CS546: Machine Learning in NLP (Spring 2020)

http://courses.engr.illinois.edu/cs546/

Lecture 2
More Intro…

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Office hours: Monday, 11am—12:30pm

Wrap-up: Syllabus for this class

Schedule

01/22	*Introduction*	Overview,Policies
01/24	*Static Word Embeddings*	Basic intro to Word2Vec, GloVe etc.
01/29	*Static Word Embeddings*	More in-depth analysis of static embeddings
01/31	*Basic Neural Architectures*	Recurrent Neural Nets and variants thereof
02/05	*Basic Neural Architectures*	Convolutional Neural Nets
02/07	*Lexical Encodings*	Character RNNs, BPE, FastText etc.
02/12	*Context-Dependent Word Embeddings*	Elmo etc.
02/14	*Advanced Neural Architectures*	Transformers
02/19	*Context-Dependent Word Embeddings*	BERT, GPT-2 et al.
02/21	*Context-Dependent Word Embeddings*	BERT, GPT-2 et al. c'td
02/26	*Advanced Neural Architectures*	Deep Learning for Graphs
02/28	*Project Proposal Presentations*	
03/04	*NLP Applications*	Neural Generation
03/06	*NLP Applications*	Neural Summarization
03/11	*NLP Applications*	Neural Machine Translation
03/13	*NLP Applications*	Neural Sequence Labeling
03/25	*NLP Applications*	Neural Structure Prediction
03/27	*NLP Applications*	Neural Question Answering
04/01	*NLP Applications*	More on Neural Question Answering
04/03	*NLP Applications*	Multimodal NLP
04/08	*NLP Applications*	Neural Dialogue
04/10	*NLP Applications*	More on Neural Dialogue
04/15	*Project Update Presentations*	
04/17	*Advanced Neural Architectures*	Intro to GANs in NLP
04/22	*Advanced Neural Architectures*	More on GANs in NLP
04/24	*Training Regimes*	Reinforcement Learning in NLP
04/29	*(TBC) Training Regimes*	(TBC) More on Reinforcement Learning in NLP
05/01	*(TBC) Training Regimes*	(TBC) Deep Learning in Low Resource Settings
05/07	*Final Project Presentations*	

Admin

You will receive an email with a link to a Google form where you can sign up for slots to present.

— Please sign up for at least three slots so that I have some flexibility in assigning you to a presentation

We will give you one week to fill this in.

You will have to meet with me the Monday before your presentation to go over your slides.

Grading criteria for presentations

- **Clarity of exposition and presentation**
- **Analysis**
  (don't just regurgitate what's in the paper)
- **Quality of slides**
  (and effort that went into making them
  - just re-using other people's slides is not enough)

Why does NLP need ML?

NLP research questions redux

How do you represent (or predict) words?

- Do you treat words in the input as atomic categories, as continuous vectors, or as structured objects?
- How do you handle rare/unseen words, typos, spelling variants, morphological information?
- Lexical semantics: do you capture word meanings/senses?

How do you represent (or predict) word sequences?

- Sequences = sentences, paragraphs, documents, dialogs,…
- As a vector, or as a structured object?

How do you represent (or predict) structures?

- Structures = labeled sequences, trees, graphs, formal languages (e.g. DB records/queries, logical representations)
- How do you represent “meaning”?

Two core problems for NLP

Ambiguity: Natural language is highly ambiguous

- Words have multiple senses and different POS
- Sentences have a myriad of possible parses
- etc.

Coverage (compounded by Zipf's Law)

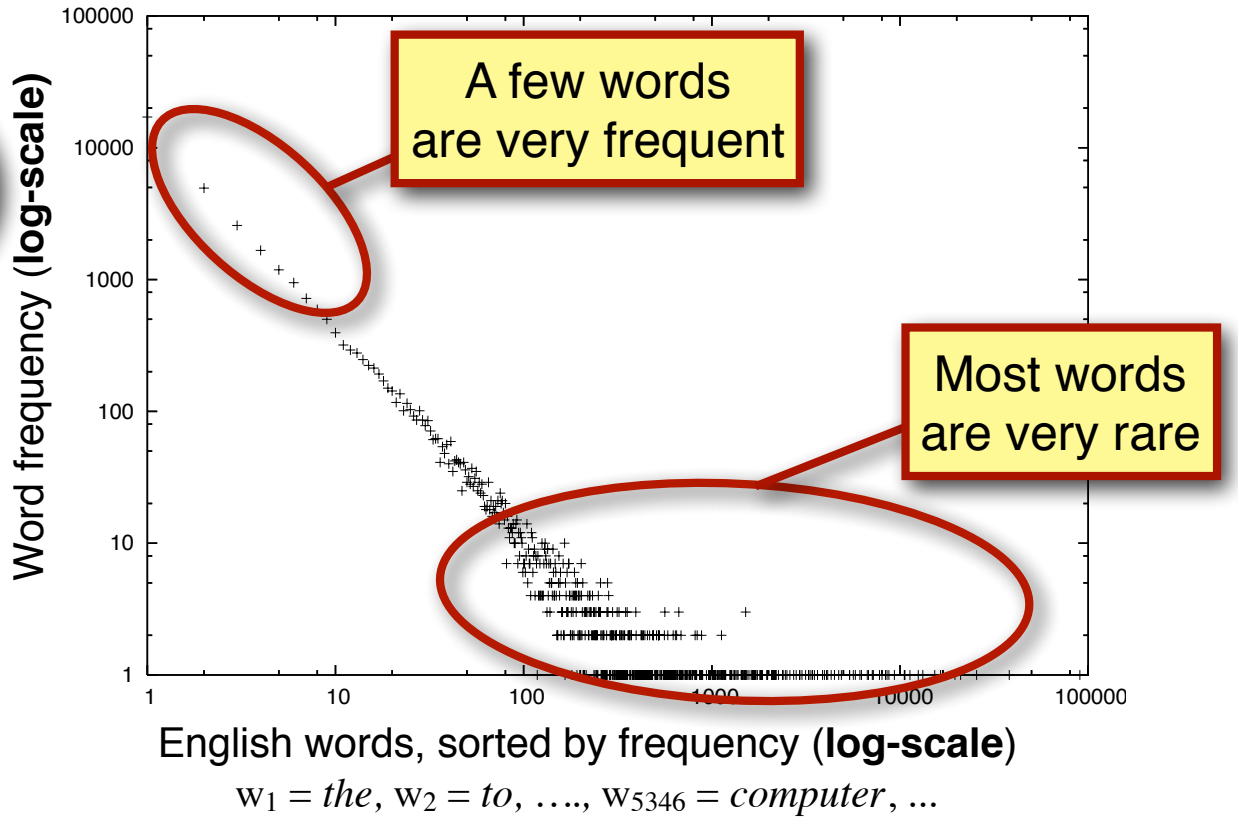
- Any (wide-coverage) NLP system will come across words or constructions that did not occur during training.
- We need to be able to generalize from the seen events during training to unseen events that occur during testing (i.e. when we actually use the system).

The coverage problem

Zipf's law: the long tail

How many words occur once, twice, 100 times, 1000 times?

the *r*-th most common word w_r has $P(w_r) \propto 1/r$



In natural language:

- A small number of events (e.g. words) occur with high frequency
- A large number of events occur with very low frequency

Implications of Zipf's Law for NLP

The good:

Any text will contain a number of words that are very **common**.

We have seen these words often enough that we know (almost) everything about them. These words will help us get at the structure (and possibly meaning) of this text.

The bad:

Any text will contain a number of words that are **rare**.

We know something about these words, but haven't seen them often enough to know everything about them. They may occur with a meaning or a part of speech we haven't seen before.

The ugly:

Any text will contain a number of words that are **unknown** to us.

We have *never* seen them before, but we still need to get at the structure (and meaning) of these texts.

Dealing with the bad and the ugly

Our systems need to be able to **generalize** from what they have seen to unseen events.

There are two (complementary) approaches to generalization:

— **Linguistics** provides us with insights about the rules and structures in language that we can exploit in the (symbolic) representations we use

E.g.: a finite set of grammar rules is enough to describe an infinite language

— **Machine Learning/Statistics** allows us to learn models (and/or representations) from real data that often work well empirically on unseen data

E.g. most statistical or neural NLP

How do we represent words?

Option 1: Words are **atomic symbols**

Can't capture syntactic/semantic relations between words

- Each (surface) word form is its own symbol
- Map different forms of a word to the same symbol
  - **Lemmatization**: map each word to its lemma (esp. in English, the lemma is still a word in the language, but lemmatized text is no longer grammatical)
  - **Stemming**: remove endings that differ among word forms (no guarantee that the resulting symbol is an actual word)
  - **Normalization:** map all variants of the same word (form) to the same canonical variant (e.g. lowercase everything, normalize spellings, perhaps spell-check)

How do we represent words?

Option 2: Represent the **structure** of each word

"books" => "book N pl" (or "book V 3rd sg")

This requires a **morphological analyzer**

The output is often a lemma plus morphological information

This is particularly useful for highly inflected languages (less so for English or Chinese)

Aims:

— the lemma/stem captures core (semantic) information

— reduce the vocabulary of highly inflected languages

How do we represent words?

Option 3: Each word is a **(high-dimensional) vector**
Advantage: Neural nets need vectors as input!

How do we represent words as vectors?

- **Naive solution**:
  as one-hot vectors
- **Distributional similarity solution**:
  as very high-dimensional sparse vectors
- **Static word embedding solution** (word2vec etc.):
  by a dictionary that maps words to
  fixed lower-dimensional dense vectors
- **Dynamic embedding solution** (Elmo etc.):
  Compute context-dependent dense embeddings

How do we represent unknown words?

Systems that use machine learning may need to have a unique representation of each word.

Option 1: the **UNK** token

Replace all rare words (in your training data) with an UNK token (for Unknown word).

Replace *all* unknown words that you come across after training (including rare training words) with the same UNK token

Option 2: **substring-based** representations

Represent (rare and unknown) words as sequences of characters or substrings

- Byte Pair Encoding: learn which character sequences are common in the vocabulary of your language

The ambiguity problem

"I made her duck"

What does this sentence mean?

*"**duck**"*: noun or verb?

*"**make**"*: "*cook X*" or "*cause X to do Y*" ?

*"**her**": "for her"* or "*belonging to her*" ?

Language has different kinds of ambiguity, e.g.:

Structural ambiguity

*"I eat sushi **with tuna**"* vs. *"I eat sushi **with chopsticks**"*

"*I saw the man **with the telescope on the hill**"*

Lexical (word sense) ambiguity

"*I went to the **bank**"*: financial institution or river bank?

Referential ambiguity

"***John** saw **Jim**. **He** was drinking coffee."*

Task: Part-of-speech-tagging

Open the pod door, Hal.

Verb Det Noun Noun , Name .

Open the pod door , Hal .

***open*:**
verb, adjective, or noun?
Verb: ***open*** *the door*
Adjective: *the* ***open*** *door*
Noun: *in the* ***open***

How do we decide?

We want to know the most likely tags T for the sentence S

$$\underset{T}{\operatorname{argmax}} P(T|S)$$

We need to define *a statistical model* of $P(T \mid S)$, e.g.:

$$\underset{T}{\operatorname{argmax}} P(T|S) = \underset{T}{\operatorname{argmax}} P(T)P(S|T)$$

$$P(T) =_{def} \prod_i P(t_i|t_{i-1})$$

$$P(S|T) =_{def} \prod_i P(w_i \mid t_i)$$

We need to estimate *the parameters* of $P(T|S)$, e.g.:

$$P(t_i = V \mid t_{i-1} = N) = 0.3$$

"I made her duck cassoulet"

(Cassoulet = a French bean casserole)

The second major problem in NLP is **coverage**:
We will always encounter unfamiliar words
and constructions.

Our models need to be able to deal with this.

This means that our models need to be able
to *generalize* from what they have been trained on
to what they will be used on.

Statistical NLP

The last big paradigm shift

Starting in the early 1990s, NLP became very empirical and data-driven due to

- success of statistical methods in machine translation (IBM systems)
- availability of large(ish) annotated corpora (Susanne Treebank, Penn Treebank, etc.)

Advantages over rule-based approaches:

- Common benchmarks to compare models against
- Empirical (objective) evaluation is possible
- Better coverage
- Principled way to handle ambiguity

Statistical models for NLP

NLP makes heavy use of statistical models as a way to handle both the ambiguity and the coverage issues.

- Probabilistic models (e.g. HMMs, MEMMs, CRFs, PCFGs)
- Other machine learning-based classifiers

Basic approach:

- Decide which output is desired
  (may depend on available labeled training data)
- Decide what kind of model to use
- Define features that could be useful (this may require further processing steps, i.e. a pipeline)
- Train and evaluate the model.
- Iterate: refine/improve the model and/or the features, etc.

Example: Language Modeling

A language model defines a distribution $P(\mathbf{w})$ over the strings $\mathbf{w} = w_1 w_2 .. w_i \ldots$ in a language

Typically we factor $P(\mathbf{w})$ such that we compute the probability word by word:

$$P(\mathbf{w}) = P(w_1)\, P(w_2 \mid w_1) \ldots\, P(w_i \mid w_1 \ldots w_{i-1})$$

Standard n-gram models make the Markov assumption that w_i depends only on the preceding n–1 words:

$$P(w_i \mid w_1 \ldots w_{i-1}) := P(w_i \mid w_{i-n+1} \ldots w_{i-1})$$

We know that this independence assumption is invalid (there are many long-range dependencies), but it is computationally and statistically necessary

(we can't store or estimate larger models)

Features in statistical NLP

Many statistical NLP systems use explicit features:

- Words (does the word "river" occur in this sentence?)
- POS tags
- Chunk information, NER labels
- Parse trees or syntactic dependencies (e.g. for semantic role labeling, etc.)

Feature design is usually a big component of building any particular NLP system.

Which features are useful for a particular task and model typically requires experimentation, but there are a number of commonly used ones (words, POS tags, syntactic dependencies, NER labels, etc.)

Neural approaches to NLP

Motivation for neural approaches to NLP: Markov assumptions

Traditional sequence models (n-gram language models, HMMs, MEMMs, CRFs) make **rigid Markov assumptions** (bigram/trigram/n-gram).

Recurrent neural nets (RNNs, LSTMs) and transformers can capture **arbitrary-length histories without requiring more parameters**.

Motivation for neural approaches to NLP: Features can be brittle

Word-based features:

How do we handle unseen/rare words?

Many features are produced by other NLP systems (POS tags, dependencies, NER output, etc.)
These systems are often trained on labeled data.

- Producing labeled data can be very expensive.
- We typically don't have enough labeled data from the domain of interest.
- We might not get accurate features for our domain of interest.

Features in neural approaches

Many of the current successful neural approaches to NLP do not use traditional discrete features.
— End-to-end models: no pipeline!

Words in the input are often represented as **dense vectors** (aka. word embeddings, e.g. word2vec)

Traditional approaches: each word in the vocabulary is a separate feature. No generalization across words that have similar meanings.

Neural approaches: Words with similar meanings have similar vectors. Models generalize across words with similar meanings

Other kinds of features (POS tags, dependencies, etc.) **are often ignored**.

What is “deep learning”?

Neural networks, typically with several hidden layers

- (depth = # of hidden layers)
- Single-layer neural nets are linear classifiers
- Multi-layer neural nets are more expressive

Very impressive performance gains in computer vision (ImageNet) and speech recognition over the last several years.

Neural nets have been around for decades.

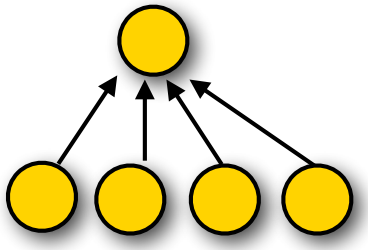
Why have they suddenly made a comeback?

- Fast computers (GPUs!) and (very) large datasets have made it possible to train these very complex models.

What are neural nets?

Simplest variant: single-layer feedforward net

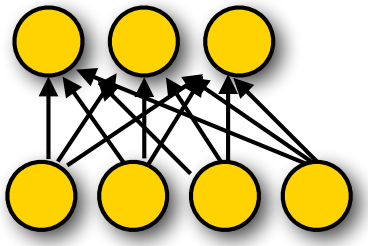
For binary classification tasks:
Single output unit
Return 1 if y > 0.5
Return 0 otherwise



Output unit: scalar *y*

Input layer: vector **x**

For multiclass classification tasks:
K output units (a vector)
Each output unit
y_i = class i
Return $\text{argmax}_i(y_i)$



Output layer: vector **y**

Input layer: vector **x**

Multiclass models: softmax(y_i)

Multiclass classification = predict one of K classes.
Return the class i with the highest score: $\text{argmax}_i(y_i)$

In neural networks, this is typically done by using the **softmax** function, which maps real-valued vectors in R^N into a distribution over the N outputs
For a vector $\mathbf{z} = (z_0 \ldots z_K)$: $P(i) = \text{softmax}(z_i) = \exp(z_i) \; / \; \sum_{k=0..K} \exp(z_k)$
(NB: This is just logistic regression)

Single-layer feedforward networks

Single-layer (linear) feedforward network

$y = \mathbf{wx} + b$ (binary classification)

w is a weight vector, b is a bias term (a scalar)

This is just a linear classifier (aka Perceptron)
(the output y is a linear function of the input **x**)

Single-layer non-linear feedforward networks:

Pass **wx** + b through a non-linear activation function,
e.g. $y = \tanh(\mathbf{wx} + b)$

Nonlinear activation functions

Sigmoid (logistic function): $\sigma(x) = 1/(1 + e^{-x})$

Useful for output units (probabilities) [0,1] range

Hyperbolic tangent: $\tanh(x) = (e^{2x} - 1)/(e^{2x}+1)$

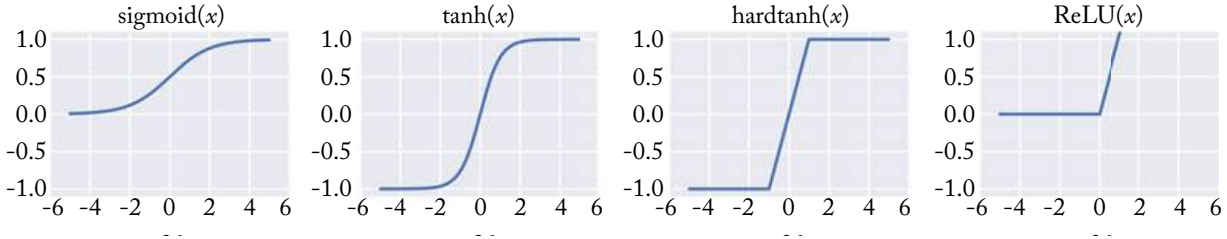
Useful for internal units: [-1,1] range

Hard tanh (approximates tanh)

$$\text{htanh}(x) = -1 \text{ for } x < -1,\ 1 \text{ for } x > 1,\ x \text{ otherwise}$$

Rectified Linear Unit: $\text{ReLU}(x) = \max(0, x)$

Useful for internal units

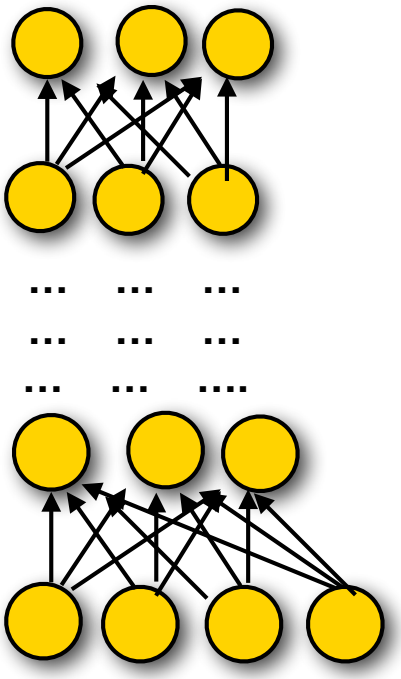


Softmax: $\text{softmax}(z_i) = \exp(z_i) \ / \ \sum_{k=0..K} \exp(z_k)$

Special case for output units (multiclass classification)

Multi-layer feedforward networks

We can generalize this to multi-layer feedforward nets



Output layer: vector **y**

Hidden layer: vector $\mathbf{h_n}$

Hidden layer: vector $\mathbf{h_1}$

Input layer: vector **x**

Challenges in using NNs for NLP

In NLP, the input and output variables are discrete: words, labels, structures.

NNs work best with continuous vectors.

- We typically want to learn a mapping (embedding) from discrete words (input) to dense vectors.
- We can do this with (simple) neural nets and related methods.

The input to a NN is (traditionally) a fixed-length vector. How do you represent a variable-length sequence as a vector?

- With recurrent neural nets: read in one word at the time to predict a vector, use that vector and the next word to predict a new vector, etc.;
- With convolutional nets: use a sliding (fixed-length) window)

How does NLP use NNs?

Word embeddings (word2vec, Glove, etc.)

- Train a NN to predict a word from its context (or the context from a word) to get a dense vector representation of each word

Neural language models:

- Use recurrent neural networks (RNNs, GRUs, LSTMs) to predict word sequences (or to obtain context-sensitive embeddings (ELMO)

Sequence-to-sequence (seq2seq) models:

- From machine translation: use one RNN to encode source string, and another RNN to decode this into a target string.
- Also used for automatic image captioning, etc.

Convolutional neural nets

- Used e.g. for text classification

Transformers

Neural Language Models

What is a language model?

Probability distribution over the strings in a language, typically factored into distributions $P(w_i \mid \ldots)$ for each word:

$$P(\mathbf{w}) = P(w_1 \ldots w_n) = \prod_i P(w_i \mid w_1 \ldots w_{i-1})$$

N-gram models assume each word depends only preceding n–1 words:

$$P(w_i \mid w_1 \ldots w_{i-1}) \;=_{\text{def}}\; P(w_i \mid w_{i-n+1} \ldots w_{i-1})$$

To handle variable length strings, we assume each string starts with n–1 start-of-sentence symbols (BOS), or ⟨S⟩
and ends in a special end-of-sentence symbol (EOS) or ⟨\S⟩

An n-gram model $P(w \mid w_1 \ldots w_k)$

— The **vocabulary** V contains *n* types (incl. UNK, BOS, EOS)
— We want to condition each word on *k* preceding words

— [Naive] Each **input word** $w_i \in V$ (that we're conditioning on)
is an ***n*-dimensional one-hot vector** $v(w) = (0,\ldots 0, 1, 0 \ldots 0)$
— Our **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $n \times k$ elements
— To predict the probability over output words,
the **output layer** is a softmax over n elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

With (say) one hidden layer $\mathbf{h}$ we'll need two sets of parameters, one for **h** and one for the output

Naive neural n-gram model

Architecture:

Input Layer: $\mathbf{x} = [v(w_1)\ldots v(w_k)]$

$v(w)$: a one-hot vector of size $\dim(V) = |V|$

Hidden Layer: $\mathbf{h} = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)$

Output Layer: $P(w \mid w1\ldots wk) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

Parameters:

Weight matrices and biases:

first layer: $\mathbf{W}^1 \in \mathbb{R}^{k \cdot \dim(V) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

second layer: $\mathbf{W}^2 \in \mathbb{R}^{\dim(\mathbf{h}) \times |V|}$ $\mathbf{b}^2 \in \mathbb{R}^{|V|}$

How many parameters do we need to learn?

Traditional n-gram model: $\dim(V)^k$ parameters

With dim(V) = 10,000 and k=3: 1,000,000,000,000

Naive neural n-gram model (one-hot encoding of vocabulary):

#parameters going to hidden layer: $k \cdot \dim(V) \cdot \dim(\mathbf{h})$,

with $\dim(\mathbf{h}) = 300$, dim(V) = 10,000 and k=3: 9,000,000

plus #parameters going to output layer: $\dim(\mathbf{h}) \cdot \dim(V)$

with $\dim(\mathbf{h}) = 300$, dim(V) = 10,000: 3,000,000

The neural model requires still a lot of parameters,
but far fewer than the traditional n-gram model

Naive neural n-gram models

Advantages over traditional n-gram models:

— The hidden layer captures interactions among context words

— Increasing the order of the n-gram requires only a small linear increase in the number of parameters.

dim($\mathbf{W}^1$) goes from $k \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$ to $(k+1) \cdot \dim(\text{emb}) \times \dim(\mathbf{h})$

A traditional k-gram model requires $\dim(V)^k$ parameters

— Increasing the vocabulary also leads only to a linear increase in the number of parameters

Better neural language models

Naive neural models have similar shortcomings as standard n-gram models

- Models get very large (and sparse) as n increases
- We can't generalize across similar contexts
- N-gram Markov (independence) assumptions are too strict

Better neural language models overcome these by…

… using **word embeddings** instead of one-hots as input:

Instead of representing context words as distinct, discrete symbols (i.e. very high-dimensional one-hot vectors), use a dense low-dimensional vector representation where similar words have similar vectors [next]

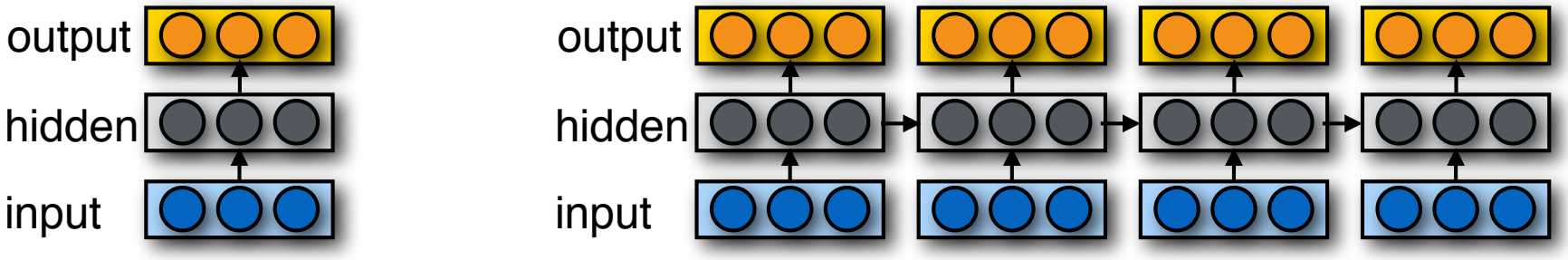
… using **recurrent nets** instead of feedforward nets:

Instead of a fixed-length (n-gram) context, use recurrent nets to encode variable-lengths contexts [later class]

Recurrent neural networks (RNNs)

Basic RNN: Modify the standard feedforward architecture (which predicts a string $w_0 \ldots w_n$ one word at a time) such that the output of the current step (w_i) is given as additional input to the next time step (when predicting the output for w_{i+1}).

"Output" — typically (the last) hidden layer.



Feedforward Net **Recurrent Net**

Word Embeddings (e.g. word2vec)

Main idea:
If you use a feedforward network to predict the probability of words that appear in the context of (near) an input word, the hidden layer of that network provides a dense vector representation of the input word.

Words that appear in similar contexts (that have high distributional similarity) wils have very similar vector representations.

These models can be trained on large amounts of raw text (and pretrained embeddings can be downloaded)

Sequence-to-sequence (seq2seq) models

Task (e.g. machine translation):

Given one variable length sequence as input, return another variable length sequence as output

Main idea:

Use one RNN to encode the input sequence (“encoder”)

Feed the last hidden state as input to a second RNN (“decoder”) that then generates the output sequence.

Use attention mechanisms (e.g. to focus on certain parts of the input when generating output)

Transformers

Non-recurrent architecture for seq2seq tasks:

— Also has an encoder and a decoder, but these process the input at once
(decoder may mask future outputs to generate output sequentially)

— May use positional embeddings to capture sequence information

— May use multiple self-attention (attention within a sequence) mechanisms in parallel

— Can be (pre)trained on very large amounts of data, and then fine-tuned for specific tasks

— Yields state-of-the-art context-dependent encodings (BERT) and language models (GPT-2)