TTIC 31210:
Advanced Natural Language Processing

Kevin Gimpel

Spring 2017

Lecture 2:
Elements of Neural NLP

- Please email me with the following:
  - name
  - email address
  - whether you are taking the course for credit
- I will use the email addresses for the course mailing list

Assignment 1

- Assignment 1 has been posted; due in one week

Roadmap

- review of TTIC 31190 (week 1)
- **deep learning for NLP (weeks 2-4)**
- generative models & Bayesian inference (week 5)
- Bayesian nonparametrics in NLP (week 6)
- EM for unsupervised NLP (week 7)
- syntax/semantics and structure prediction (weeks 8-9)
- applications (week 10)

What is a classifier?

- a function from inputs *x* to classification labels *y*
- one simple type of classifier:
  - for any input *x*, assign a score to each label *y*, parameterized by vector θ:

$$\mathrm{score}(x, y, \boldsymbol{\theta})$$

  - classify by choosing highest-scoring label:

$$\mathrm{classify}(x, \boldsymbol{\theta}) = \operatorname*{argmax}_{y} \ \mathrm{score}(x, y, \boldsymbol{\theta})$$

Modeling, Inference, Learning

inference: solve argmax

modeling: define score function

$$\text{classify}(x, \boldsymbol{\theta}) = \underset{y}{\text{argmax}} \; \text{score}(x, y, \boldsymbol{\theta})$$

learning: choose $\boldsymbol{\theta}$

Notation

- We'll use boldface for vectors:

$$\boldsymbol{\theta}$$

- Individual entries will use subscripts and no boldface, e.g., for entry i:

$$\theta_i$$

What is a neural network?

- just think of a neural network as a function
- it has inputs and outputs
- the term “neural” typically means a particular type of functional building block (“neural layers”), but the term has expanded to mean many things
- neural modeling is now better thought of as a modeling strategy (leveraging “distributed representations” or “representation learning”), or a family of related methods

Classifier Framework

$$\mathrm{classify}(\boldsymbol{x}, \boldsymbol{\theta}) = \operatorname*{argmax}_{\boldsymbol{y} \in \mathcal{L}} \ \mathrm{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$$

- linear model score function:

$$\mathrm{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = \sum_i \theta_i f_i(\boldsymbol{x}, \boldsymbol{y})$$

- we can also use a neural network for the score function!

neural layer = affine transform + nonlinearity

$$\boldsymbol{z}^{(1)} = \underset{\text{nonlinearity}}{g}\left(\underbrace{W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}}_{\text{affine transform}}\right)$$

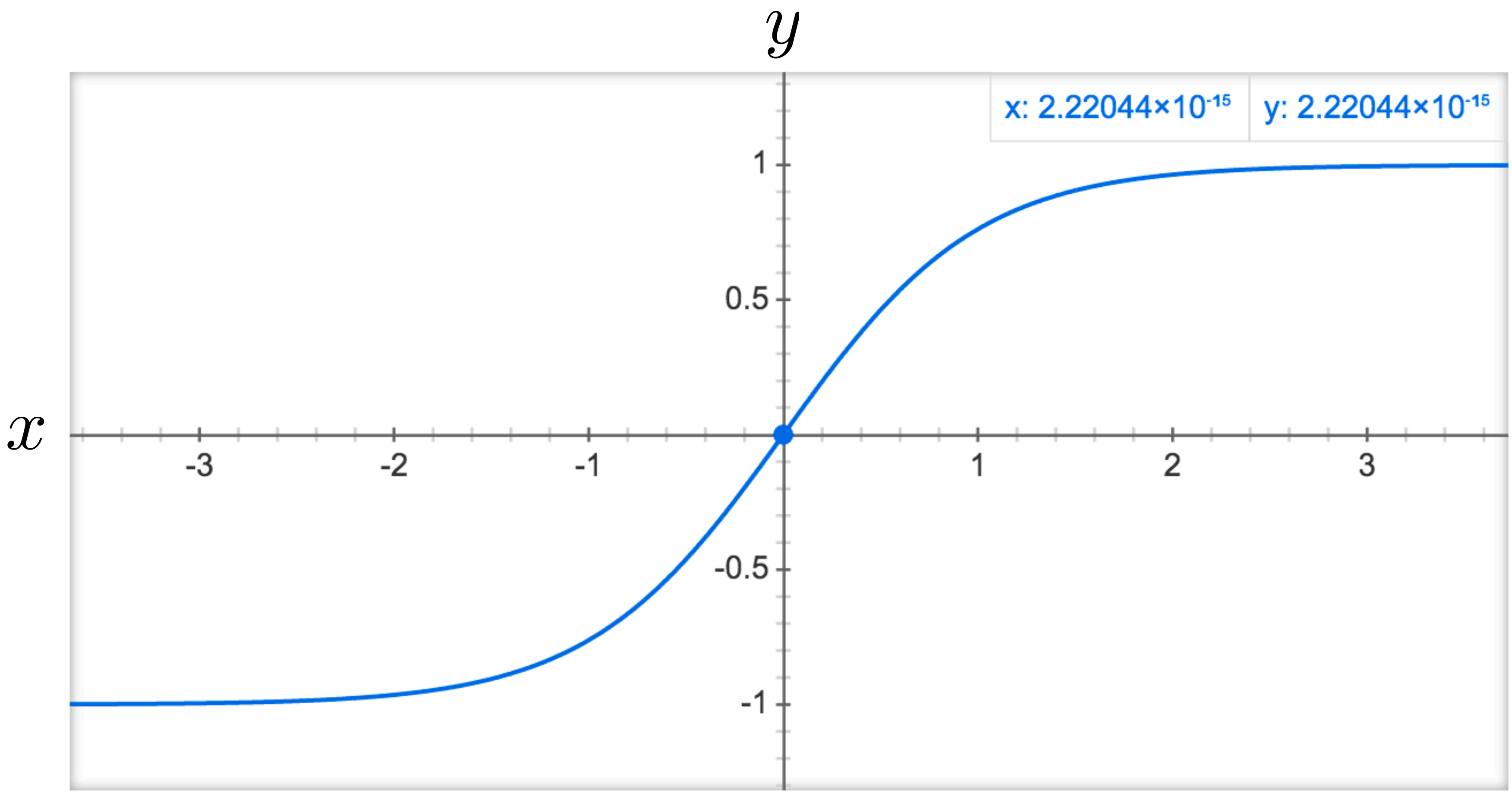
- this is a single "layer" of a neural network
- input vector is $\boldsymbol{x}$
- vector of "hidden units" is $\boldsymbol{z}^{(1)}$

Nonlinearities

$$\boldsymbol{z}^{(1)} = \boxed{g}\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

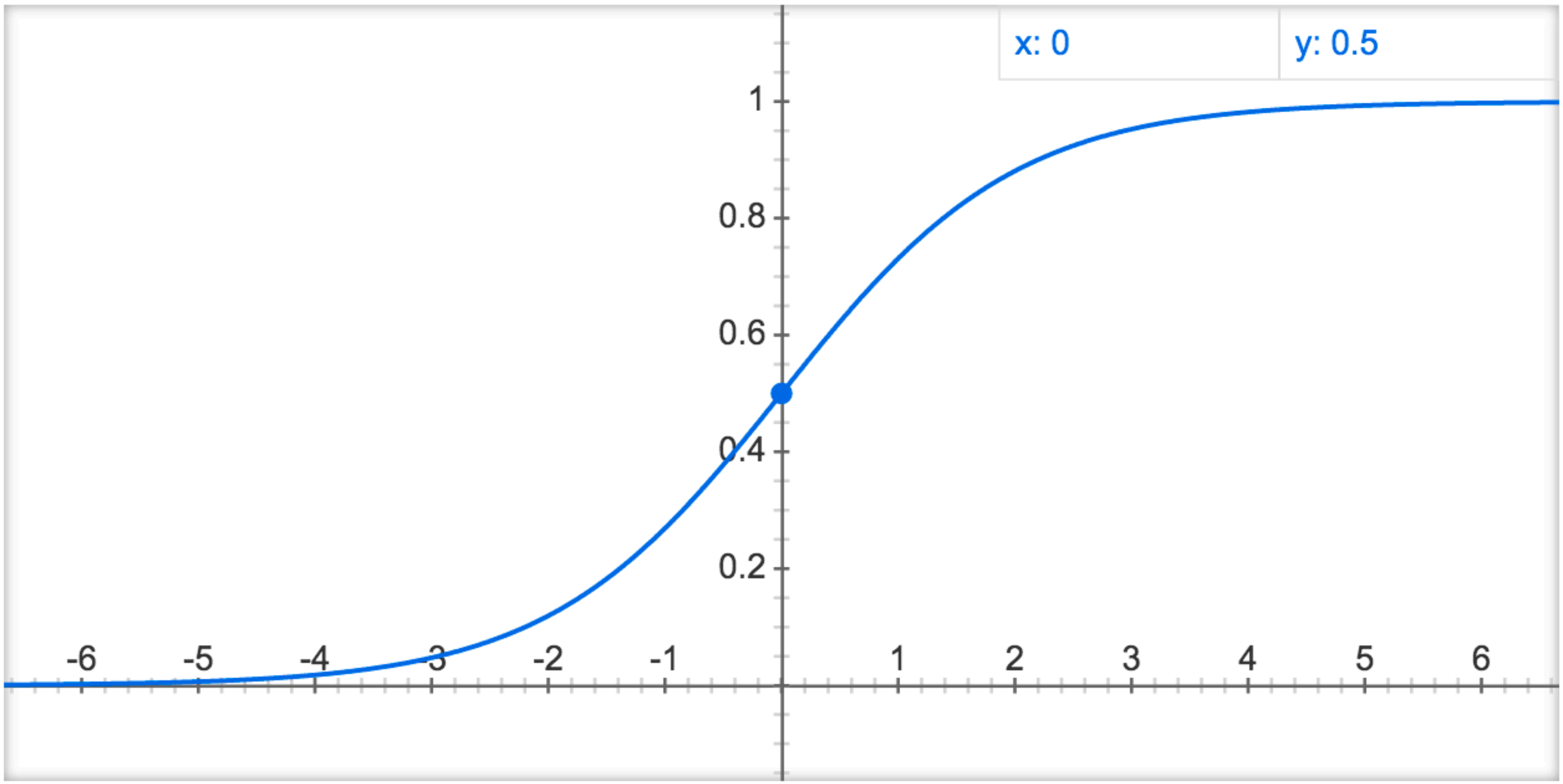
- most common: elementwise application of *g* function to each entry in vector
- examples…

tanh: $y = \tanh(x)$



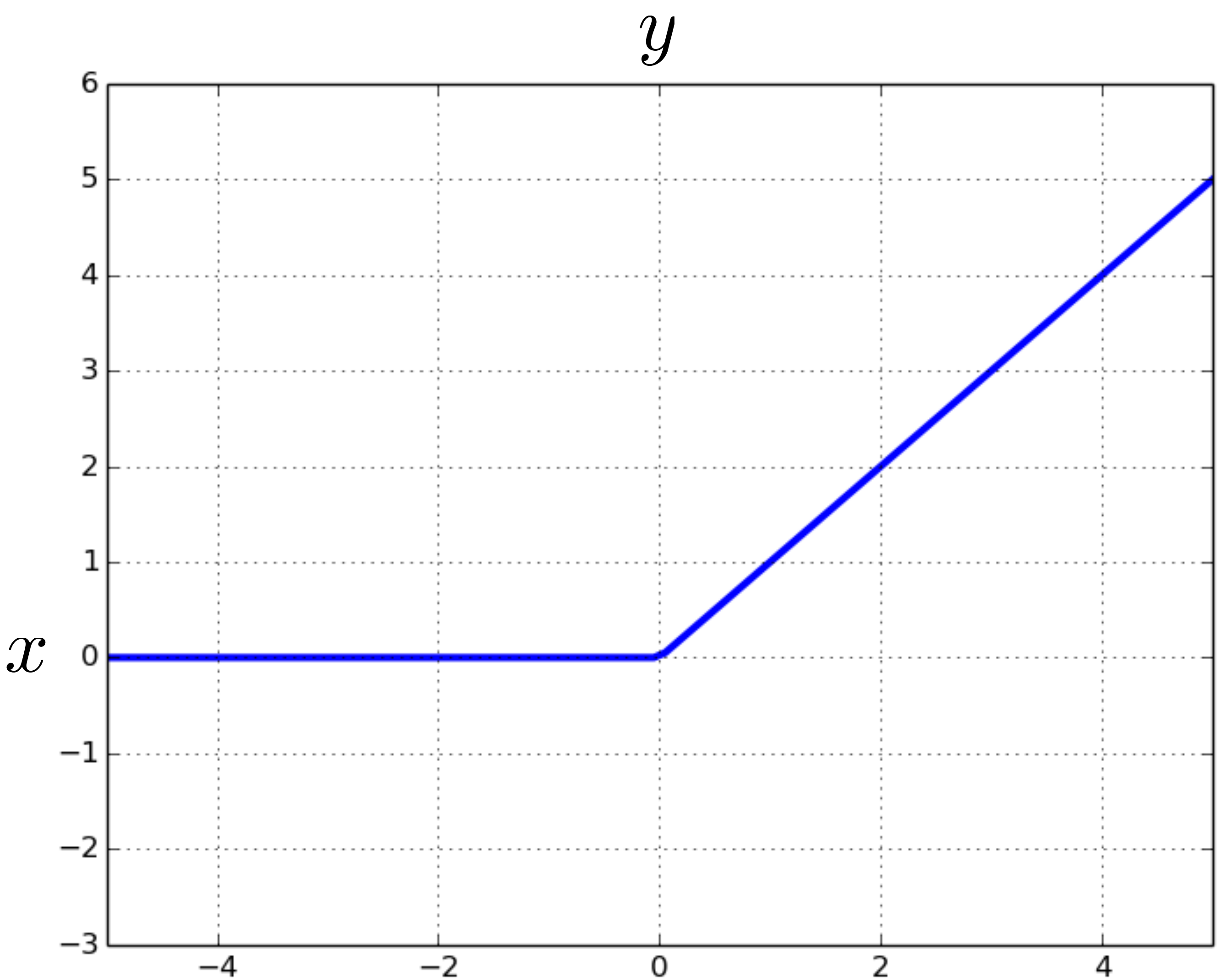
(logistic) sigmoid: $$y = \frac{1}{1 + \exp\{-x\}}$$

y



x

rectified linear unit (ReLU): $y = \max(0, x)$



2-layer network

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

- this is a 2-layer neural network
- input vector is $\boldsymbol{x}$
- output vector is $\boldsymbol{s}$

2-layer neural network for sentiment classification

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}$$

$$\boldsymbol{s} = \begin{bmatrix} \mathrm{score}(\boldsymbol{x}, \mathrm{positive}, \boldsymbol{\theta}) \\ \mathrm{score}(\boldsymbol{x}, \mathrm{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

Use softmax function to convert scores into probabilities

$$\mathrm{softmax}(\boldsymbol{s}) = \begin{bmatrix} \frac{\exp\{s_1\}}{\sum_i \exp\{s_i\}} \\ \cdots \\ \frac{\exp\{s_d\}}{\sum_i \exp\{s_i\}} \end{bmatrix}$$

$$\boldsymbol{s} = \begin{bmatrix} \mathrm{score}(\boldsymbol{x}, \mathrm{positive}, \boldsymbol{\theta}) \\ \mathrm{score}(\boldsymbol{x}, \mathrm{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

$$\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{s}) = \begin{bmatrix} \frac{\exp\{\mathrm{score}(\boldsymbol{x}, \mathrm{positive}, \boldsymbol{\theta})\}}{Z} \\ \frac{\exp\{\mathrm{score}(\boldsymbol{x}, \mathrm{negative}, \boldsymbol{\theta})\}}{Z} \end{bmatrix}$$

$$Z = \exp\{\mathrm{score}(\boldsymbol{x}, \mathrm{positive}, \boldsymbol{\theta})\} + \exp\{\mathrm{score}(\boldsymbol{x}, \mathrm{negative}, \boldsymbol{\theta})\}$$

Why nonlinearities?

2-layer network:

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$
$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

written in a single equation:

$$\boldsymbol{s} = g\left(W^{(1)} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}\right)$$

- if g is linear, then we can rewrite the above as a single affine transform
- can you prove this? (use distributivity of matrix multiplication)

Understanding the score function

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

entry 2 of bias vector

$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(W^{(1)}_{1,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_1\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(\boxed{W^{(1)}_{2,*}} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_2\right)$$

row vector corresponding to row 2 of $W^{(1)}$

Parameter sharing

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

parameters NOT shared between labels

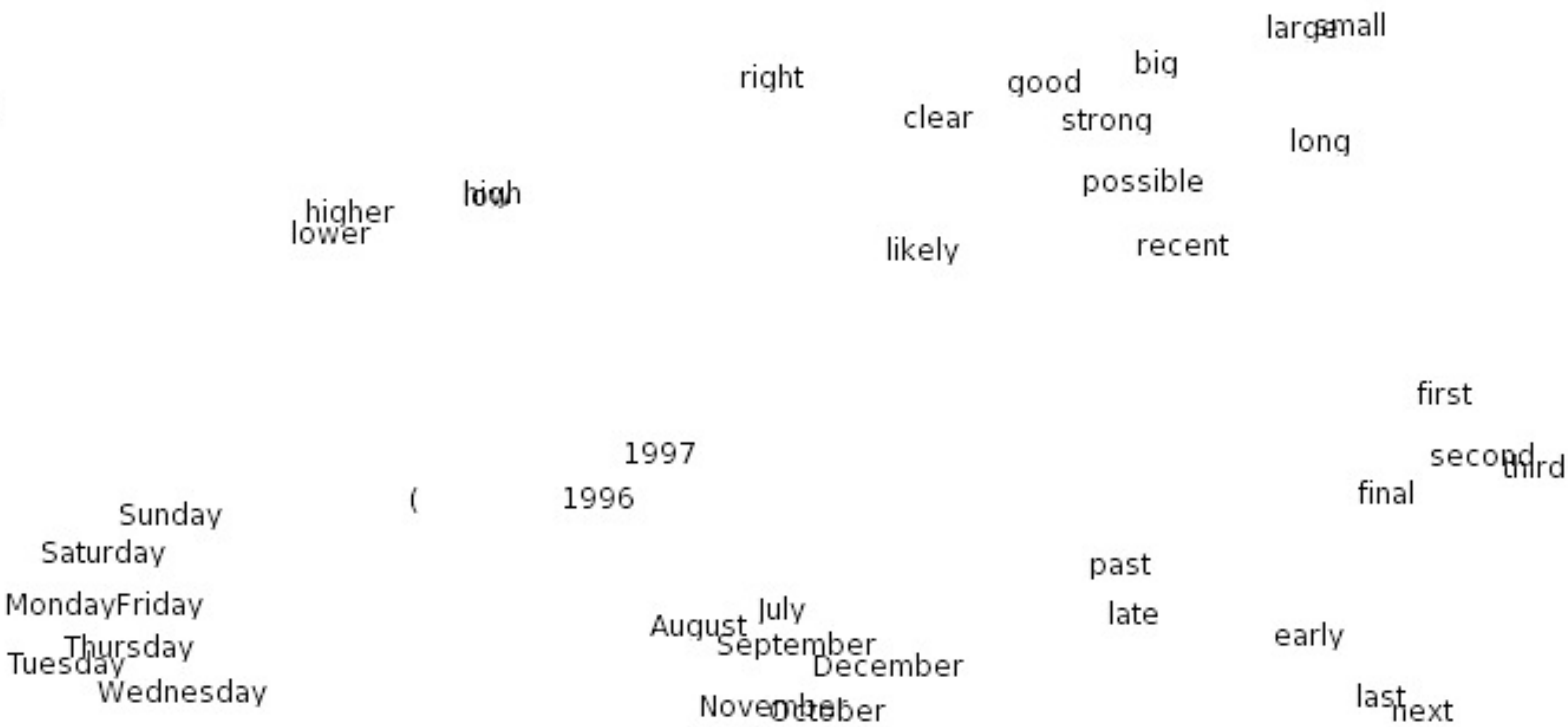
$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(\boxed{W^{(1)}_{1,*}}\, g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boxed{\boldsymbol{b}^{(1)}_1}\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(\boxed{W^{(1)}_{2,*}}\, \underbrace{g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)}_{\text{parameters shared between labels}} + \boxed{\boldsymbol{b}^{(1)}_2}\right)$$

Word Embeddings

- “Basic unit” of neural NLP
- We’ll talk about ways to learn word embeddings next week
- Today: we’ll assume we have word embeddings as a black box

Word Embeddings



Turian et al. (2010)

Two Ways to Represent Word Embeddings

- $\mathcal{V}$ = vocabulary , $|\mathcal{V}|$ = size of vocab
- 1: create $|\mathcal{V}|$-dimensional "one-hot" vector for each word, multiply by word embedding matrix:

$$emb(x) = W \mathrm{onehot}(\mathcal{V}, x)$$

Two Ways to Represent Word Embeddings

- $\mathcal{V}$ = vocabulary , $|\mathcal{V}|$ = size of vocab
- 1: create $|\mathcal{V}|$-dimensional “one-hot” vector for each word, multiply by word embedding matrix:

$$emb(x) = W \mathrm{onehot}(\mathcal{V}, x)$$

- 2: store embeddings in a hash/dictionary data structure, do lookup to find embedding for word:

$$emb(x) = \mathrm{lookup}(W, x)$$

Two Ways to Represent Word Embeddings

- $\mathcal{V}$ = vocabulary , $|\mathcal{V}|$ = size of vocab
- 1: create $|\mathcal{V}|$-dimensional "one-hot" vector for each word, multiply by word embedding matrix:

$$emb(x) = W \mathrm{onehot}(\mathcal{V}, x)$$

- 2: store embeddings in a hash/dictionary data structure, do lookup to find embedding for word:

$$emb(x) = \mathrm{lookup}(W, x)$$

- These are equivalent, second can be much faster (though first can be fast if using sparse operations)

Encoders

- encoder: a function to represent a word sequence as a vector
- simplest: average word embeddings:

$$f_{avg}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{x}|} \sum_{i=1}^{|\boldsymbol{x}|} emb(x_i)$$

- many other functions possible!

Attention

- attention is a useful generic tool
- often used to replace a sum with an attention-weighted sum

Attention

- attention is a useful generic tool
- often used to replace a sum with an attention-weighted sum
- e.g., for a word summing encoder:

$$f_{sum}^{att}(\boldsymbol{x}) = \sum_{i=1}^{|\boldsymbol{x}|} att(x_i, i, \boldsymbol{x}) emb(x_i)$$

$$\sum_{i=1}^{|\boldsymbol{x}|} att(x_i, i, \boldsymbol{x}) = 1$$

- many other functions possible!

Ling et al. (EMNLP 2015)

- Not All Contexts Are Created Equal: Better Word Representations with Variable Attention

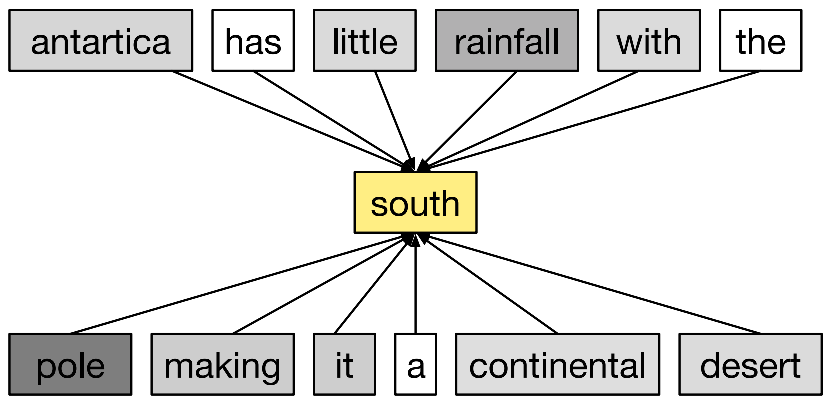
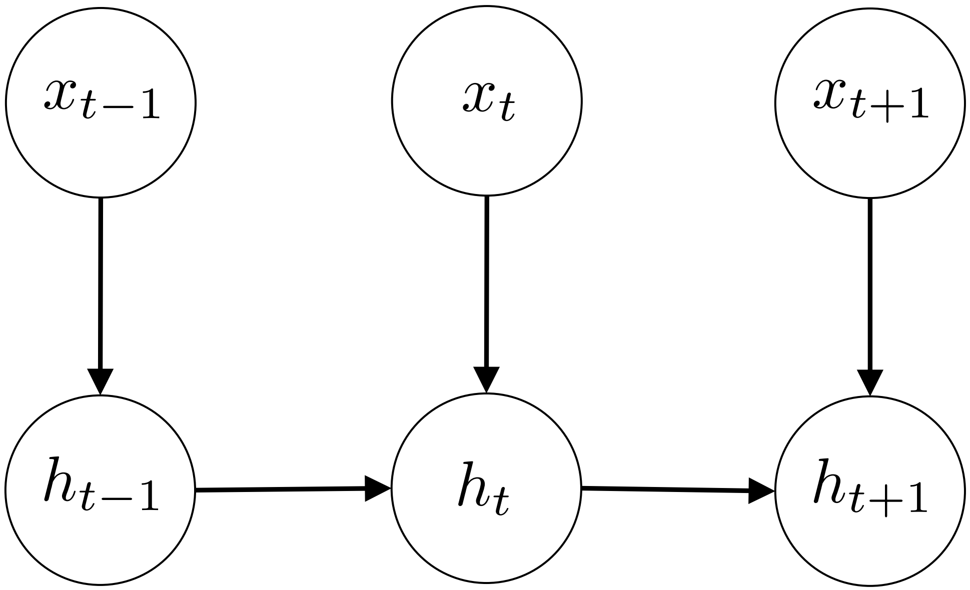


Figure 1: Illustration of the inferred attention parameters for a sentence from our training data when predicting the word *south*; darker cells indicate higher weights.

Recurrent Neural Networks

$$h_t = \tanh\left(W^{(xh)}x_t + W^{(hh)}h_{t-1} + b^{(h)}\right)$$

“hidden vector”



Neural Similarity Learning

- A common need:
  compute similarity/affinity of two things
  - maybe two things of the same type,
  - two things with different types being mapped to same space, or
  - two things with different types being mapped to different spaces, but being compared with a learned similarity function
- Examples?

Synonym Pairs

- Faruqui et al. (NAACL 2014), Wieting et al. (TACL 2014), *inter alia*

contamination	pollution
converged	convergence
captioned	subtitled
outwit	thwart
bad	villain
broad	general
permanent	permanently
bed	sack
carefree	reckless
absolutely	urgently
…	…

Translation Pairs

- Haghighi et al. (ACL 2008), Mikolov et al. (2013), Faruqui and Dyer (EACL 2014)

dog	hund
man	mann
woman	frau
city	stadt
person	man
the	der
the	die
the	das
…	…

Sentence Pairs

this was also true for pompeii , where the temple of jupiter that was already there was enlarged and made more roman when the romans took over .

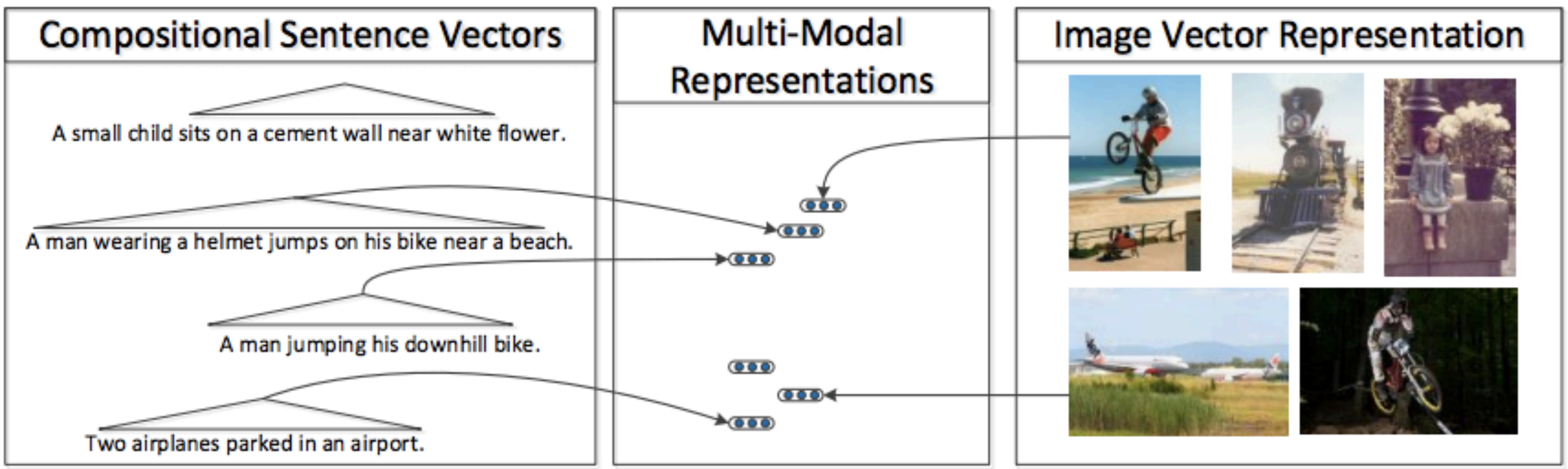
this held true for pompeii , where the previously existing temple of jupiter was enlarged and romanized upon conquest .

WIKIPEDIA
The Free Encyclopedia

WIKIPEDIA
Simple English

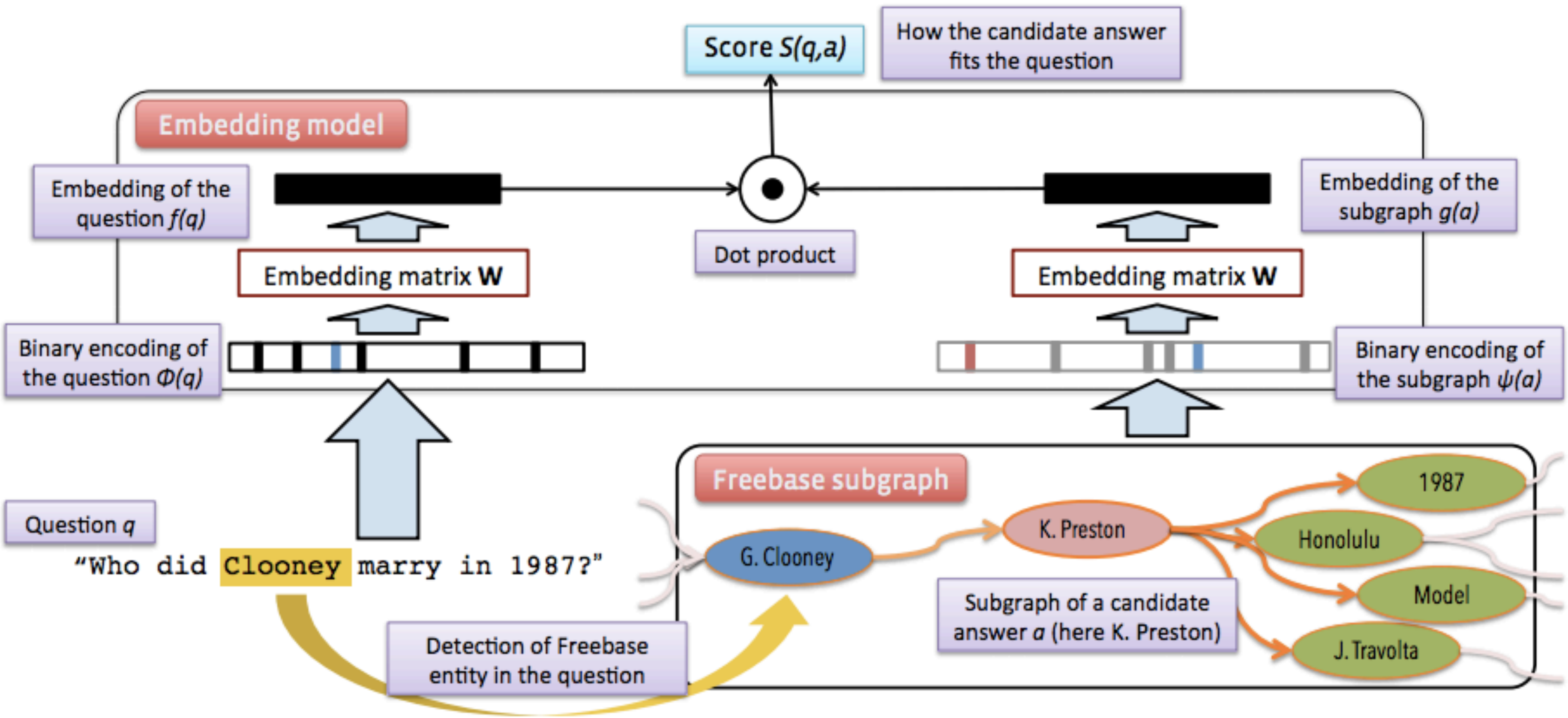
Captions and Images

- Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, Andrew Y. Ng. “Grounded Compositional Semantics For Finding And Describing Images With Sentences,” *TACL* 2014.



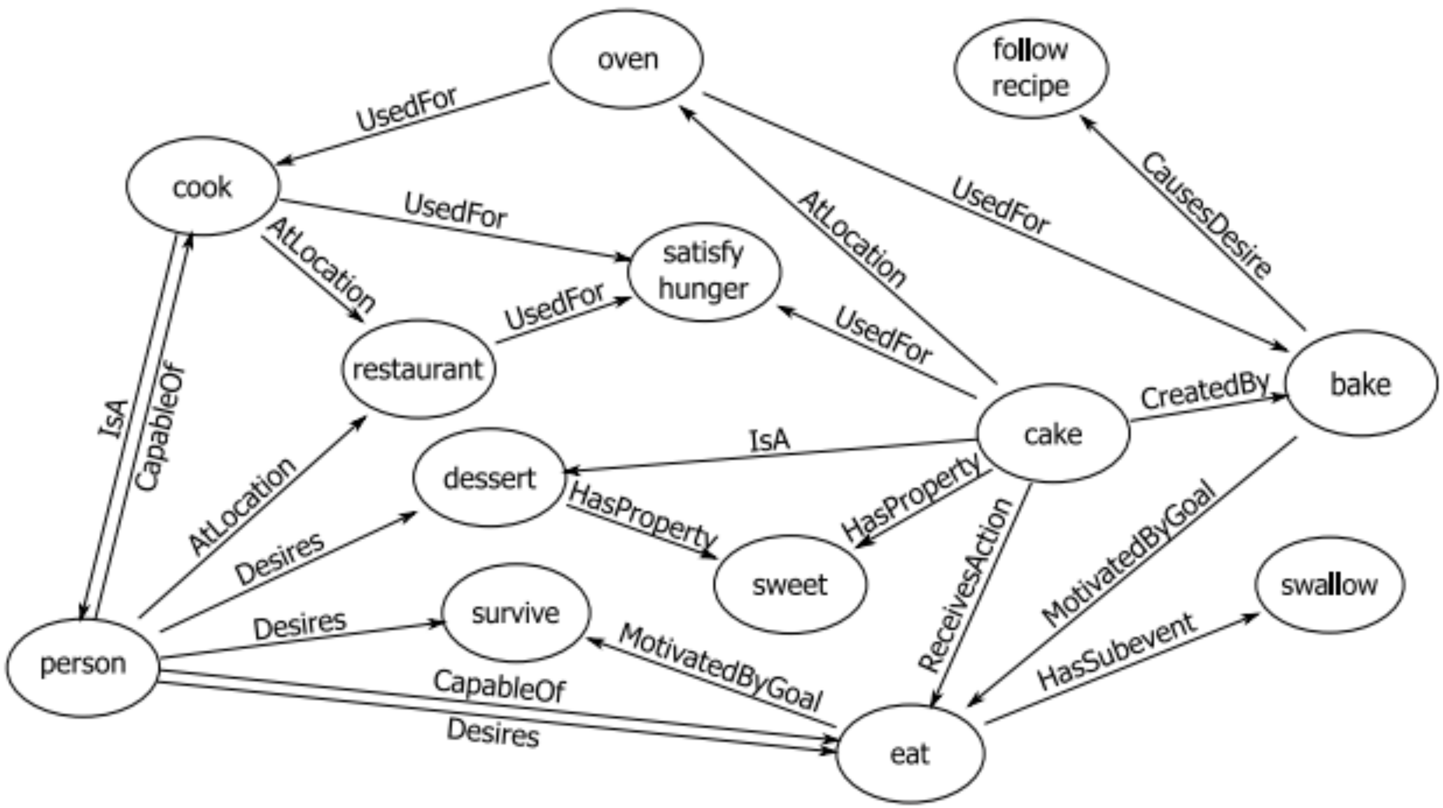
Questions and Answers

- Mohit Iyyer, Jordan Boyd-Graber, Leonardo Claudino, Richard Socher, and Hal Daumé III. "A Neural Network for Factoid Question Answering over Paragraphs," *EMNLP* 2014
- Antoine Bordes, Sumit Chopra, and Jason Weston. "Question Answering with Subgraph Embeddings," *EMNLP* 2014.



Commonsense Knowledge Tuples

- Xiang Li, Aynaz Taheri, Lifu Tu, Kevin Gimpel. “Commonsense Knowledge Base Completion,” *ACL* 2016.



<“cake”, UsedFor, “satisfy hunger”>

Stories and Endings

- story:
  – Jennifer has a big exam tomorrow. She got so stressed, she pulled an all-nighter. She went into class the next day, weary as can be. Her teacher stated that the test is postponed for next week.
- ending:
  – Jennifer felt bittersweet about it.

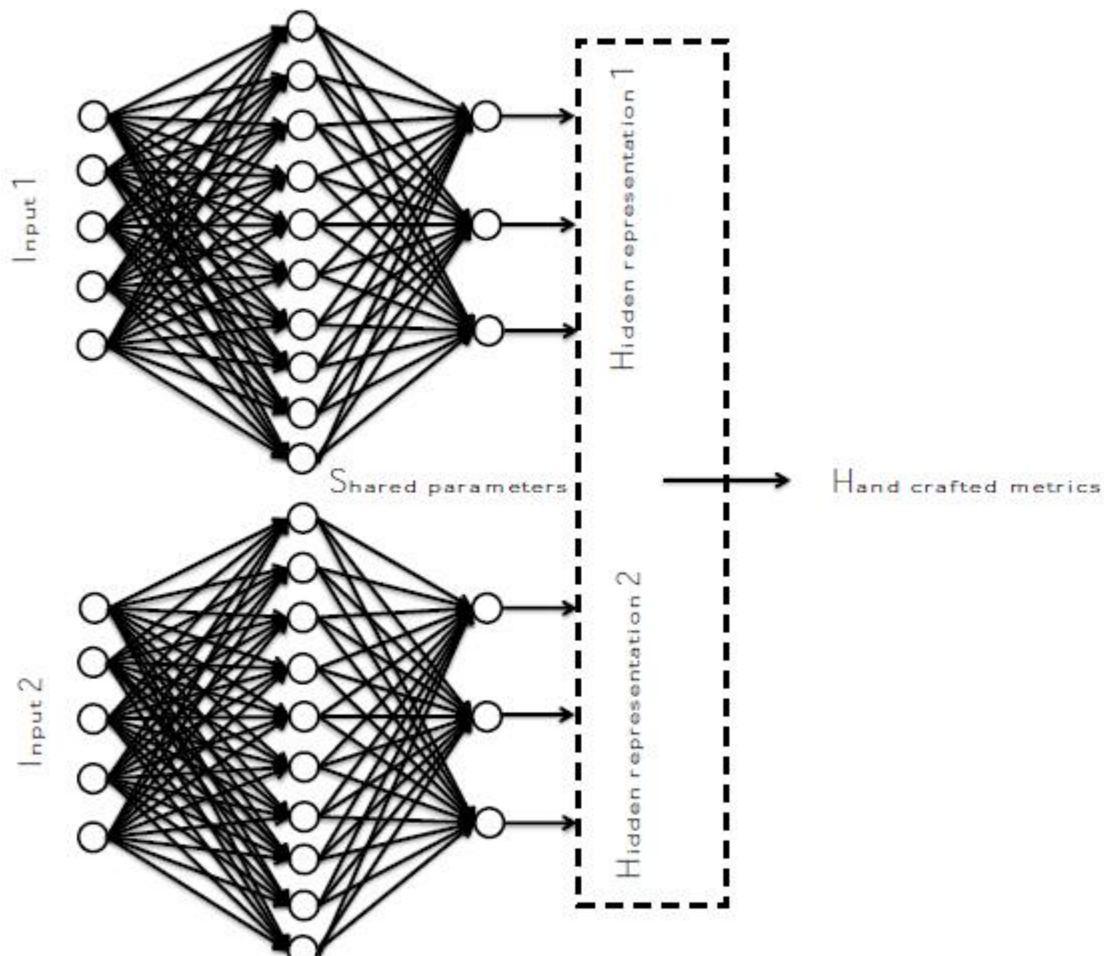
- from ROC Story Corpus (Mostafazadeh et al., 2016)

- Other examples/applications you can think of?

- Sometimes direction matters, sometimes not
- Sometimes there is a particular kind of relation being named for each pair, sometimes not (i.e., just one kind)

Neural Similarity Modeling

- "Siamese networks" (Bromley et al., 1993)
  - these typically share parameters, hence the name



Similarity Modeling

- Siamese networks typically share parameters across the two networks

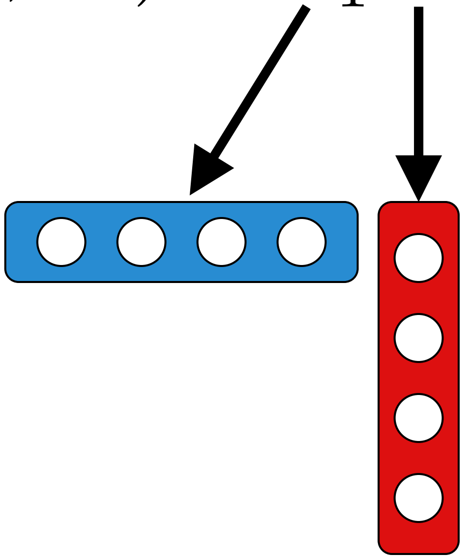
- but it's also common to not share parameters when the items have different types, but we still want to relate them in some way
  - whether map to same space + compute sim or map each to some other space + compute sim

Similarity Functions

- many choices for similarity functions
- we'll go over some in the next few slides
- throughout, keep in mind:
  - output range
  - symmetric? $sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = sim(\boldsymbol{x}_2, \boldsymbol{x}_1)$
  - introduces new parameters?
  - connections among similarity functions?
  - notes/tips on using these

Dot Product

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1^\top \boldsymbol{x}_2$$



range? $\mathbb{R}$

symmetric or asymmetric? **symmetric**

introduces parameters? **no**

Cosine Similarity

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = \cos(\boldsymbol{x}_1, \boldsymbol{x}_2) = \frac{\boldsymbol{x}_1^\top \boldsymbol{x}_2}{||\boldsymbol{x}_1|| \; ||\boldsymbol{x}_2||}$$

$$||\boldsymbol{x}|| = \sqrt{\sum_i x_i^2}$$

range? $[-1, 1]$

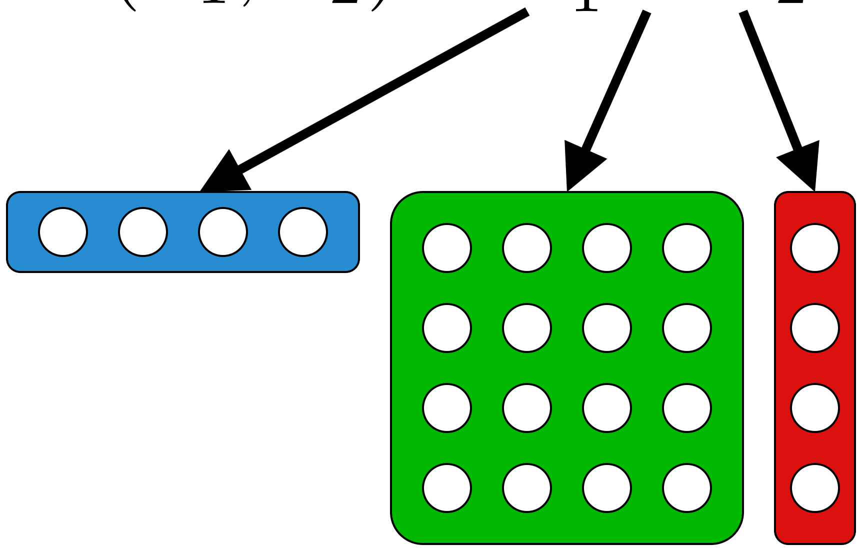
symmetric or asymmetric? **symmetric**

introduces parameters? **no**

generalizes anything? **dot product when vectors have norm 1**

Bilinear Function

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1^\top W \boldsymbol{x}_2$$



range? $\mathbb{R}$

symmetric or asymmetric? **asymmetric in general**

introduces parameters? **yes**

generalizes anything? **dot product if W is identity**

Notes on Using Bilinear Functions

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = \boldsymbol{x}_1^\top W \boldsymbol{x}_2$$

similarity can depend on relation by using different bilinear weight matrices for different relations:

$$sim(\boldsymbol{x}_1, r, \boldsymbol{x}_2) = \boldsymbol{x}_1^\top W_r \boldsymbol{x}_2$$

often W is initialized to the identity matrix (and regularized back to it)

potential issue: W might be very huge

L1/L2 Distances

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = ||\boldsymbol{x}_1 - \boldsymbol{x}_2||_2^2$$

range? $\mathbb{R}_{\geq 0}$

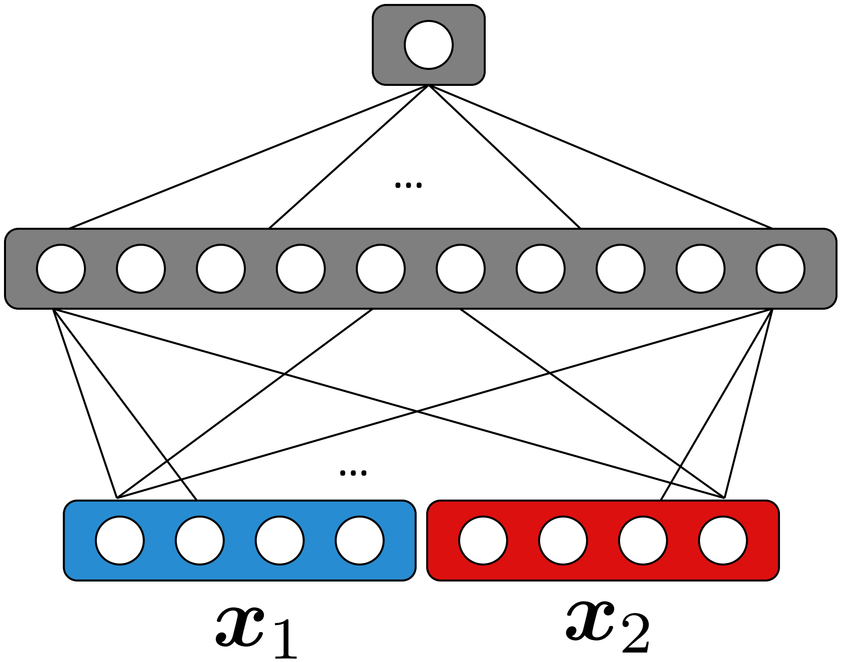
symmetric or asymmetric? **symmetric**

introduce parameters? **no**

Deep Neural Network

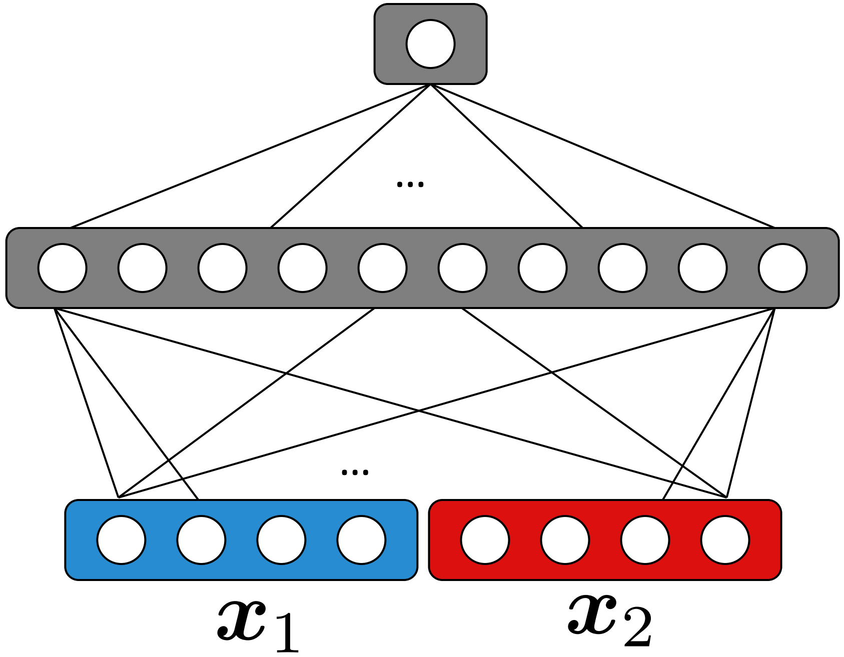
$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = DNN(\mathrm{cat}(\boldsymbol{x}_1, \boldsymbol{x}_2))$$

concatenate vectors, pass to DNN, use scalar for final output:



Deep Neural Network

$$sim(\boldsymbol{x}_1, \boldsymbol{x}_2) = DNN(\text{cat}(\boldsymbol{x}_1, \boldsymbol{x}_2))$$



range? **depends on nonlinearity**

symmetric? **asymmetric**

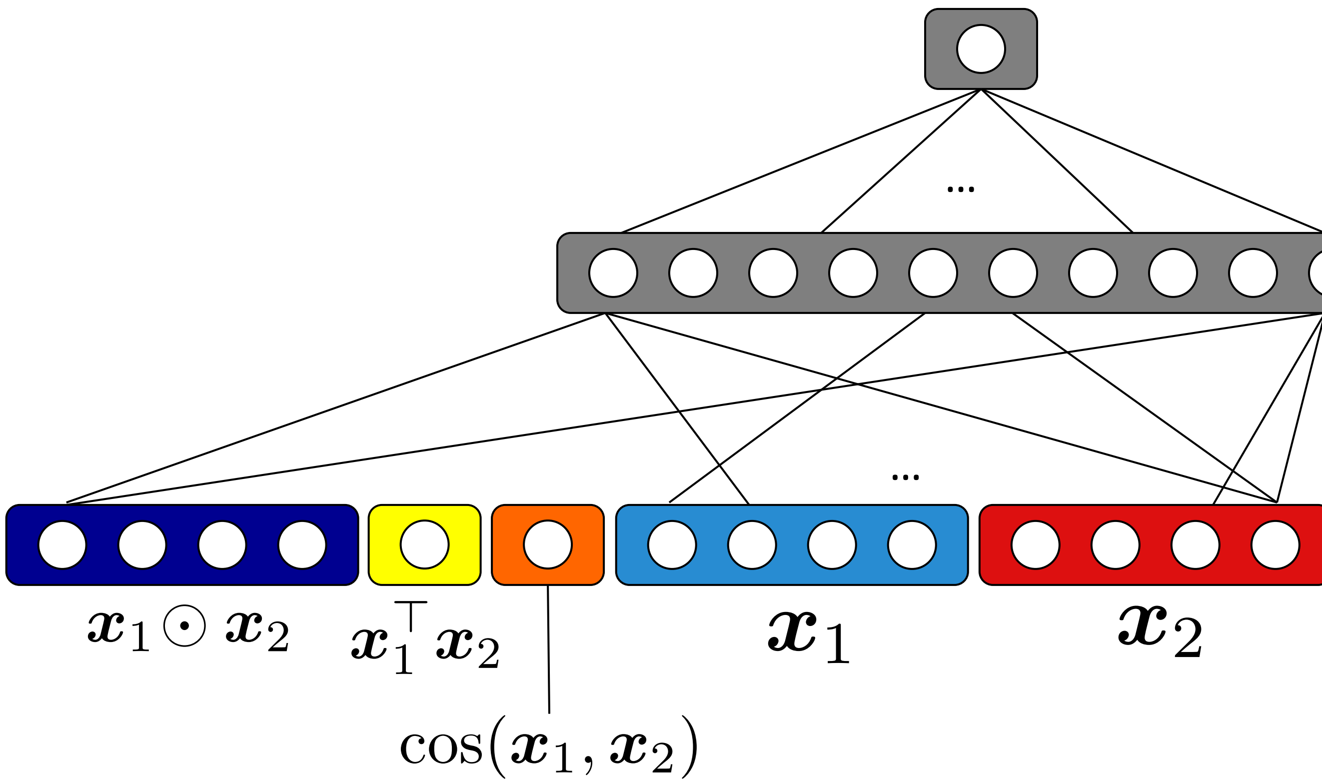
introduces parameters? **yes**

generalizes anything? **yes, can represent any function!**

Notes on DNN Similarity Functions

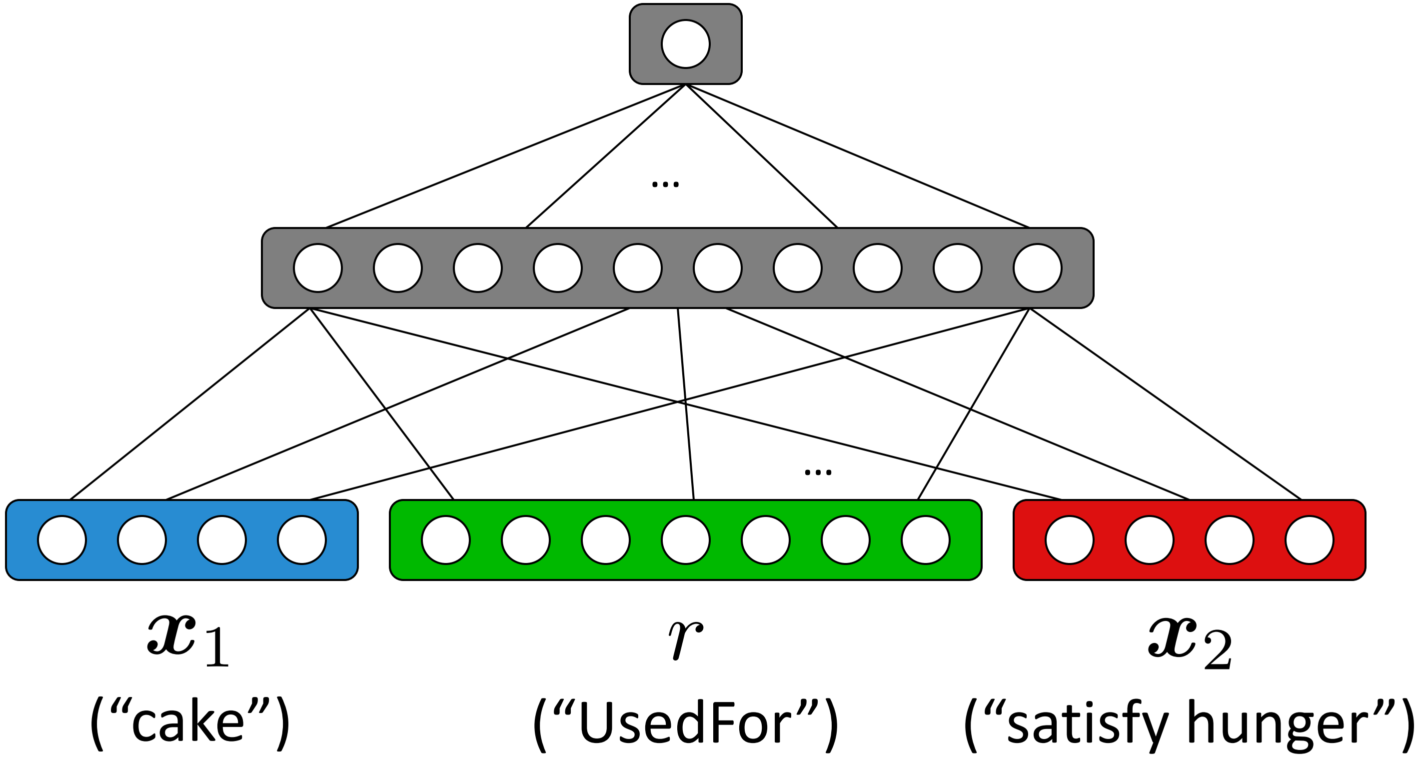
since DNNs are so powerful, things could go horribly wrong.

in practice, often pass additional quantities:



Deep Neural Network

similarity can depend on relation:



Learning for Similarity

- We want to learn input representations as well as all parameters of $sim(\boldsymbol{x}_1, \boldsymbol{x}_2)$
- We'll just write all these parameters as $\boldsymbol{\theta}$
- How about this?

$$\min_{\boldsymbol{\theta}} \sum_{\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle \in \text{train}} -sim(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

- Any potential problems with this?

(Better) Learning for Similarity

- Contrastive hinge loss:

$$\min_{\boldsymbol{\theta}} \sum_{\langle \boldsymbol{x}_1, \boldsymbol{x}_2 \rangle \in \text{train}} [-sim(\boldsymbol{x}_1, \boldsymbol{x}_2) + sim(\boldsymbol{x}_1, \boldsymbol{v})]_+$$

$$[a]_+ = \max(0, a)$$

- $\boldsymbol{v}$ is a “negative” example
- Any potential problems with this?

Learning with Neural Networks

$$\text{classify}(\boldsymbol{x}, \boldsymbol{\theta}) = \underset{\boldsymbol{y} \in \mathcal{L}}{\text{argmax}} \ \text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$$

$$\text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) = s_1 = g\left(W^{(1)}_{1,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_1\right)$$

$$\text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) = s_2 = g\left(W^{(1)}_{2,*} g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right) + \boldsymbol{b}^{(1)}_2\right)$$

- we can use any of our loss functions from before, as long as we can compute (sub)gradients
- algorithm for doing this efficiently: **backpropagation**
- it’s basically just the chain rule of derivatives

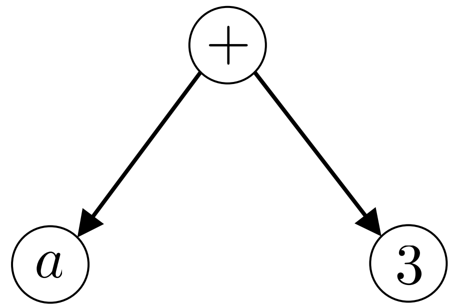
Computation Graphs

- a useful way to represent the computations performed by a neural model (or any model!)
- why useful? makes it easy to implement automatic differentiation (backpropagation)
- many neural net toolkits let you define your model in terms of computation graphs (Theano, (Py)Torch, TensorFlow, CNTK, DyNet, PENNE, etc.)

Backpropagation

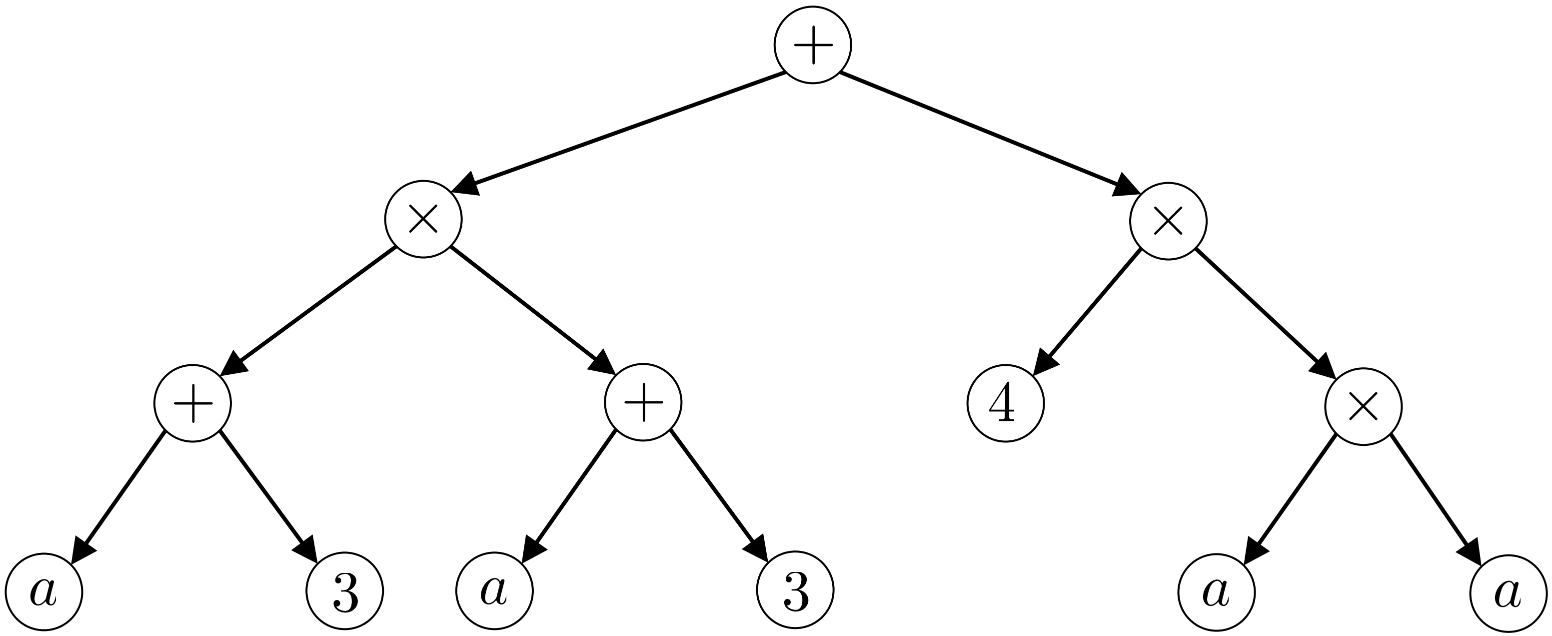
- backpropagation has become associated with neural networks, but it’s much more general
- I also use backpropagation to compute gradients in **linear** models for structured prediction

A simple computation graph:



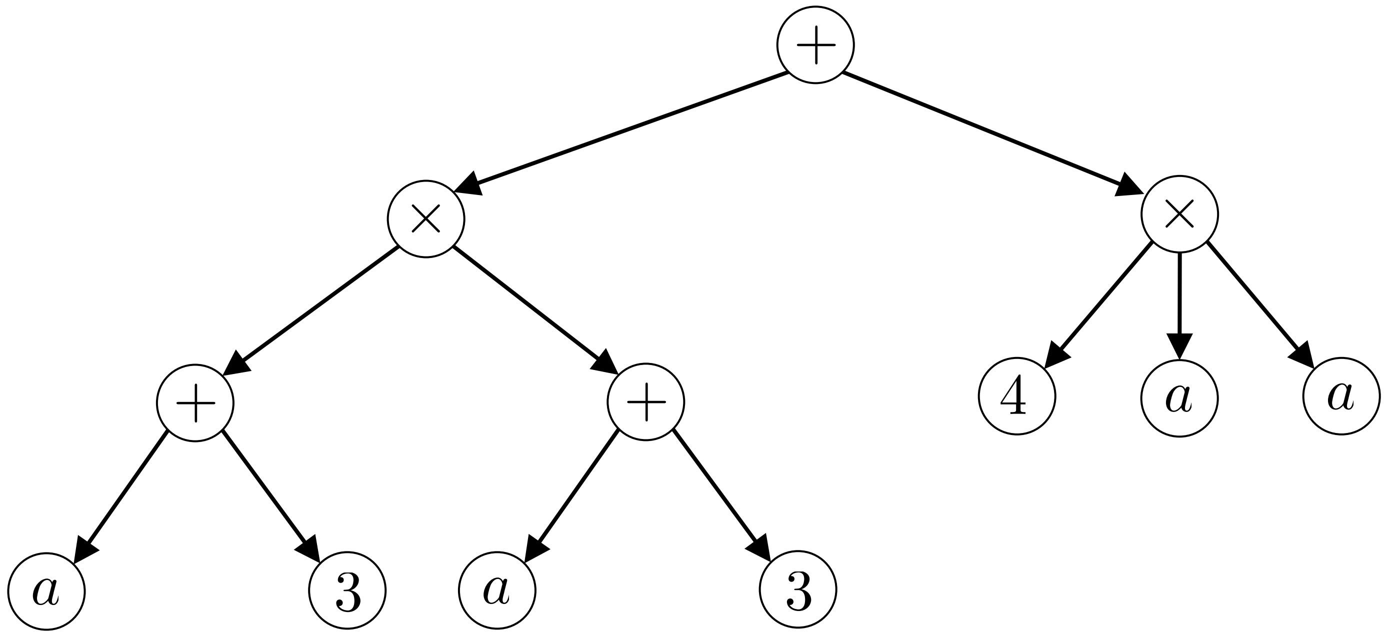
- represents expression “a + 3”

A slightly bigger computation graph:

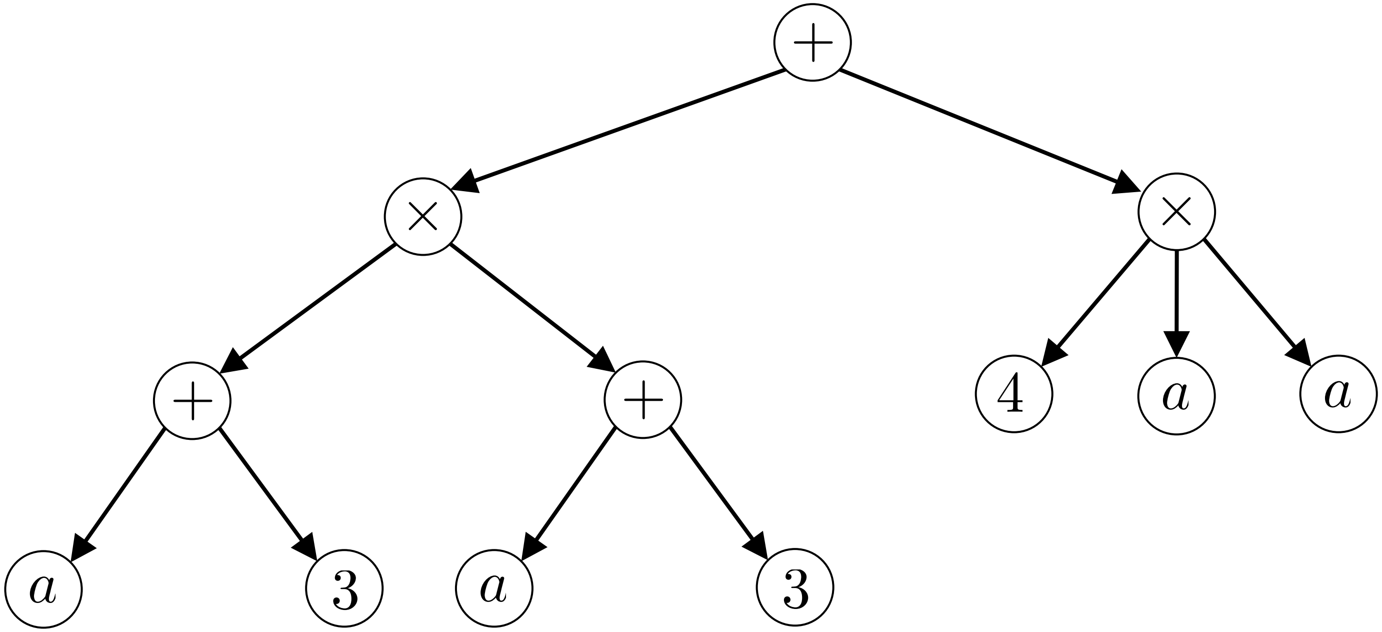


- represents expression “$(a + 3)^2 + 4a^2$”

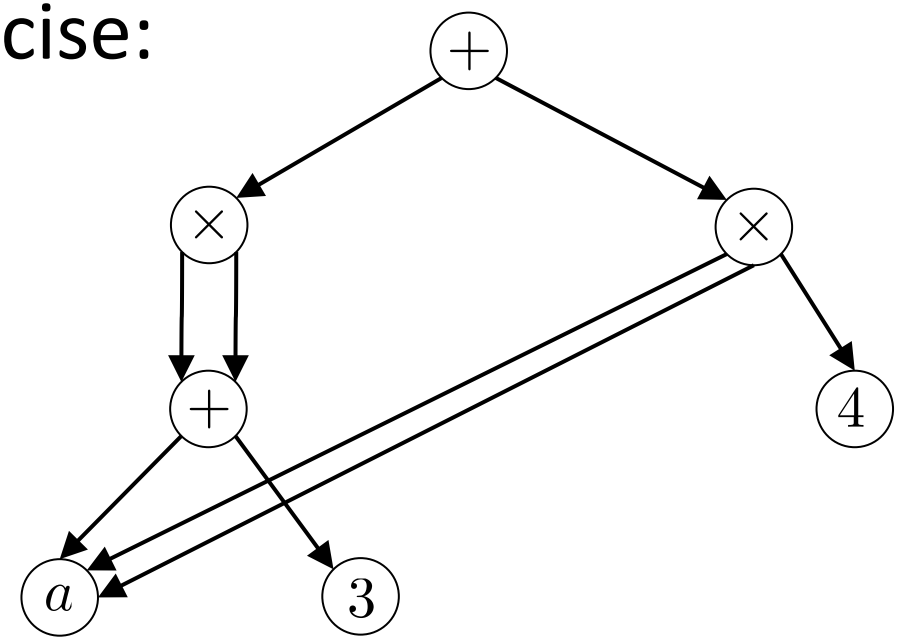
Operators can have more than 2 operands:



- still represents expression "(a + 3)^2 + 4a^2"



- more concise:



Overfitting & Regularization

- when we can fit any function, **overfitting** becomes a big concern
- overfitting: learning a model that does well on the training set but doesn't generalize to new data
- there are many strategies to reduce overfitting (we'll use the general term **regularization** for any such strategy)
- you used **early stopping** in Assignment 1, which is one kind of regularization

Regularization Terms

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \sum_{i=1}^{|\mathcal{T}|} \operatorname{loss}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta})$$

- most common: penalize large parameter values
- intuition: large parameters might be instances of overfitting
- examples:

***L_2* regularization**: $R_{\mathrm{L2}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2^2 = \sum_i \theta_i^2$

(also called Tikhonov regularization or ridge regression)

***L_1* regularization**: $R_{\mathrm{L1}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1 = \sum_i |\theta_i|$

(also called basis pursuit or LASSO)

Regularization Terms

***L_2* regularization**: $R_{\mathrm{L2}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_2^2 = \sum_i \theta_i^2$

differentiable, widely-used

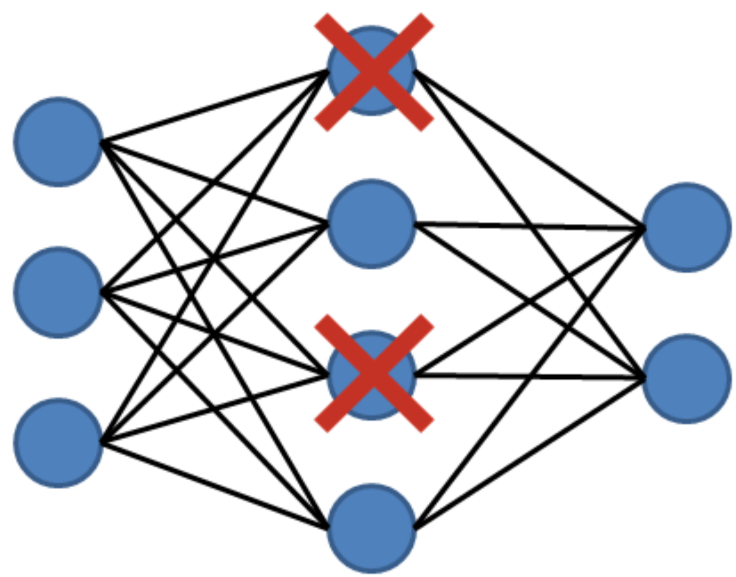
***L_1* regularization**: $R_{\mathrm{L1}}(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1 = \sum_i |\theta_i|$

not differentiable (but is subdifferentiable)

leads to sparse solutions (many parameters become zero!)

Dropout

- popular regularization method for neural networks
- randomly “drop out” (set to zero) some of the vector entries in the layers



Optimization Algorithms

- many choices:
  - SGD
  - AdaGrad
  - AdaDelta
  - RMSProp
  - Adam
  - SGD with momentum
- we don’t have time to go through these in class, but you should try using them! (most toolkits have implementations of these and others)

2-transformation (1-layer) network

$$z^{(1)} = g\left(W^{(0)}x + b^{(0)}\right)$$

$$s = g\left(W^{(1)}z^{(1)} + b^{(1)}\right)$$

vector of label scores

- we'll call this a "2-transformation" neural network, or a "1-layer" neural network
- input vector is x
- score vector is s
- one hidden vector $z^{(1)}$ ("hidden layer")

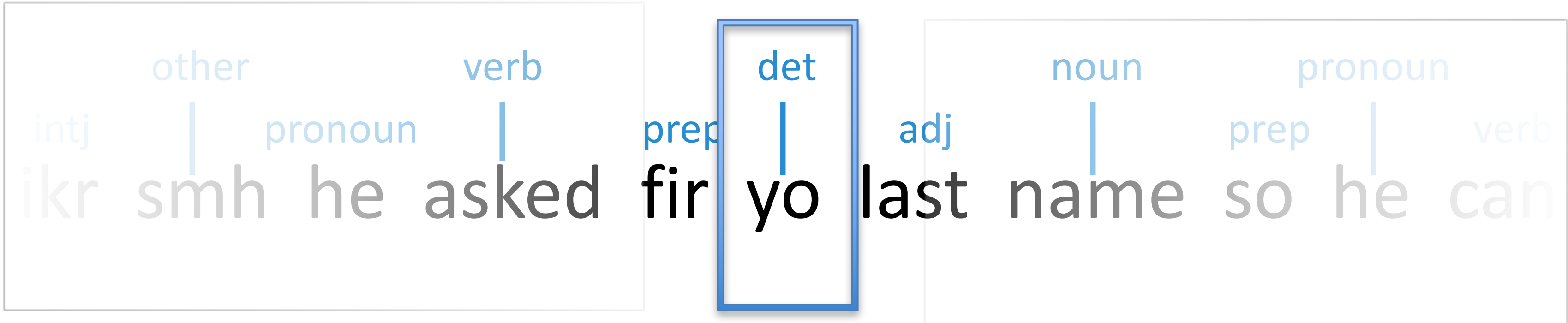
1-layer neural network for sentiment classification

$$\boldsymbol{z}^{(1)} = g\left(W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}\right)$$

$$\boldsymbol{s} = g\left(W^{(1)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{\theta}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{\theta}) \end{bmatrix}$$

Neural Networks for Twitter Part-of-Speech Tagging



- let's use the center word + two words to the right:

$$\boldsymbol{x} = [\underbrace{0.4 \;\; \ldots \;\; 0.9}_{\text{vector for } \textit{yo}} \;\; \underbrace{0.2 \;\; \ldots \;\; 0.7}_{\text{vector for } \textit{last}} \;\; \underbrace{0.3 \;\; \ldots \;\; 0.6}_{\text{vector for } \textit{name}}]^\top$$

- if *name* is to the right of *yo*, then *yo* is probably a form of *your*
- but our ***x*** above uses separate dimensions for each position!
  - i.e., *name* is two words to the right
  - what if *name* is one word to the right?

Convolution

$\boldsymbol{c}$ = "feature map", has an entry for each word position in context window / sentence

$$\boldsymbol{x} = [\underbrace{0.4 \;\ldots\; 0.9}_{\text{vector for } \textit{yo}} \; \underbrace{0.2 \;\ldots\; 0.7}_{\text{vector for } \textit{last}} \; \underbrace{0.3 \;\ldots\; 0.6}_{\text{vector for } \textit{name}}]^\top$$

$$c_1 = \boldsymbol{w} \cdot \boldsymbol{x}_{1:d}$$

$$c_2 = \boldsymbol{w} \cdot \boldsymbol{x}_{d+1:2d}$$

$$c_3 = \boldsymbol{w} \cdot \boldsymbol{x}_{2d+1:3d}$$

Pooling

$\boldsymbol{c}$ = "feature map", has an entry for each word position in context window / sentence

how do we convert this into a fixed-length vector?
use **pooling**:
max-pooling: returns maximum value in $\boldsymbol{c}$
average pooling: returns average of values in $\boldsymbol{c}$

vector for *yo* vector for *last* vector for *name*

$$c_1 = \boldsymbol{w} \cdot \boldsymbol{x}_{1:d}$$

$$c_2 = \boldsymbol{w} \cdot \boldsymbol{x}_{d+1:2d}$$

$$c_3 = \boldsymbol{w} \cdot \boldsymbol{x}_{2d+1:3d}$$

Pooling

$\boldsymbol{c}$ = "feature map", has an entry for each word position in context window / sentence

how do we convert this into a fixed-length vector?
use **pooling**:
    max-pooling: returns maximum value in $\boldsymbol{c}$
    average pooling: returns average of values in $\boldsymbol{c}$

vector for *yo*   vector for *last*   vector for *name*

$$c_1 = \boldsymbol{w} \cdot \boldsymbol{x}_{1:d}$$

then, this single filter $\boldsymbol{w}$ produces a single feature value (the output of some kind of pooling).
in practice, we use many filters of many different lengths (e.g., *n*-grams rather than words).

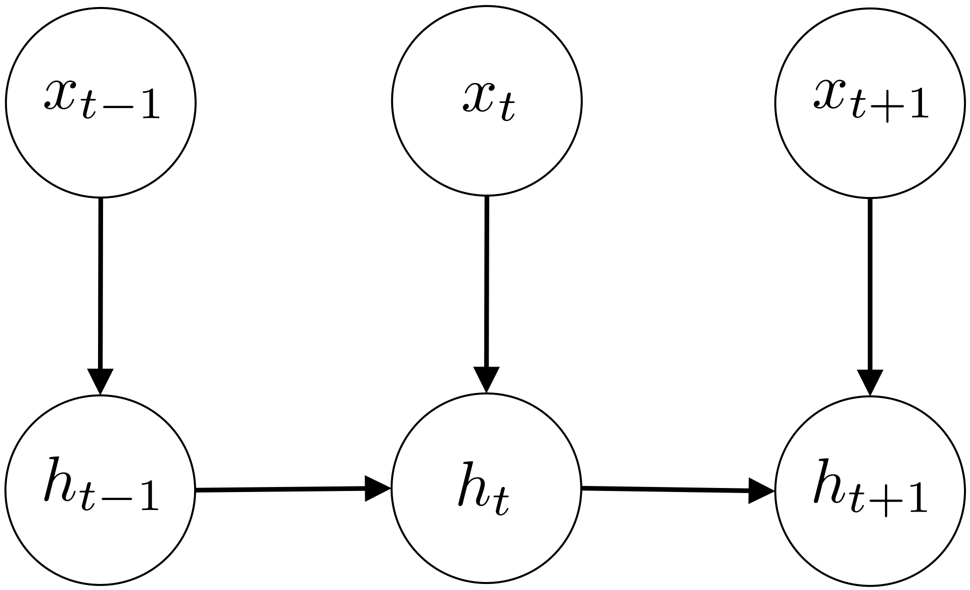
Convolutional Neural Networks

- convolutional neural networks (**convnets** or **CNNs**) use filters that are "convolved with" (matched against all positions of) the input
- think of convolution as "perform the same operation everywhere on the input in some systematic order"
- "convolutional layer" = set of filters that are convolved with the input vector (whether $\boldsymbol{x}$ or hidden vector)
- could be followed by more convolutional layers, or by a type of pooling
- often used in NLP to convert a sentence into a feature vector

Recurrent Neural Networks

$$h_t = \tanh\left(W^{(xh)}x_t + W^{(hh)}h_{t-1} + b^{(h)}\right)$$

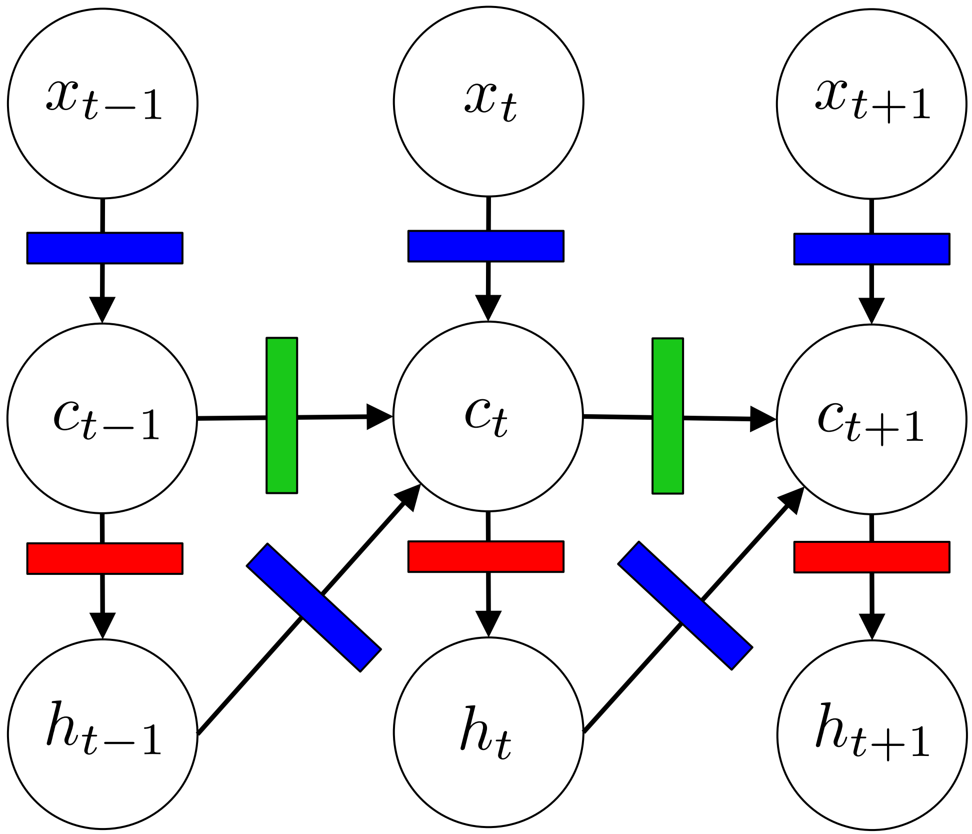
“hidden vector”



Long Short-Term Memory (LSTM) Recurrent Neural Networks

$$c_t = f_t c_{t-1} + i_t \tanh\left(W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)}\right)$$

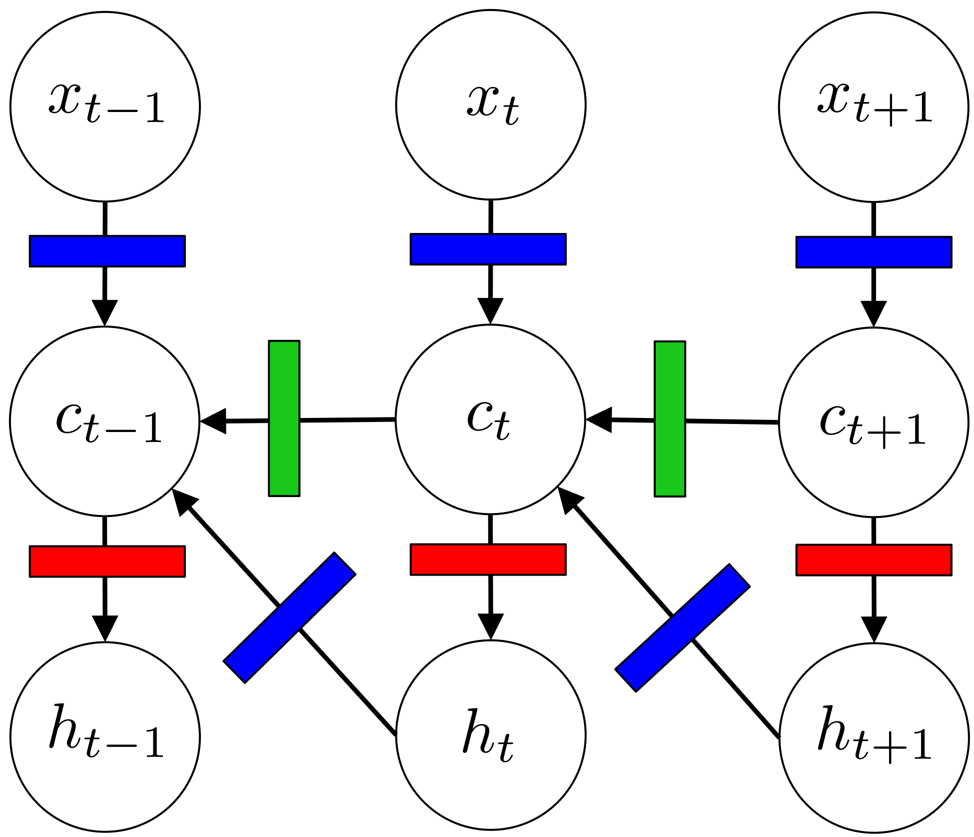
$$h_t = o_t \tanh(c_t)$$



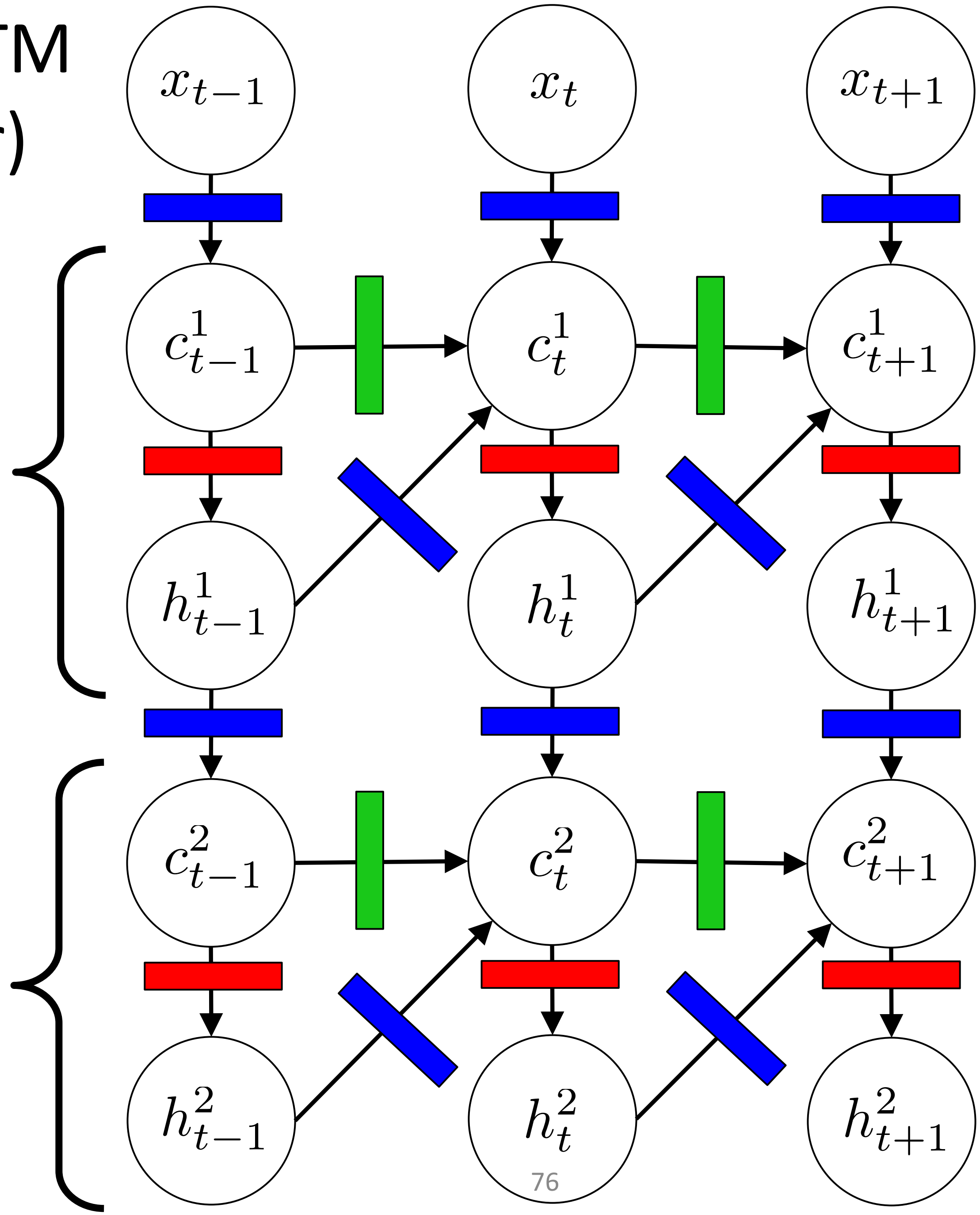
Backward & Bidirectional LSTMs

bidirectional:

if shallow, just use forward and backward LSTMs in parallel, concatenate final two hidden vectors, feed to softmax

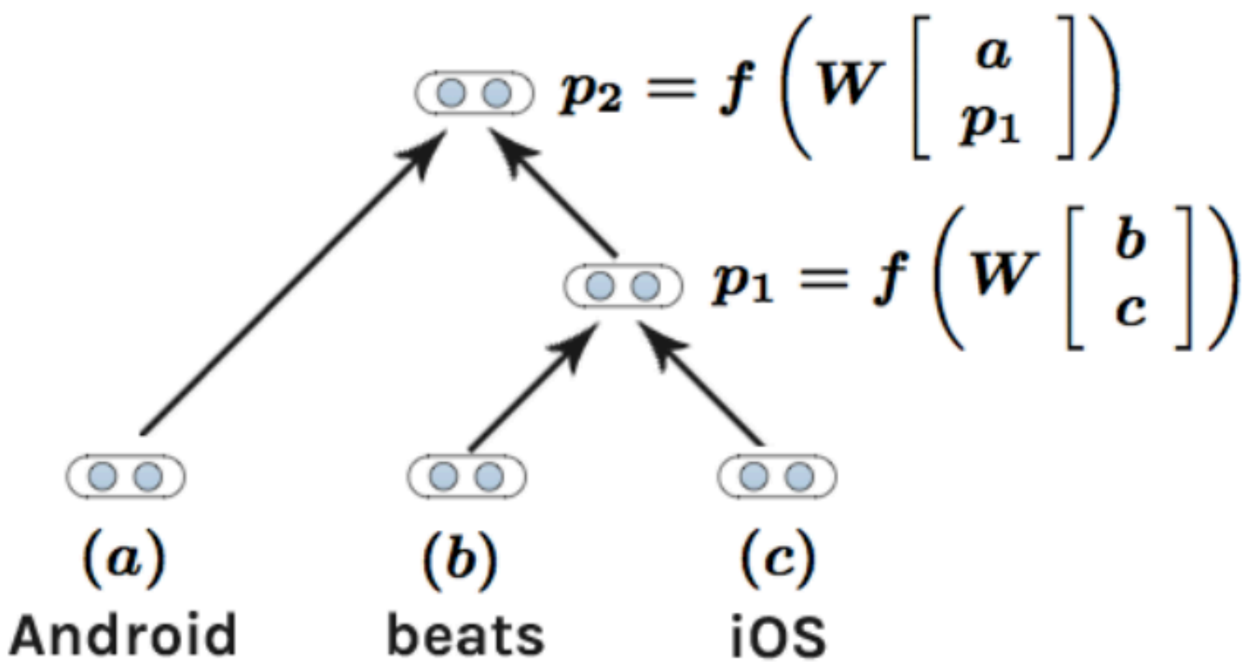


Deep LSTM (2-layer)



Recursive Neural Networks for NLP

- first, run a constituent parser on the sentence
- convert the constituent tree to a binary tree (each rewrite has exactly two children)
- construct vector for sentence recursively at each rewrite ("split point"):



Cost Functions

- cost function: scores output against a gold standard

$$\text{cost} : \mathcal{L} \times \mathcal{L} \to \mathbb{R}_{\geq 0}$$

- should reflect the evaluation metric for your task

- usual conventions: $\text{cost}(y, y) = 0$ $\quad\quad$ $\text{cost}(y, y') = \text{cost}(y', y)$

- for classification, what cost should we use?

$$\text{cost}(y, y') = \mathbb{I}[y \neq y']$$