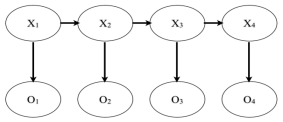
Deep Learning for NLP

CS224N

Christopher Manning

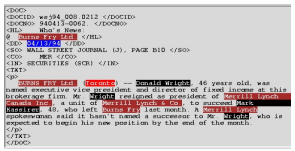
(Many slides borrowed from ACL 2012/NAACL 2013
Tutorials by me, Richard Socher and Yoshua Bengio)

Machine Learning and NLP

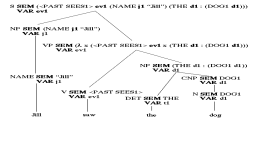


NER

WordNet



SRL

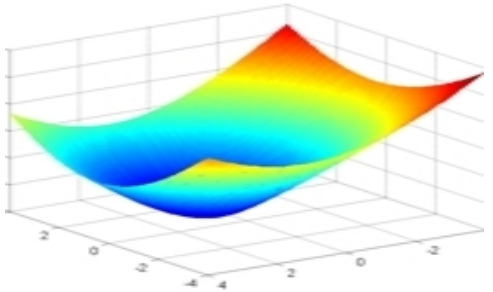


Parser

Usually machine learning works well because of human-designed representations and input features

Machine learning becomes just optimizing weights to best make a final prediction

PA1 and PA3?



Representation learning attempts to automatically learn good features or representations

Deep Learning and its Architectures

Deep learning attempts to learn multiple levels of representation
Focus: Multi-layer neural networks

Output layer

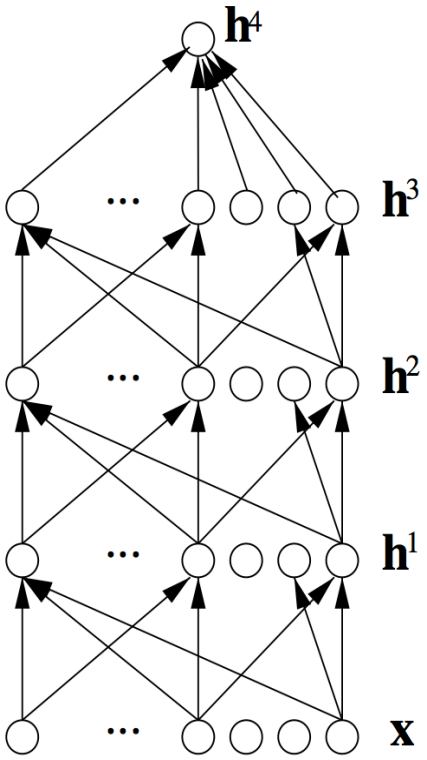
Here predicting a supervised target

Hidden layers

These learn more abstract representations as you head up

Input layer

Raw sensory inputs (roughly)



Outline for next two lectures

1. The Basics
    1. From logistic regression to neural networks
    2. Word vector representations
    3. Unsupervised word vector learning (backpropagation)
    4. Learning word-level classifiers: POS and NER
    5. Sharing statistical strength and Autoencoders
2. Sentence Level Sequence Models:
    1. Recursive Neural Networks
    2. Sentiment Analysis, Parsing, Relation Classification
3. Other Deep NLP Applications and Practical Tricks

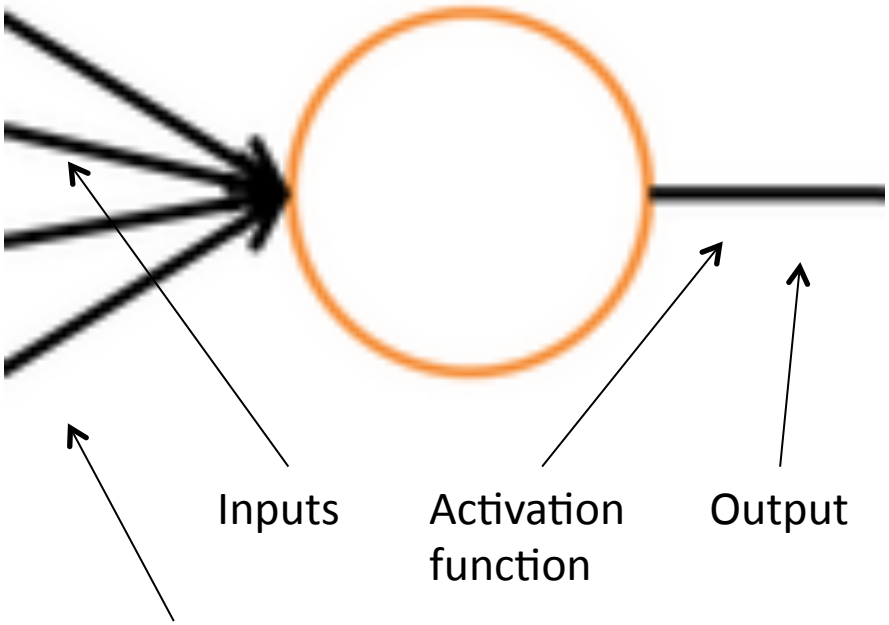
From logistic regression to neural nets

Demystifying neural networks

Terminology

Math is similar to maxent/ logistic regression

A single neuron
A computational unit with *n* (3) inputs
and 1 output
and parameters *W, b*



From Maxent Classifiers to Neural Networks

Refresher: Maxent classifier (from last lecture)

$$P(c \mid d, \lambda) = \frac{\exp \sum_i \lambda_i f_i(c,d)}{\sum_{c' \in C} \exp \sum_i \lambda_i f_i(c',d)}$$

Supervised learning gives us a distribution for datum *d* over classes in *C*

New: Vector form: $P(c \mid d, \lambda) = \frac{e^{\lambda^\top f(c,d)}}{\sum_{c'} e^{\lambda^\top f(c',d)}}$

Such a classifier is used as-is in a neural network ("a softmax layer")

- Often as the top layer

But for now we'll derive a two-class logistic model for one neuron

From Maxent Classifiers to Neural Networks

Vector form: $$P(c \mid d, \lambda) = \frac{e^{\lambda^\mathsf{T} f(c,d)}}{\sum_{c'} e^{\lambda^\mathsf{T} f(c',d)}}$$

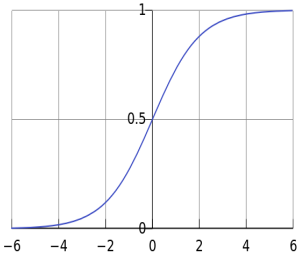
Make two class:

$$P(c_1 \mid d, \lambda) = \frac{e^{\lambda^\mathsf{T} f(c_1,d)}}{e^{\lambda^\mathsf{T} f(c_1,d)} + e^{\lambda^\mathsf{T} f(c_2,d)}} = \frac{e^{\lambda^\mathsf{T} f(c_1,d)}}{e^{\lambda^\mathsf{T} f(c_1,d)} + e^{\lambda^\mathsf{T} f(c_2,d)}} \cdot \frac{e^{-\lambda^\mathsf{T} f(c_1,d)}}{e^{-\lambda^\mathsf{T} f(c_1,d)}}$$

$$= \frac{1}{1 + e^{\lambda^\mathsf{T}[f(c_2,d) - f(c_1,d)]}} = \frac{1}{1 + e^{-\lambda^\mathsf{T} x}}$$

for $x = f(c_1,d) - f(c_2,d)$

$$= f(\lambda^\mathsf{T} x)$$



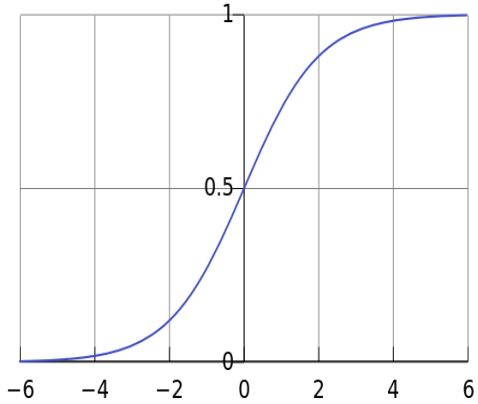
for $f(z) = 1/(1 + \exp(-z))$, the logistic function – a sigmoid non-linearity.

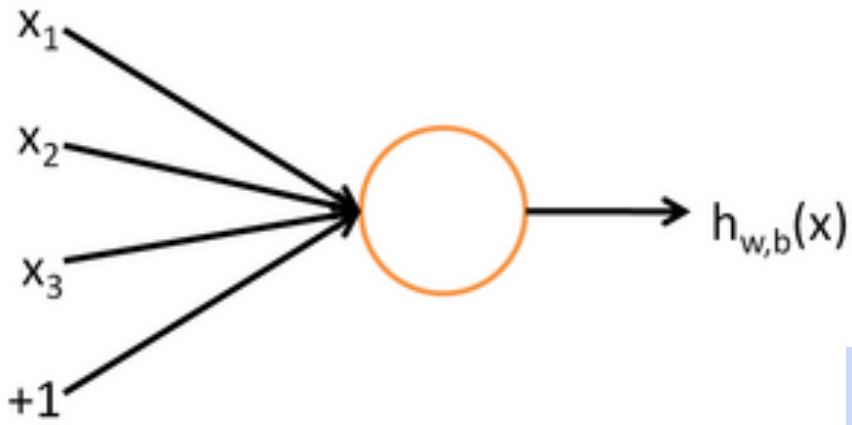
This is exactly what an artificial "neuron" computes

$$h_{w,b}(x) = f(w^{\mathsf{T}}x + b)$$

b: We can have an "always on" feature, which gives a class prior, or separate it out, as a bias term

$$f(z) = \frac{1}{1+e^{-z}}$$

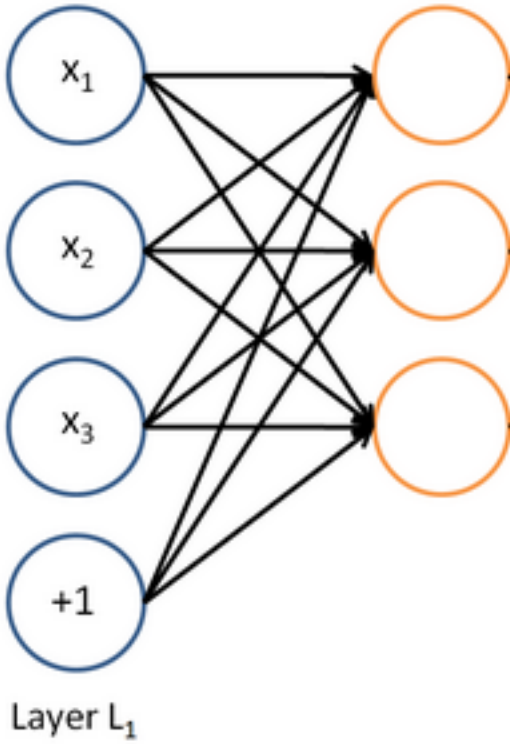




w, *b* are the parameters of this neuron i.e., this logistic regression model

A neural network = running several logistic regressions at the same time

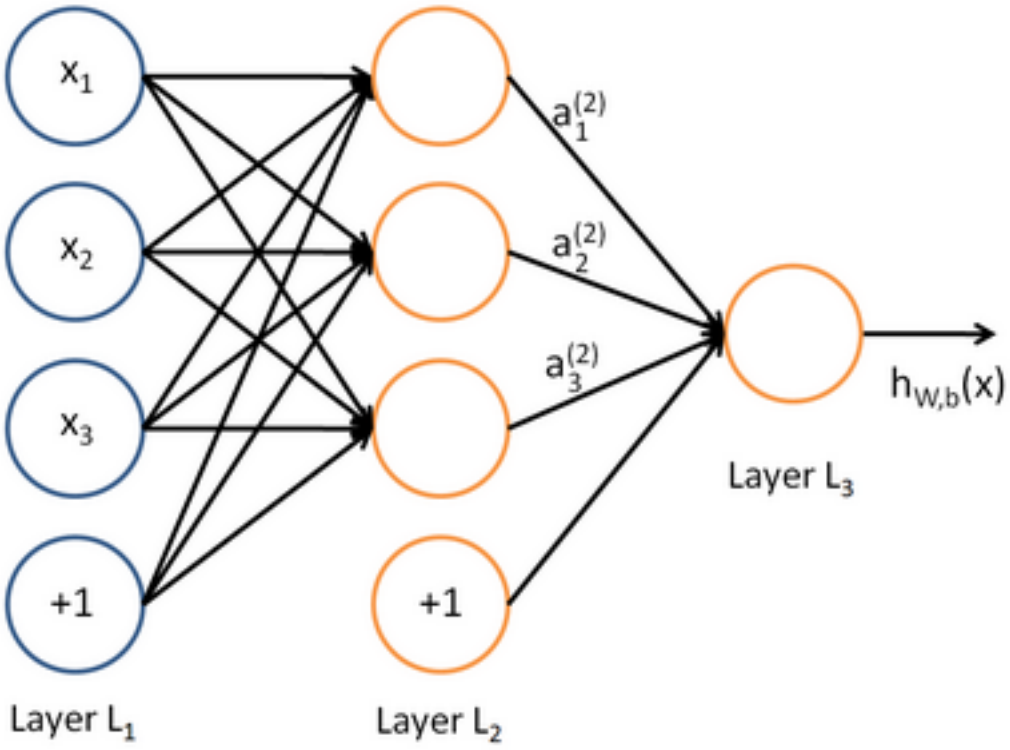
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs



But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

A neural network = running several logistic regressions at the same time

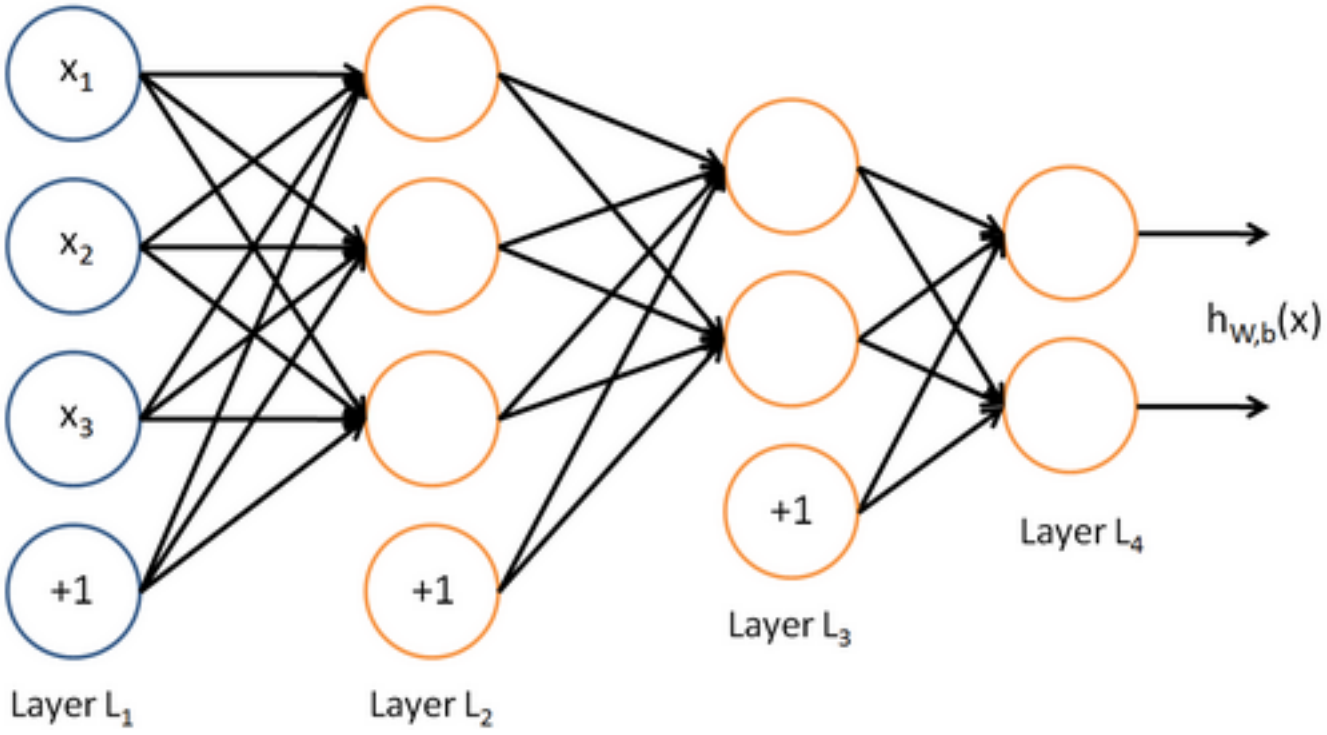
… which we can feed into another logistic regression function



and it is the training criterion that will decide what those intermediate binary variables should be, so as to do a good job of predicting the targets for the next layer, etc.

A neural network = running several Logistic regressions at the same time

Before we know it, we have a multilayer neural network….



Matrix notation for a layer

We have

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

etc.

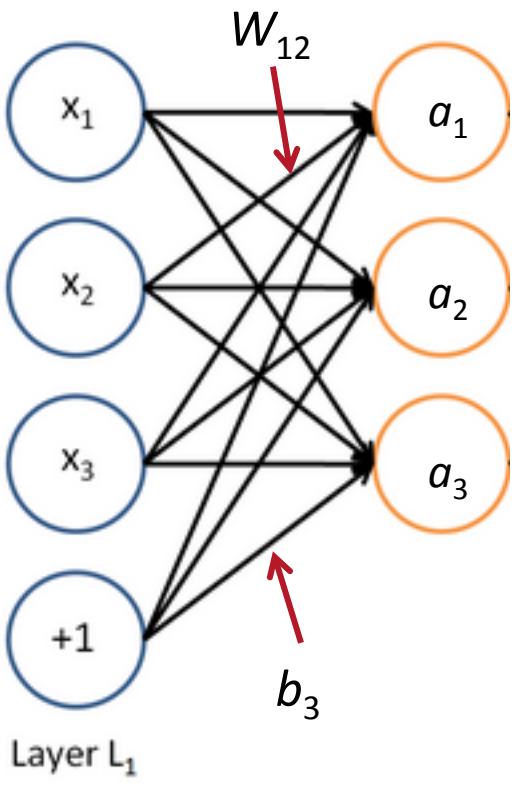
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

where *f* is applied element-wise:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

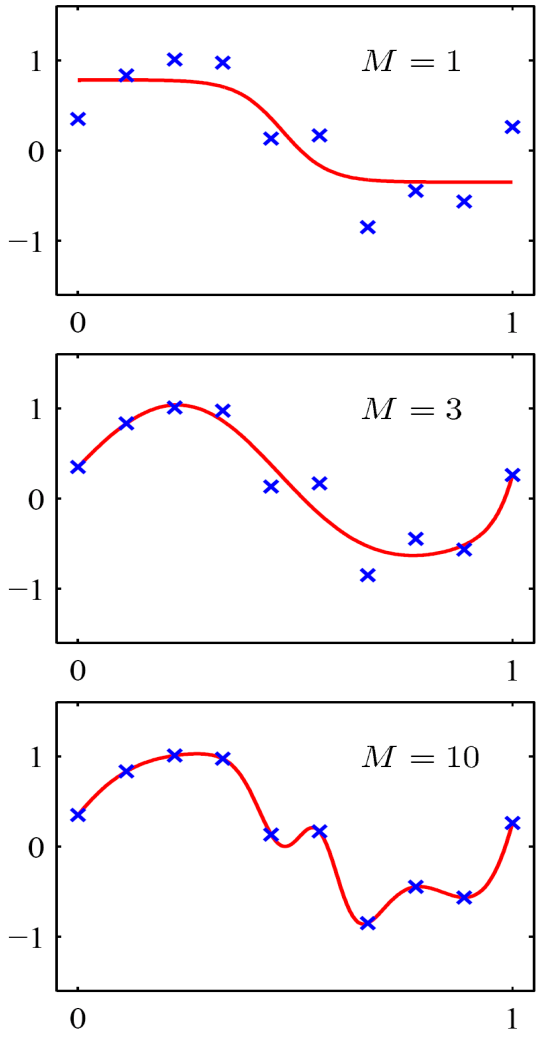


How do we train the weights *W*?

- For a supervised neural net, we can train the model just like a maxent model – we calculate and use gradients
  - Stochastic gradient descent (SGD)
  - Conjugate gradient or L-BFGS
- We can use the same ideas and techniques
  - Just without guarantees …
- This leads to "backpropagation", which we cover later

Non-Linearities: Why they're needed

- For logistic regression: map to probabilities
- Here: function approximation,
  e.g., regression or classification
  - Without non-linearities, deep neural networks can't do anything more than a linear transform
    - Extra layers could just be compiled down into a single linear transform
- People often use other non-linearities, such as **tanh, ReLU**, as we'll discuss in part 3



Summary
Knowing the meaning of words!

You now understand the basics and the relation to other models

- Neuron = logistic regression or similar function
- Input layer = input training/test vector
- Bias unit = intercept term/always on feature
- Activation = response
- Activation function is a logistic or other nonlinearity
- Backpropagation = running stochastic gradient descent through a multilayer network efficiently
- Weight decay = regularization / Bayesian prior smoothing

Word Representations

The standard word representation

The vast majority of rule-based **and** statistical NLP work regards words as atomic symbols: hotel, conference, walk

In vector space terms, this is a vector with one 1 and a lot of zeroes

[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

Dimensionality: 20K (speech) – 50K (PTB) – 500K (big vocab) – 13M (Google 1T)

We call this a "one-hot" representation. Its problem:

motel [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] AND
hotel [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0] = 0

Distributional similarity based representations

You can get a lot of value by representing a word by means of its neighbors

> "You shall know a word by the company it keeps"
>
> (J. R. Firth 1957: 11)

One of the most successful ideas of modern statistical NLP

government debt problems turning into	banking	crises as has happened in
saying that Europe needs unified	banking	regulation to replace the hodgepodge

↖ These words will represent *banking* ↗

You can vary whether you use local or large context to get a more syntactic or semantic clustering

Two traditional word representations: Class-based and soft clustering

Class based models learn word classes of similar words based on distributional information (~ class HMM)

- Brown clustering (Brown et al. 1992, Liang 2005)
- Exchange clustering (Martin et al. 1998, Clark 2003)

Soft clustering models learn for each cluster/topic a distribution over words of how likely that word is in each cluster

- Latent Semantic Analysis (LSA/LSI), Random projections
- Latent Dirichlet Analysis (LDA), HMM clustering

Neural word embeddings as a distributed representation

Similar idea

Combine vector space semantics with the prediction of probabilistic models (Bengio et al. 2003, Collobert & Weston 2008, Huang et al. 2012)

In all of these approaches, including deep learning models, a word is represented as a dense vector

linguistics =

$$\begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

Neural word embeddings - visualization

Advantages of the neural word embedding approach

Compared to other methods, neural word embeddings can become more meaningful through adding supervision from one or multiple tasks

- For instance, sentiment is usually not captured in unsupervised word embeddings but can be in neural word vectors

We can build compositional vector representations for longer phrases (next lecture)

Unsupervised word vector learning

A neural network for learning word vectors (Collobert et al. JMLR 2011)

Idea: A word and its context is a positive training sample; a random word in that same context gives a negative training sample:

cat chills **on** a mat

cat chills Ohio a mat

A neural network for learning word vectors

How do we formalize this idea? Ask that

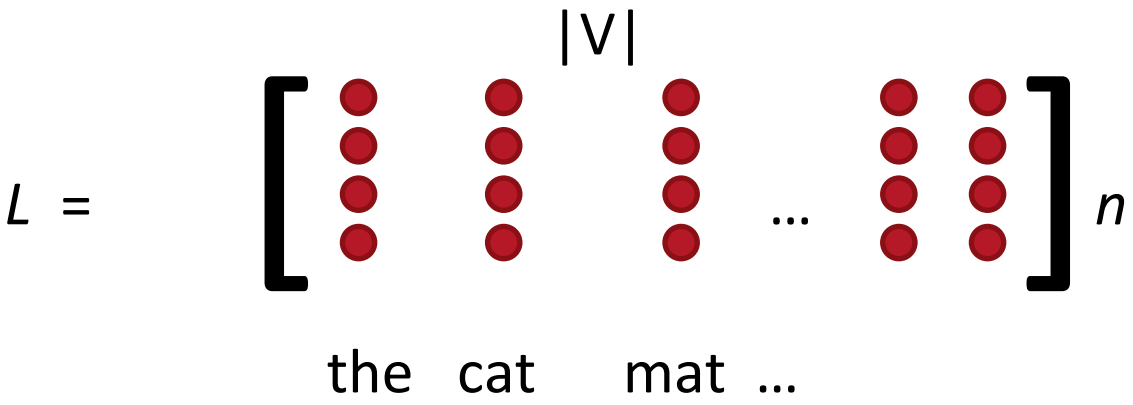
score(cat chills on a mat) > score(cat chills Ohio a mat)

How do we compute the score?

- With a neural network
- Each word is associated with an *n*-dimensional vector

Word embedding matrix

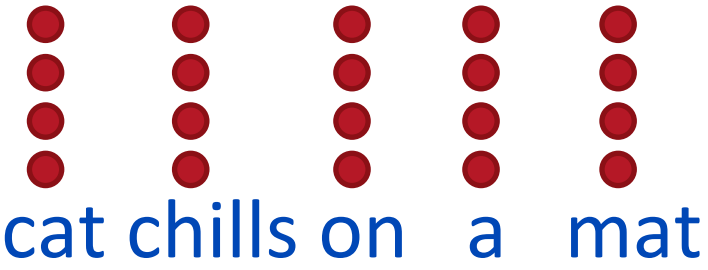
- Initialize all word vectors randomly to form a word embedding matrix $L \in \mathbb{R}^{n \times |V|}$



- These are the word features we want to learn
- Also called a look-up table
  - Mathematically you get a word's vector by multiplying L with a one-hot vector *e*: $x = Le$

Word vectors as input to a neural network

- score(cat chills on a mat)
- To describe a phrase, retrieve (via index) the corresponding vectors from *L*



- Then concatenate them to form a 5*n* vector:
- *x* =[●●●● ●●●● ●●●● ●●●● ●●●●]
- How do we then compute score(x)?

Scoring a Single Layer Neural Network

- A single layer is a combination of a linear layer and a nonlinearity:

$$z = Wx + b$$
$$a = f(z)$$

- The neural activations can then be used to compute some function.
- For instance, the score we care about:

$$score(x) = U^T a \in \mathbb{R}$$

Summary: Feed-forward Computation

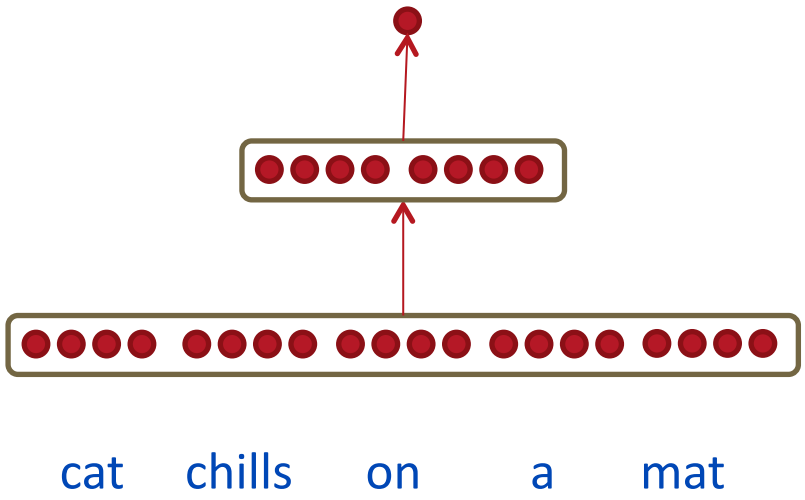
Computing a window's score with a 3-layer Neural Net: *s = score*(cat chills on a mat)

$$s = U^T f(Wx + b) \qquad x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$

$$\begin{aligned} s &= U^T a \\ a &= f(z) \\ z &= Wx + b \end{aligned}$$

$$x = [x_{cat} \quad x_{chills} \quad x_{on} \quad x_a \quad x_{mat}]$$

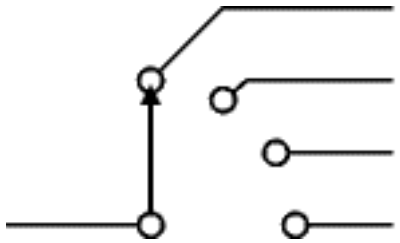
$$L \in \mathbb{R}^{n\times|V|}$$



Summary: Feed-forward Computation

- s = score(cat chills on a mat)
- s_c = score(cat chills Ohio a mat)
- Idea for training objective: make score of true window larger and **c**orrupt window's score lower (until they're good enough): minimize

$$J = \max(0, 1 - s + s_c)$$



- This is continuous, can perform SGD
  - Look at a few examples, nudge weights to make *J* smaller

Training with Backpropagation

$$J = \max(0, 1 - s + s_c)$$

$$s = U^T f(Wx + b)$$
$$s_c = U^T f(Wx_c + b)$$

Assuming cost *J* is > 0, it is simple to see that we can compute the derivatives of *s* and s_c wrt all the involved variables: *U, W, b, x*

$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a \qquad \frac{\partial s}{\partial U} = a$$

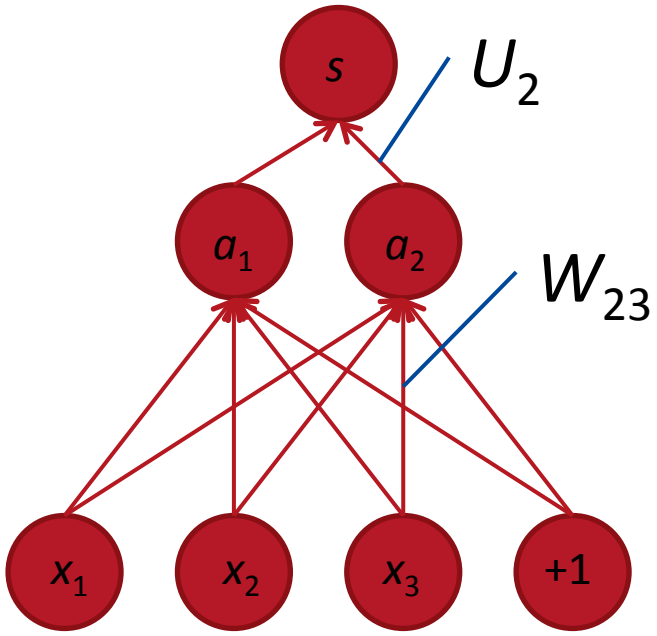
Training with Backpropagation

- Let's consider the derivative of a single weight W_{ij}

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

- This only appears inside a_i

- For example: W_{23} is only used to compute a_2



Training with Backpropagation

$$\frac{\partial s}{\partial W} = \frac{\partial}{\partial W} U^T a = \frac{\partial}{\partial W} U^T f(z) = \frac{\partial}{\partial W} U^T f(Wx + b)$$

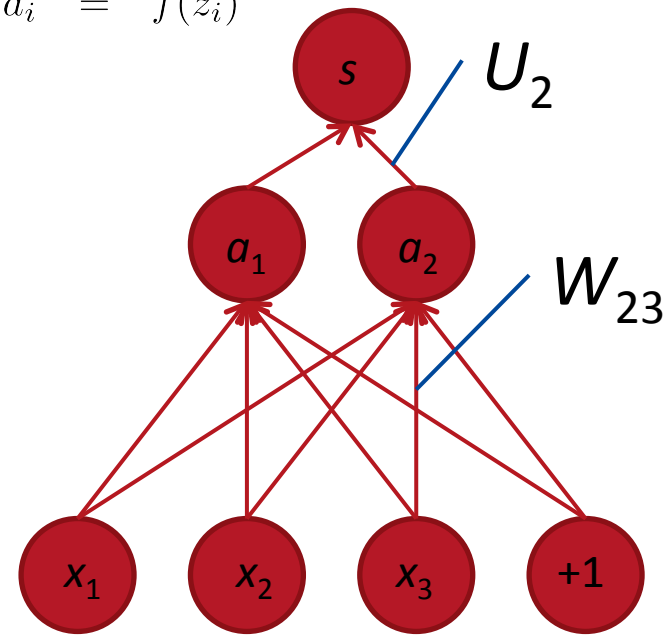
Derivative of weight W_{ij}:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x}$$

$$\frac{\partial}{\partial W_{ij}} U^T a \quad \rightarrow \quad \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$\begin{aligned} U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}} \\ &= U_i f'(z_i) \frac{\partial W_i . x + b_i}{\partial W_{ij}} \end{aligned}$$

$$\begin{aligned} z_i &= W_i . x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i \\ a_i &= f(z_i) \end{aligned}$$



Training with Backpropagation

Derivative of single weight W_{ij} :

$$U_i \frac{\partial}{\partial W_{ij}} a_i$$

$$= U_i f'(z_i) \frac{\partial W_i.x + b_i}{\partial W_{ij}}$$

$$= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k$$

$$= \underbrace{U_i f'(z_i)}_{\delta_i} x_j$$

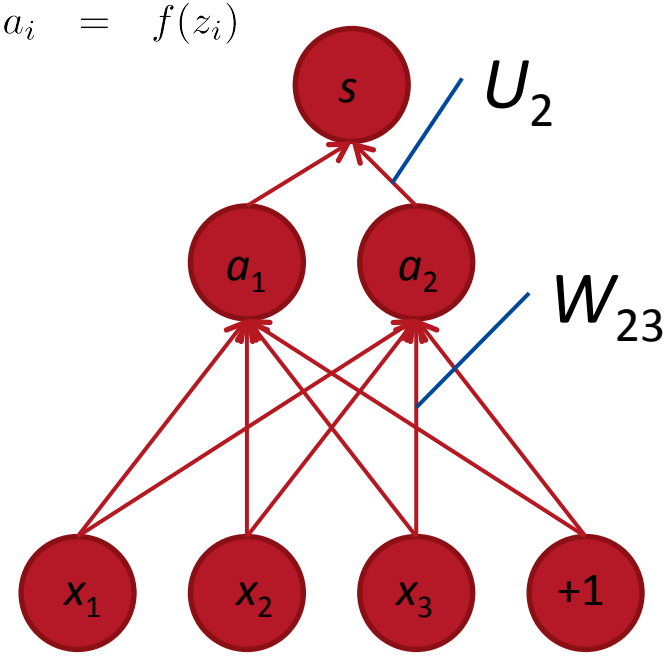
$$= \delta_i \; x_j$$

Local error signal

Local input signal

$$z_i = W_i.x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$



where $f'(z) = f(z)(1 - f(z))$ for logistic f

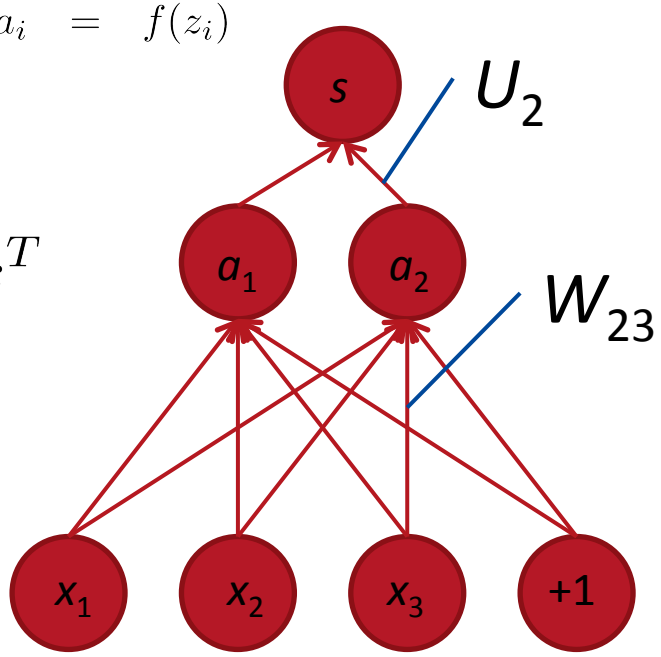
Training with Backpropagation

- From single weight W_{ij} to full W:

$$\begin{aligned}\frac{\partial J}{\partial W_{ij}} &= \underbrace{U_i f'(z_i)}_{\delta_i} x_j \\ &= \delta_i \quad x_j\end{aligned}$$

- We want all combinations of $i = 1, 2$ and $j = 1, 2, 3$
- Solution: Outer product: $\frac{\partial J}{\partial W} = \delta x^T$ where $\delta \in \mathbb{R}^{2\times 1}$ is the "responsibility" coming from each activation a

$$\begin{aligned} z_i &= W_{i\cdot}x + b_i = \sum_{j=1}^{3} W_{ij}x_j + b_i \\ a_i &= f(z_i)\end{aligned}$$

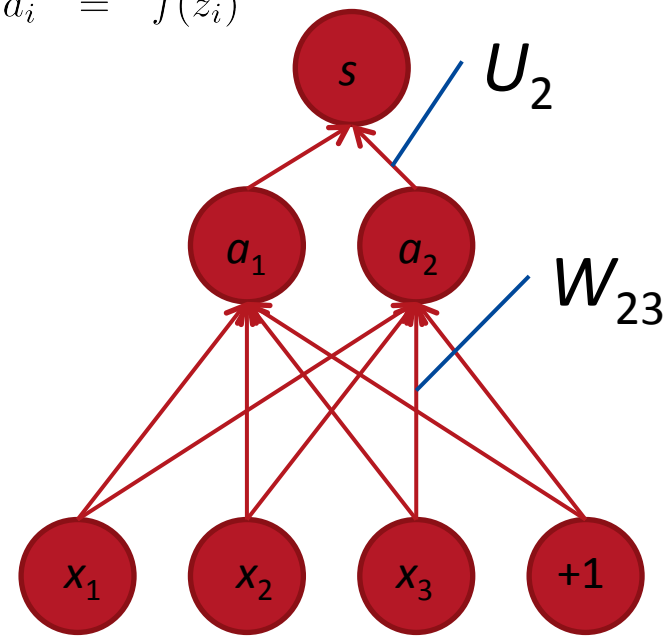


Training with Backpropagation

- For biases *b*, we get:

$$
\begin{aligned}
& U_i \frac{\partial}{\partial b_i} a_i \\
= \quad & U_i f'(z_i) \frac{\partial W_i.x + b_i}{\partial b_i} \\
= \quad & \delta_i
\end{aligned}
$$

$$
\begin{aligned}
z_i &= W_i.x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i \\
a_i &= f(z_i)
\end{aligned}
$$



Training with Backpropagation

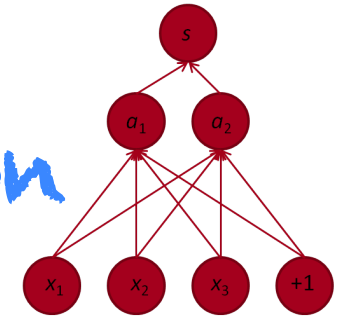
That's almost backpropagation

It's simply taking derivatives and using the chain rule!

Remaining trick: we can re-use derivatives computed for higher layers in computing derivatives for lower layers

Example: last derivatives of model, the word vectors in *x*

Training with Backpropagation



- Take derivative of score with respect to single word vector (for simplicity a 1d vector, but same if it were longer)
- Now, we cannot just take into consideration one a_i because each x_j is connected to all the neurons above and hence x_j influences the overall score through all of these, hence:

$$
\begin{aligned}
\frac{\partial s}{\partial x_j} &= \sum_{i=1}^{2} \frac{\partial s}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
&= \sum_{i=1}^{2} \frac{\partial U^T a}{\partial a_i} \frac{\partial a_i}{\partial x_j} \\
&= \sum_{i=1}^{2} U_i \frac{\partial f(W_i.x + b)}{\partial x_j} \\
&= \sum_{i=1}^{2} \underbrace{U_i f'(W_i.x + b)}_{} \frac{\partial W_i.x}{\partial x_j} \\
&= \sum_{i=1}^{2} \delta_i W_{ij} \\
&= \underbrace{\delta^T}_{} W_{\cdot j}
\end{aligned}
$$

Re-used part of previous derivative

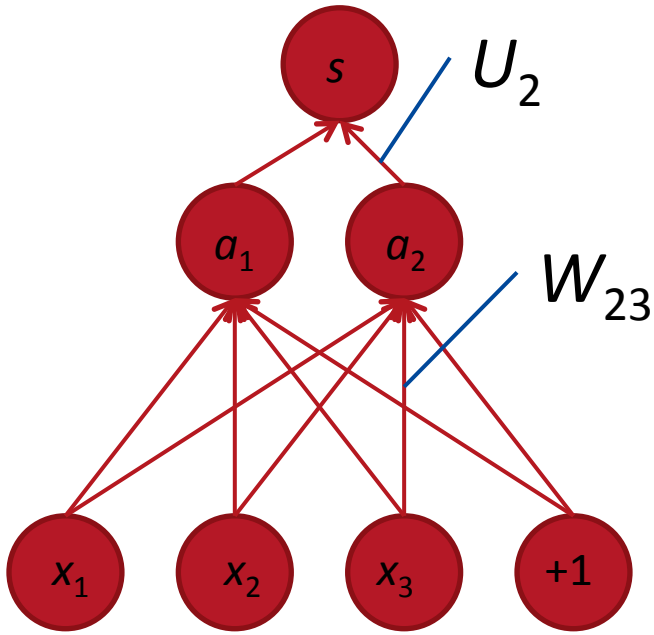
Learning word-level classifiers: POS and NER

The Model

(Collobert & Weston 2008;
Collobert et al. 2011)

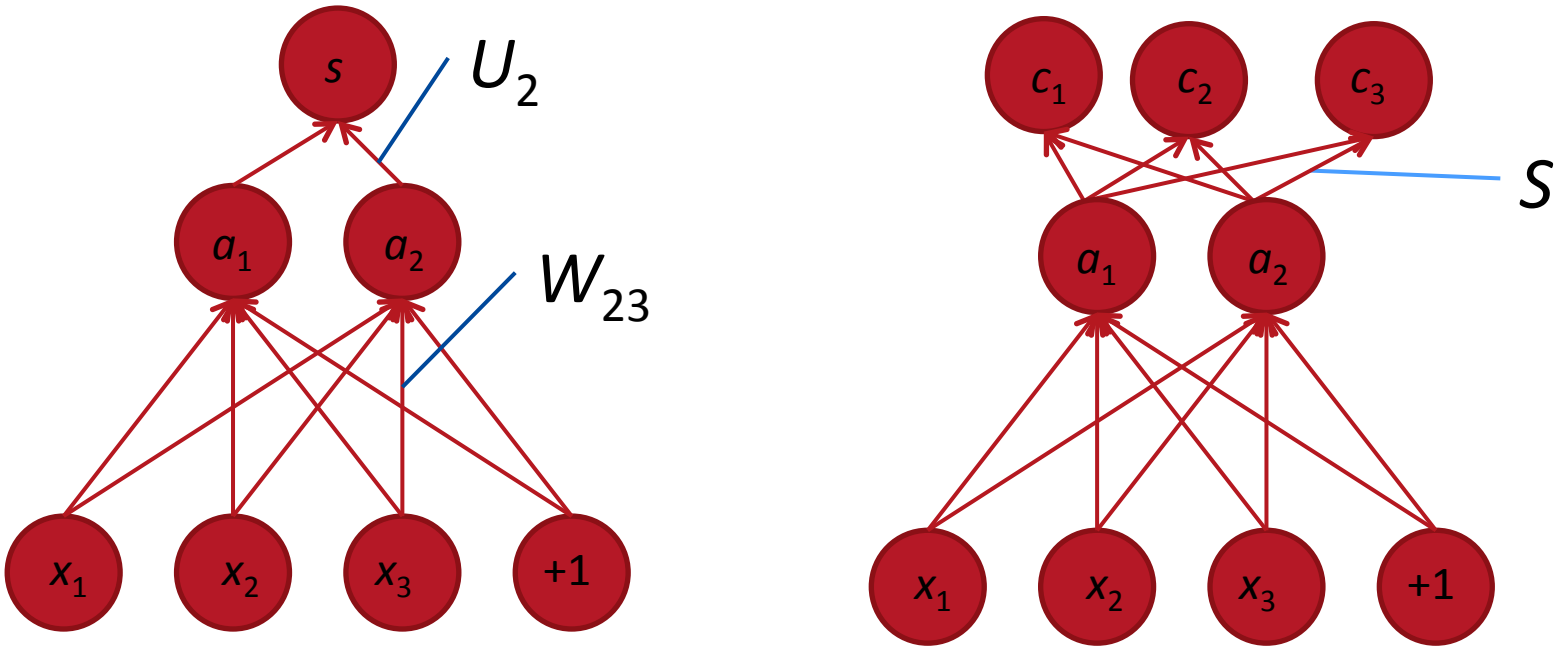
- Similar to word vector learning but replaces the single scalar score with a *Softmax*/Maxent classifier

- Training is again done via backpropagation which gives an error similar to (*but not the same as!*) the score in the unsupervised word vector learning model



The Model – Training

- We already know softmax/MaxEnt and how to optimize it
- The interesting twist in deep learning is that the input features are also learned, similar to learning word vectors with a score:



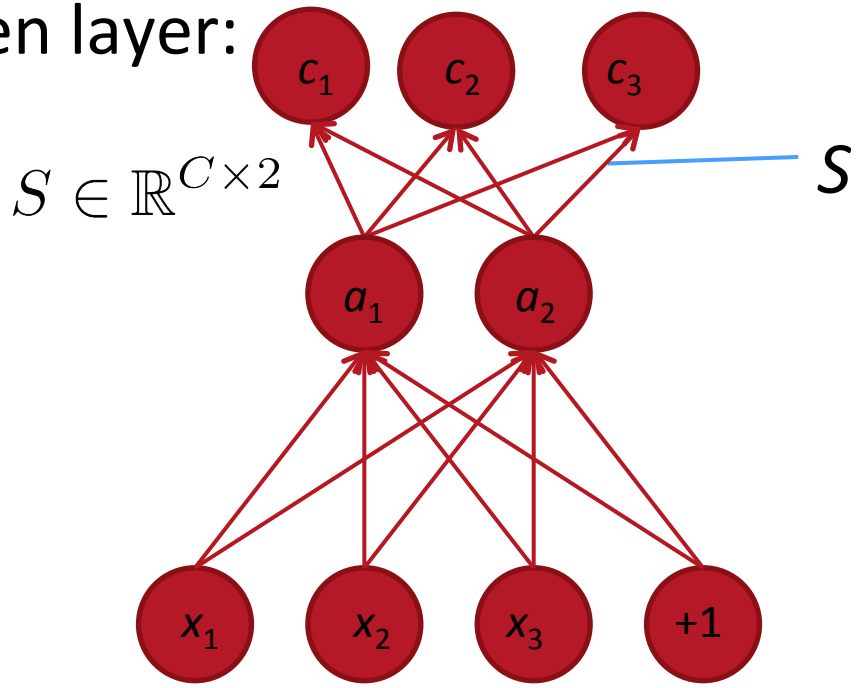
Training with Backpropagation: softmax

What is the major benefit of learned word vectors?

Ability to also propagate labeled information into them, via softmax/maxent and hidden layer:

$$P(c \mid d, \lambda) = \frac{e^{\lambda^\top f(c,d)}}{\sum_{c'} e^{\lambda^\top f(c',d)}}$$

$$p(c|x) = \frac{\exp(S_c.a)}{\sum_{c'} \exp(S_{c'}.a)}$$



For small supervised data sets, unsupervised pre-training helps a lot

	POS WSJ (acc.)	NER CoNLL (F1)
State-of-the-art*	97.24	89.31
Supervised NN	**96.37**	**81.47**
Unsupervised pre-training followed by supervised NN**	**97.20**	**88.87**
+ hand-crafted features***	97.29	89.59

* Representative systems: POS: (Toutanova et al. 2003), NER: (Ando & Zhang 2005)

** 130,000-word embedding trained on Wikipedia and Reuters with 11 word window, 100 unit hidden layer – **for 7 weeks!** – then supervised task training

***Features are character suffixes for POS and a gazetteer for NER

Supervised refinement of the unsupervised word representation helps

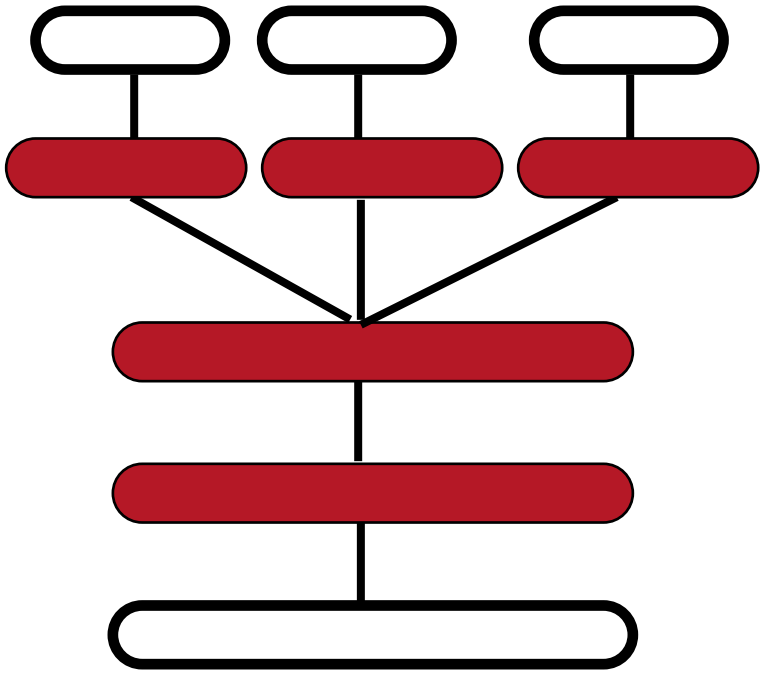
	POS WSJ (acc.)	NER CoNLL (F1)
Supervised NN	96.37	81.47
NN with Brown clusters	96.92	87.15
Fixed embeddings*	**97.10**	**88.87**
C&W 2011**	**97.29**	**89.59**

* Same architecture as C&W 2011, but word embeddings are kept constant during the supervised training phase

** C&W is unsupervised pre-train + supervised NN + features model of last slide

Multi-Task Learning

- Generalizing better to new tasks is crucial to approach AI
- Deep architectures learn good intermediate representations that can be shared across tasks
- Good representations make sense for many tasks



Part 1.5: The Basics

Backpropagation Training

Back-Prop

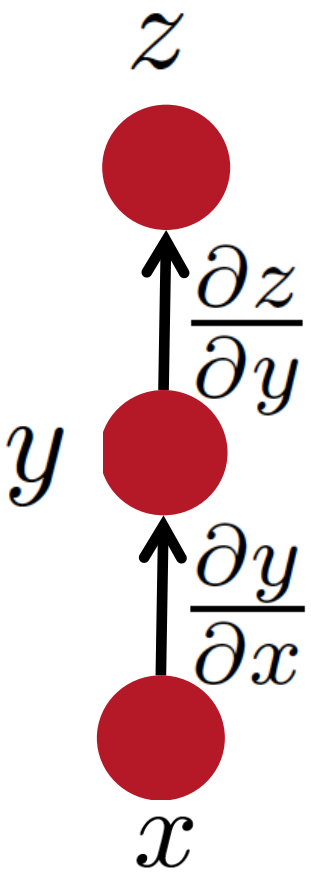
- Compute gradient of example-wise loss wrt parameters

- Simply applying the derivative chain rule wisely

$$z = f(y) \quad y = g(x) \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

- If computing the loss(example, parameters) is $O(n)$ computation, then so is computing the gradient

Simple Chain Rule



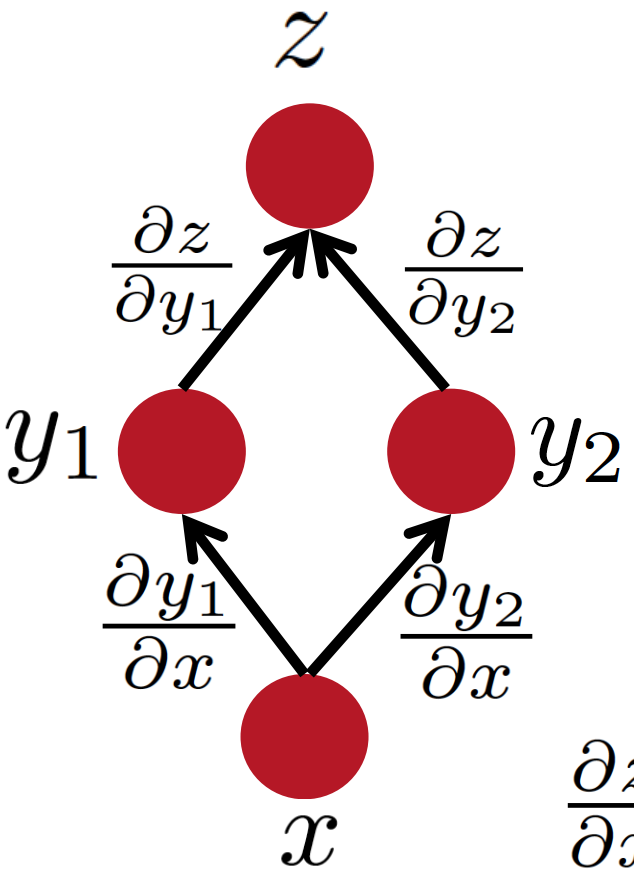
$$\Delta z = \frac{\partial z}{\partial y} \Delta y$$

$$\Delta y = \frac{\partial y}{\partial x} \Delta x$$

$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x$$

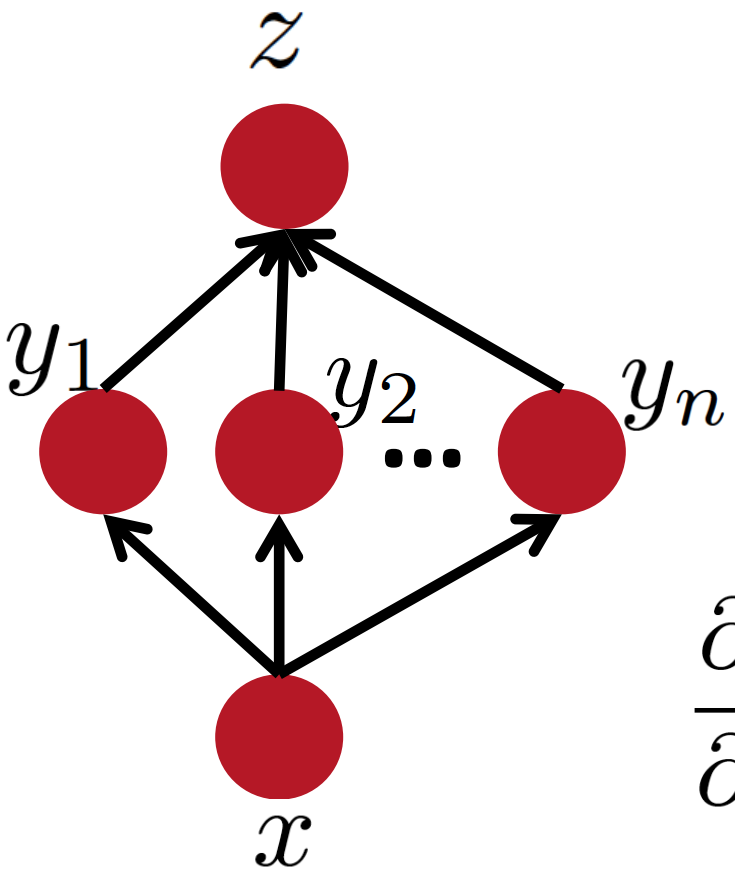
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Multiple Paths Chain Rule



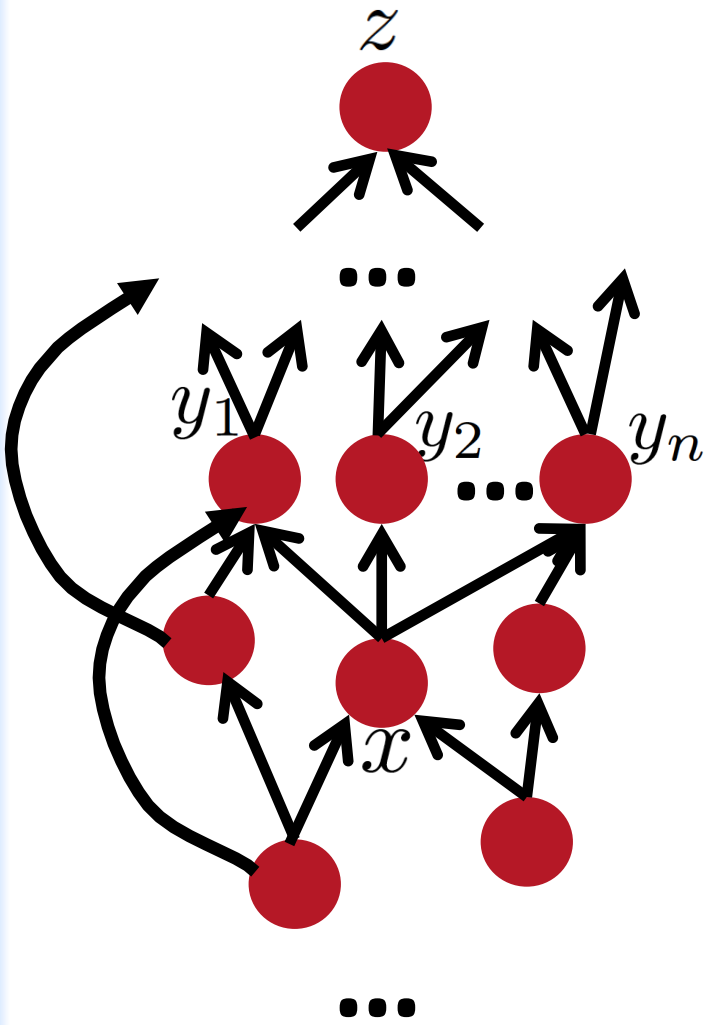
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Multiple Paths Chain Rule - General



$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Chain Rule in Flow Graph

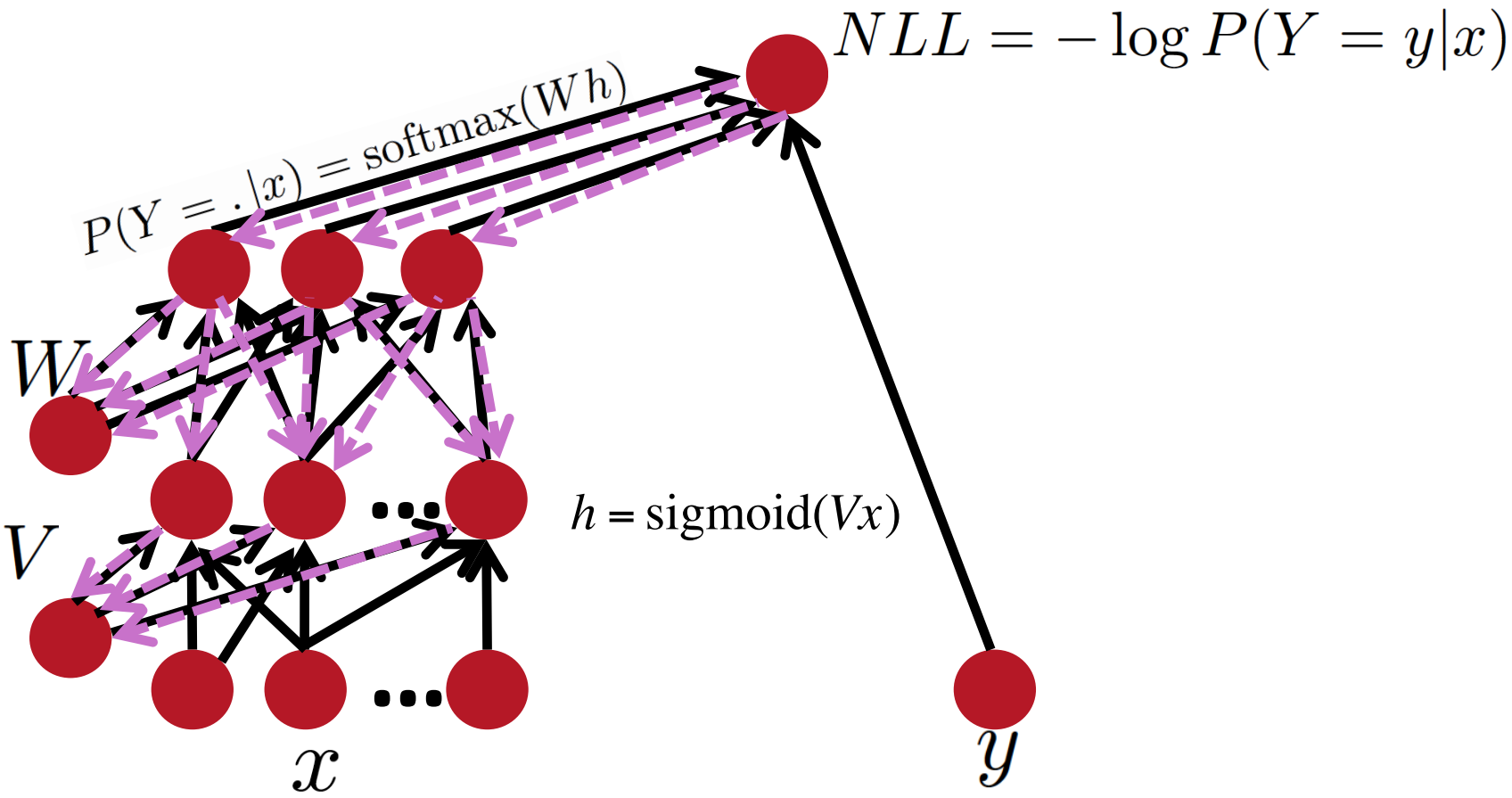


Flow graph: any directed acyclic graph
node = computation result
arc = computation dependency

$\{y_1, y_2, \ldots y_n\}$ = successors of x

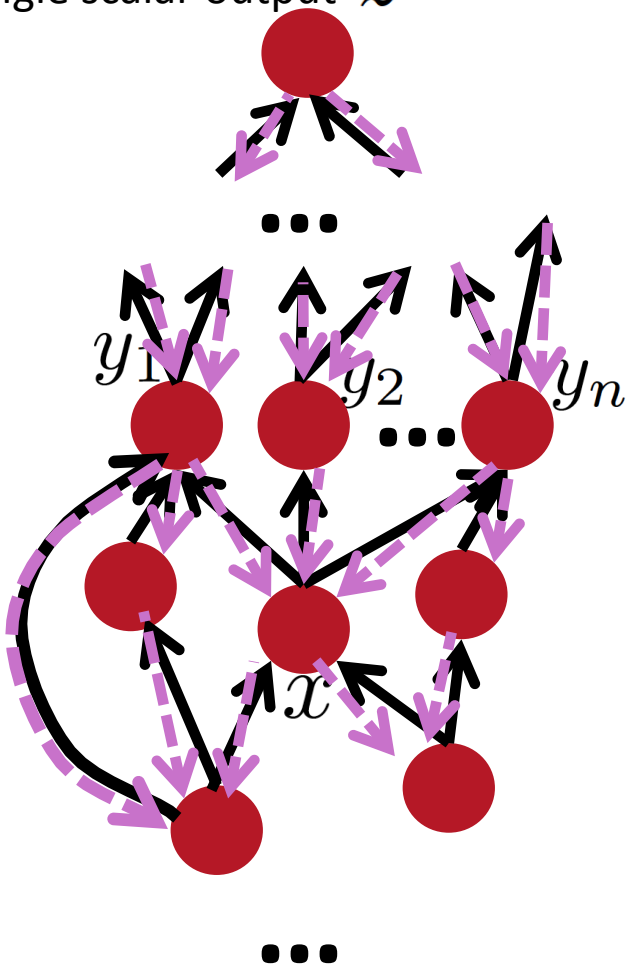
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Back-Prop in Multi-Layer Net



Back-Prop in General Flow Graph

Single scalar output z

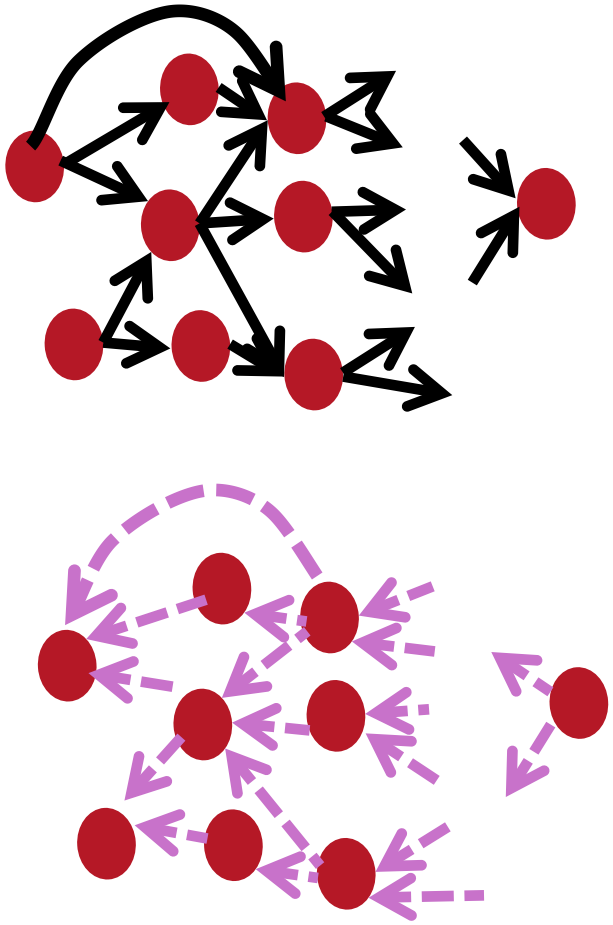


1. Fprop: visit nodes in topo-sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
     Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Automatic Differentiation



- The gradient computation can be automatically inferred from the symbolic expression of the fprop.
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output.
- Easy and fast prototyping
  - See Theano (Python)

Sharing statistical strength

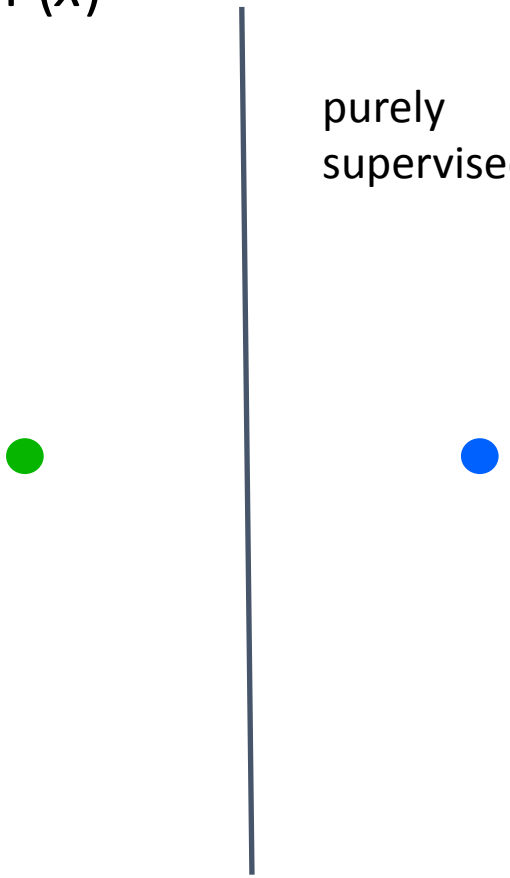
Sharing Statistical Strength

- Besides very fast prediction, the main advantage of deep learning is statistical

- Potential to learn from less labeled examples because of sharing of statistical strength:
  - Unsupervised pre-training & Multi-task learning
  - Semi-supervised learning →

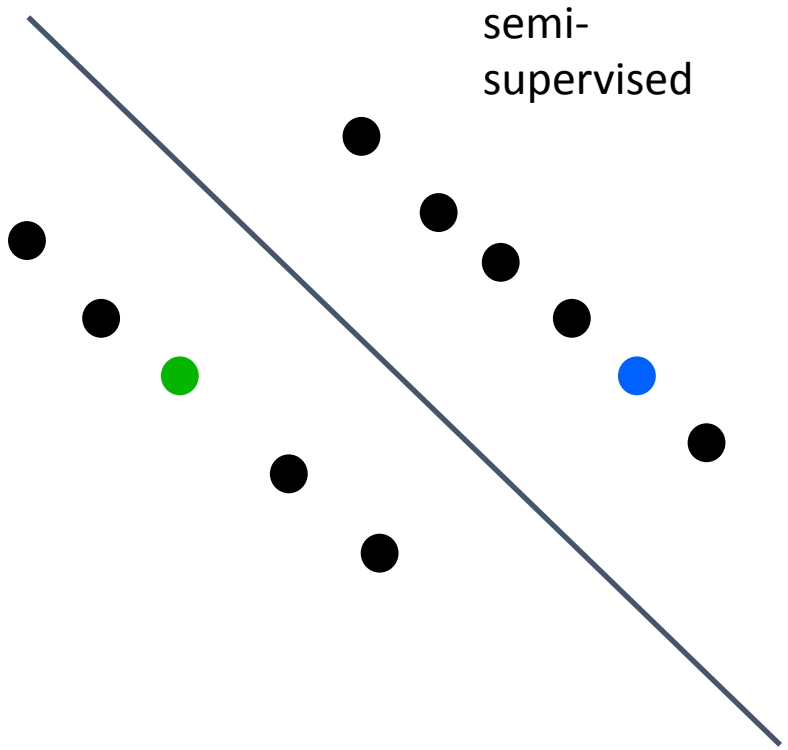
Semi-Supervised Learning

- Hypothesis: P(*c*|*x*) can be more accurately computed using shared structure with P(*x*)



Semi-Supervised Learning

- Hypothesis: P(*c*|*x*) can be more accurately computed using shared structure with P(*x*)



Advantages of Deep Learning

#1 Learning representations

Handcrafting features is time-consuming

The features are often both over-specified and incomplete

The work has to be done again for each task/domain/…

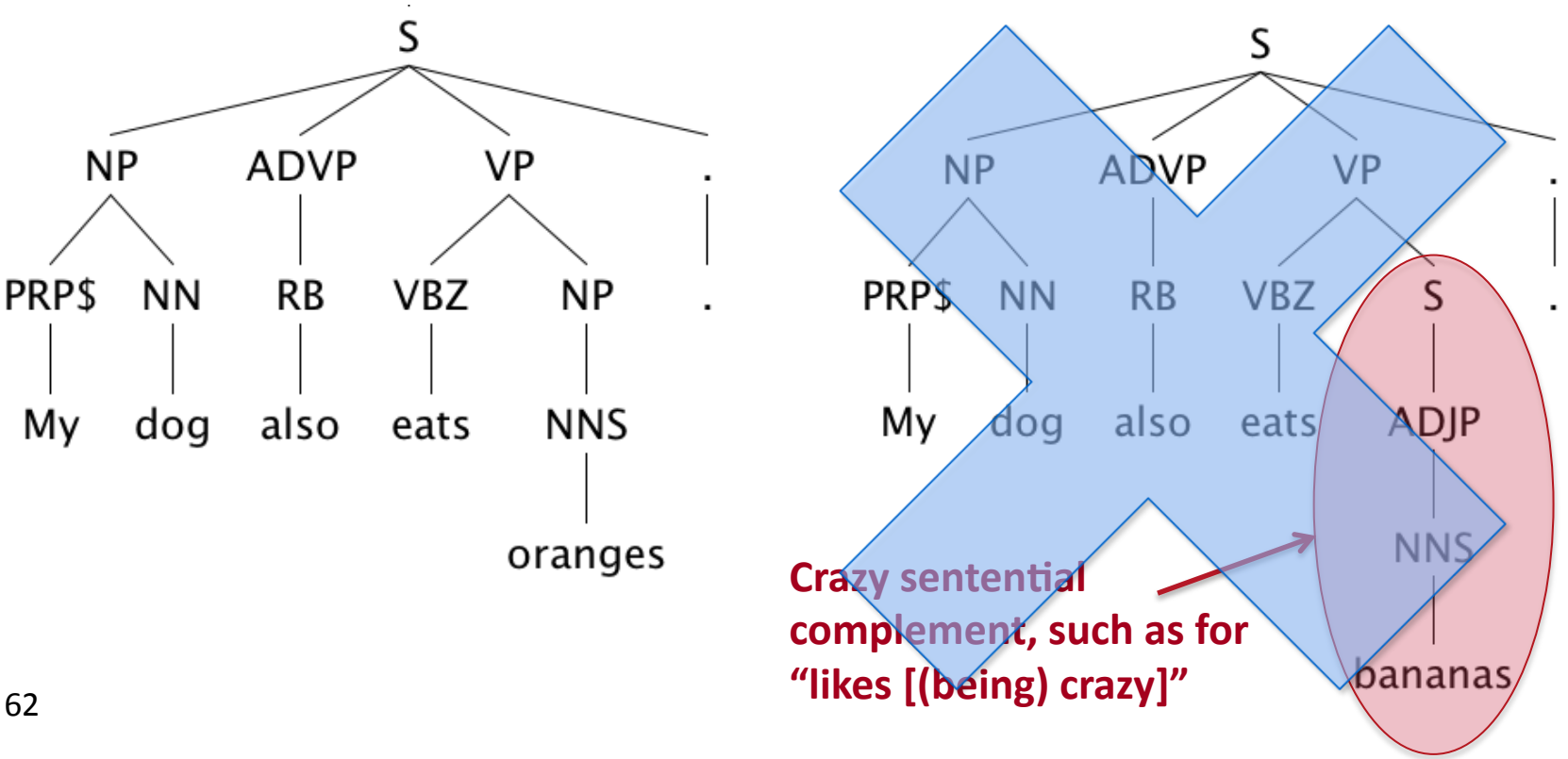
Humans develop representations for learning and reasoning

Our computers should do the same

Deep learning provides a way of doing this
Machine Learning

#2 The need for distributed representations

Current NLP systems are incredibly fragile because of their atomic symbol representations



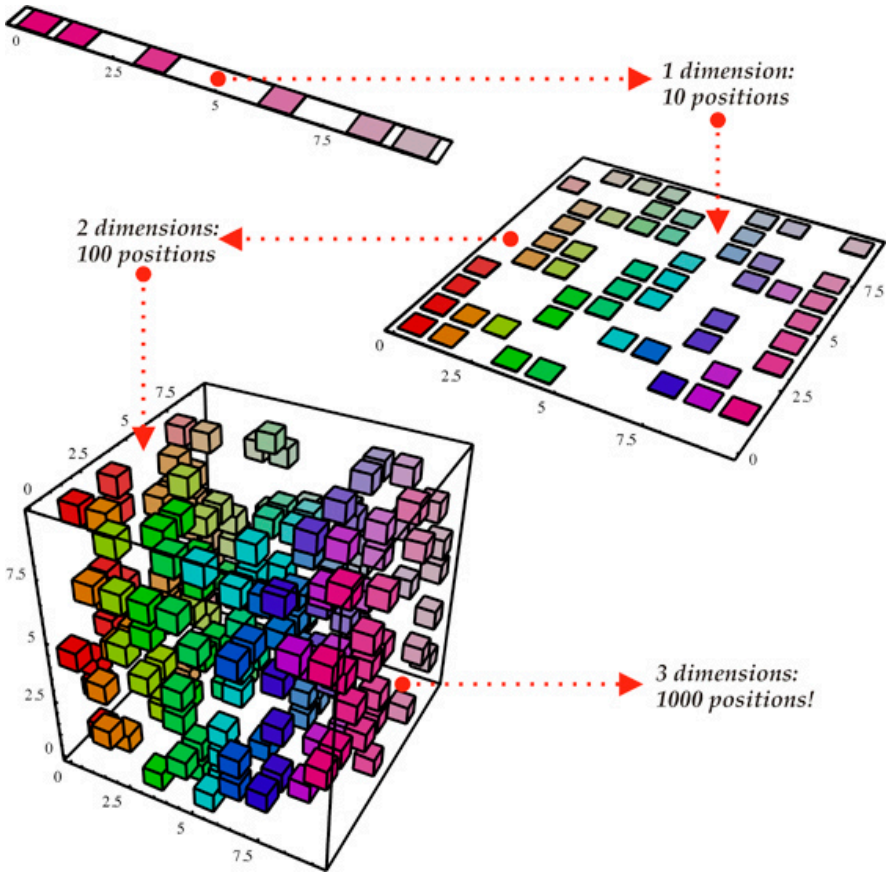
Distributed representations deal with the curse of dimensionality

Generalizing locally (e.g., nearest neighbors) requires representative examples for all relevant variations!

Classic solutions:

- Manual feature design
- Assuming a smooth target function (e.g., linear models)
- Kernel methods (linear in terms of kernel based on data points)

Neural networks parameterize and learn a "similarity" kernel



#3 Unsupervised feature learning

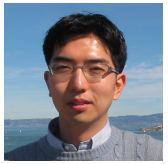
Today, most practical, good NLP& ML methods require labeled training data (i.e., supervised learning)

- But almost all data is unlabeled

Most information must be acquired **unsupervised**

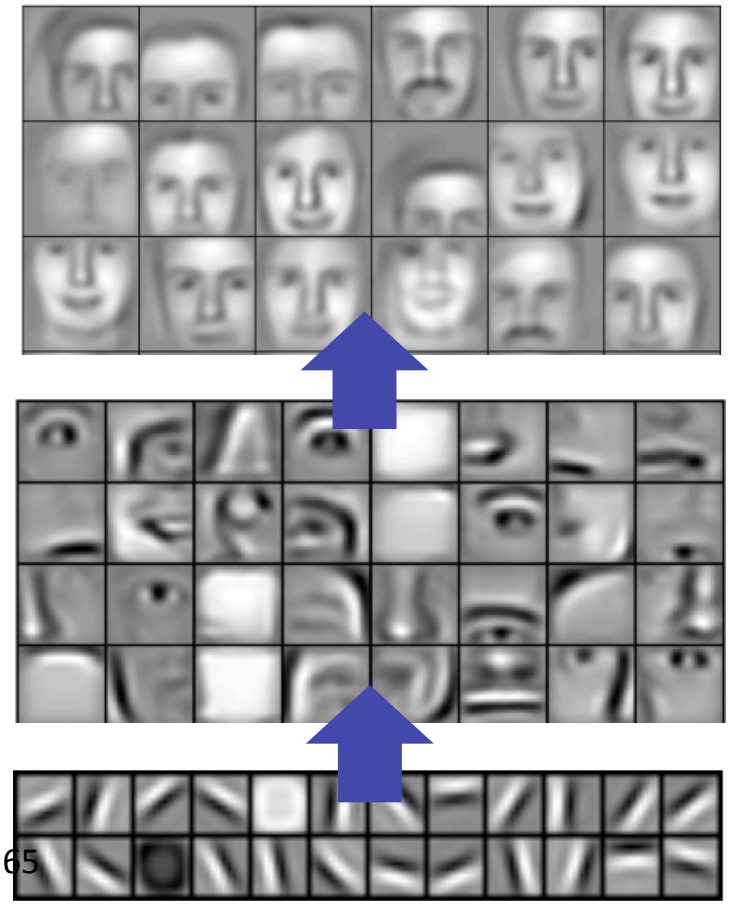
- Fortunately, a good model of observed data can really help you learn classification decisions

#4 Learning multiple levels of representation



[Lee et al. ICML 2009; Lee et al. NIPS 2009]

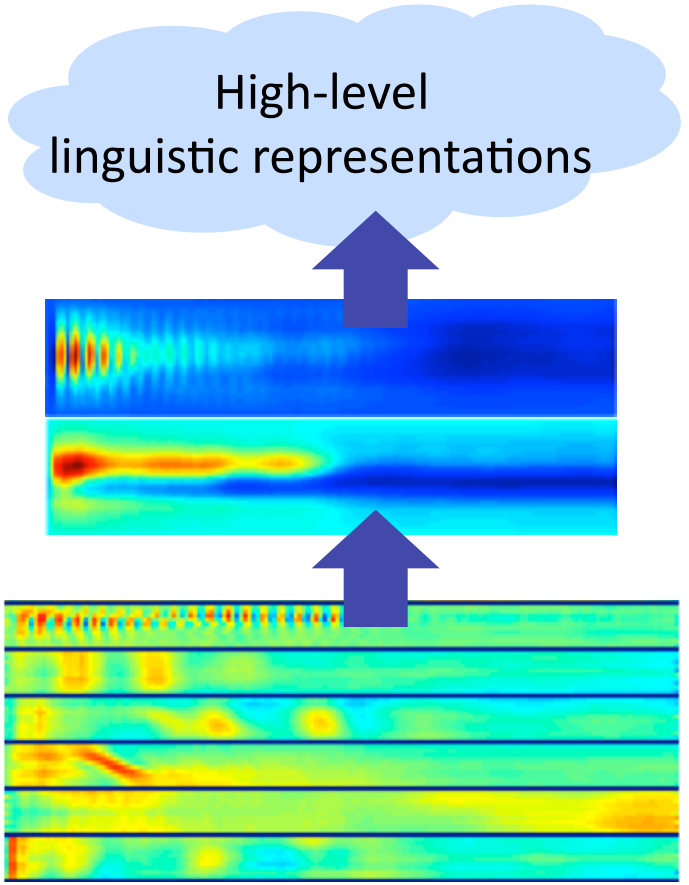
Successive model layers learn deeper intermediate representations



Layer 3

Layer 2

Layer 1

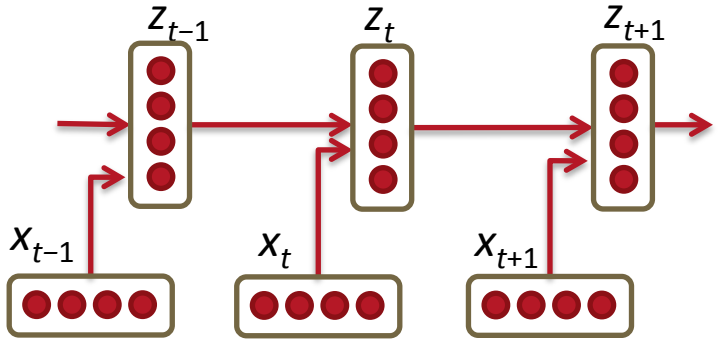


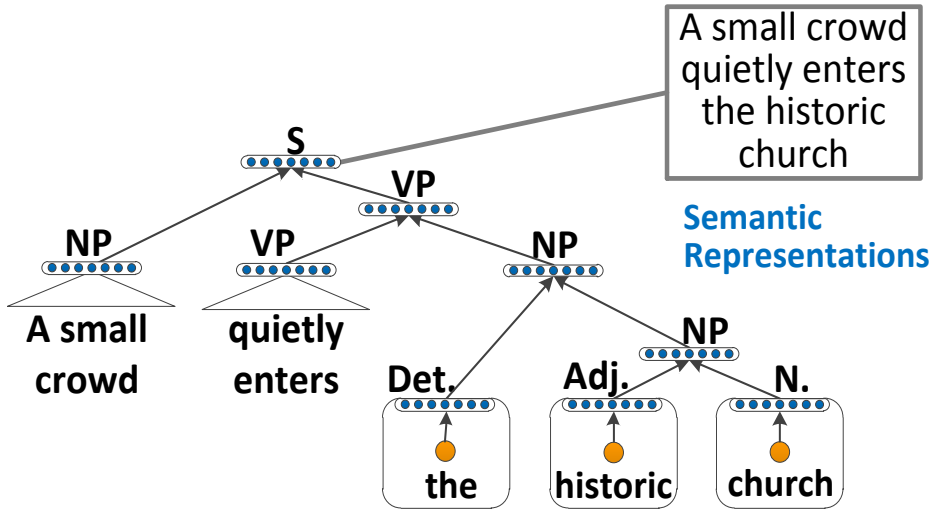
#4 Handling the recursivity of Language

Human sentences are composed from words and phrases

We need compositionality in our ML models

Recursion: the same operator (same parameters) is applied repeatedly on different components





#5 Why now?

Despite prior investigation and understanding of many of the algorithmic techniques …

Before 2006 training deep architectures was **unsuccessful** ☹

What has changed?

- New methods for unsupervised pre-training have been developed (Restricted Boltzmann Machines = RBMs, autoencoders, contrastive estimation, etc.)
- More efficient parameter estimation methods
- Better understanding of model regularization
- **More data** and **more computational power**

Deep Learning models have already achieved impressive results for HLT

Neural Language Model

[Mikolov et al. Interspeech 2011]

Model \ WSJ ASR task	Eval WER
KN5 Baseline	**17.2**
Discriminative LM	**16.9**
Recurrent NN combination	**14.4**

MSR MAVIS Speech System

[Dahl et al. 2012; Seide et al. 2011; following Mohamed et al. 2011]

"The algorithms represent the first time a company has released a deep-neural-networks (DNN)-based speech-recognition algorithm in a commercial product."

Acoustic model & training	Recog \ WER	RT03S FSH	Hub5 SWB
GMM 40-mix, BMMI, SWB 309h	1-pass –adapt	**27.4**	**23.6**
DBN-DNN 7 layer x 2048, SWB 309h	1-pass –adapt	**18.5** (–33%)	**16.1** (–32%)
GMM 72-mix, BMMI, FSH 2000h	*k*-pass +adapt	**18.6**	**17.1**

Deep Learn Models Have Interesting Performance Characteristics

Deep learning models can now be very fast in some circumstances

- SENNA [Collobert et al. 2011] can do POS or NER faster than other SOTA taggers (16x to 122x), using 25x less memory
- WSJ POS 97.29% acc; CoNLL NER 89.59% F1; CoNLL Chunking 94.32% F1

Changes in computing technology favor deep learning

- In NLP, speed has traditionally come from exploiting sparsity
- But with modern machines, branches and widely spaced memory accesses are costly
- Uniform parallel operations on dense vectors are faster

These trends are even stronger with multi-core CPUs and GPUs