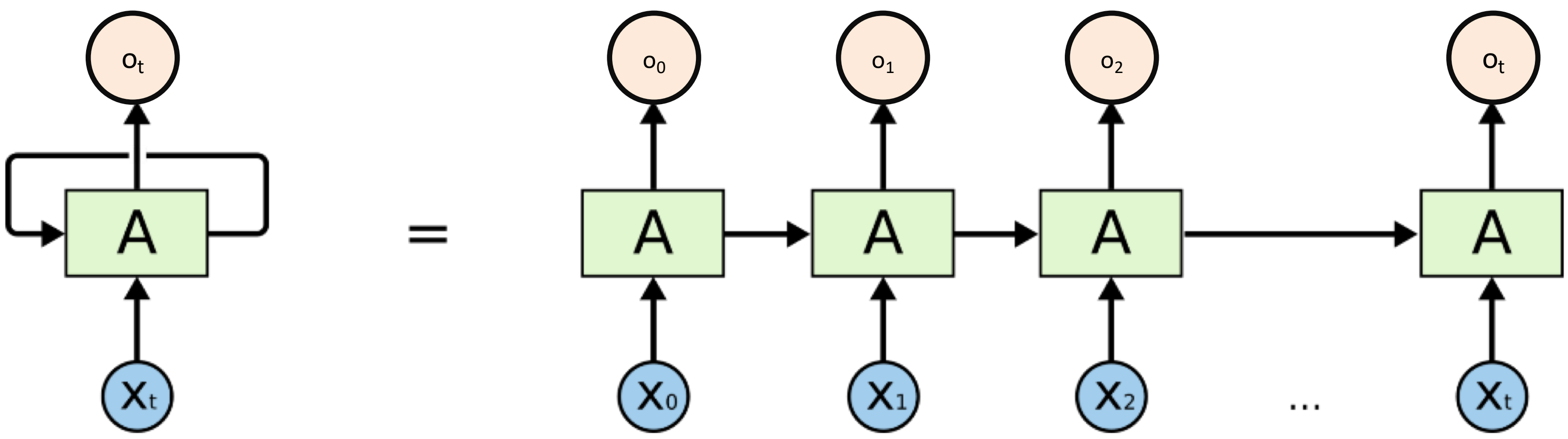


- Recurrent Neural Network have an internal state
- State is passed from input x_t to x_{t+1}

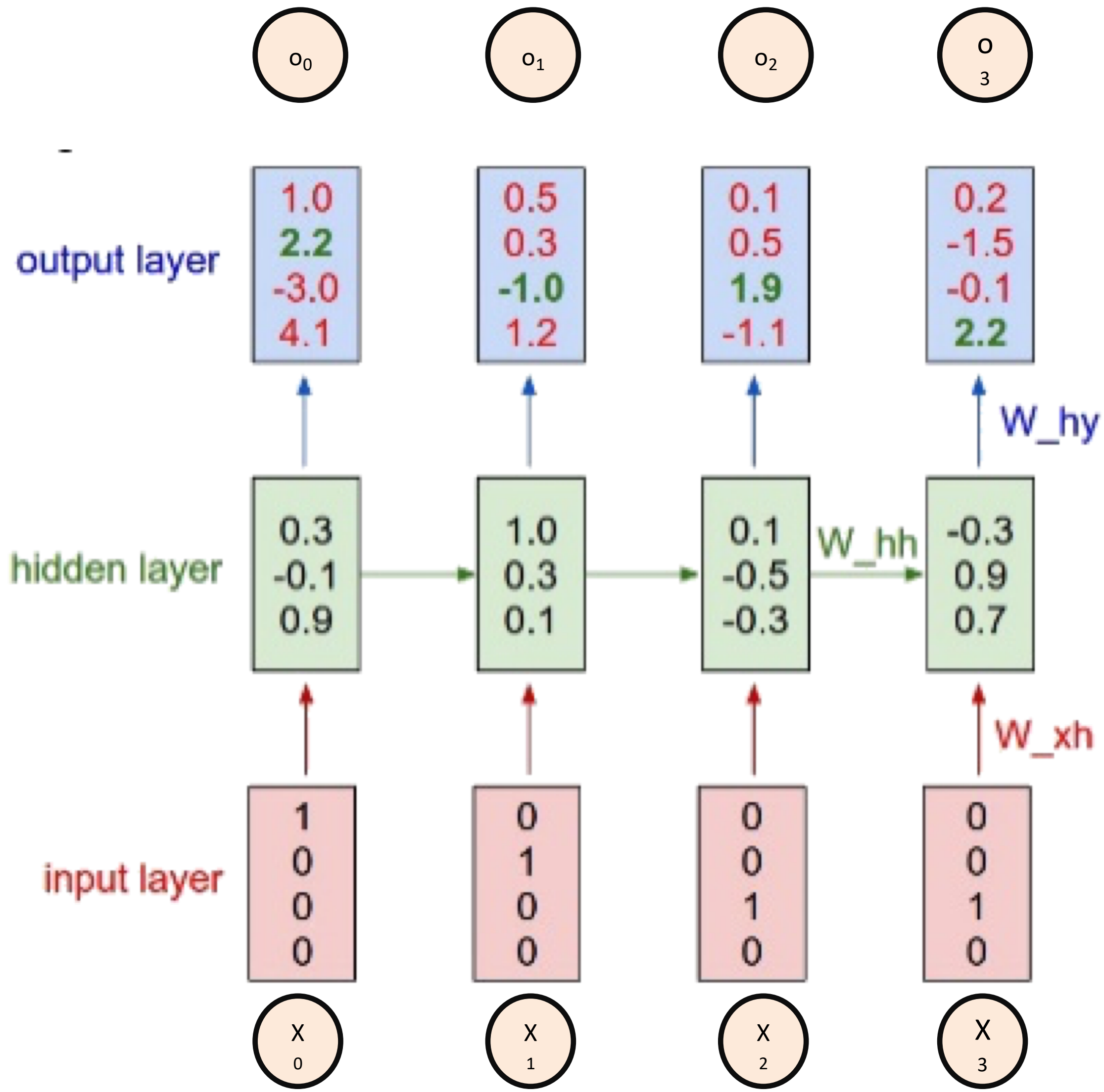
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Language Models with RNN

- Let x_0, x_1, x_2… denote words (input)
- Let o_0, o_1, o_2… denote the probability of the sentence(output)
- Memory requirement scales nicely (linear with the number of word embeddings / number of character)

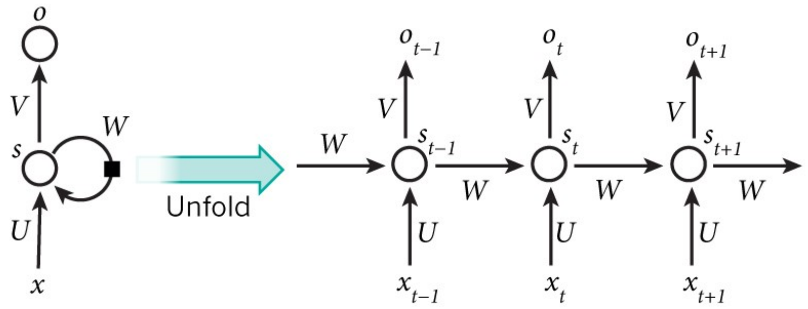


Recurrent Neural Network



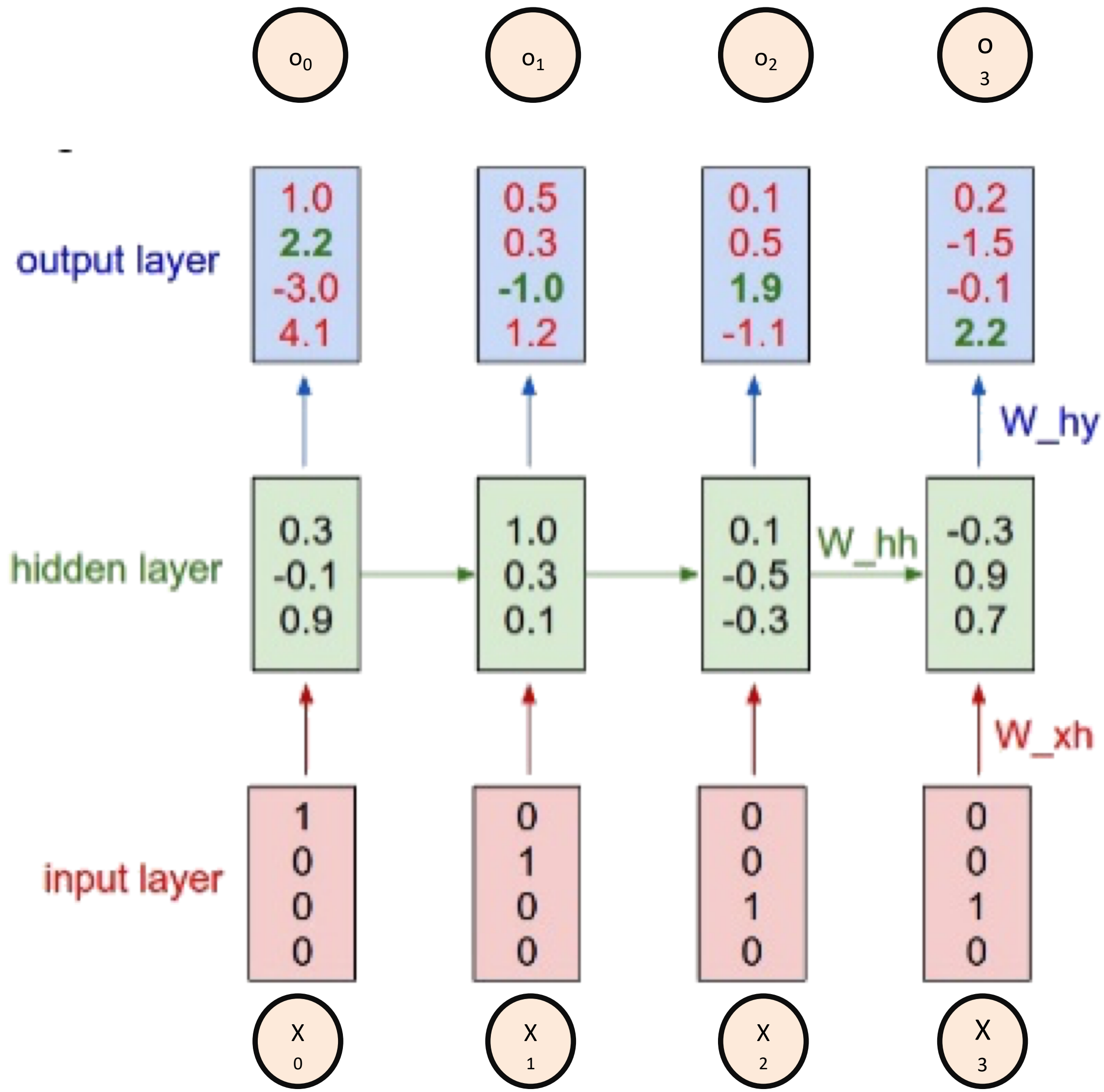
Recurrent neural networks

- RNN being unrolled (or unfolded) into full network
- Unrolling: write out network for complete sequence



- Image credits: Nature

RNN (Problem Revisited)



Img Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

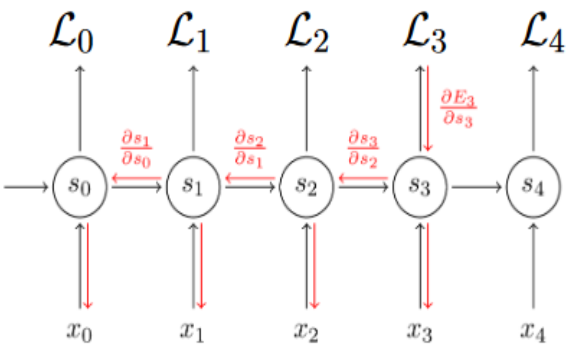
No Magic Involved (in Theory)

- You unroll your data in time
- You compute the gradients
- You use back propagation to train your network
- Karpathy presents a Python implementation for Char-RNN with 112 lines

- Training RNNs is hard:
  - Inputs from many time steps ago can modify output
  - Vanishing / Exploding Gradient Problem

- Vanishing gradients can be solved by Gated-RNNs like Long-Short-Term-Memory (LSTM) Models
  - LSTM became popular in NLP in 2015

Vanishing and exploding gradients



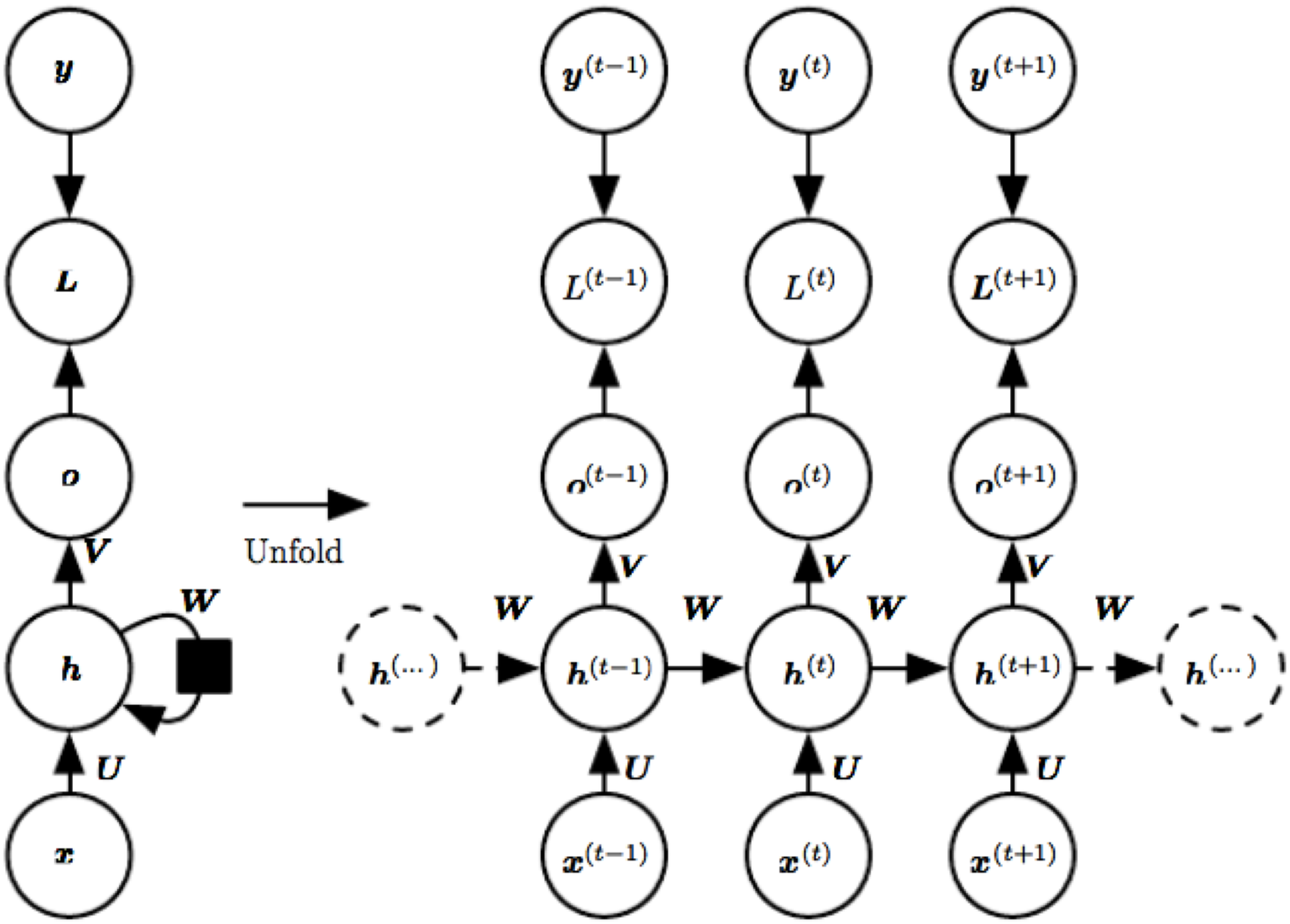
- For training RNNs, calculate gradients for U, V, W – ok for V but for W and U ...
- Gradients for W:

$$\frac{\partial \mathcal{L}_3}{\partial W} = \frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial W} = \sum_{k=0}^{3}\frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

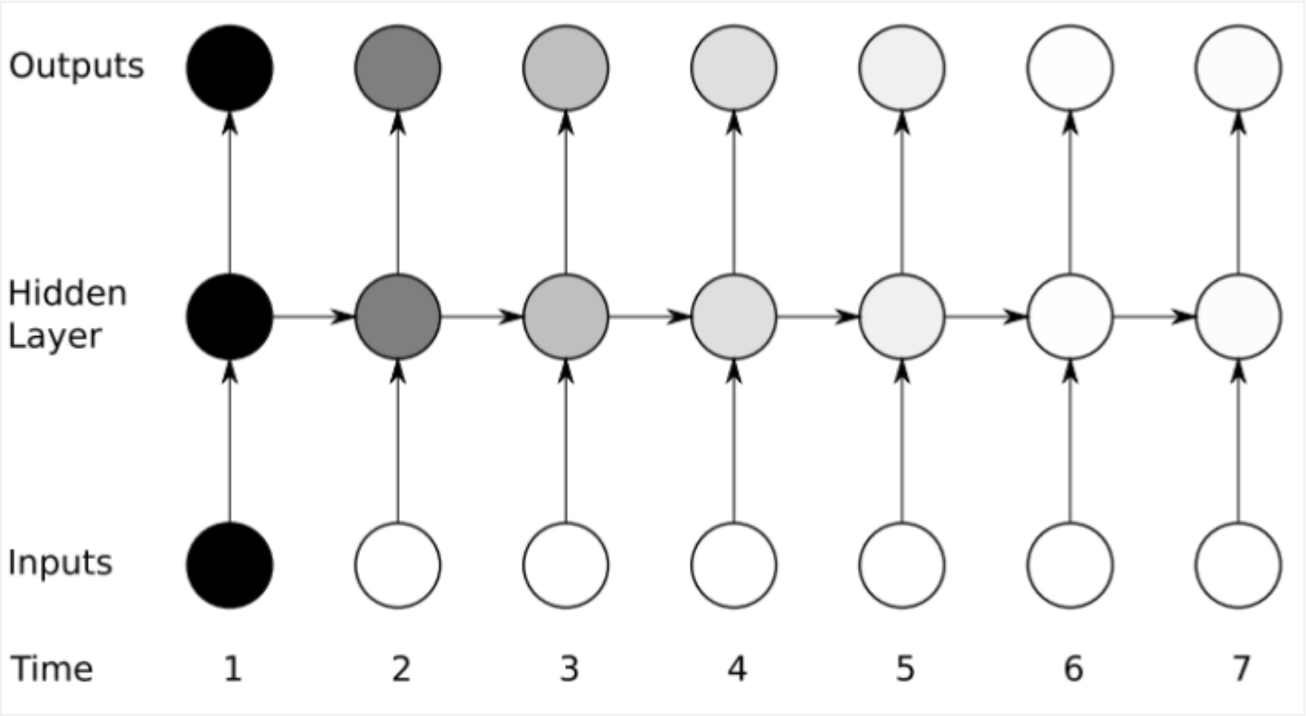
- More generally: $\frac{\partial \mathcal{L}}{\partial s_t} = \frac{\partial \mathcal{L}}{\partial s_m} \cdot \underbrace{\frac{\partial s_m}{\partial s_{m-1}}}_{<1} \cdot \underbrace{\frac{\partial s_{m-1}}{\partial s_{m-2}}}_{<1} \cdot \dots \cdot \underbrace{\frac{\partial s_{t+1}}{\partial s_t}}_{<1} \Rightarrow \ll 1$
- Gradient contributions from far away steps become zero: state at those steps doesn't contribute to what you are learning

L_i – Loss, U, V, W – Parameters, S_i - states

Vanishing and exploding gradients



Vanishing and exploding gradients



Heatmap

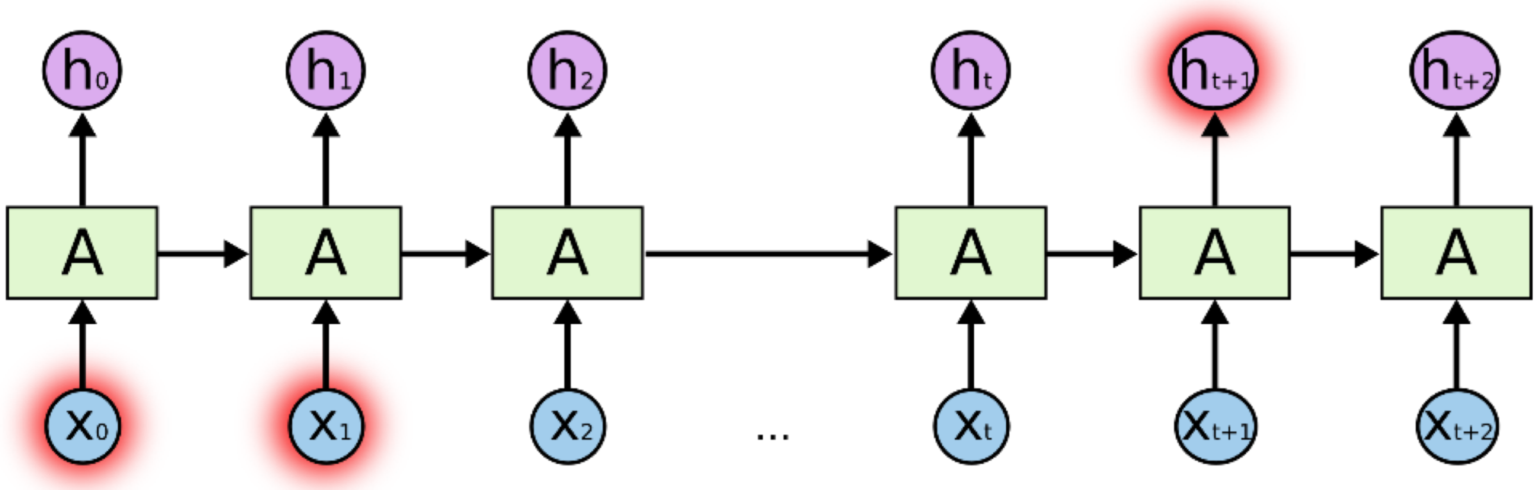
Long Short Term Memory [Hochreiter and Schmidhuber, 1997]

LSTMs designed to combat vanishing gradients through gating mechanism

How LSTM calculates hidden state s_t

$$
\begin{aligned}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{aligned}
$$

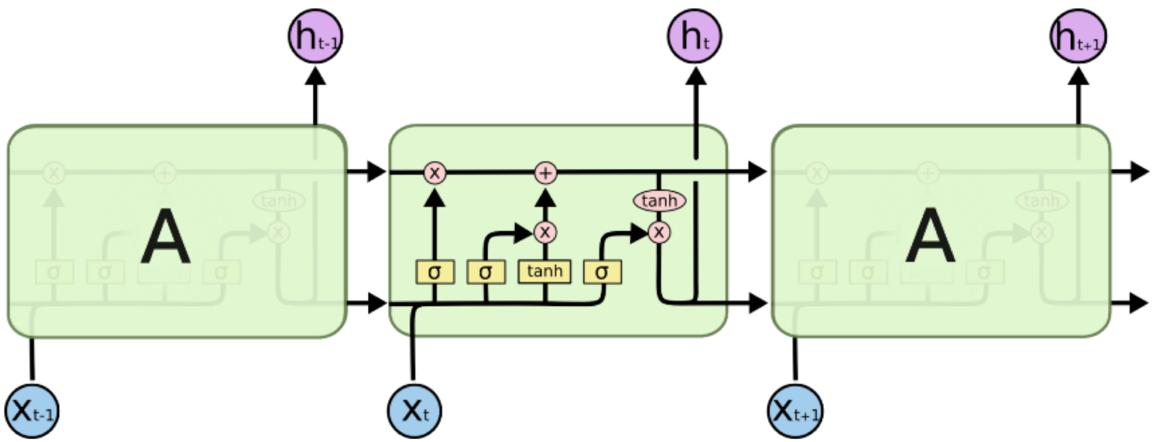
Long-Short-Term Memory (LSTM)



- Long-term dependencies:
  I grew up in France and lived there until I was 18. Therefore I speak fluent ???
- Presented (vanilla) RNN is unable to learn long term dependencies
  - Issue: More recent input data has higher influence on the output
- Long-Short-Term Memory (LSTM) models solves this problem

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

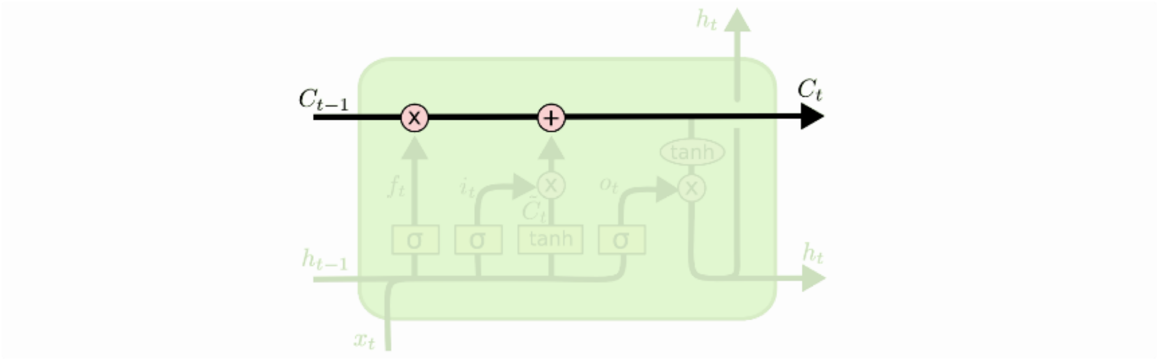
LSTM Model



- The LSTM model implements a *forget-gate* and an *add-gate*
- The models learns when to forget something and when to update internal storage

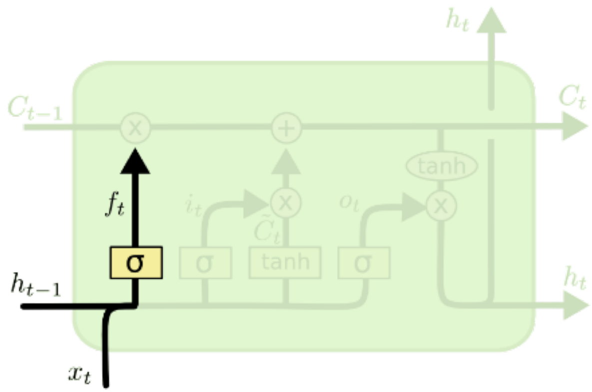
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Model



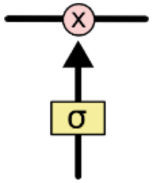
- Core: Cell-state *C* (a vector of certain size)
- The model has the ability to remove or add information using Gates

Forget-Gate



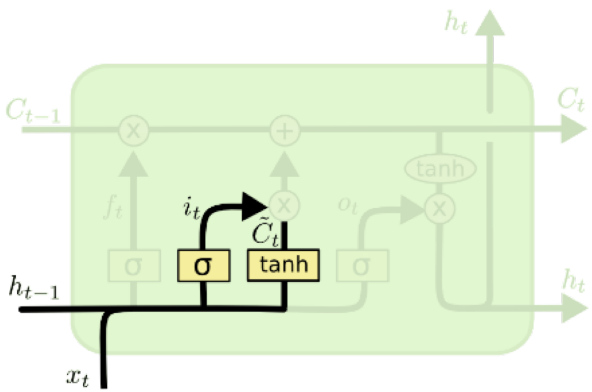
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- Sigmoid function σ output a value between 0 and 1
- The output is point-wise multiplied with the cell state C_{t-1}
- Interpretation:
  - 0: *Let nothing through*
  - 1: *Let everything through*



- Example: When we see a new subject, forget gender of old subject

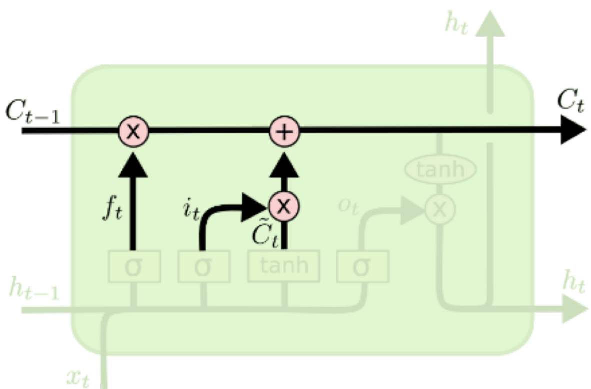
Set-Gate



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

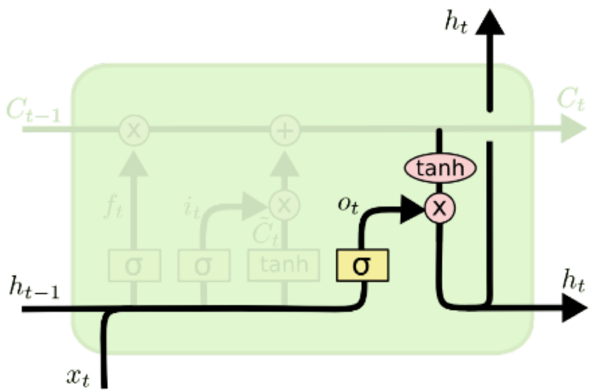
- Compute i_t which cells we want to update and to which degree (σ: 0 … 1)
- Compute the new cell value using the *tanh* function

Update Internal Cell State



$$C_t = \underbrace{f_t * C_{t-1}}_{\text{Forget state cells}} + \underbrace{i_t * \tilde{C}_t}_{\text{Update state cells}}$$

Compute Output h_t



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- We use the updated cell state C_t to compute the output
- We might not need the complete cell state as output
  - Compute o_t, defining how relevant each cell is for the output
  - Pointwise multiply o_t with *tanh(C_t)*

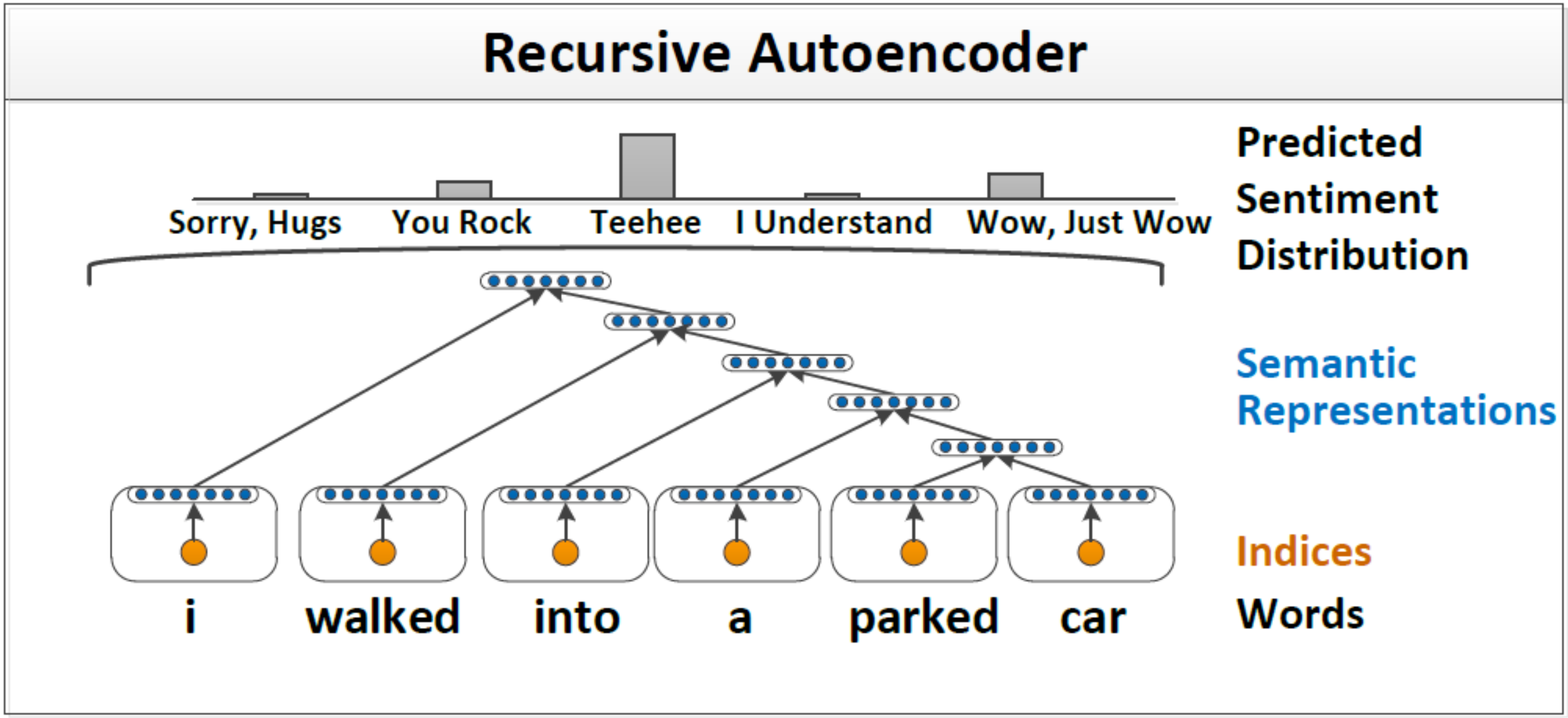
- Cell state C_t and output h_t is passed to the next time step

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recursive Neural Networks

- Socher et al., 2011, *Semi-supervised recursive autoencoders for predicting sentiment distributions*
- Socher et al., 2013, *Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank*

Recursive Autoencoders



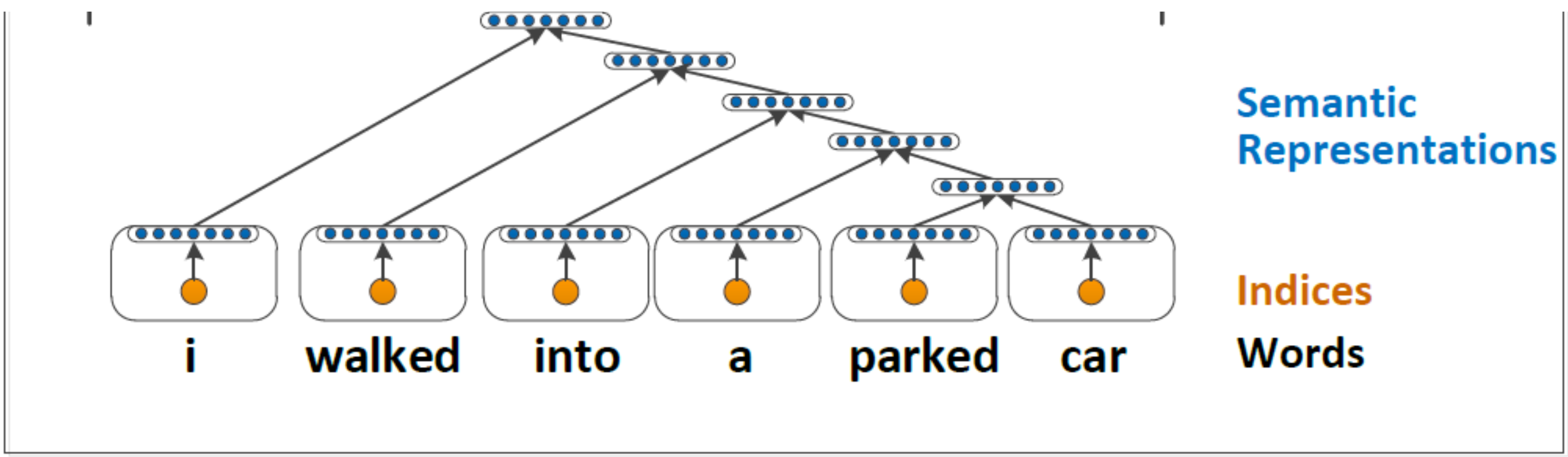
- In a first step, words are mapped to dense vectors (word embeddings)
- Iteratively they are combined and reduced to form a single compact representation of the sentence

Recursive Autoencoders (RAE)

- Given two embeddings x_1, x_2 each with length *n*
- The autoencoder takes $[x_1; x_2]$ as input and maps it to a hidden layer of size *n:*

$$y_1 = f(W[x_1; x_2] + b)$$

- The function is repeatedly applied for the whole sentence until we receive a single vector of size *n*, representing the semantic of this sentence



Selecting the nodes that should be combined

- The previous slides showed a joining of the vectors from right to left
- However, we can define any tree structure for the combination of two vectors, for example a parse tree
- Socher et al. present a greedy approach for the combination of vectors
  - Compute the reconstruction error for all neighboring vectors.
  - The two neighbors with the lowest error are selected and their nodes are replaced by the compressed representation.
  - Repeat the previous two steps until we end up with a single vector representing the semantics of the sentence
- A different, more recent approach, is to use parse trees

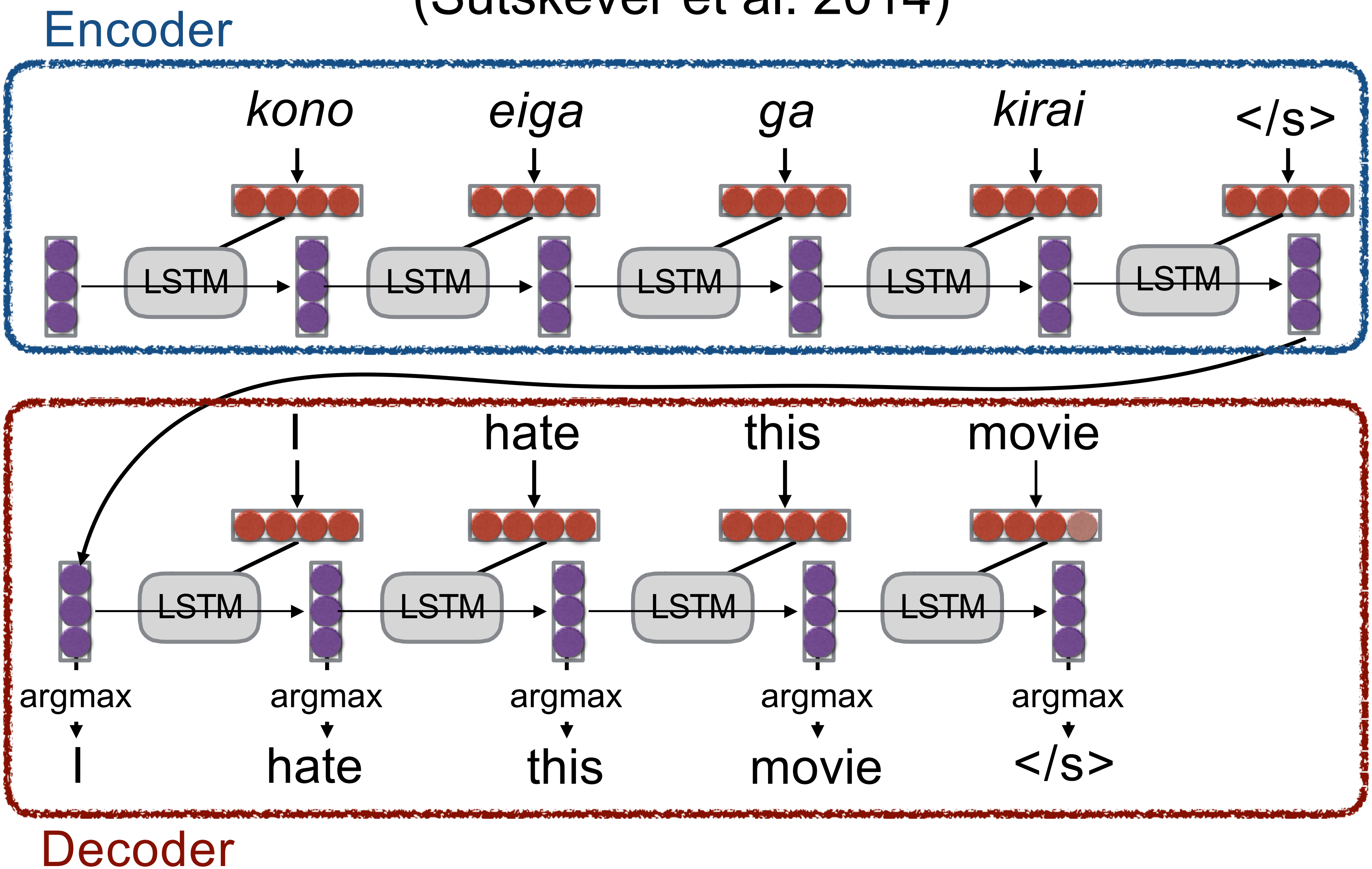
- The output of the recursive autoencoder can be used for a classification task by adding a final softmax layer:

$$o = softmax(W^{label} y_m)$$

Machine translation

Encoder-decoder Models

(Sutskever et al. 2014)

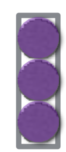


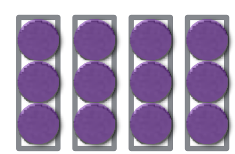
Sentence Representations

Problem!

"You can't cram the meaning of a whole %&!$ing sentence into a single $&!*ing vector!"
— Ray Mooney

- But what if we could use multiple vectors, based on the length of the sentence.

this is an example → 

this is an example → 

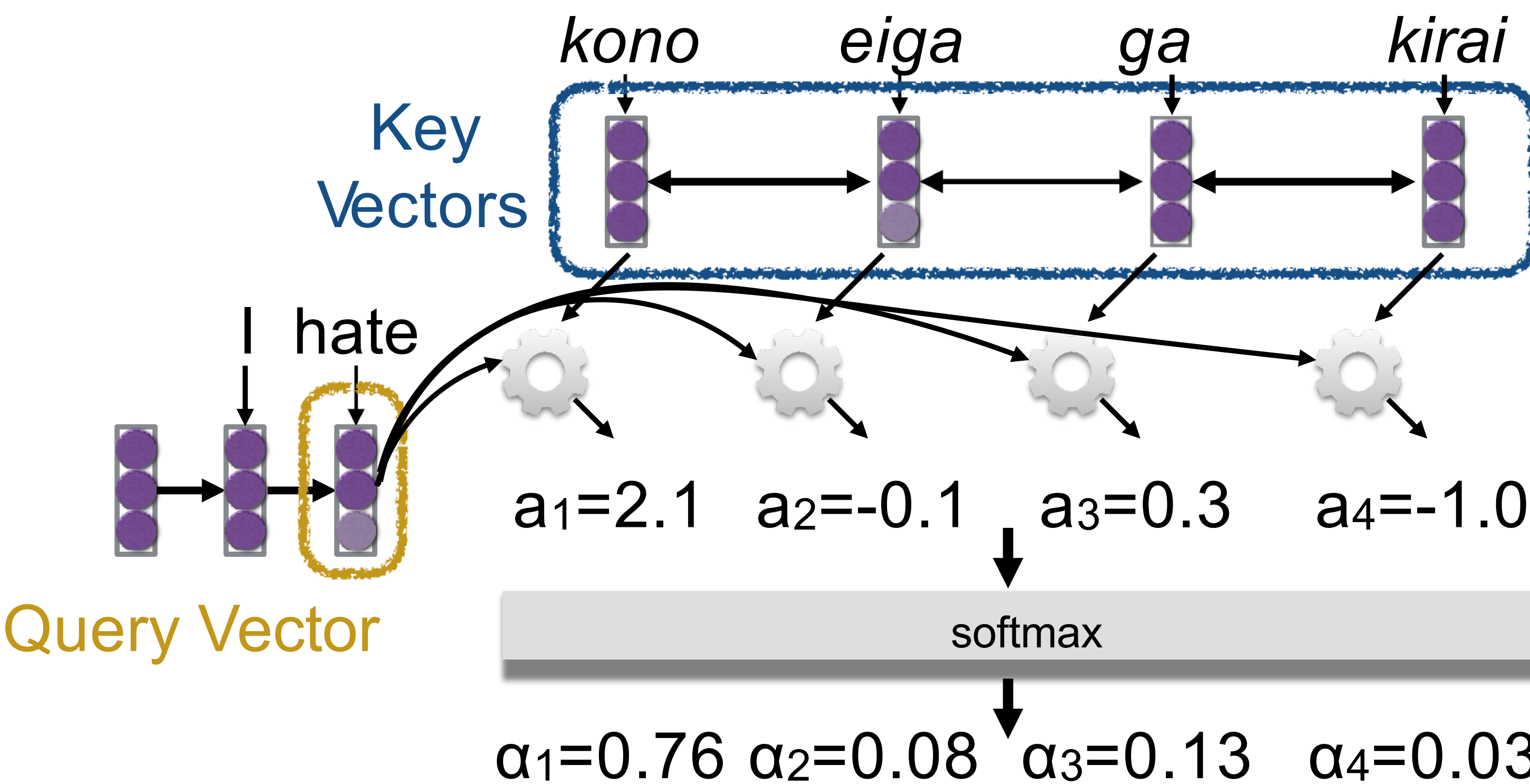
Attention - Basic Idea

(Bahdanau et al. 2015)

- Encode each word in the sentence into a vector
- When decoding, perform a linear combination of these vectors, weighted by “attention weights”
- Use this combination in picking the next word

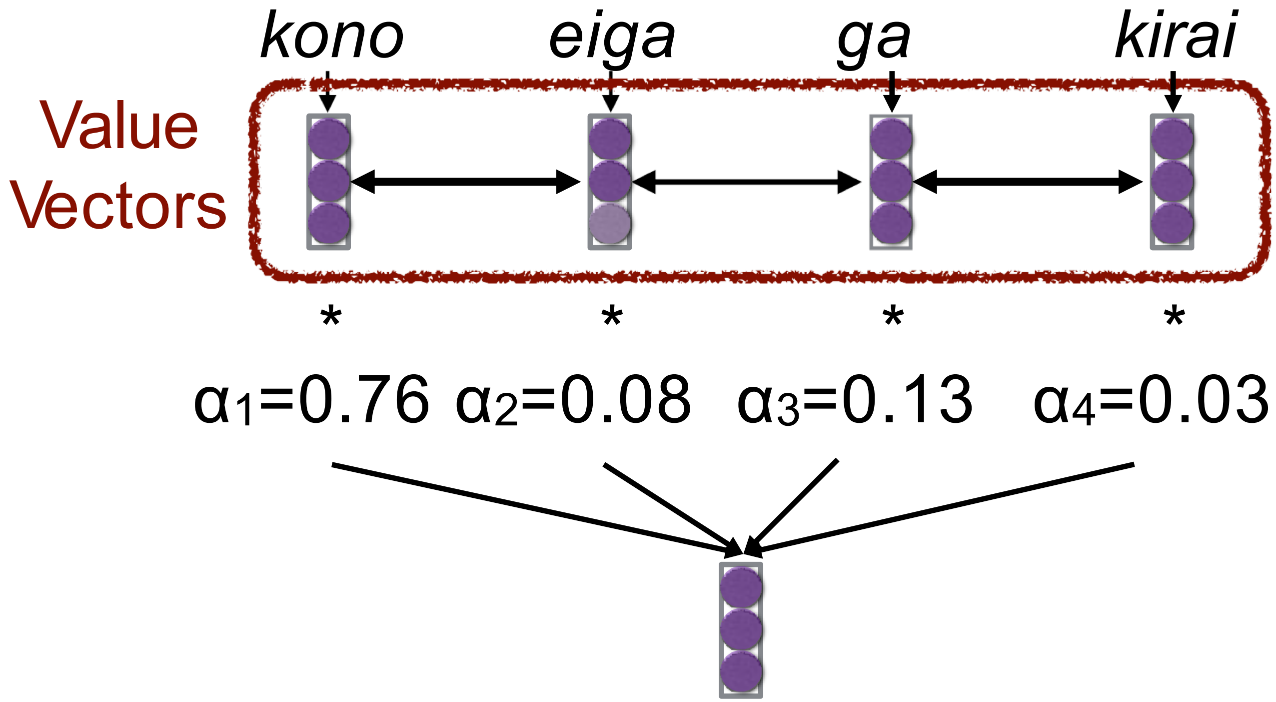
Calculating Attention (1)

- Use "query" vector (decoder state) and "key" vectors (all encoder states)
- For each query-key pair, calculate weight
- Normalize to add to one using softmax



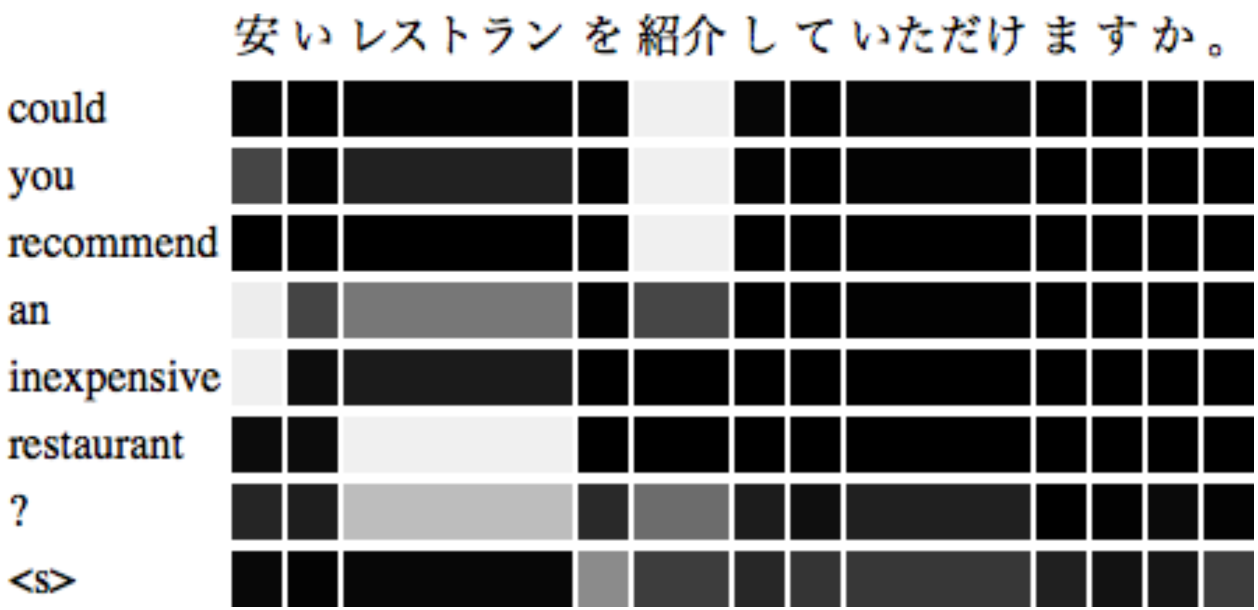
Calculating Attention (2)

- Combine together value vectors (usually encoder states, like key vectors) by taking the weighted sum



- Use this in any part of the model you like

A Graphical Example



Attention Score Functions (1)

- ***q*** is the query and ***k*** is the key
- **Multi-layer Perceptron** (Bahdanau et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{w}_2^{\intercal} \tanh(W_1[\boldsymbol{q}; \boldsymbol{k}])$$

  - Flexible, often very good with large data
- **Bilinear** (Luong et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^{\intercal} W \boldsymbol{k}$$

Attention Score Functions (2)

- **Dot Product** (Luong et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\intercal \boldsymbol{k}$$

  - No parameters! But requires sizes to be the same.

- **Scaled Dot Product** (Vaswani et al. 2017)

  - Problem: scale of dot product increases as dimensions get larger

  - Fix: scale by size of the vector

$$a(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q}^\intercal \boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}}$$

References

- **Deep Learning for NLP - **Nils Reimers. https://github.com/UKPLab/deeplearning4nlp-tutorial/tree/master/2017-07_Seminar
- CS231n: Convolutional Neural Networks for Visual Recognition. Andrej Karpathy http://cs231n.github.io/convolutional-networks/
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- **Neural Networks for Information Retrieval**. SIGIR 2017 Tutorial http://nn4ir.com/
- **CSE 446 - Machine Learning - Spring 2015,** University of Washington. Pedro Domingos. https://courses.cs.washington.edu/courses/cse446/15sp/