Deep Learning for NLP

(without Magic)

Richard Socher and Christopher Manning

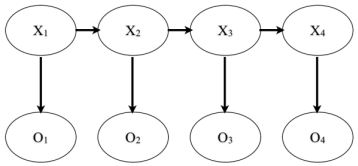
Stanford University

NAACL 2013, Atlanta

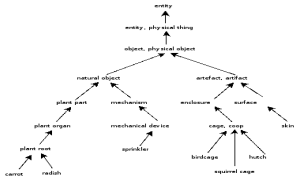
http://nlp.stanford.edu/courses/NAACL2013/

*with a big thank you to Yoshua Bengio, with whom we participated in the previous ACL 2012 version of this tutorial

Deep Learning

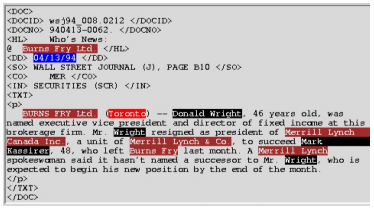


NER

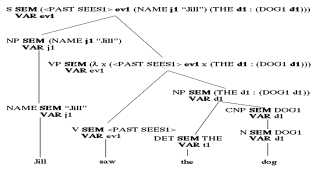


WordNet

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

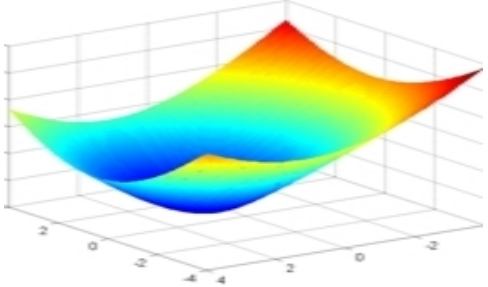


SRL



Parser

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



Representation learning attempts to automatically learn good features or representations

Deep learning algorithms attempt to learn multiple levels of representation of increasing complexity/abstraction

A Deep Architecture

Mainly, work has explored deep belief networks (DBNs), Markov Random Fields with multiple layers, and various types of multiple-layer neural networks

Output layer

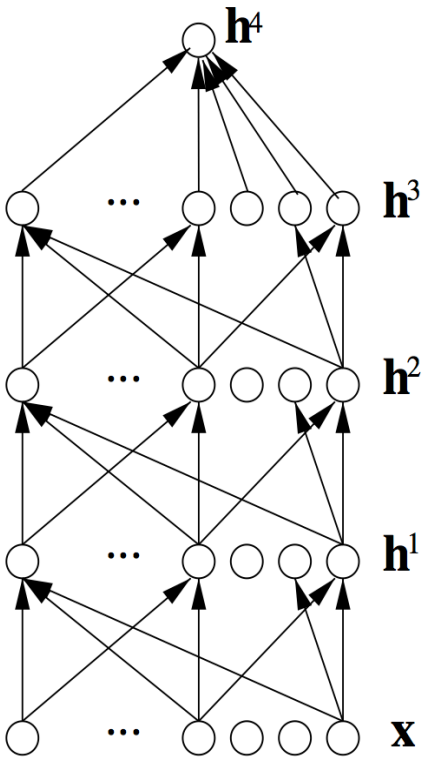
Here predicting a supervised target

Hidden layers

These learn more abstract representations as you head up

Input layer

Raw sensory inputs (roughly)



Part 1.1: The Basics

Five Reasons to Explore 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/…

We must move beyond handcrafted features and simple ML

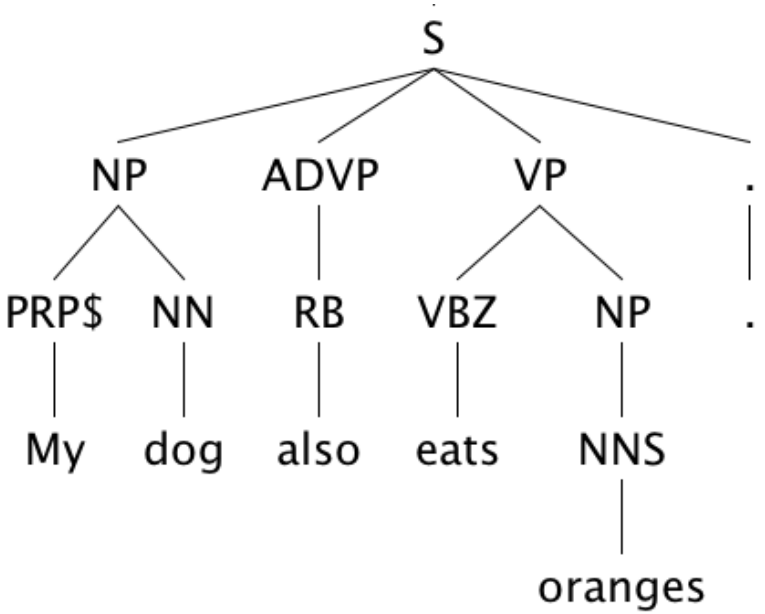
Humans develop representations for learning and reasoning

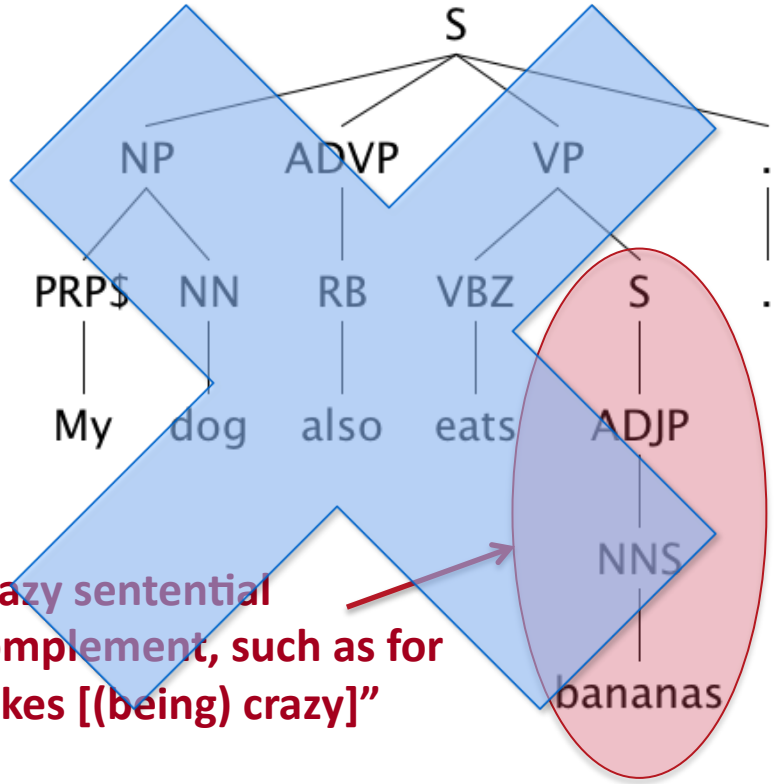
Our computers should do the same

Deep learning provides a way of doing this

#2 The need for distributed representations

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





#2 The need for distributional & distributed representations

Learned word representations help enormously in NLP

They provide a powerful similarity model for words

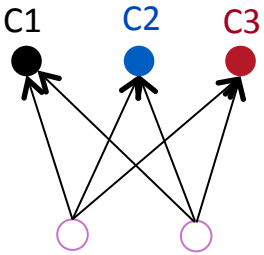
Distributional similarity based word clusters greatly help most applications

+1.4% F1 Dependency Parsing 15.2% error reduction (Koo & Collins 2008, Brown clustering)

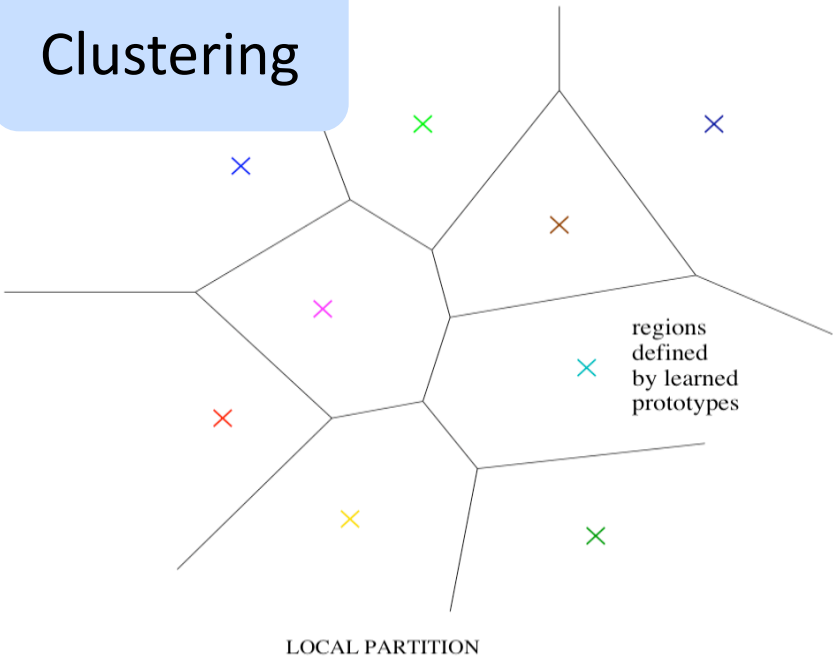
+3.4% F1 Named Entity Recognition 23.7% error reduction (Stanford NER, exchange clustering)

Distributed representations can do even better by representing more dimensions of similarity

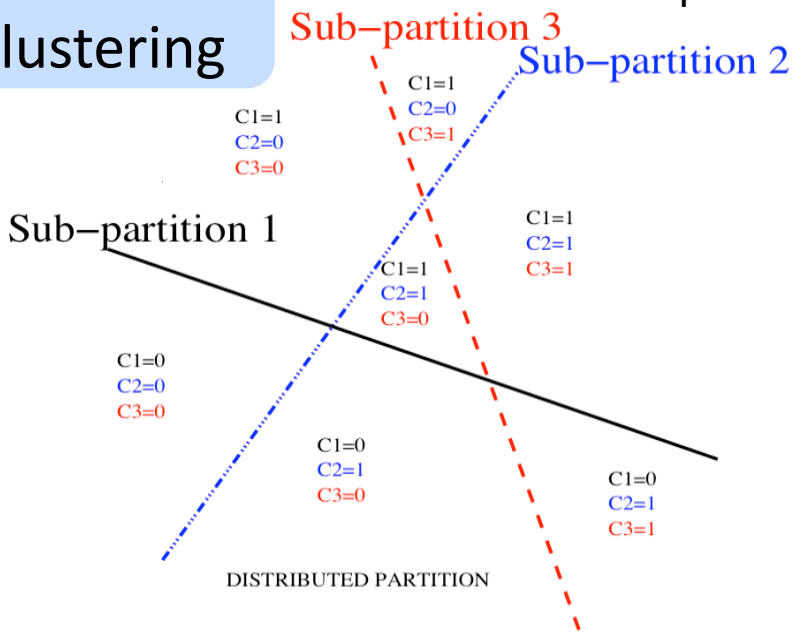
#2 The need for distributed representations



Clustering



Multi-Clustering



Learning features that are not mutually exclusive can be exponentially more efficient than nearest-neighbor-like or clustering-like models

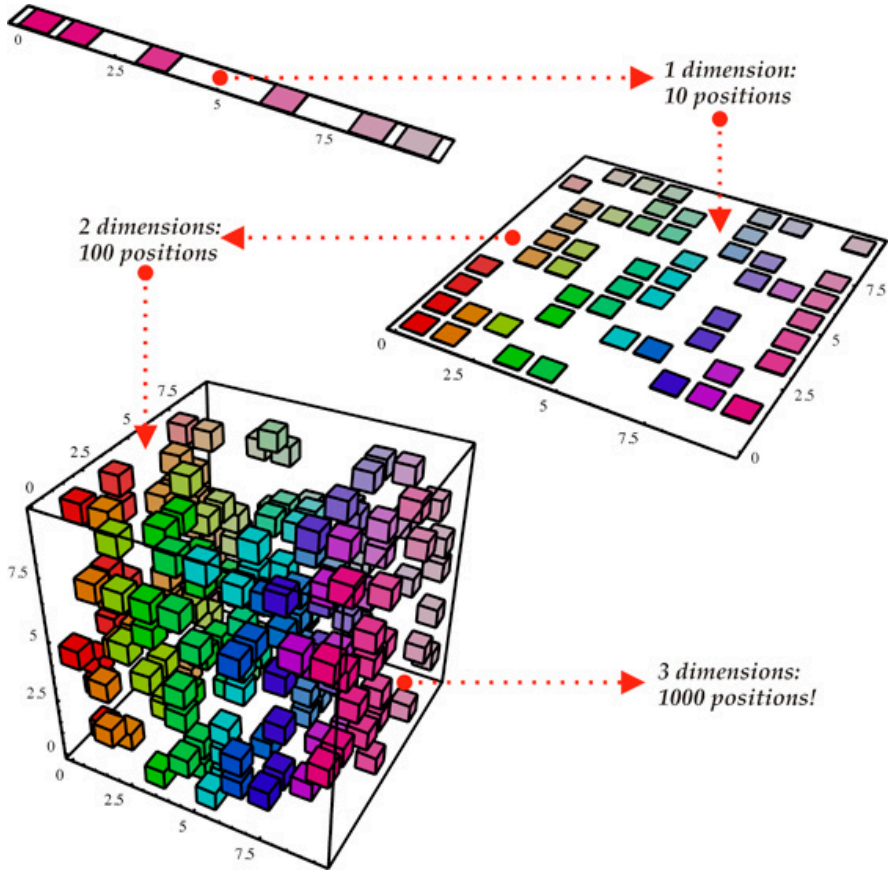
Distributed representations deal with the curse of dimensionality

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 and weight learning

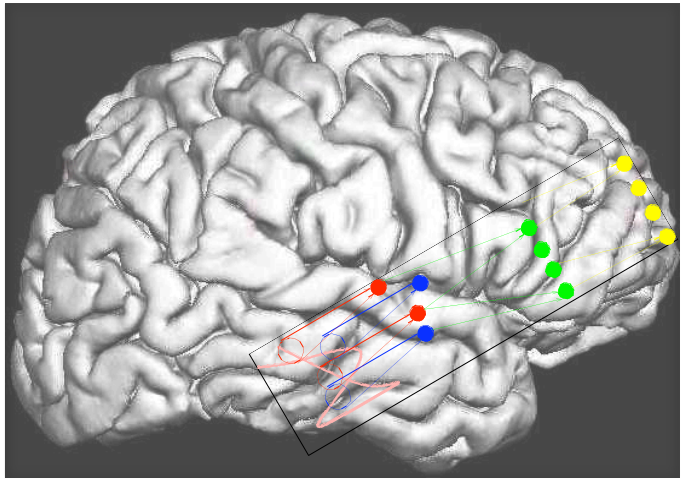
Today, most practical, good NLP& ML methods require labeled training data (i.e., supervised learning)

- But almost all data is unlabeled

Most information must be acquired **unsupervised**

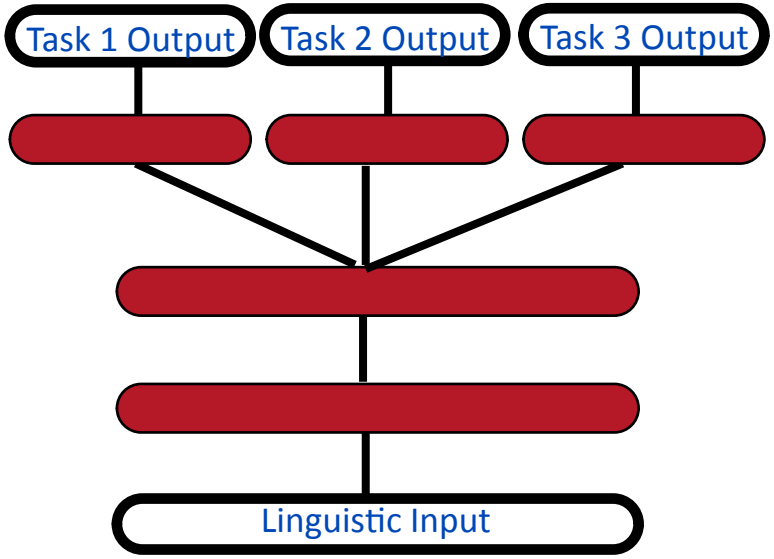
- Fortunately, a good model of observed data can really help you learn classification decisions

#4 Learning multiple levels of representation



Biologically inspired learning

- The cortex seems to have a generic learning algorithm
- The brain has a deep architecture

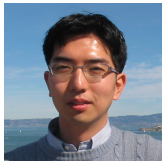


We need good intermediate representations that can be shared across tasks

Multiple levels of latent variables allow combinatorial sharing of statistical strength

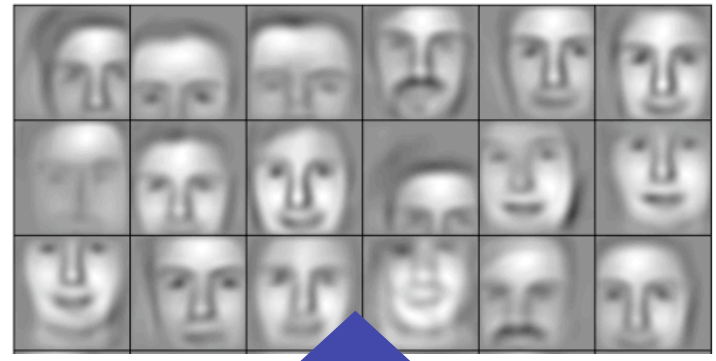
- Insufficient model depth can be exponentially inefficient

#4 Learning multiple levels of representation

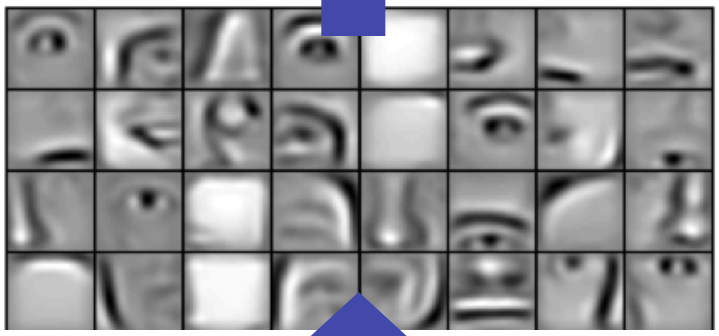


[Lee et al. ICML 2009; Lee et al. NIPS 2009]

Successive model layers learn deeper intermediate representations

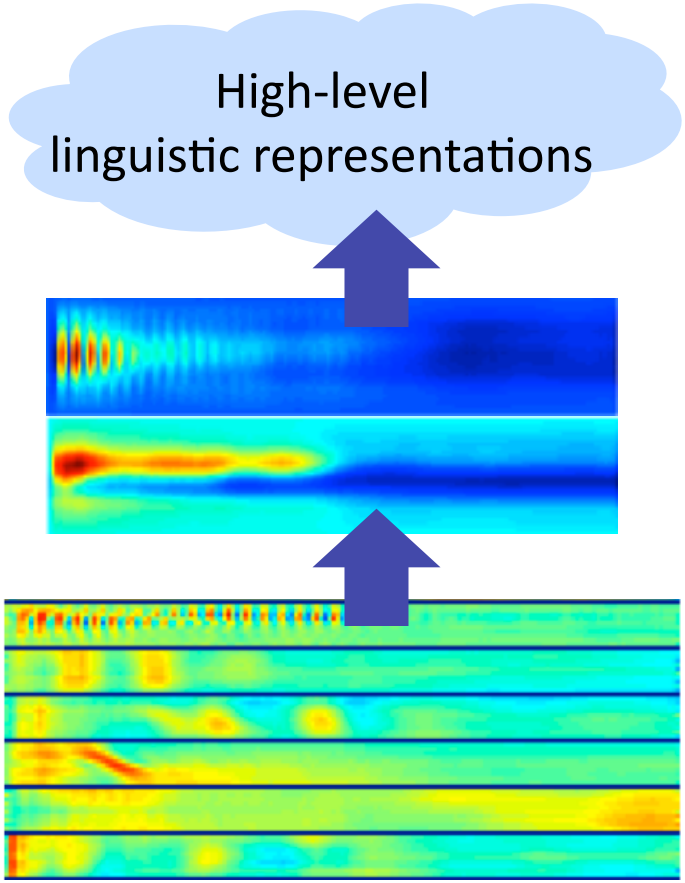


Layer 3



Layer 2

Layer 1

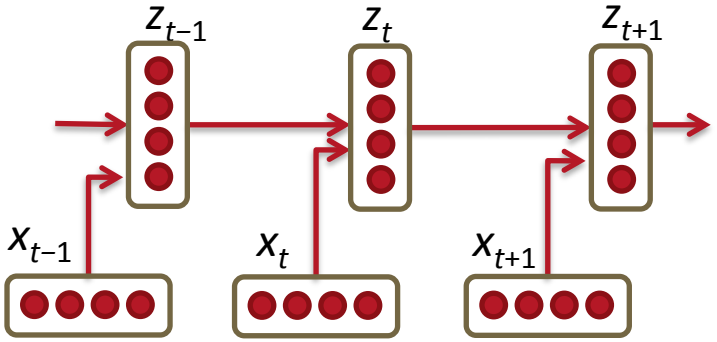


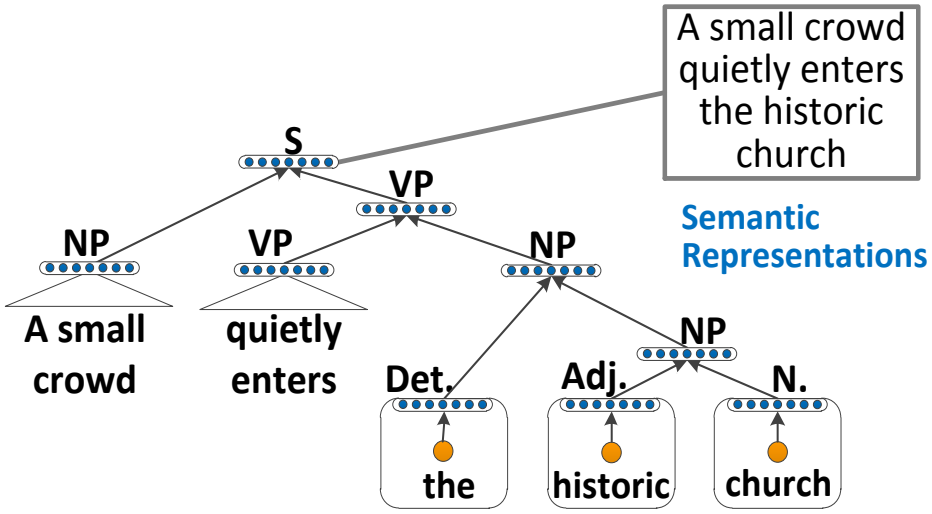
Handling the recursivity of human language

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

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

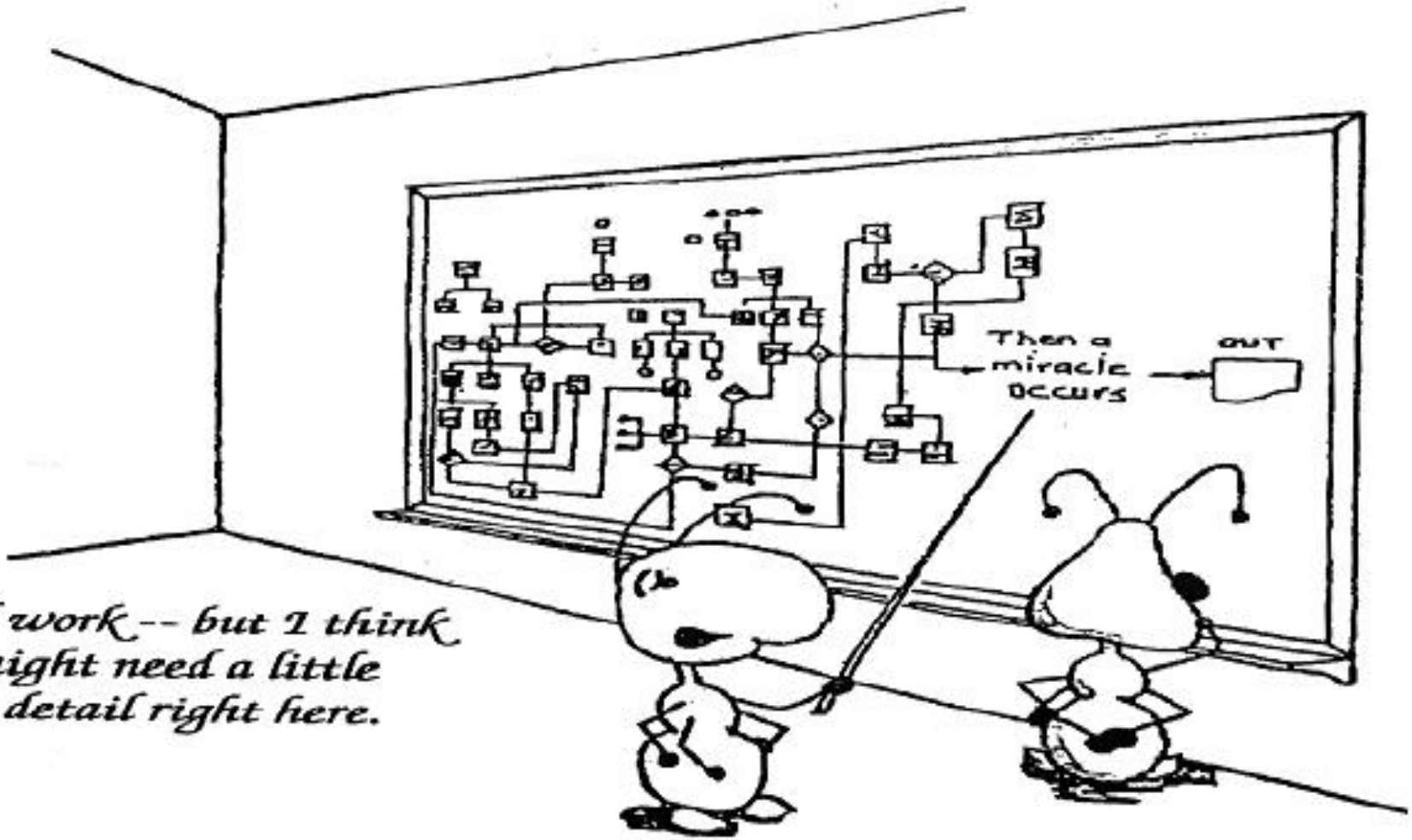
Deep learning models can now be very fast in some circumstances

- 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



Good work -- but I think we might need a little more detail right here.

Outline of the Tutorial

1. The Basics
    1. Motivations
    2. From logistic regression to neural networks
    3. Word representations
    4. Unsupervised word vector learning
    5. Backpropagation Training
    6. Learning word-level classifiers: POS and NER
    7. Sharing statistical strength
2. Recursive Neural Networks
3. Applications, Discussion, and Resources

Outline of the Tutorial

1. The Basics
2. Recursive Neural Networks
    1. Motivation
    2. Recursive Neural Networks for Parsing
    3. Optimization and Backpropagation Through Structure
    4. Compositional Vector Grammars: Parsing
    5. Recursive Autoencoders: Paraphrase Detection
    6. Matrix-Vector RNNs: Relation classification
    7. Recursive Neural Tensor Networks: Sentiment Analysis
3. Applications, Discussion, and Resources

Outline of the Tutorial

1. The Basics
2. Recursive Neural Networks
3. Applications, Discussion, and Resources
    1. Assorted Speech and NLP Applications
    2. Deep Learning: General Strategy and Tricks
    3. Resources (readings, code, …)
    4. Discussion

Part 1.2: The Basics

From logistic regression to neural nets

Demystifying neural networks

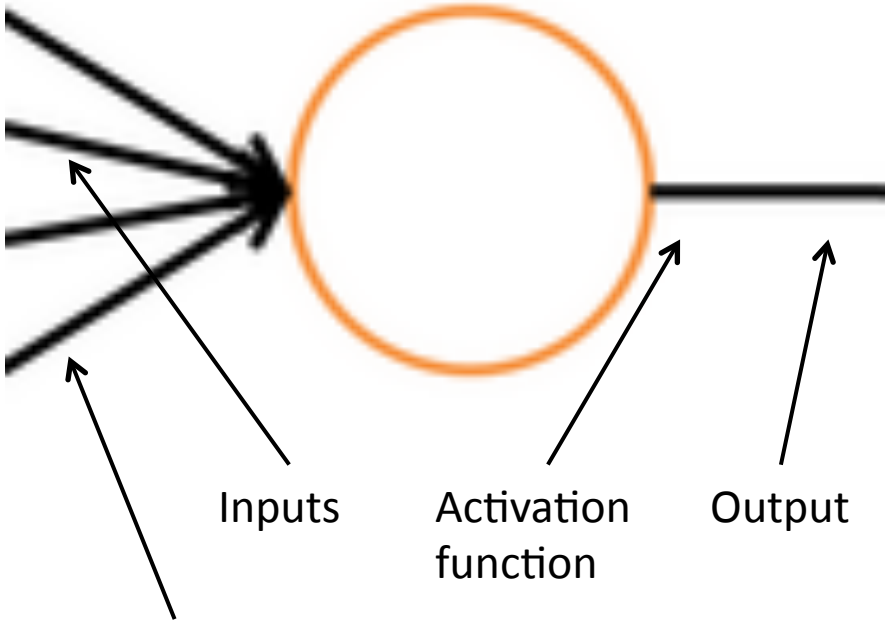
Neural networks come with their own terminological baggage

… just like SVMs

But if you understand how logistic regression or maxent models work

Then **you already understand** the operation of a basic neural network neuron!

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



From Maxent Classifiers to Neural Networks

In NLP, a maxent classifier is normally written as:

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

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

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: $J = \text{softmax}(\lambda \cdot x)$

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

From Maxent Classifiers to Neural Networks

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

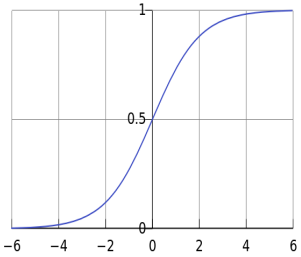
Make two class:

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

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

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

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



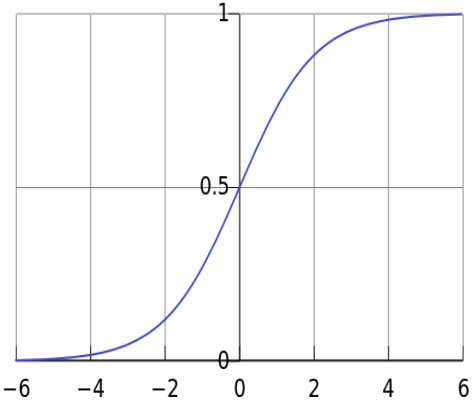
for $f(z) = 1/(1 + \exp(-z))$, the logistic function – a sigmoid non-linearity.

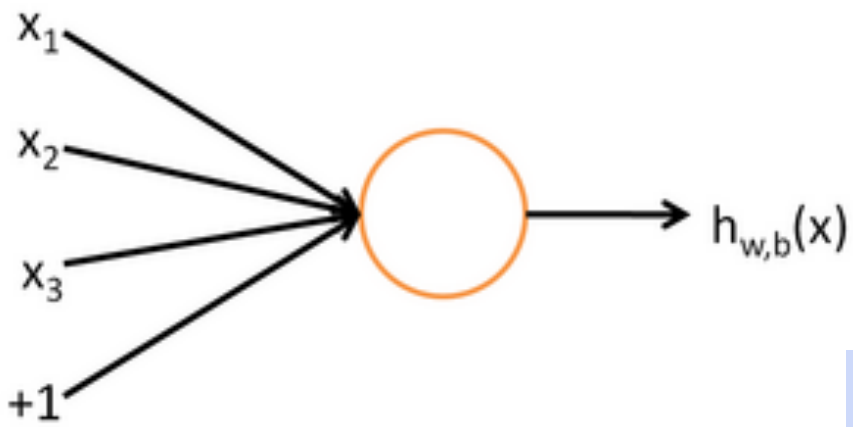
This is exactly what a neuron computes

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

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

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

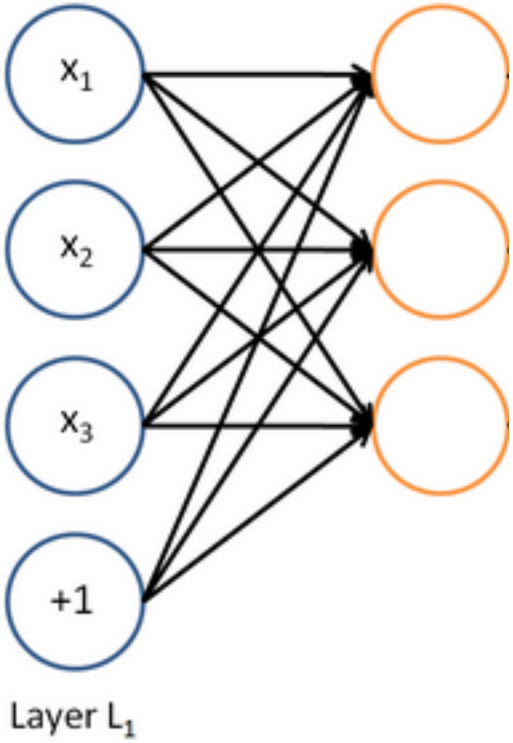




w, *b* are the parameters of this neuron i.e., this logistic regression model

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

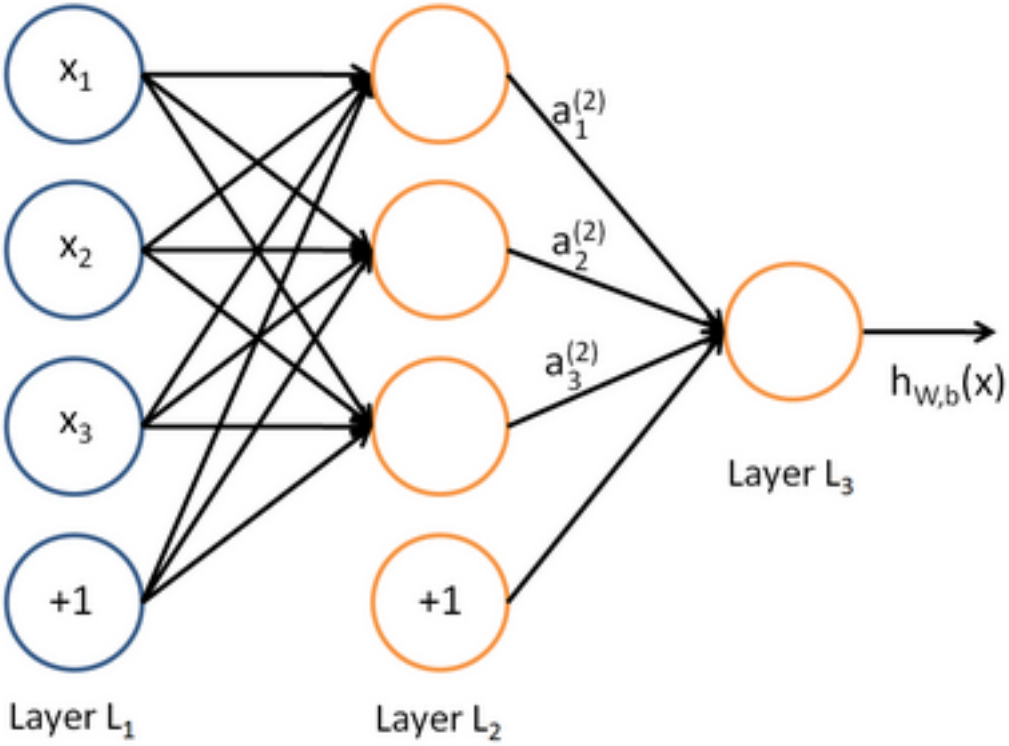
If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs …



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

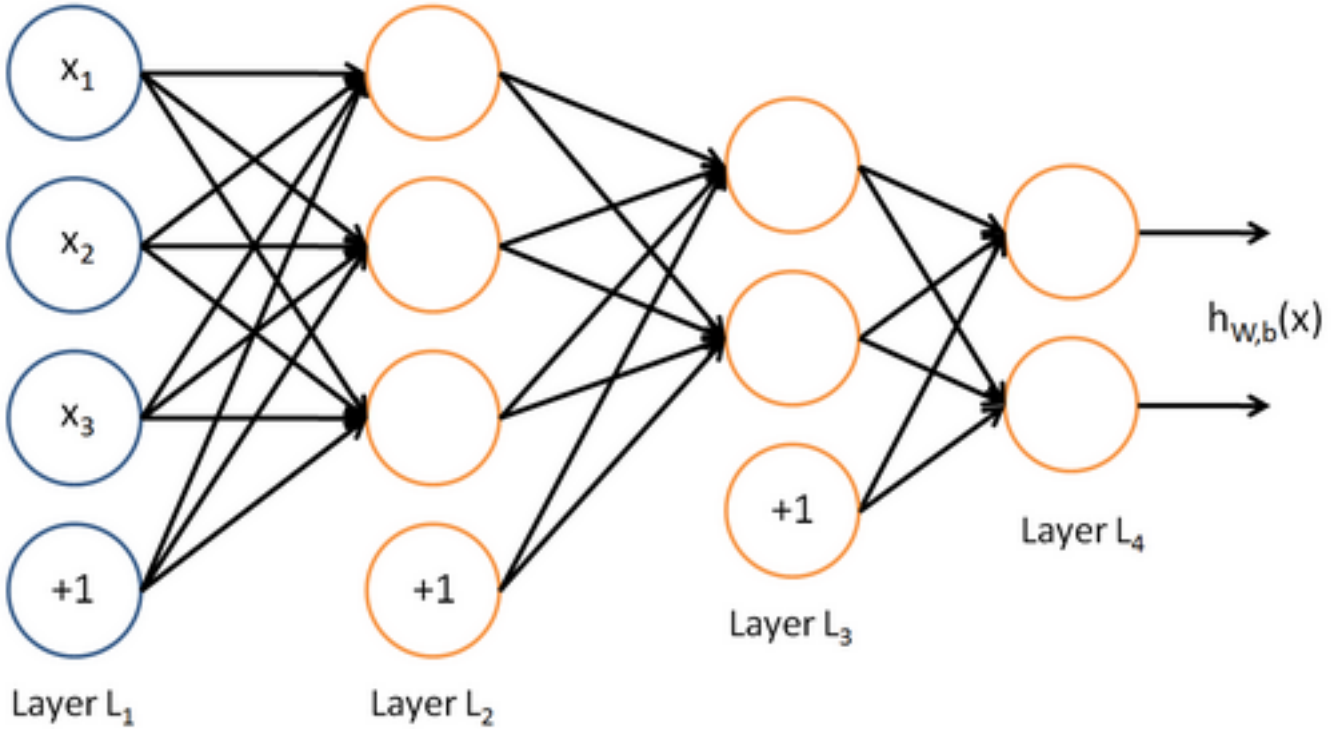
… which we can feed into another logistic regression function



It is the training criterion that will direct what the intermediate hidden variables should be, so as to do a good job at 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.

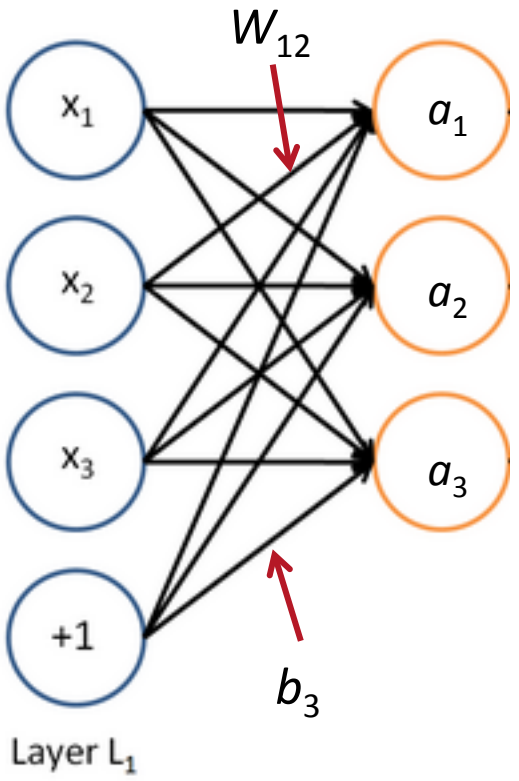
In matrix notation

$$z = Wx + b$$

$$a = f(z)$$

where f is applied element-wise:

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



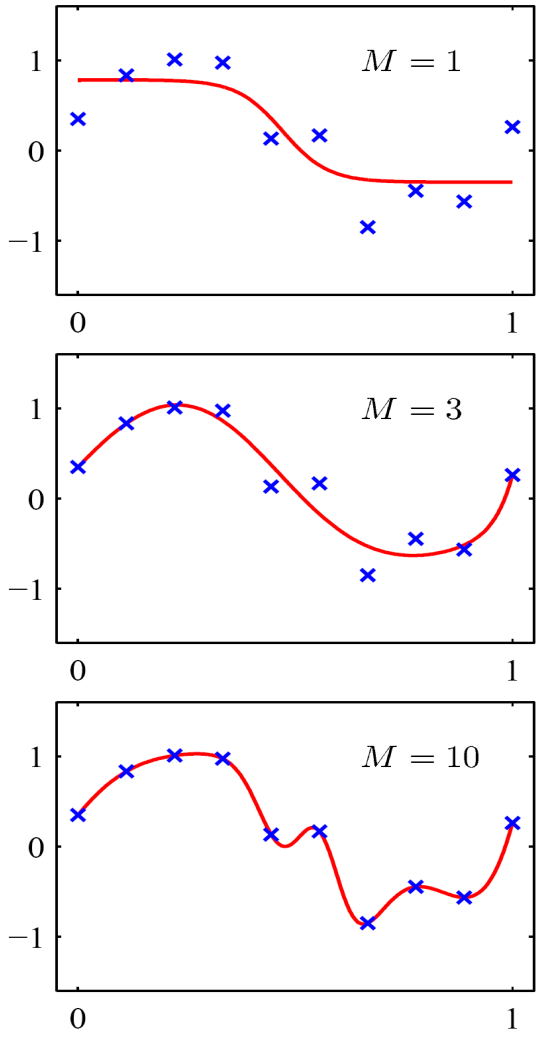
How do we train the weights W?

- For a single supervised layer, we train just like a maxent model – we calculate and use error derivatives (gradients) to improve
  - Online learning: Stochastic gradient descent (SGD)
    - Or improved versions like AdaGrad (Duchi, Hazan, & Singer 2010)
  - Batch learning: Conjugate gradient or L-BFGS

- A multilayer net could be more complex because the internal ("hidden") logistic units make the function non-convex … just as for hidden CRFs [Quattoni et al. 2005, Gunawardana et al. 2005]
  - But we can use the same ideas and techniques
    - Just without guarantees …
  - We "backpropagate" error derivatives through the model

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
  - Probabilistic interpretation unnecessary except in the Boltzmann machine/graphical models
    - People often use other non-linearities, such as **tanh**, as we'll discuss in part 3



Summary
Knowing the meaning of words!

You now understand the basics and the relation to other models

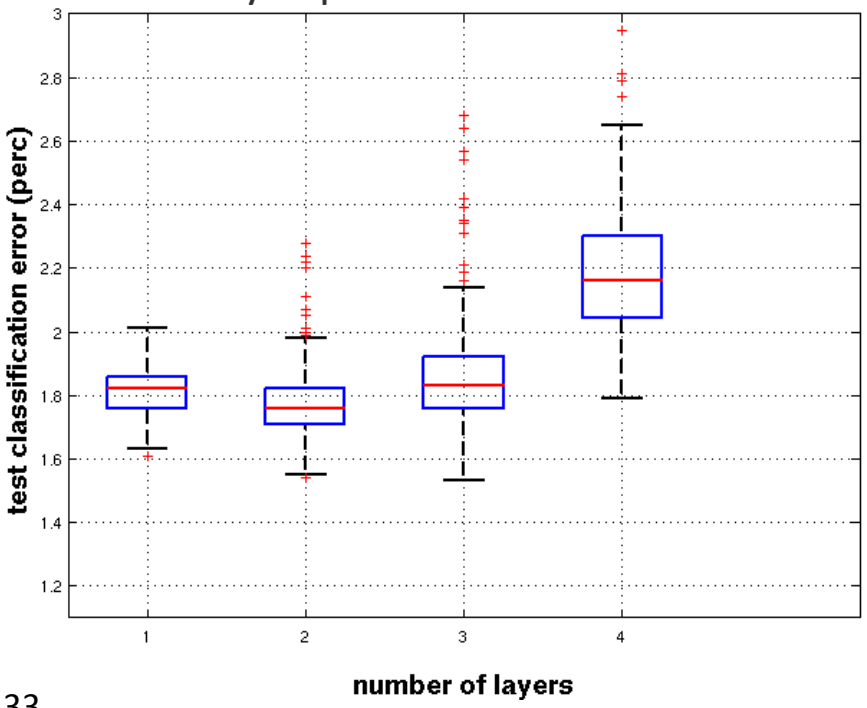
- Neuron = logistic regression or similar function
- Input layer = input training/test vector
- Bias unit = intercept term/always on feature
- Activation = response
- Activation function is a logistic (or similar "sigmoid" nonlinearity)
- Backpropagation = running stochastic gradient descent backward layer-by-layer in a multilayer network
- Weight decay = regularization / Bayesian prior

Effective deep learning became possible through unsupervised pre-training

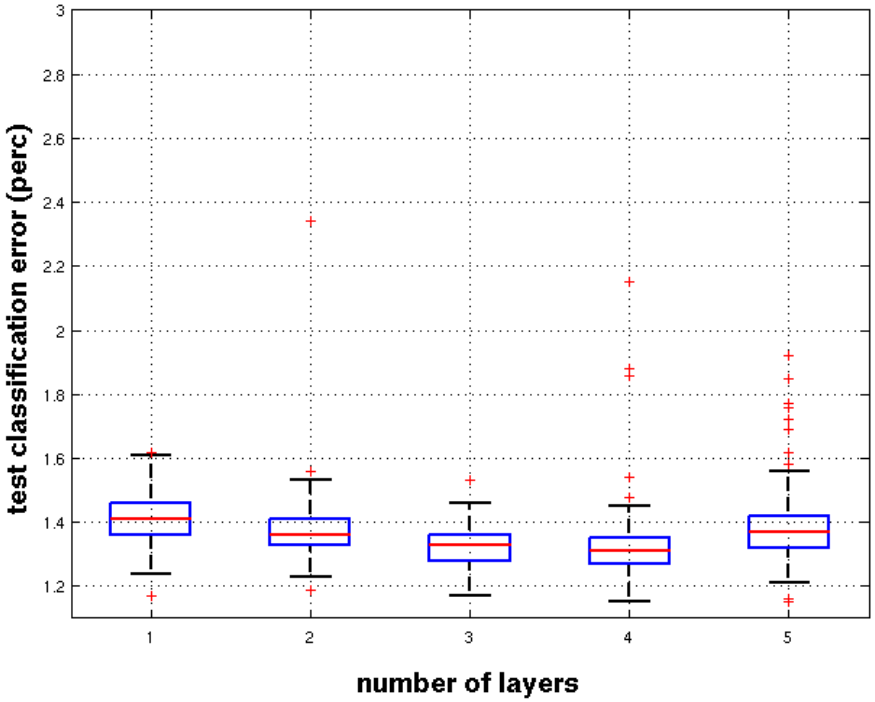
[Erhan et al., JMLR 2010]

(with RBMs and Denoising Auto-Encoders)

Purely supervised neural net



With unsupervised pre-training



0–9 handwritten digit recognition error rate (MNIST data)

Part 1.3: The Basics

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

Class-based (hard) and soft clustering word representations

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

- Brown clustering (Brown et al. 1992)
- Exchange clustering (Martin et al. 1998, Clark 2003)
- Desparsification and great example of unsupervised pre-training

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, Turian et al. 2010)

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 Embeddings

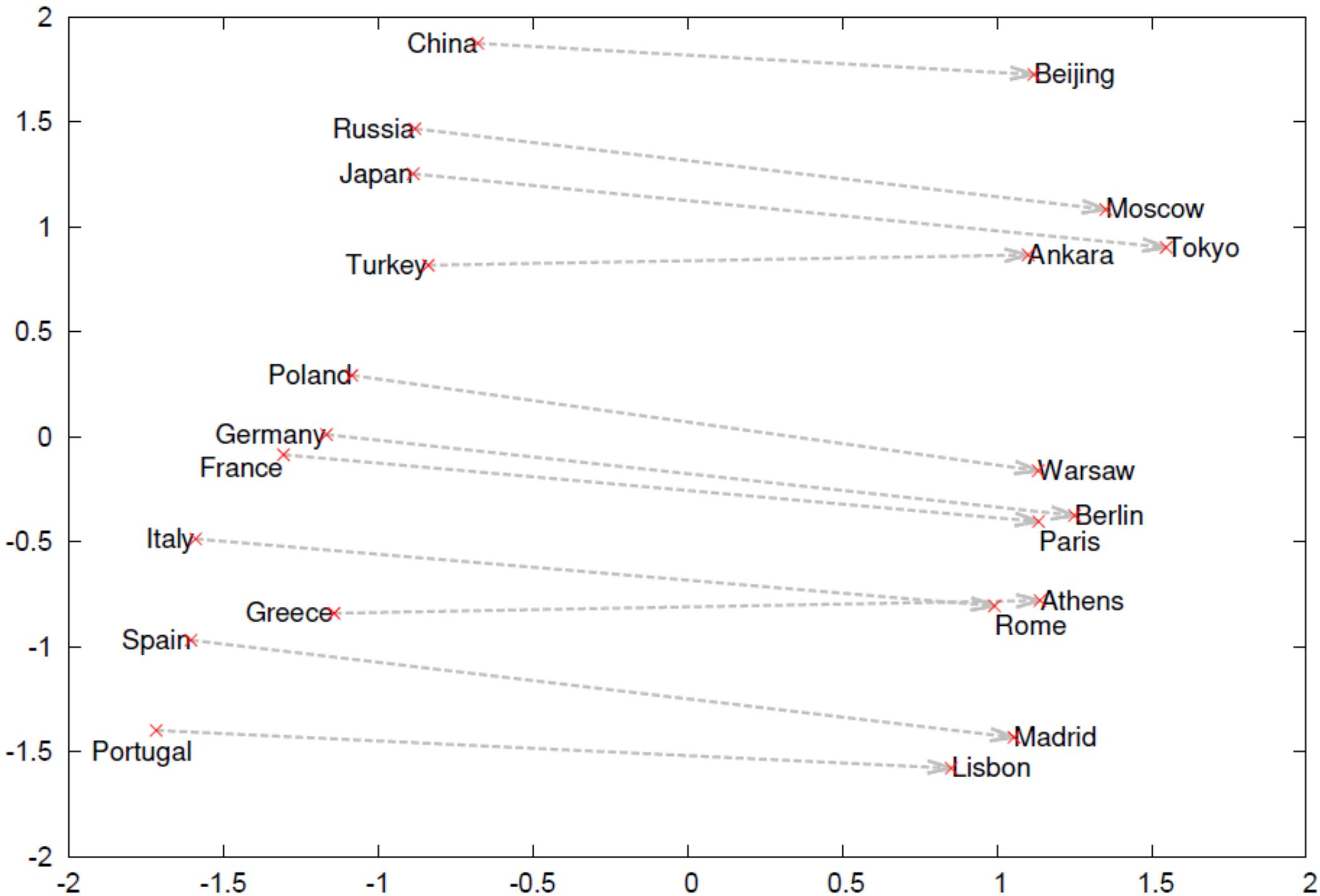
- **Dense** vectors
- Each dimension is a **latent** feature
- Common software package: **word2vec**

$$Italy\colon\ (-7.35, 9.42, 0.88, \dots) \in \mathbb{R}^{100}$$

- **“Magic”**

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

(analogies)


China
Beijing
Russia
Japan
Moscow
Tokyo
Turkey
Ankara
Poland
Germany
France
Warsaw
Berlin
Paris
Italy
Athens
Greece
Rome
Spain
Madrid
Portugal
Lisbon
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2

Mikolov et al. (2013a,b,c)

- Neural embeddings have interesting geometries
- These patterns capture “relational similarities”
- Can be used to solve analogies:

man is to woman as king is to queen

Mikolov et al. (2013a,b,c)

- Neural embeddings have interesting geometries
- These patterns capture “relational similarities”
- Can be used to solve analogies:

a is to a^* as b is to b^*

Mikolov et al. (2013a,b,c)

- Neural embeddings have interesting geometries
- These patterns capture “relational similarities”
- Can be used to solve analogies:

$$a \text{ is to } a^* \text{ as } b \text{ is to } b^*$$

- With simple vector arithmetic:

$$a - a^* = b - b^*$$

Mikolov et al. (2013a,b,c)

$$a - a^* = b - b^*$$

Mikolov et al. (2013a,b,c)

$$b - a + a^* = b^*$$

Mikolov et al. (2013a,b,c)

$$b \qquad a \qquad a^* \qquad b^*$$

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

Mikolov et al. (2013a,b,c)

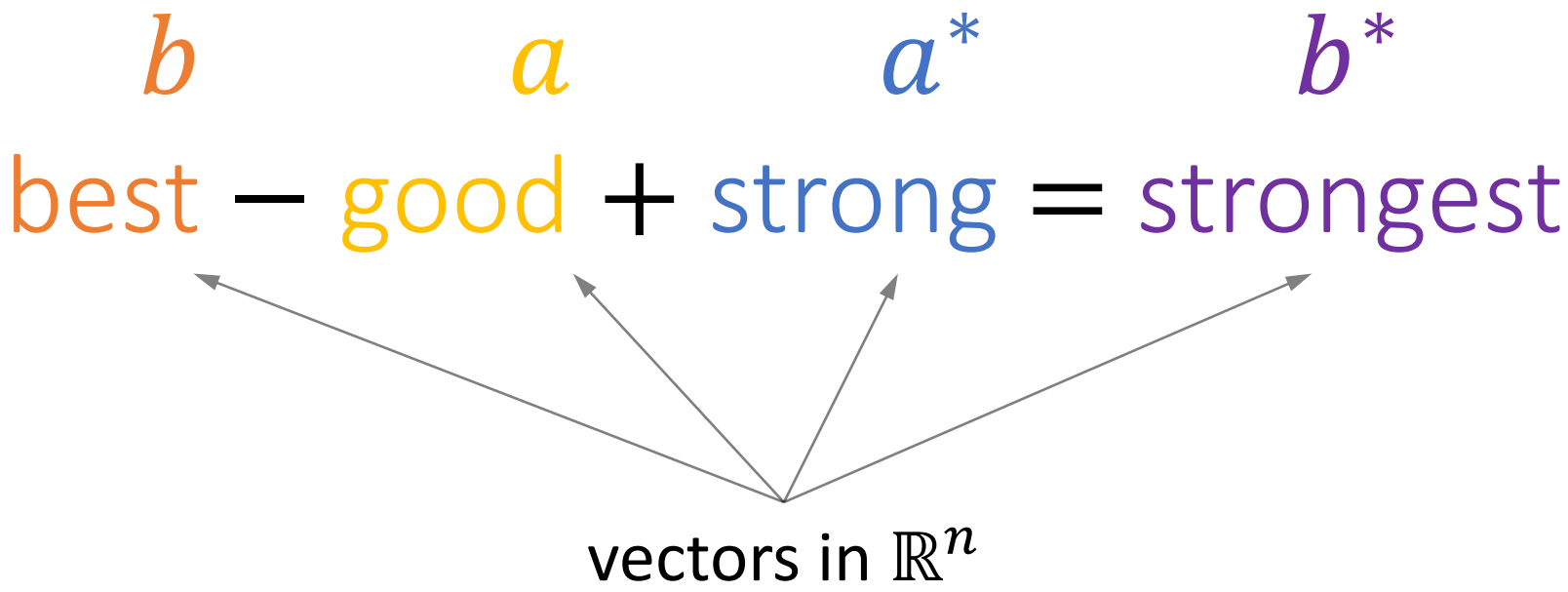
$$b \qquad a \qquad a^* \qquad b^*$$
$$\text{Tokyo} - \text{Japan} + \text{France} = \text{Paris}$$

Mikolov et al. (2013a,b,c)

$$b \qquad a \qquad a^* \qquad b^*$$

$$\text{best} - \text{good} + \text{strong} = \text{strongest}$$

Mikolov et al. (2013a,b,c)



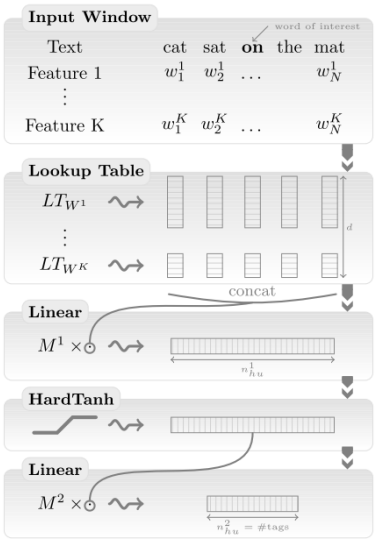


Figure 3: The C&W model without ranking objective (Collobert et al., 2011)

The resulting language model produces embeddings that already possess many of the relations word embeddings have become known for, e.g. countries are clustered close together and syntactically similar words occupy similar locations in the vector space. While their ranking objective eliminates the complexity of the softmax, they keep the intermediate fully-connected hidden layer (**2.**) of Bengio et al. around (the **HardTanh** layer in Figure 3), which constitutes another source of expensive computation. Partially due to this, their full model trains for seven weeks in total with $|V| = 130000$.

Word2Vec

Let us now introduce arguably the most popular word embedding model, the model that launched a thousand word embedding papers: word2vec, the subject of two papers by Mikolov et al. in 2013. As word embeddings are a key building block of deep learning models for NLP, word2vec is often assumed to belong to the same group. Technically however, word2vec is not be considered to be part of deep learning, as its architecture is neither deep nor uses non-linearities (in contrast to Bengio's model and the C&W model).

In their first paper [2], Mikolov et al. propose two architectures for learning word embeddings that are computationally less expensive than previous models. In their second paper [3], they improve upon these models by employing additional strategies to enhance training speed and accuracy.
These architectures offer two main benefits over the C&W model and Bengio's language model:

- They do away with the expensive hidden layer.
- They enable the language model to take additional context into account.

As we will later show, the success of their model is not only due to these changes, but especially due to certain training strategies.

In the following, we will look at both of these architectures:

Continuous bag-of-words (CBOW)

While a language model is only able to look at the past words for its predictions, as it is evaluated on its ability to predict each next word in the corpus, a model that just aims to generate accurate word embeddings does not suffer from this restriction. Mikolov et al. thus use both the n words before and after the target word w_t to predict it as depicted in Figure 4. They call this continuous bag-of-words (CBOW), as it uses continuous representations whose order is of no importance.

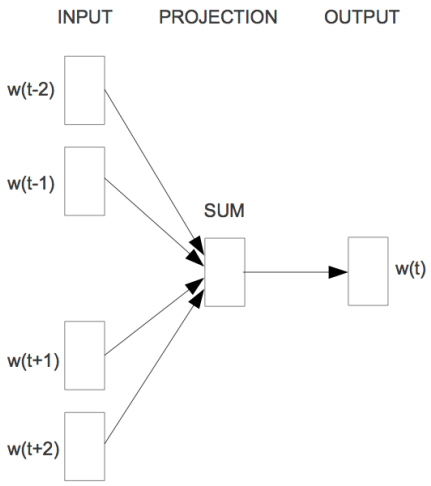


Figure 4: Continuous bag-of-words (Mikolov et al., 2013)
The objective function of CBOW in turn is only slightly different than the language model one:

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T} \log p(w_t \mid w_{t-n}, \cdots, w_{t-1}, w_{t+1}, \cdots, w_{t+n}).$$

Instead of feeding n previous words into the model, the model receives a window of n words around the target word w_t at each time step t.

Skip-gram

While CBOW can be seen as a precognitive language model, skip-gram turns the language model objective on its head: Instead of using the surrounding words to predict the centre word as with CBOW, skip-gram uses the centre word to predict the surrounding words as can be seen in Figure 5.

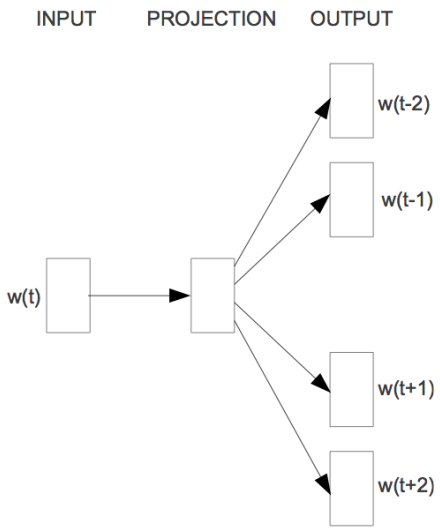


Figure 5: Skip-gram (Mikolov et al., 2013)

The skip-gram objective thus sums the log probabilities of the surrounding n words to the left and to the right of the target word w_t to produce the following objective:

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T} \sum_{-n \leq j \leq n, \neq 0} \log p(w_{t+j} \mid w_t).$$

To gain a better intuition of how the skip-gram model computes $p(w_{t+j} \mid w_t)$, let's recall the definition of our softmax:

$$p(w_t \mid w_{t-1}, \cdots, w_{t-n+1}) = \frac{\exp(h^\top v'_{w_t})}{\sum_{w_i \in V} \exp(h^\top v'_{w_i})}.$$

Instead of computing the probability of the target word w_t given its previous words, we calculate the probability of the surrounding word w_{t+j} given w_t. We can thus simply replace these variables in the equation:

$$p(w_{t+j} \mid w_t) = \frac{\exp(h^\top v'_{w_{t+j}})}{\sum_{w_i \in V} \exp(h^\top v'_{w_i})}.$$

As the skip-gram architecture does not contain a hidden layer that produces an intermediate state vector h, h is simply the word embedding v_{w_t} of the input word w_t. This also makes it clearer why we want to have different

representations for input embeddings v_w and output embeddings v'_w, as we would otherwise multiply the word embedding by itself. Replacing h with v_{w_t} yields:

$$p(w_{t+j} \mid w_t) = \frac{\exp(v_{w_t}^\top v'_{w_{t+j}})}{\sum_{w_i \in V} \exp(v_{w_t}^\top v'_{w_i})}.$$

Note that the notation in Mikolov's paper differs slightly from ours, as they denote the centre word with w_I and the surrounding words with w_O. If we replace w_t with w_I, w_{t+j} with w_O, and swap the vectors in the inner product due to its commutativity, we arrive at the softmax notation in their paper:

$$p(w_O|w_I) = \frac{\exp(v'^\top_{w_O} v_{w_I})}{\sum_{w=1}^{V} \exp(v'^\top_w v_{w_I})}.$$

In the next post, we will discuss different ways to approximate the expensive softmax as well as key training decisions that account for much of skip-gram's success. We will also introduce GloVe [5], a word embedding model based on matrix factorisation and discuss the link between word embeddings and methods from distributional semantics.

Did I miss anything? **Let me know in the comments below.**

Other blog posts on word embeddings

If you want to learn more about word embeddings, these other blog posts on word embeddings are also available:

- On word embeddings - Part 2: Approximating the

Max-pooling The most common pooling operation is *max pooling*, taking the maximum value across each dimension.

$$\boldsymbol{c}_{[j]} = \max_{1<i\leq m} \boldsymbol{p_i}_{[j]} \qquad \forall j \in [1, \ell], \tag{13.4}$$

$\boldsymbol{p_i}_{[j]}$ denotes the jth component of $\boldsymbol{p_i}$. The effect of the max-pooling operation is to get the most salient information across window positions. Ideally, each dimension will "specialize" in a particular sort of predictors, and max operation will pick on the most important predictor of each type.

Figure 13.2 provides an illustration of the convolution and pooling process with a max-pooling operation.

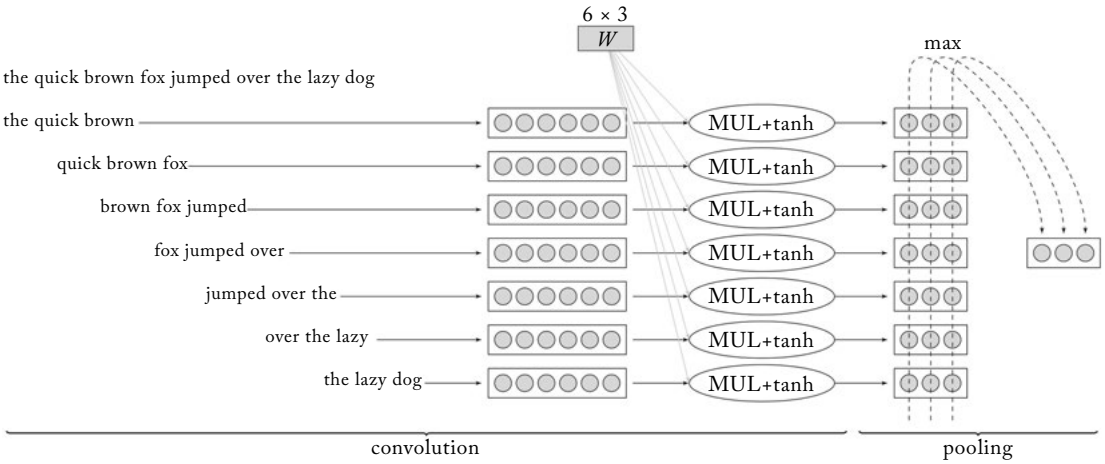


Figure 13.2: 1D convolution+pooling over the sentence "the quick brown fox jumped over the lazy dog." This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transfered through a 6×3 filter (linear transformation followed by element-wise tanh), resulting in seven 3-dimensional filtered representations. Then, a max-pooling operation is applied, taking the max over each dimension, resulting in a final 3-dimensional pooled vector.

Average Pooling The second most common pooling type being *average-pooling*—taking the average value of each index instead of the max:

$$\boldsymbol{c} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{p_i}. \tag{13.5}$$

14.1 THE RNN ABSTRACTION

We use $\boldsymbol{x_{i:j}}$ to denote the sequence of vectors $\boldsymbol{x_i}, \ldots, \boldsymbol{x_j}$. On a high-level, the RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\boldsymbol{x_{1:n}} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$, $(\boldsymbol{x_i} \in \mathbb{R}^{d_{in}})$ and returns as output a single d_{out} dimensional vector $\boldsymbol{y_n} \in \mathbb{R}^{d_{out}}$:

$$\boldsymbol{y_n} = \text{RNN}(\boldsymbol{x_{1:n}}) \tag{14.1}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{y_n} \in \mathbb{R}^{d_{out}}.$$

This implicitly defines an output vector $\boldsymbol{y_i}$ for each prefix $\boldsymbol{x_{1:i}}$ of the sequence $\boldsymbol{x_{1:n}}$. We denote by $\text{RNN}^\star$ the function returning this sequence:

$$\begin{aligned} \boldsymbol{y_{1:n}} &= \text{RNN}^\star(\boldsymbol{x_{1:n}}) \\ \boldsymbol{y_i} &= \text{RNN}(\boldsymbol{x}_{1:i}) \end{aligned} \tag{14.2}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}.$$

The output vector $\boldsymbol{y_n}$ is then used for further prediction. For example, a model for predicting the conditional probability of an event e given the sequence $\boldsymbol{x_{1:n}}$ can be defined as $p(e = j|\boldsymbol{x_{1:n}}) = \text{softmax}(\text{RNN}(\boldsymbol{x_{1:n}}) \cdot \boldsymbol{W} + \boldsymbol{b})_{[j]}$, the jth element in the output vector resulting from the softmax operation over a linear transformation of the RNN encoding $\boldsymbol{y_n} = \text{RNN}(\boldsymbol{x_{1:n}})$. The RNN function provides a framework for conditioning on the entire history $\boldsymbol{x_1}, \ldots, \boldsymbol{x_i}$ without resorting to the Markov assumption which is traditionally used for modeling sequences, described in Chapter 9. Indeed, RNN-based language models result in very good perplexity scores when compared to ngram-based models.

Looking in a bit more detail, the RNN is defined recursively, by means of a function R taking as input a state vector $\boldsymbol{s_{i-1}}$ and an input vector $\boldsymbol{x_i}$ and returning a new state vector $\boldsymbol{s_i}$. The state vector $\boldsymbol{s_i}$ is then mapped to an output vector $\boldsymbol{y_i}$ using a simple deterministic function $O(\cdot)$.[2] The base of the recursion is an initial state vector, $\boldsymbol{s_0}$, which is also an input to the RNN. For brevity, we often omit the initial vector $\boldsymbol{s_0}$, or assume it is the zero vector.

When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\boldsymbol{x_i}$ as well as the dimensions of the outputs $\boldsymbol{y_i}$. The dimensions of the states $\boldsymbol{s_i}$ are a function of the output dimension.[3]

[2]Using the O function is somewhat non-standard, and is introduced in order to unify the different RNN models to to be presented in the next chapter. For the Simple RNN (Elman RNN) and the GRU architectures, O is the identity mapping, and for the LSTM architecture O selects a fixed subset of the state.

[3]While RNN architectures in which the state dimension is independent of the output dimension are possible, the current popular architectures, including the Simple RNN, the LSTM, and the GRU do not follow this flexibility.

$$\begin{aligned} \text{RNN}^\star(\boldsymbol{x_{1:n}}; \boldsymbol{s_0}) &= \boldsymbol{y_{1:n}} \\ \boldsymbol{y_i} &= O(\boldsymbol{s_i}) \\ \boldsymbol{s_i} &= R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i}) \end{aligned} \tag{14.3}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}, \quad \boldsymbol{s_i} \in \mathbb{R}^{f(d_{out})}.$$

The functions R and O are the same across the sequence positions, but the RNN keeps track of the states of computation through the state vector $\boldsymbol{s_i}$ that is kept and being passed across invocations of R.

Graphically, the RNN has been traditionally presented as in Figure 14.1.

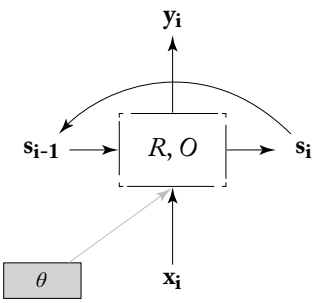


Figure 14.1: Graphical representation of an RNN (recursive).

This presentation follows the recursive definition, and is correct for arbitrarily long sequences. However, for a finite sized input sequence (and all input sequences we deal with are finite) one can *unroll* the recursion, resulting in the structure in Figure 14.2.

While not usually shown in the visualization, we include here the parameters θ in order to highlight the fact that the same parameters are shared across all time steps. Different instantiations of R and O will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. However, they all adhere to the same abstract interface. We will provide details of concrete instantiations of R and O—the Simple RNN, the LSTM, and the GRU—in Chapter 15. Before that, let's consider working with the RNN abstraction.

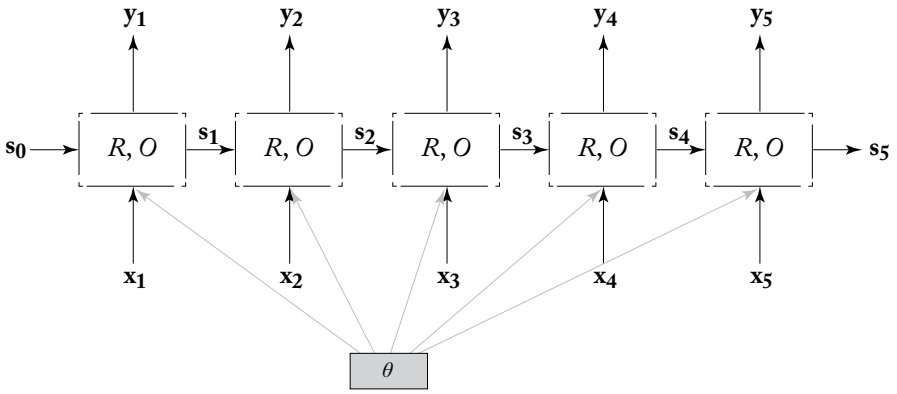


Figure 14.2: Graphical representation of an RNN (unrolled).

First, we note that the value of s_i (and hence y_i) is based on the entire input $x_1, \ldots, x_i$. For example, by expanding the recursion for $i = 4$ we get:

$$\begin{aligned} s_4 &= R(s_3, x_4) \\ &= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\ &= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\ &= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4). \end{aligned} \tag{14.4}$$

Thus, s_n and y_n can be thought of as *encoding* the entire input sequence.[4] Is the encoding useful? This depends on our definition of usefulness. The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are tying to solve.

14.2 RNN TRAINING

Viewed as in Figure 14.2 it is easy to see that an unrolled RNN is just a very deep neural network (or rather, a very large *computation graph* with somewhat complex nodes), in which the same parameters are shared across many parts of the computation, and additional input is added at various layers. To train an RNN network, then, all we need to do is to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph, and then use the backward

[4]Note that, unless R is specifically designed against this, it is likely that the later elements of the input sequence have stronger effect on s_n than earlier ones.