INFO-I590 Fundamentals and Applications of LLMs

Word Representation and Text Classifiers

Jisun An

Many slides from Graham Neubig
and Dan Jurafsky

Overview

- Key components and development of NLP techniques for LLM
  - **Subword Models**: Tokenization methods used to process text
  - **Word Representations**: Techniques for encoding the semantic meaning of words
  - **Neural Networks**: The foundation of modern NLP architectures
  - **Sequence Models**: (next lecture) Techniques for understanding context in sequential data,

NLP System Building Overview

A General Framework for NLP Systems

- Create a function to map an input X into an output Y, where X and/or Y involve language

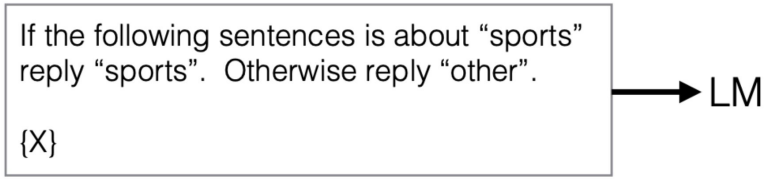
Input *X*	Output *Y*	Task
Text	Continuing Text	Language Modeling
Text	Text in Other Language	Translation
Text	Label	Text Classification
Text	Linguistic Structure	Language Analysis
Image	Text	Image Captioning

Methods for Creating NLP Systems

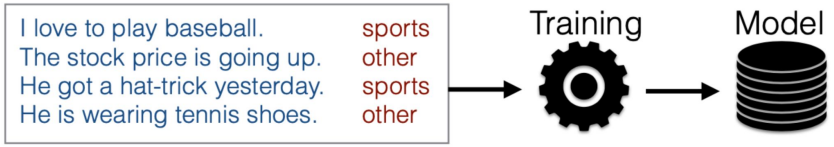
- Rules: Manual creation of rules

```
def classify(x: str) -> str:
    sports_keywords = ["baseball", "soccer", "football", "tennis"]
    if any(keyword in x for keyword in sports_keywords):
        return "sports"
    else:
        return "other"
```

- Prompting: Prompting a language model w/o training



- Fine-tuning: Machine learning from paired data <X, Y>



Let's Try to Make a Rule-based NLP System!

Example Task: Review Sentiment Analysis

- Given a review on a reviewing website (*x*), decide whether its label (*y*) is positive (1), negative (-1), or neutral (0)
  - Input: a review text
  - Output: a sentiment label

- Examples:
  - “I hate this movie”
  - “I love this movie”
  - “I saw this movie”

Example Task: Review Sentiment Analysis

- Given a review on a reviewing website (*x*), decide whether its label (*y*) is positive (1), negative (-1), or neutral (0)
  - Input: a review text
  - Output: a sentiment label

- Examples:
  - “I hate this movie” → Positive (1)
  - “I love this movie” → Negative (-1)
  - “I saw this movie” → neutral (0)

A Three-Step Process for Text Classification

1. **Feature Extraction**: Identify and extract the key features from the text that are crucial for decision-making.
2. **Score Calculation**: Compute a score for one or more potential outcomes based on the extracted features.
3. **Decision Function**: Select the most appropriate option among the possible outcomes.

Formally

1. Feature Extraction: $\mathbf{h} = f(\mathbf{x})$

2. Score Calculation: binary, multi-class

$$s = \mathbf{w} \cdot \mathbf{h} \quad \mathbf{s} = W\mathbf{h}$$

3. Decision: $\hat{y} = \text{decide}(\mathbf{s})$

Rule-based Sentiment Classifier

- Lexicons
  - good_words = ['love', 'good', 'nice', 'great', 'enjoy', 'enjoyed']
  - bad_words = ['hate', 'bad', 'terrible', 'disappointing', 'sad', 'lost', 'angry']
- Features
  - Feature 1 (f_1): number of good words
  - Feature 2 (f_2): number of bad words
- Score function: $s = w_1 * f_1 + w_2 * f_2 + b$
  - $w_1 = 1.0$, $w_2 = -1.0$, b = 0.5 # b is a “bias,” a default value
- Decision: If s > 0: 1; elif s < 0: -1; else 0
- Evaluation: Accuracy on test data?

Some Difficult Cases

Low-frequency Words

The action switches between past and present , but the material link is too **tenuous** to anchor the emotional connections that **purport** to span a 125-year divide .

negative

Here 's yet another studio horror franchise **mucking** up its storyline with **glitches** casual fans could correct in their sleep .

negative

Solution?: Keep working till we get all of them? Incorporate external resources such as sentiment dictionaries?

Conjugation

An operatic , sprawling picture that 's **entertainingly** acted , **magnificently** shot and gripping enough to sustain most of its 170-minute length .

positive

It 's basically an **overlong** episode of Tales from the Crypt .

negative

Solution?: Use the root form and POS of word?

Note: Would require morphological analysis.

Negation

This one is not nearly as dreadful as expected .

positive

Serving Sara does n't serve up a whole lot of laughs .

negative

Solution?: If a negation modifies a word, disregard it. Note: Would probably need to do syntactic analysis.

Metaphor, Analogy

Puts a human face on a land most Westerners are unfamiliar with.

positive

Green might want to hang onto that ski mask , as robbery may be the only way to pay for his next project .

negative

Has all the depth of a wading pool .

negative

Solution?: ???

Other Languages

끝까지 몰입하며 즐길 수 있는 훌륭한
작품이었습니다. positive

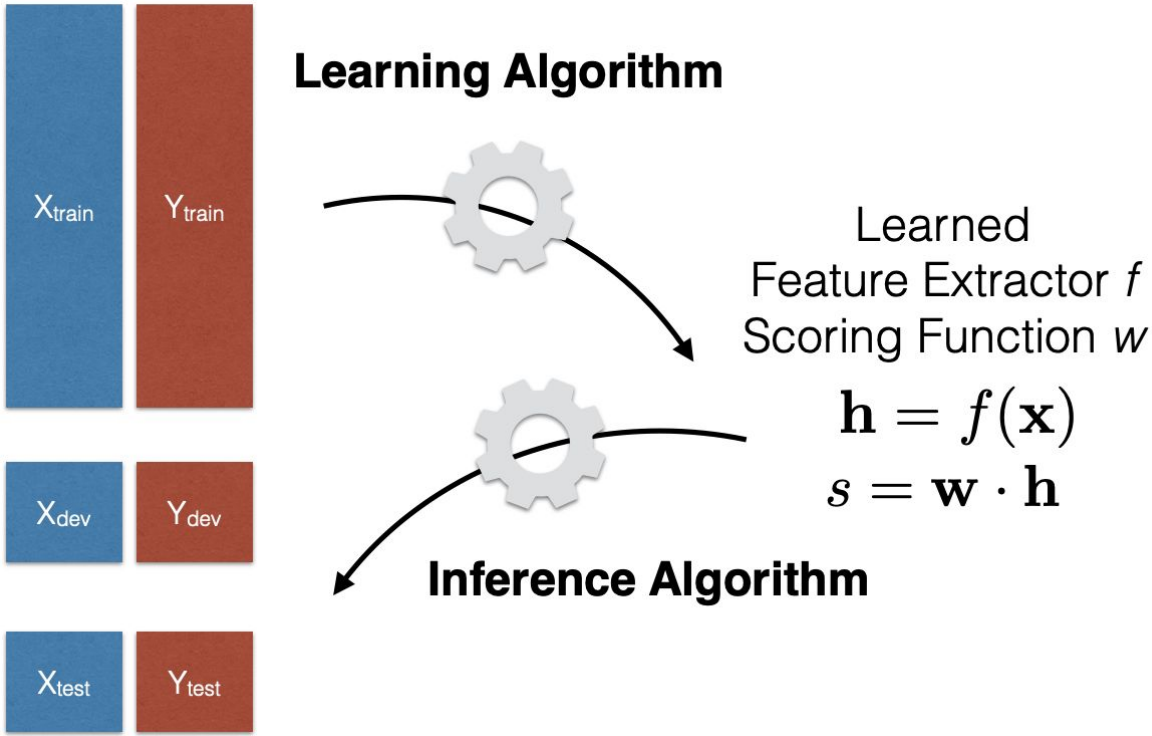
스토리 전개가 엉성하고 캐릭터들의 감정선도
전혀 공감되지 않아 실망스러웠습니다.

negative

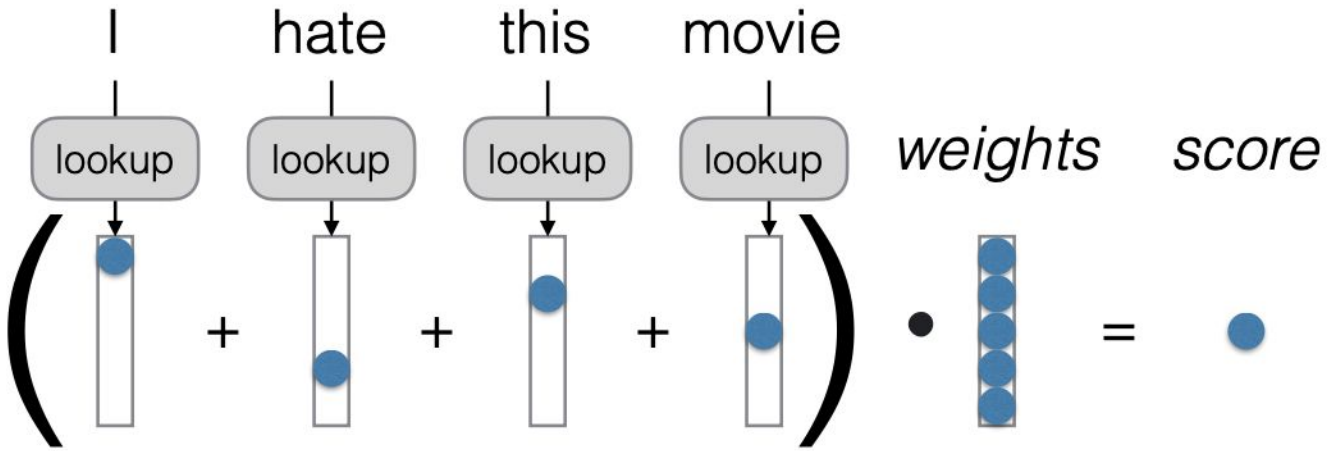
Solution?: Learn Korean?

Machine Learning based NLP

Machine Learning



A First Attempt: Bag of Words (BOW)



Features f are based on word identity, weights w learned

Word Representation – One-hot encoding

Human-Readable

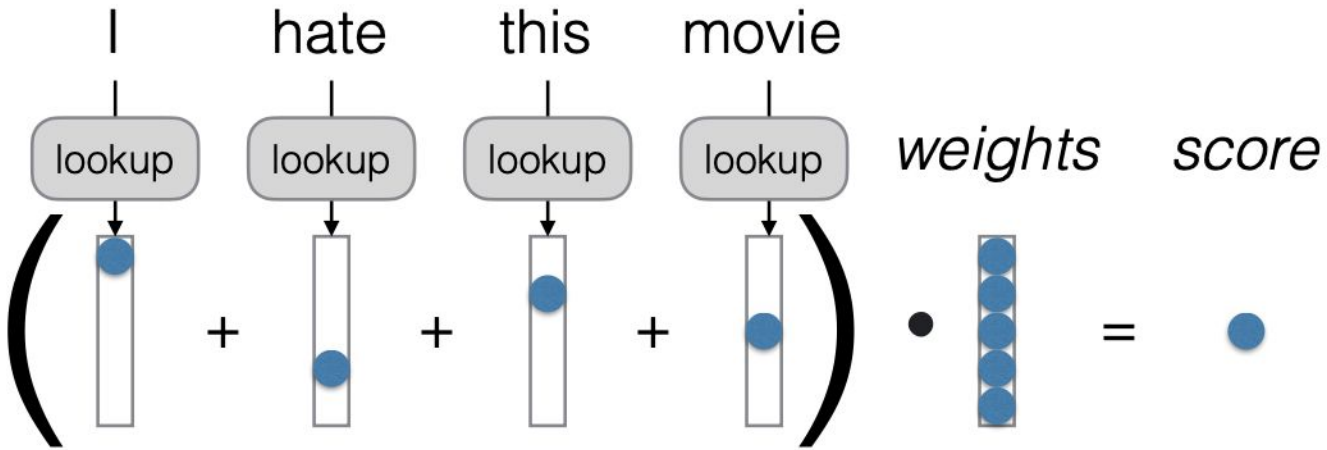
Pet
Cat
Dog
Turtle
Fish
Cat

Machine-Readable

Cat	Dog	Turtle	Fish
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0

Img: https://medium.com/analytics-vidhya/stop-one-hot-encoding-your-categorical-variables-bbb0fba89809

A First Attempt: Bag of Words (BOW)

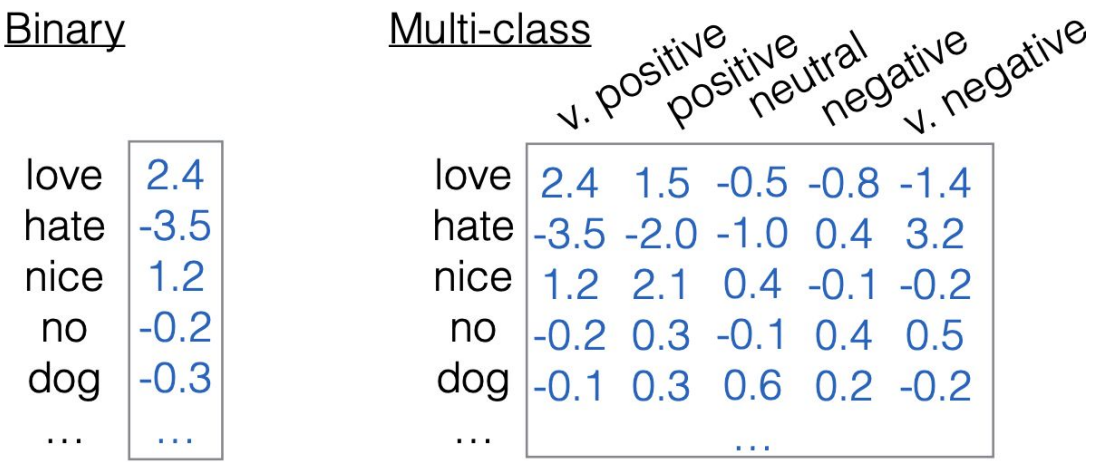


Features f are based on word identity, weights w learned

Which problems mentioned before would this solve?

What do Our Vectors Represent?

- Binary classification: Each word has a single scalar, positive indicating “yes” and negative indicating “no”
- Multi-class classification: Each word has its own 5 elements corresponding to [very good, good, neutral, bad, very bad]



What's missing in BOW?

- Handling of *conjugated* or *compound words*
  - I **love** this movie → I **loved** this movie

Subword Models

- Hanging of word similarity
  - I **love** this movie → I **adore** this movie

Word Embeddings

- Handling of combination features
  - I **love** this movie → I **don't love** this movie
  - I **hate** this movie → I **don't hate** this movie

Neural Networks

- Handing of sentence structure
  - It has an interesting story, **but** is boring overall

Sequence Models

Subword Model (Tokenization)

What's missing in BOW?

- Handling of *conjugated* or *compound words*
  - I **love** this movie → I **loved** this movie

Subword Models

- Hanging of word similarity
  - I **love** this movie → I **adore** this movie

Word Embeddings

- Handling of combination features
  - I **love** this movie → I **don't love** this movie
  - I **hate** this movie → I **don't hate** this movie

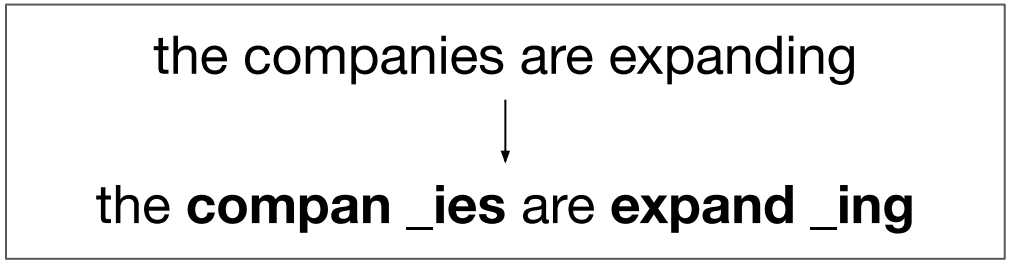
Neural Networks

- Handing of sentence structure
  - It has an interesting story, **but** is boring overall

Sequence Models

Basic Idea

- Split less common words into multiple subword tokens



- Use the data to tell us how to tokenize
- Benefits:
  - **Share parameters** between word variants, compound words
  - Reduce parameter size, **save compute+memory**

Subword Model

- Three common algorithms:
  - **Byte-Pair Encoding (BPE)** (Sennrich et al. 2016)
  - **WordPiece** (Schuster and Nakajima, 2012)
  - **Unigram language modeling tokenization** (Kudo, 2018)
- All have two parts:
  - A token **learner** that takes a raw training corpus and induces a vocabulary (a set of tokens)
  - A token **segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary

Byte Pair Encoding (BPE) token learner (Sennrich et al 2015)

- Used in: a lot of Transformer models
  - e.g., GPT, GPT-2, RoBERTa, BART, and DeBERTa.
- Let vocabulary be the set of all individual characters

  = {A, B, C, D, … , a, b, c, d …}
- Repeat:
  - Choose the two symbols that are most frequently adjacent in the training corpus (say ‘A’. ‘B’)
  - Add a new merged symbol ‘AB’ to the vocabulary
  - Replace every adjacent ‘A’ ‘B’ in the corpus with ‘AB’
- Until *k* merges have been done

Sennrich, R. (2015). Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909.

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

Sennrich, R. (2015). Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909.

Byte Pair Encoding (BPE) Addendum

- Most subword algorithms are run inside space-separated tokens.
- So we commonly first add a special end-of-word symbol ‘__’ before space in training corpus
- Next, separate into letters

BPE Token Learner

- Original corpus:

  low low low low low lowest lowest newer newer newer

  newer newer newer wider wider wider new new
- Add end-of-word tokens, resulting in this vocabulary:
  - __, d, e, i, l, n, o, r, s, t, w
- Corpus representation

  5 l o w __

  2 l o w e s t __

  6 n e w e r __

  3 w i d e r __

  2 n e w __

BPE Token Learner

corpus

5 l o w __
2 l o w e s t __
6 n e w e r __
3 w i d e r __
2 n e w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w

Merge e r to er

corpus

5 l o w __
2 l o w e s t __
6 n e w er __
3 w i d er __
2 n e w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w, er

BPE Token Learner

corpus

5 l o w __
2 l o w e s t __
6 n e w er __
3 w i d er __
2 n e w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w, er

Merge er __ to er__

corpus

5 l o w __
2 l o w e s t __
6 n e w er__
3 w i d er__
2 n e w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w, er, er__

BPE Token Learner

corpus

5 l o w __
2 l o w e s t __
6 n e w er__
3 w i d er__
2 n e w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w, er, er__

Merge n e to ne

corpus

5 l o w __
2 l o w e s t __
6 ne w er__
3 w i d er__
2 ne w __

vocabulary

__, d, e, i, l, n, o, r, s, t, w, er, er__, ne

BPE Token Learner

The next merges are:

Merge	**Current Vocabulary**
(ne, w)	__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new
(l, o)	__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new, lo
(lo, w)	__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new, lo, low
(new, er__)	__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new, lo, low, newer__
(low, __)	__, d, e, i, l, n, o, r, s, t, w, er, er__, ne, new, lo, low, newer__, low__

BPE Token Segmenter Algorithm

- On the test data, run each merge learned from the training data
  - Greedily
  - In the order we learned them
  - (test frequencies don't play a role)
- So, merge every **e r** to **er**, then merge **er** __ to **er**__, etc.
- Result:
  - Test set "n e w e r __" would be tokenized as a full word
  - Test set "l o w e r __" would be two tokens: "low er_"

Properties of BPE tokens

- Usually include frequent words
- And frequent subwords
  - Which are often morphemes like *-est* or *-er*
- A **morpheme** is the smallest meaning-bearing unit of a language
  - *Unlikeliest* has 3 morphemes *un-*, *likely*, and *-est*

WordPiece (Schuster and Nakajima 2012)

- Used in: BERT, DistilBERT, Electra
- Learns merge rules like BPE, but,
  - instead of selecting the most frequent pair, WordPiece computes a score for each pair, using the following formula:

$$\text{score} = (\text{freq\_of\_pair})/(\text{freq\_of\_first\_element} \times \text{freq\_of\_second\_element})$$

    , which evaluates what it loses by merging two symbols to ensure it's worth it
- Saves only the final vocabulary, not merge rules
  - Finds the longest subword that is in the vocabulary, then splits on it.

Schuster, M., & Nakajima, K. (2012). Japanese and korean voice search. In 2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)

Unigram Models (Kudo 2018)

- Used in: AlBERT, T5, mBART, Big Bird, and XLNet.
- Start from a big vocabulary and removes tokens from it until it reaches the desired vocabulary size
- Use a *unigram* LM that generates all words in the sequence independently
- **Pick a vocabulary** that maximizes the log likelihood of the corpus given a fixed vocabulary size
- **Find the segmentation** of the input that maximizes unigram probability

Kudo, T. (2018). Subword regularization: Improving neural network translation models with multiple subword candidates. arXiv preprint arXiv:1804.10959.

SentencePiece

- A highly optimized library that makes it possible to train and use BPE and Unigram models.
- Python bindings also available
  - https://github.com/google/sentencepiece

Kudo, T. (2018). Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. arXiv preprint arXiv:1808.06226.

Comparing Trained LLM Tokenizers

```
text = """

English and CAPITALIZATION

🎵鸟
show_tokens False None elif == >= else: two tabs:" " Three tabs: "   "

12.0*50=600

"""
```

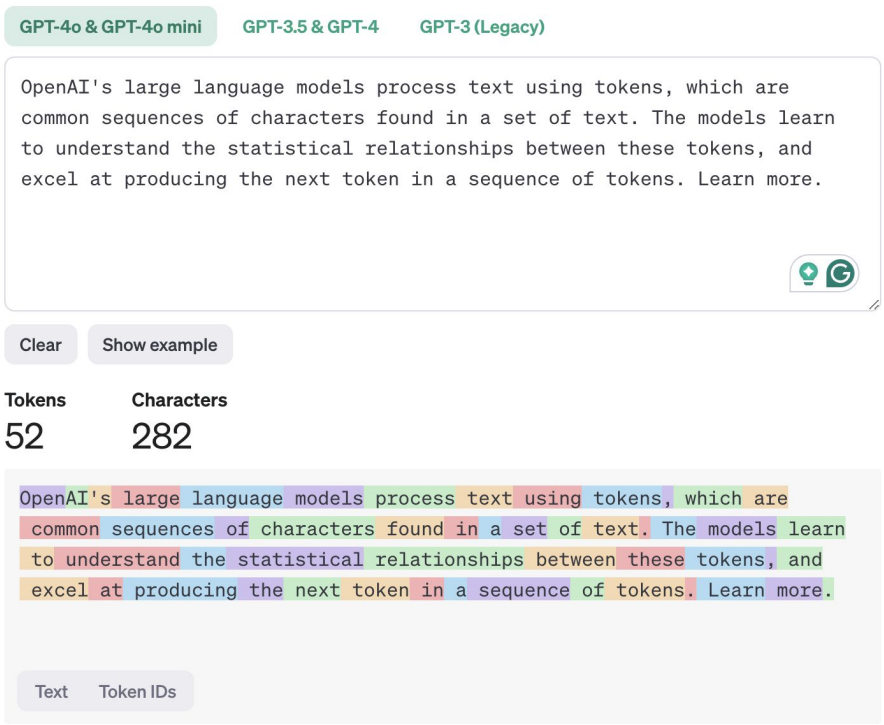
BERT base model (uncased)	[CLS] english and capital ##ization [UNK] [UNK] show _ token ##s false none eli ##f = = > = else : two tab ##s : " " three tab ##s : " " 12 . 0 * 50 = 600 [SEP] *WordPiece*
BERT base model (cased)	[CLS] English and CA ##PI ##TA ##L ##I ##Z ##AT ##ION [UNK] [UNK] show _ token ##s F ##als ##e None el ##if = = > = else : two ta ##bs : " " Three ta ##bs : " " 12 . 0 * 50 = 600 [SEP] *WordPiece*
GPT-2	English and CAP ITAL IZ ATION 🎵鸟 show _ t ok ens False None el if == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600 *BPE*
FLAN-T5	English and CA PI TAL IZ ATION <unk> <unk> show _ to ken s Fal s e None e l if = = > = else : two tab s : " " Three tab s : " " 12. 0 * 50 = 600 </s> *SentencePiece*
GPT-4	English and CAPITAL IZATION 🎵鸟 show _tokens False None elif == >= else : two tabs :" " Three tabs : " " 12 . 0 * 50 = 600 *BPE*
Phi-3 and Llama 2	<s> English and C AP IT AL IZ ATION 🎵鸟 show _ to kens False None elif == >= else : two tabs :" " Three tabs : " " 1 2 . 0 * 5 0 = 6 0 0 *BPE*

Img: Hands on Large Language Models

Subword

- **Multilinguality**: Subword models are hard to use multilingually because they will over-segment less common languages naively
  - Work-around: Upsample less represented languages

Multilinguality



https://platform.openai.com/tokenizer

Continuous Word Embeddings

What's missing in BOW?

- Handling of *conjugated* or *compound words*
  - I **love** this movie → I **loved** this movie

Subword Models

- Hanging of word similarity
  - I **love** this movie → I **adore** this movie

Word Embeddings

- Handling of combination features
  - I **love** this movie → I **don't love** this movie
  - I **hate** this movie → I **don't hate** this movie

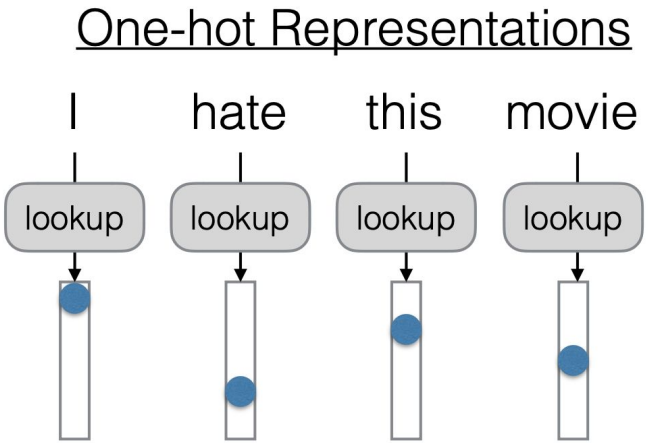
Neural Networks

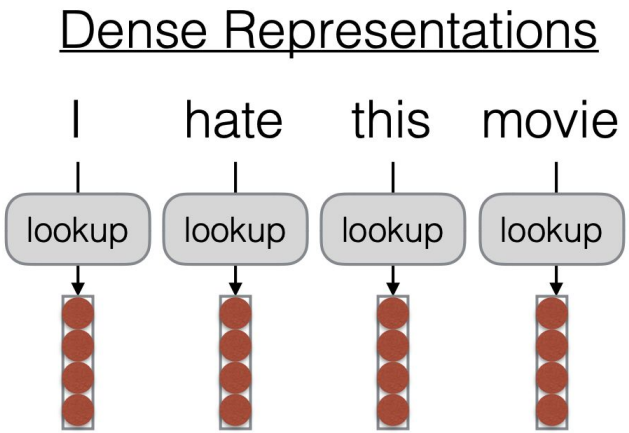
- Handing of sentence structure
  - It has an interesting story, **but** is boring overall

Sequence Models

Basic Idea

- Previously we represented words with a *sparse* vector with a single "1" — a **one-hot** vector
- Continuous word embeddings look up a *dense* vector



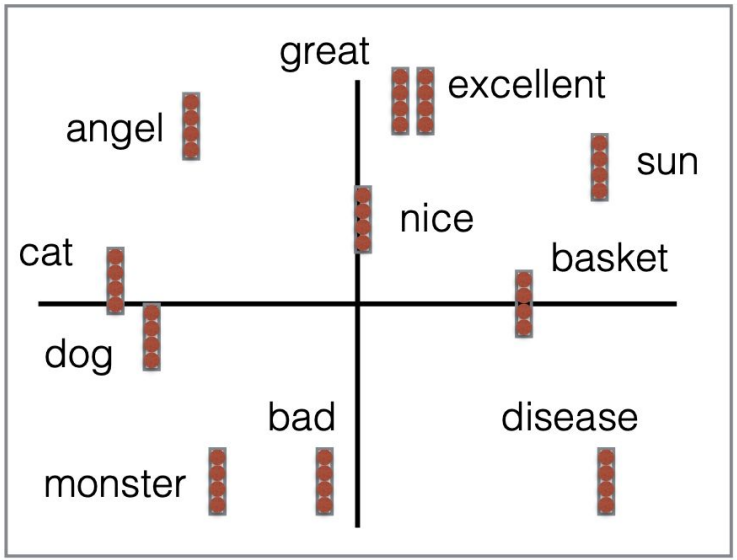


Why dense vector?

- Dense vectors may **generalize** better than explicit counts
- Dense vectors may do better at capturing **synonym**
- Consider sentiment analysis:
  - With words, a feature is a word identity
    - Feature 5: ‘The previous word was “terrible”’
    - requires exact same word to be in training and test
  - With embeddings:
    - Feature is a word vector
    - ‘The previous word was vector [35, 22, 17, …]
    - Now in the test set we might see a similar vector [24, 21, 15]
    - We can generalize to similar but unseen words.

What do Our Vectors Represent?

- No guarantees, but we hope that:
  - Words that are **similar** (syntactically, semantically, same language, etc.) are close in vector space
  - Each vector element is a **features** (e.g. is this an animate object? is this a positive word, etc.)



Shown in 2D, but in reality we use 512, 1024D, etc

Common methods for getting short dense vectors

- “Neural Language Model” – inspired models
  - Word2vec (skipgram, Continuous Bag of Words (CBOW)), GloVe
- Singular Value Decomposition (SVD)
  - A special case of this is called LSA — Latent Semantic Analysis
- Alternative to these “static embeddings”:
  - Contextual Embeddings (ELMo, BERT)

Word2vec

- Train a classifier on a binary prediction task:
  - Is *w* likely to show up near "*apricot*"?
- We don't actually care about this task
  - But we'll take the learned classifier weights as the word embeddings
- Big ideas: **self-supervision**:
  - A word *c* that occurs near 'apricot' in the corpus is treated as the gold "correct answer" for supervised learning
  - No need for human labels
  - Bengio et al. (2003); Collobert et al. (2011)

Approach: predict if candidate word *c* is a “neighbor”

1. Treat the target word *t* and a neighboring context word *c* as **positive examples**
2. Randomly sample other words in the lexicon to get **negative examples**
3. Use logistic regression to train a classifier to distinguish those two cases
4. Use the learned weights as the embeddings

Skip-Gram Classifier

(assuming a +/= 2 word window)

… lemon, a [tablespoon of apricot jam, a] pinch

c1 c2 [target] c3 c4

Goal: train a classifier that is given a candidate (word, context) pair

(apricot, jam)

(apricot, aardvark)

…

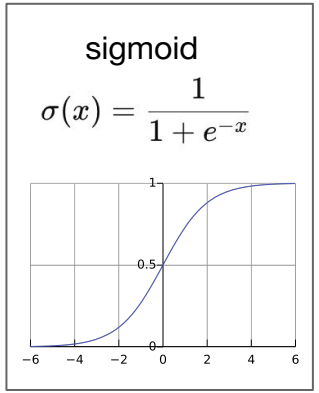
And assigns each pair a probability:

$P(+ \mid w, c)$

$P(- \mid w, c) = 1 - P(+ \mid w, c)$

Turning vector similarity into probabilities

- The intuition of the skip-gram model is to base *P*(+/- | *w, c*) on the embedding similarity.
- Two vectors are similar if they have a high doc product.
  - Similarity (w, c) ~ w · c
- To turn this into a probability, we use the sigmoid:



$$P(+ \mid w, c) = \sigma(c \cdot w) = \frac{1}{1 + \exp(-c \cdot w)}$$

$$P(- \mid w, c) = 1 - P(+ \mid w, c) = \sigma(-c \cdot w) = \frac{1}{1 + \exp(c \cdot w)}$$

How Skip-Gram Classifier computes *P*(+ | *w, c*)

$$P(+ \mid w, c) = \sigma(c \cdot w) = \frac{1}{1 + \exp(-c \cdot w)}$$

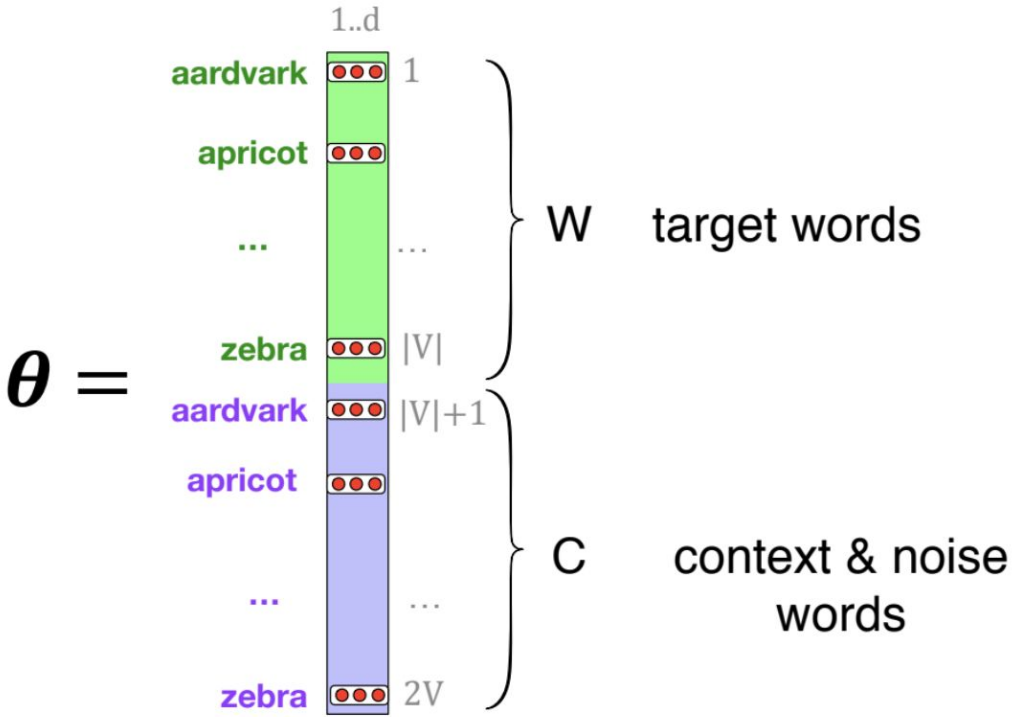
This is for one context word, but we have lots of context words.

We'll assume independence and just multiply them.

$$P(+ \mid w, c_{1:L}) = \prod_{i=1}^{L} \sigma(c_i \cdot w)$$

$$\log P(+ \mid w, c_{1:L}) = \sum_{i=1}^{L} \log \sigma(c_i \cdot w)$$

These embeddings we'll need: a set for w, a set for c



Skip-Gram Training data

… lemon, a [tablespoon of apricot jam, a] pinch

c1 c2 [target] c3 c4

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

t	c	t	c
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

For each positive example, we will grab k negative examples, sampling by frequency

Word2vec: how to learn vectors

- Given the set of positive and negative training instances, and an initial set of embedding vectors
- The goal of learning is to adjust those word vectors such that we:
  - **Maximize** the similarity of the target word, context word pairs (w, c_pos) drawn from the positive data.
  - **Minimize** the similarity of the (w, c_neg) pairs drawn from the negative data.

Loss function for one w with c_{pos}, c_{neg1}, … c_{negk}

- Maximize the similarity of the target with the actual context words, and minimize the similarity of the target with the k negative sampled non-neighbor words.

$$\mathcal{L}(w) = -\log\left[P(+ \mid w, c_{\text{pos}}) \prod_{i=1}^{k} P(- \mid w, c_{\text{neg}_i})\right]$$

$$= -\left[\log P(+ \mid w, c_{\text{pos}}) + \sum_{i=1}^{k} \log P(- \mid w, c_{\text{neg}_i})\right]$$

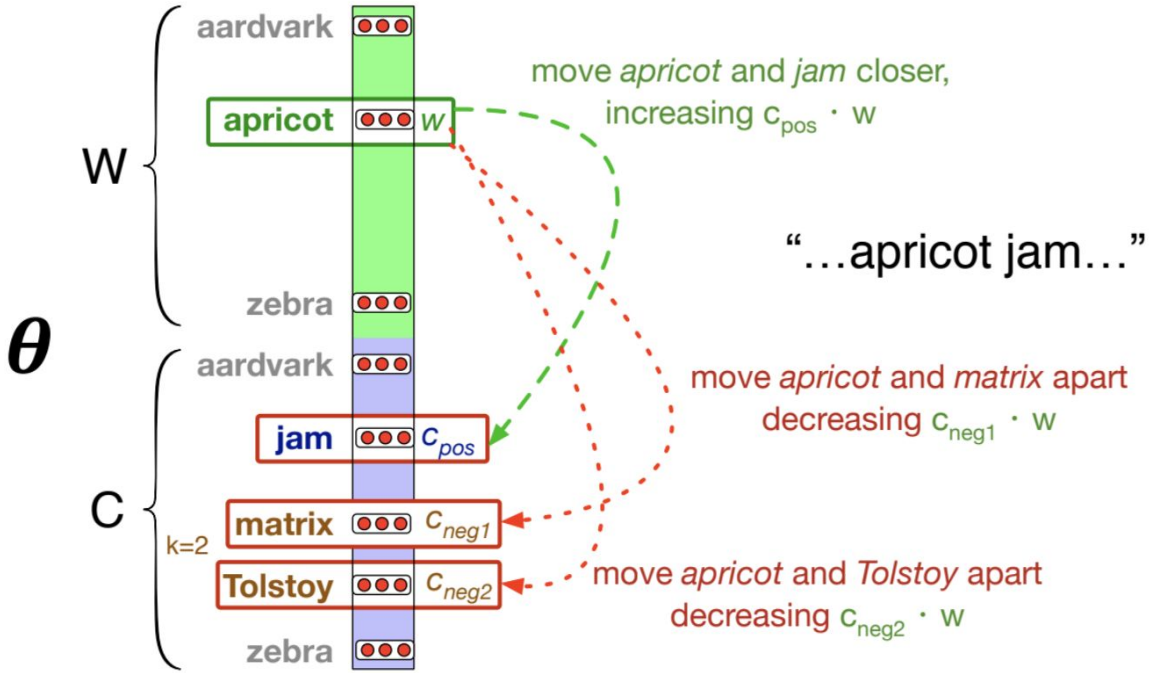
$$= -\left[\log \sigma(c_{\text{pos}} \cdot w) + \sum_{i=1}^{k} \log\left(1 - \sigma(c_{\text{neg}_i} \cdot w)\right)\right]$$

$$= -\left[\log \sigma(c_{\text{pos}} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{\text{neg}_i} \cdot w)\right]$$

Learning the classifier

- How to learn?
  - Stochastic gradient descent!
- We'll adjust the word weights to
  - make the positive pairs more likely
  - and the negative pairs less likely
  - over the entire training set

Intuition of one step of gradient descent



Gradient descent

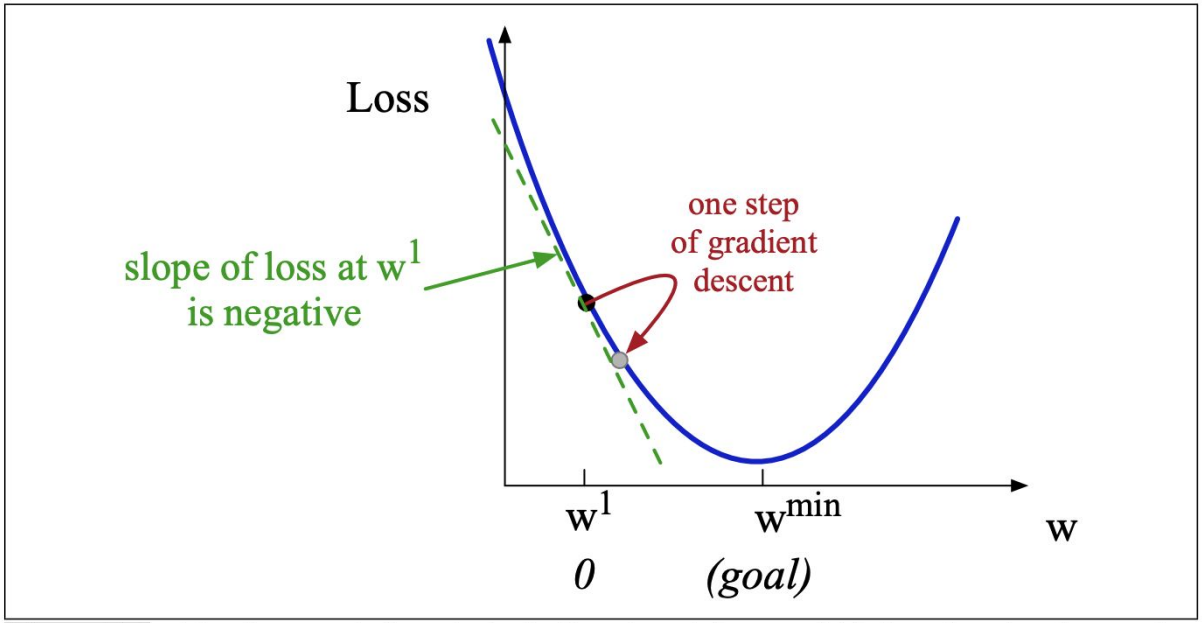
- At each step
  - Direction: We move in the reverse direction from the gradient of the loss function
  - Magnitude: we move the value of the gradient weighted by a **learning rate η**
  - Higher learning rate means move w faster

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \cdot \frac{d}{d\mathbf{w}} L(f(\mathbf{x}; \mathbf{w}), y)$$

Where:

- $\mathbf{w}^{(t)}$: Parameters (e.g., embeddings) at iteration t.
- $\mathbf{w}^{(t+1)}$: Updated parameters after the gradient step.
- η: Learning rate.
- $\frac{d}{d\mathbf{w}} L()$: Gradient of the loss function L with respect to $\mathbf{w}$.

Gradient descent intuition



The derivatives of the loss function

$$\mathcal{L}(w) = -\left[\log \sigma(c_{\text{pos}} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{\text{neg}_i} \cdot w)\right]$$

$$\frac{\partial \mathcal{L}(w)}{\partial c_{\text{pos}}} = \big(\sigma(c_{\text{pos}} \cdot w) - 1\big) w$$

$$\frac{\partial \mathcal{L}(w)}{\partial c_{\text{neg}_i}} = \sigma(c_{\text{neg}_i} \cdot w) w$$

$$\frac{\partial \mathcal{L}(w)}{\partial w} = (\sigma(c_{\text{pos}} \cdot w) - 1) c_{\text{pos}} + \sum_{i=1}^{k} \sigma(c_{\text{neg}_i} \cdot w) c_{\text{neg}_i}$$

Update equation with SGD

Start with randomly initialized C and W matrics, then incrementally do updates

$$\mathbf{c}_{\text{pos}_t}^{(t+1)} = \mathbf{c}_{\text{pos}_t}^{(t)} - \eta\big(\sigma(\mathbf{c}_{\text{pos}_t} \cdot \mathbf{w}_t) - 1\big)\mathbf{w}_t$$

$$\mathbf{c}_{\text{neg}_t}^{(t+1)} = \mathbf{c}_{\text{neg}_t}^{(t)} - \eta\sigma(\mathbf{c}_{\text{neg}_t} \cdot \mathbf{w}_t)\mathbf{w}_t$$

$$\mathbf{w}_t^{(t+1)} = \mathbf{w}_t^{(t)} - \eta\left[\sum_{c \in \text{Context}(w_t)} (\sigma(\mathbf{c} \cdot \mathbf{w}_t) - 1)\mathbf{c} + \sum_{c' \in \text{NegativeSamples}(w_t)} \sigma(\mathbf{c}' \cdot \mathbf{w}_t)\mathbf{c}'\right]$$

Two sets of embeddings

- SGNS learns two sets of embeddings
  - Target embeddings matrix W
  - Context embeddings matrix C
- It's common to just add them together, representing word *i* as the vector $w_i + c_i$

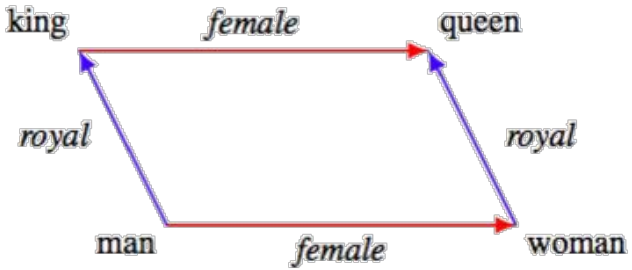
Properties of Embeddings

Analogical relations

Analogical relations are identified using the parallelogram method with word embeddings (Mikolov et al. 2013b).

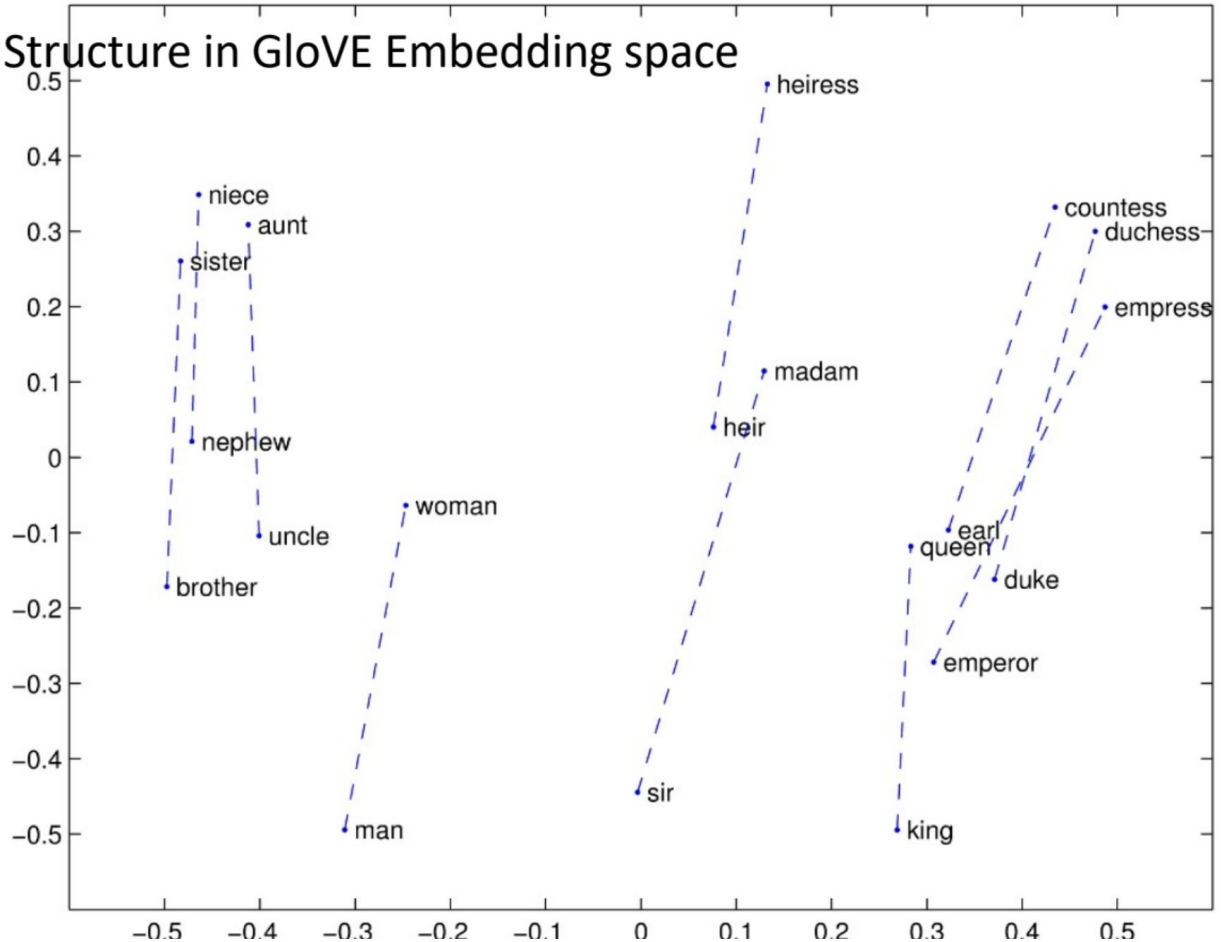
$$\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$$

$$\vec{\text{Paris}} - \vec{\text{France}} + \vec{\text{Italy}} \approx \vec{\text{Rome}}$$



Img: https://kawine.github.io/blog/nlp/2019/06/21/word-analogies.html

Structure in GloVE Embedding space



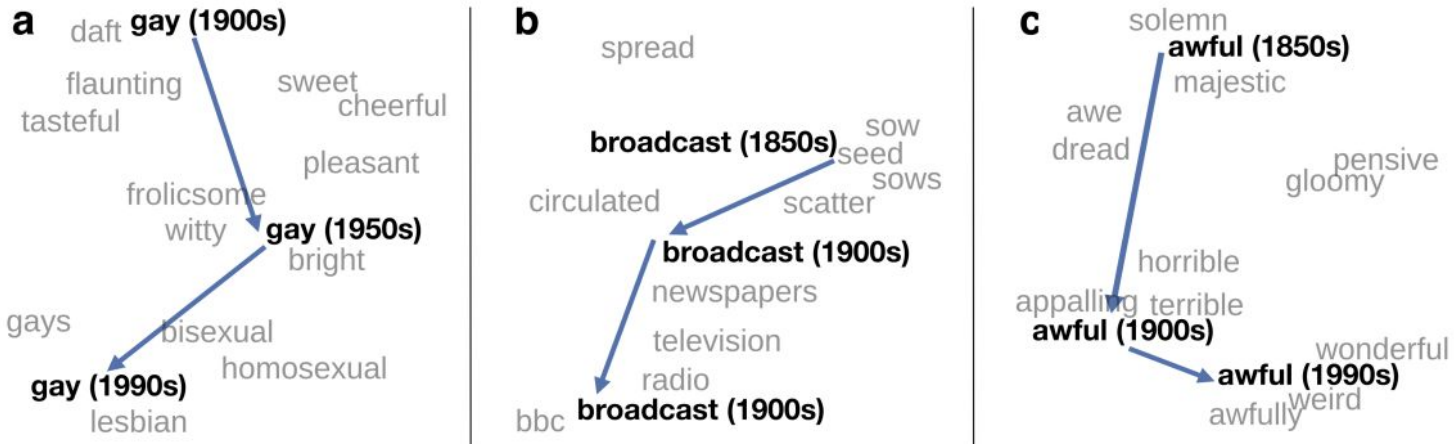
Caveats with the parallelogram method

- It only seems to work for frequent words, small distances and certain relations (relating countries to capitals, or parts of speech), but not others (Linzen 2016, Gladkova et al. 2016, Ethayarajh et al. 2019a).

- Understanding analogy is an open area of research (Peterson et al. 2020).

Embeddings as a window onto historical semantics

- Train embeddings on different decades of historical text to see meanings shift: ~30 million books, 1850-1990, Google Books data



Hamilton, et al 2016. https://aclanthology.org/P16-1141/

Embeddings reflect cultural bias!

Ask “Paris : France :: Tokyo : x”

- x = Japan

Ask “father : doctor :: mother : x”

- x = nurse

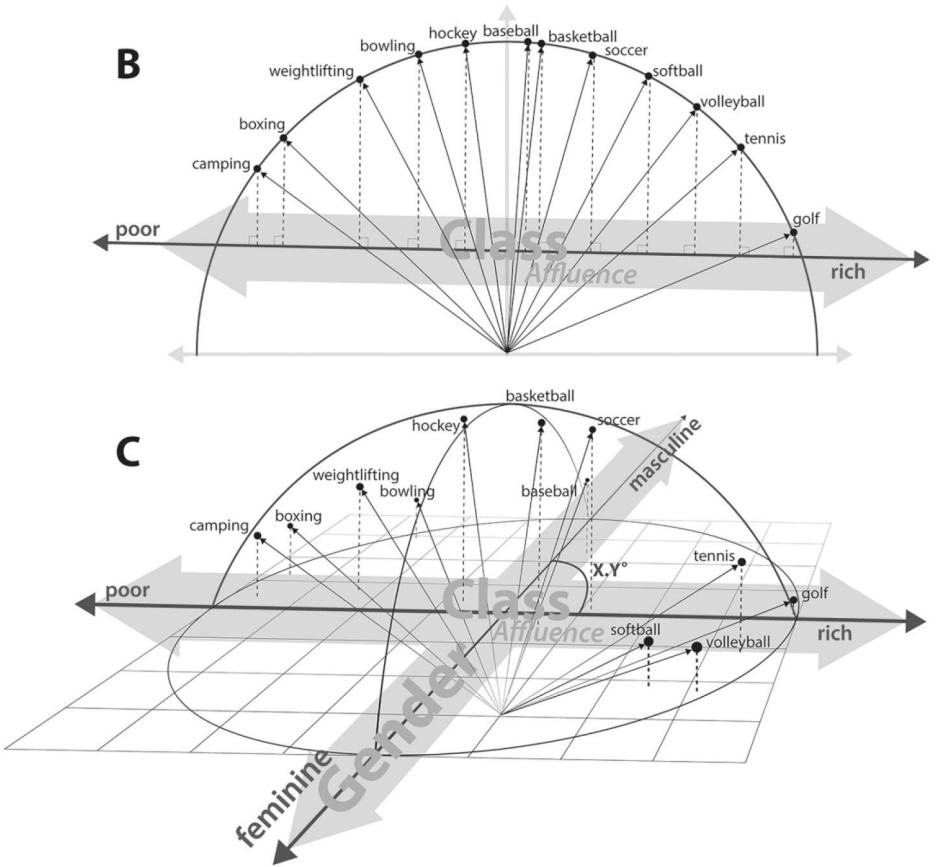
Ask “man : computer programmer :: woman : x”

- x = homemaker

Algorithms that use embeddings as part of e.g., hiring searches for programmers, might lead to bias in hiring

Bolukbasi et at 2016. https://arxiv.org/abs/1607.06520

Meanings of Class through Word Embeddings

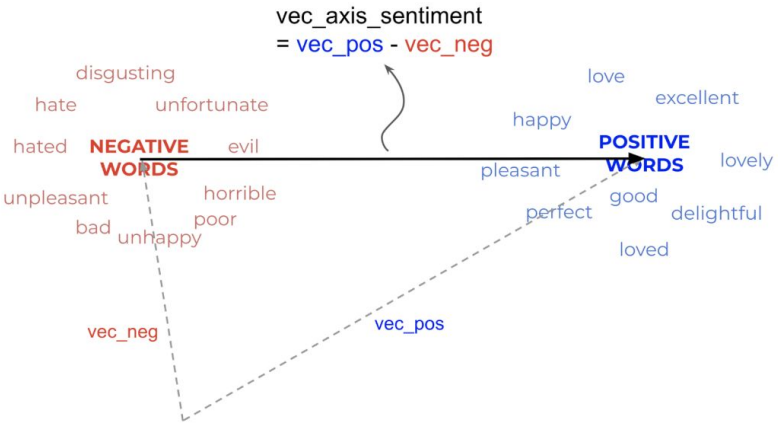


Kozlowski et al (2019) https://journals.sagepub.com/doi/full/10.1177/0003122419877135

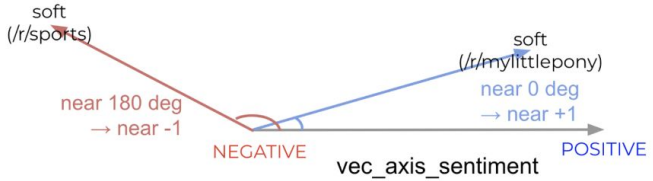
SemAxis for Analyzing Domain-specific Word Semantics

soft in /r/mylittlepony vs soft in /r/sports vs

Step 1: Define a semantic axis



Step 2. Locate words on the semantic axis



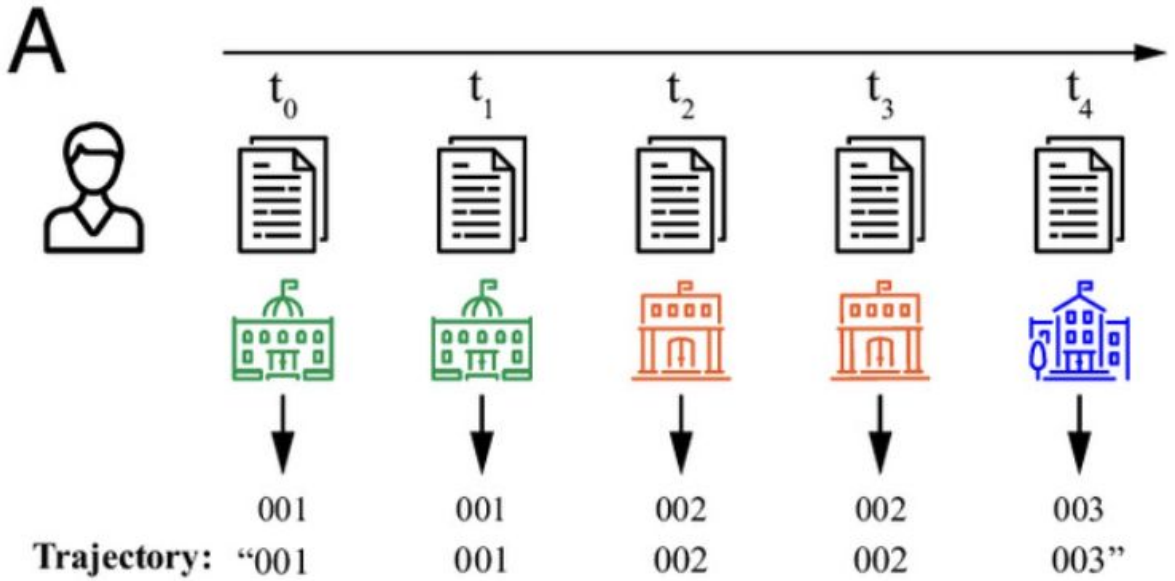
Compute a cosine similarity between a word vector and axis vector

An et al 2018. https://aclanthology.org/P18-1228/

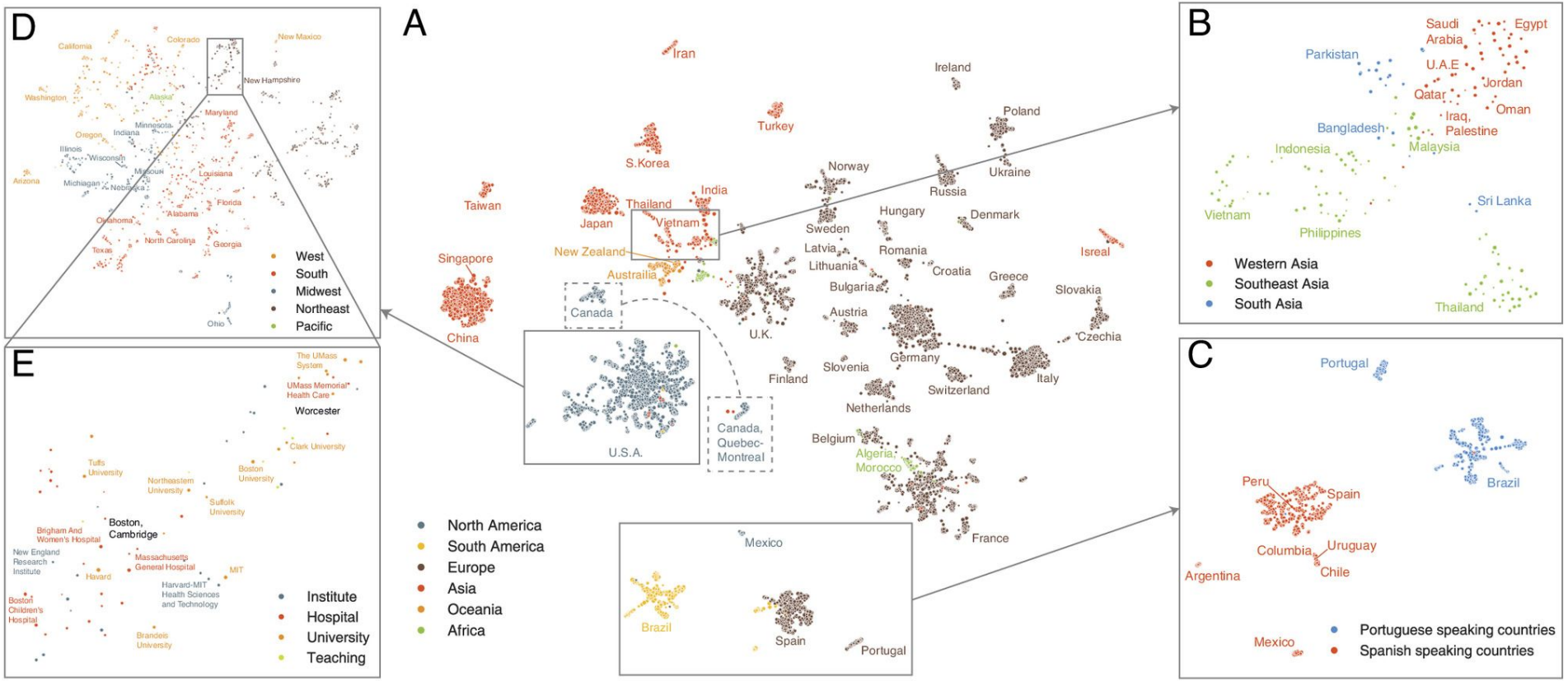
Building a Recommendation System using Word2vec

https://www.analyticsvidhya.com/blog/2019/07/how-to-build-recommendation-system-word2vec-python/

Embeddings for Understanding Scientific migration

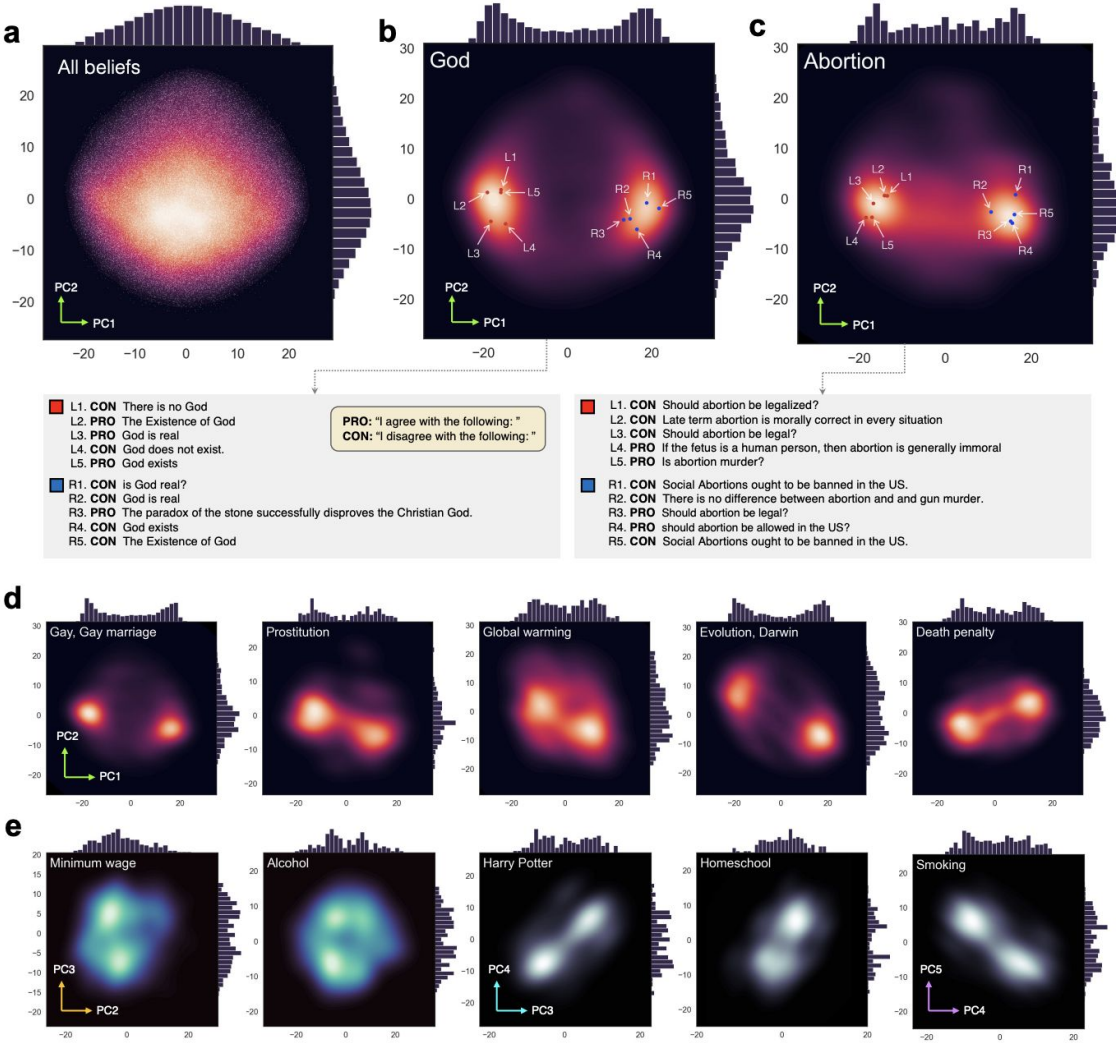


Murray et al (2023). https://www.pnas.org/doi/full/10.1073/pnas.2305414120



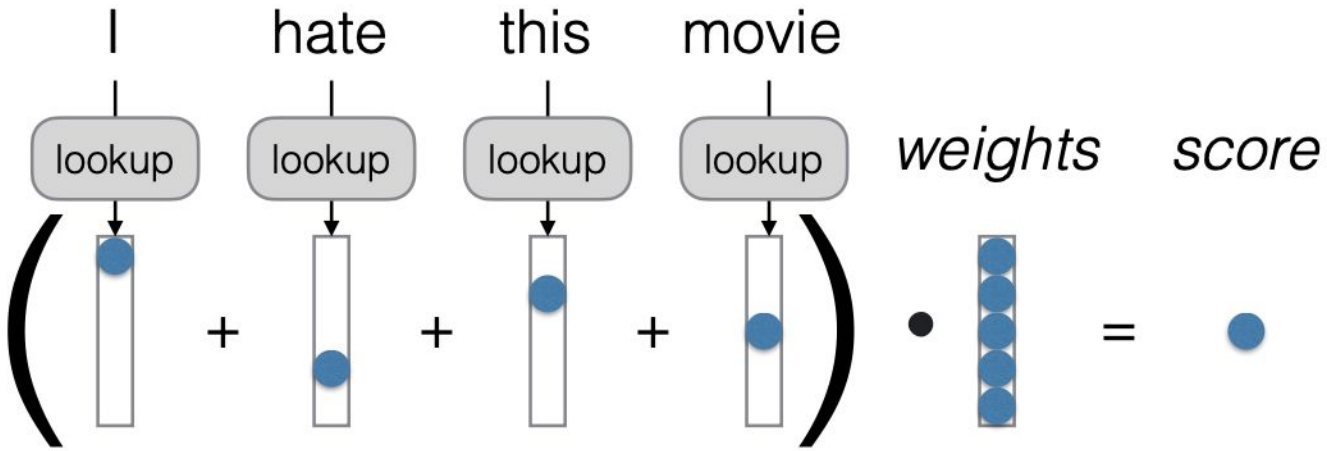
Murray et al (2023). https://www.pnas.org/doi/full/10.1073/pnas.2305414120

Belief Embeddings



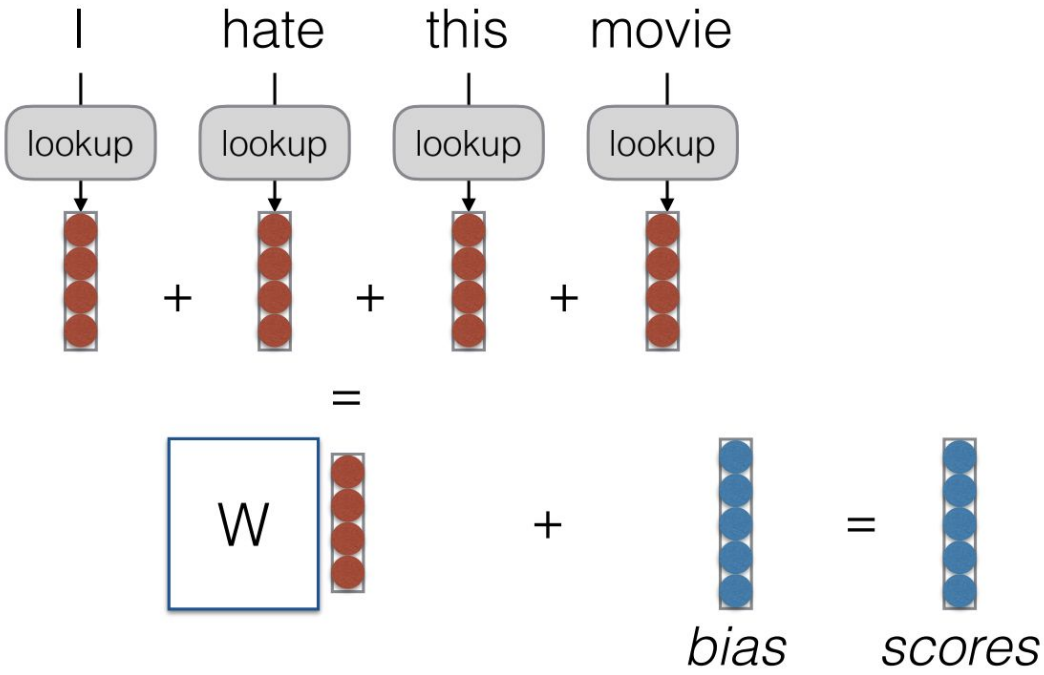
Lee et al. (2024) https://arxiv.org/abs/2408.07237

A First Attempt: Bag of Words (BOW)



Features f are based on word identity, weights w learned

Continuous Bag of Words (CBOW)



Combination Features

What's missing in BOW?

- Handling of *conjugated* or *compound words*
  - I **love** this movie → I **loved** this movie

Subword Models

- Hanging of word similarity
  - I **love** this movie → I **adore** this movie

Word Embeddings

- Handling of combination features
  - I **love** this movie → I **don't love** this movie
  - I **hate** this movie → I **don't hate** this movie

Neural Networks

- Handing of sentence structure
  - It has an interesting story, **but** is boring overall

Sequence Models

source: CMU CS 11-711, Advanced NLP

Combination Features

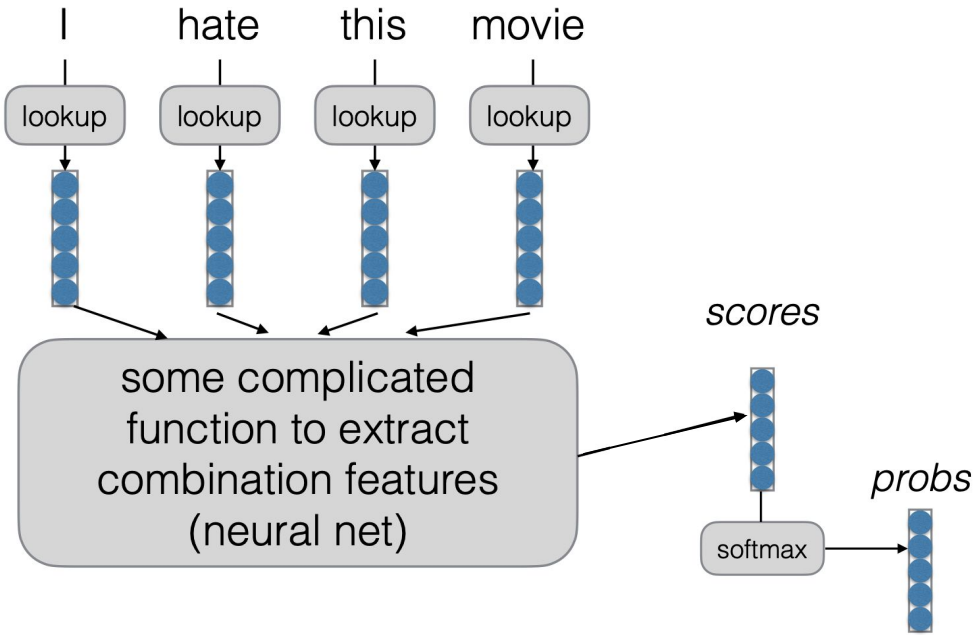
I don't love this movie

very good
good
neutral
bad
very bad

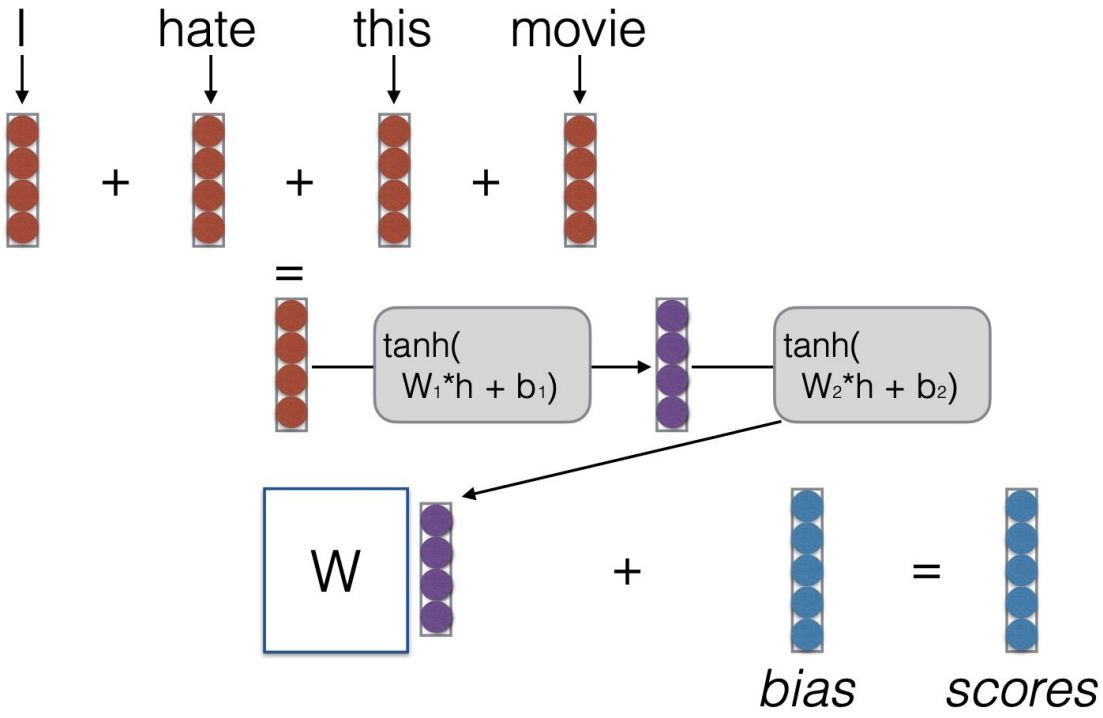
There's nothing I don't love about this movie

very good
good
neutral
bad
very bad

Basic Idea of Neural Networks (for NLP Prediction Tasks)



Deep CBOW

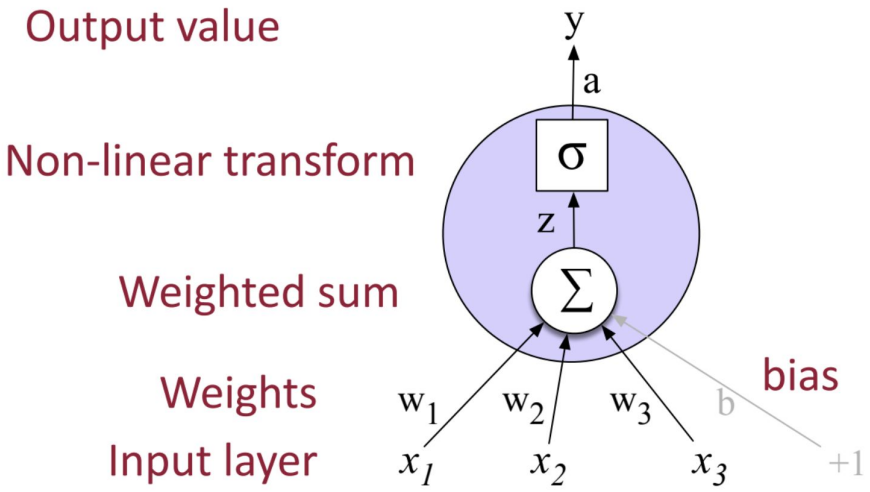


What do Our Vectors Represent?

- Now things are more interesting!
- We can learn feature combinations (a node in the second layer might be “feature 1 AND feature 5 are active”)
- e.g. capture things such as “not” AND “hate”

What is a Neural Net?

Neural Network Unit



- Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

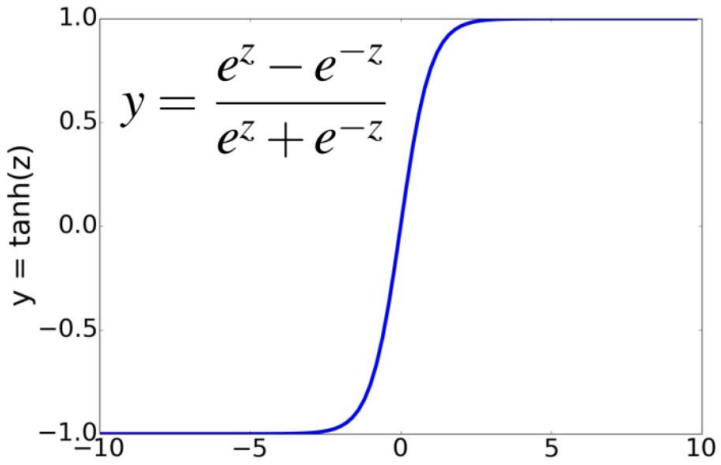
- Apply a nonlinear activation function f:

$$y = a = f(z)$$

- If the non-linear activation function is sigmoid:

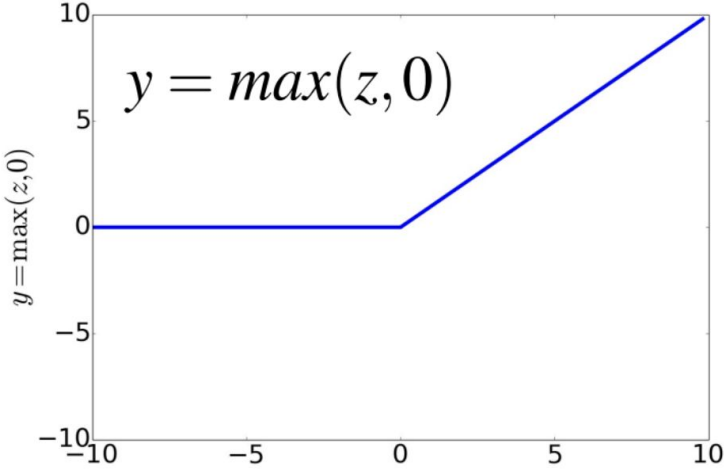
$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Non-Linear Activation Functions besides sigmoid



tanh

Most Common:

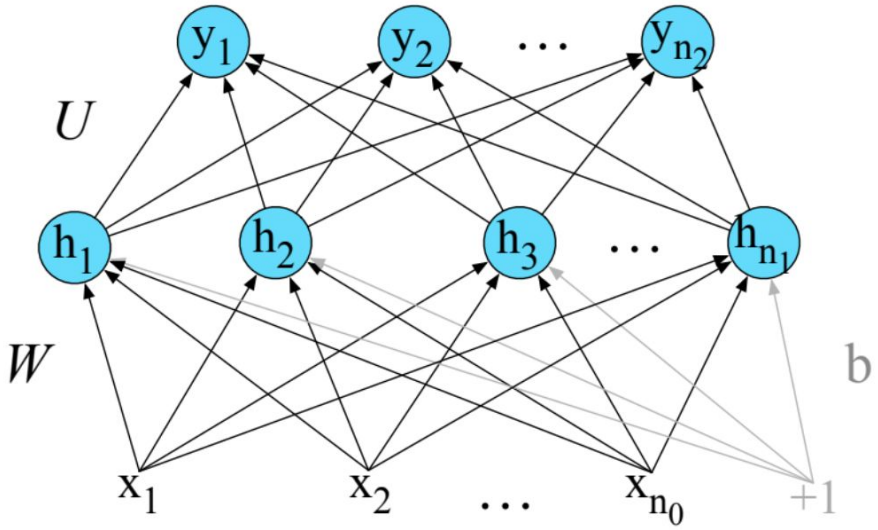


ReLU

Rectified Linear Unit

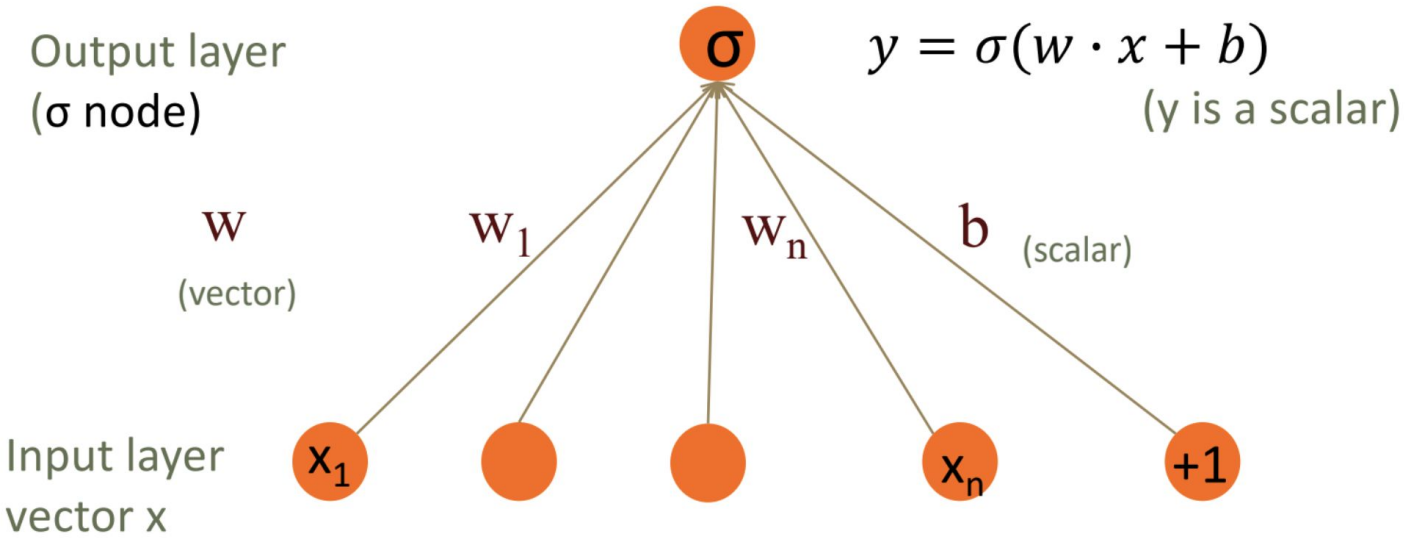
Feedforward Neural Networks

Can also be called multi-layer perceptrons (or MLPs) for historical reasons



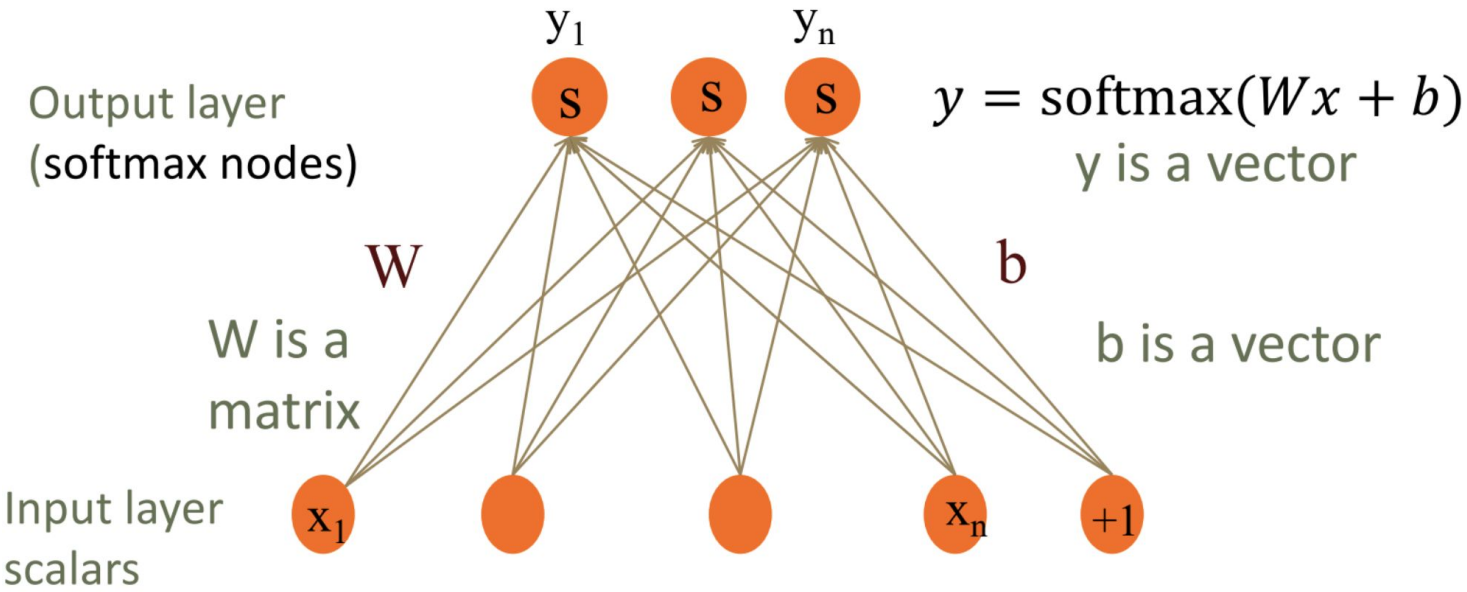
Binary Logistic Regression as a 1-layer Network

(we don't count the input layer in counting layers!)



Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



Softmax: a generalization of sigmoid

- For a vector z of dimensionality k, the softmax is

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \leq i \leq k$$

- Example

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

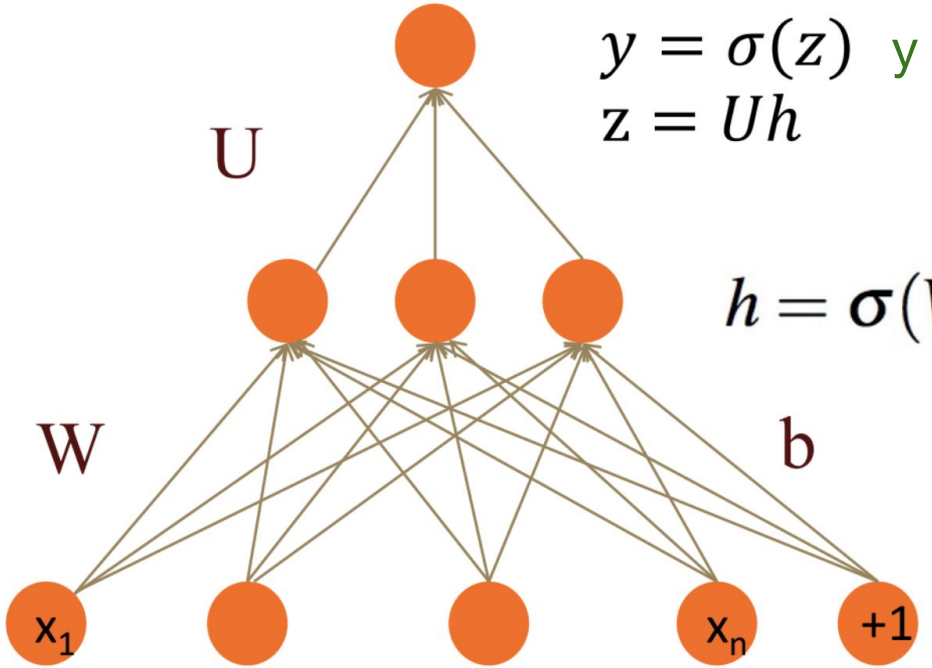
$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



$$y = \sigma(z)$$
$$z = Uh$$

y is a scalar

$$h = \sigma(Wx + b)$$

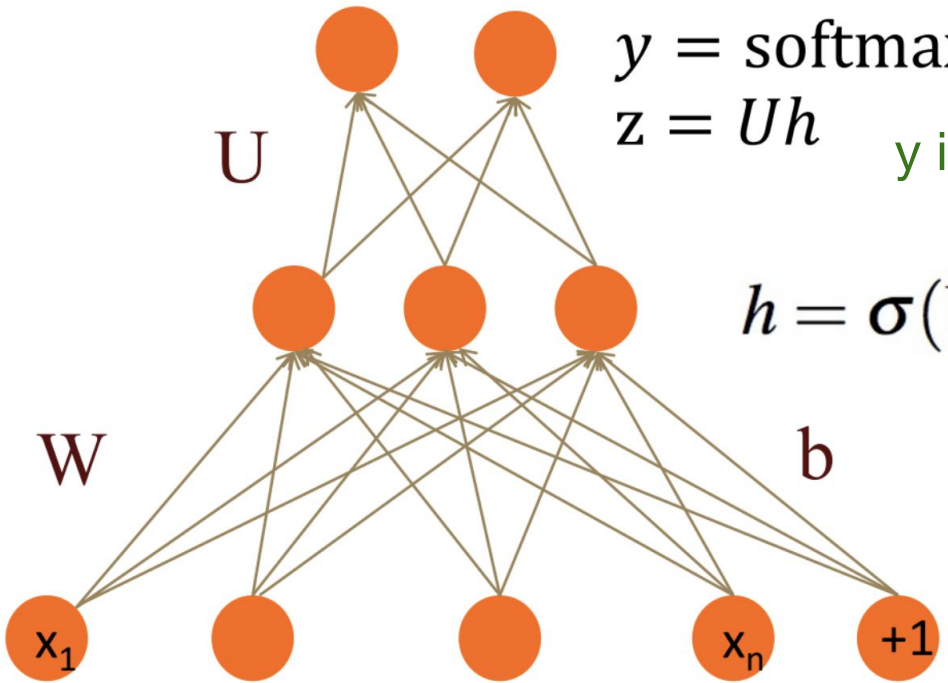
Could be ReLU
Or tanh

Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



$$y = \text{softmax}(z)$$
$$z = Uh$$

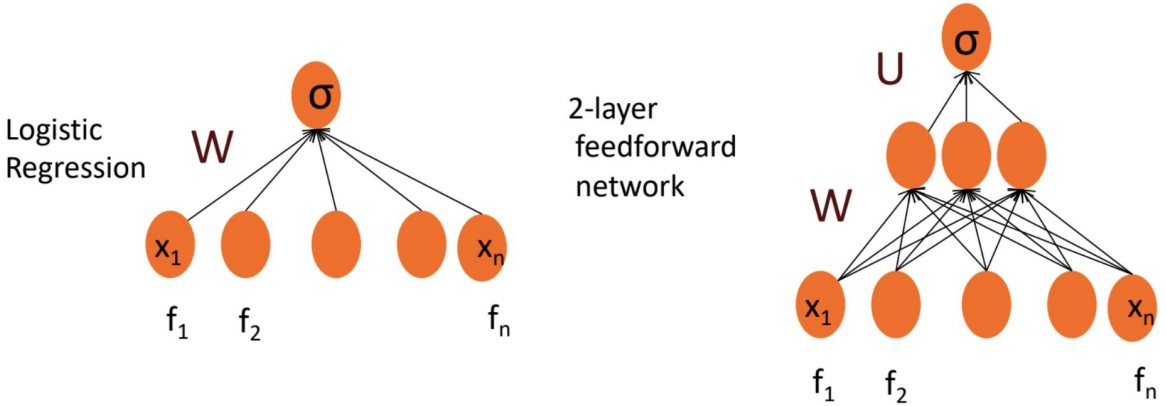
y is a vector

$$h = \sigma(Wx + b)$$

Could be ReLU
Or tanh

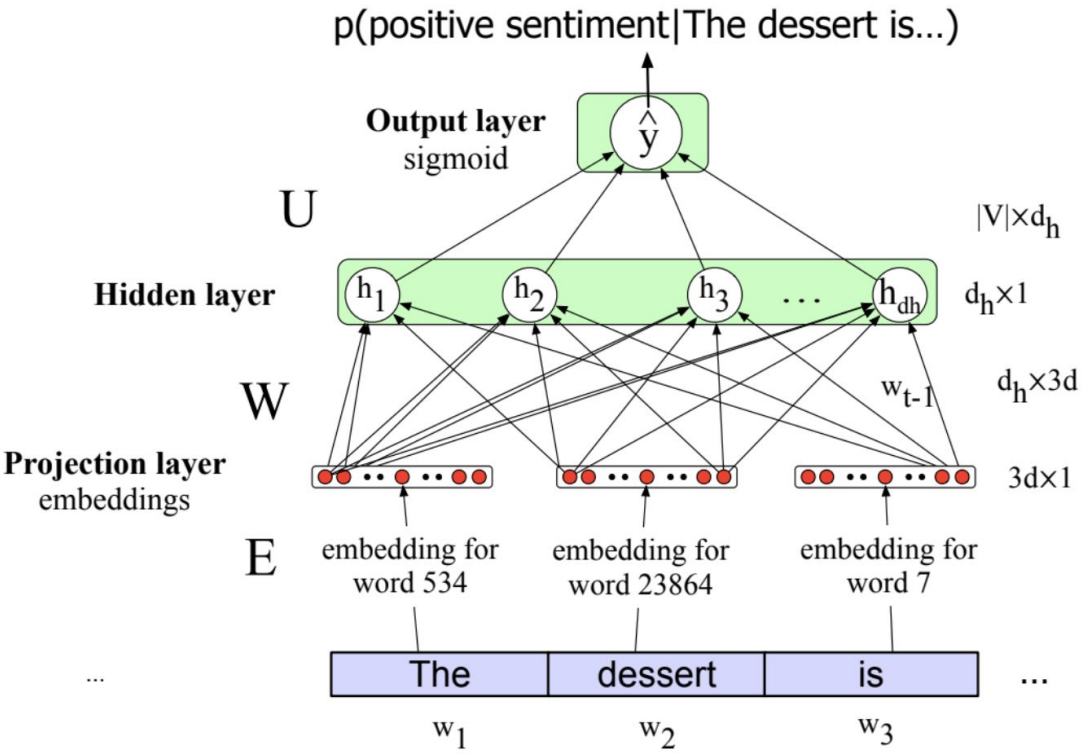
Feedforward nets for Sentiment Classification

- Input layer are features, and output layer is 0 or 1

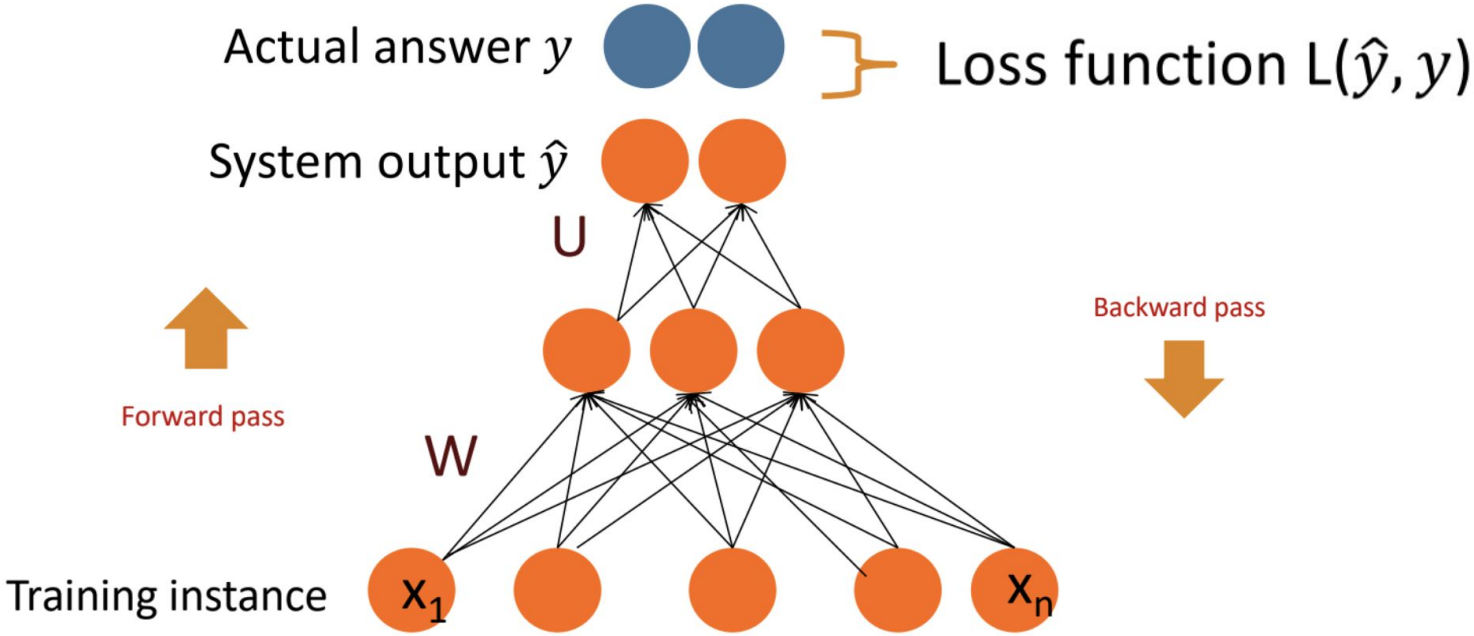


- Just adding a hidden layer to logistic regression allows the network to use non-linear interactions between features
- The real power of deep learning comes from the ability to learn features from the data

Neural Net Classification with embeddings as input features!



Intuition: Training a 2-layer Network



Intuition: Training a 2-layer Network

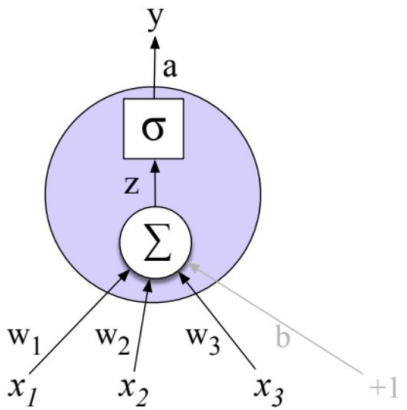
For every training tuple (*x, y*)

- Run *forward* computation to find our estimate *y_hat*
- Run *backward* computation to update weights:
  - For every output node
    - Compute loss *L* between true *y* and the estimated *y_hat*
  - For every weight *w* from hidden layer to the output layer
    - Update the weight
  - For every hidden node
    - Assess how much blame it deserves for the current answer
    - For every weight *w* from input layer to the hidden layer
      - Update the weight

Where did that derivative come from?

- Using the chain rule!

$$f(x) = u(v(x)) \qquad \frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$



Derivative of the weighted sum

Derivative of the Activation

Derivative of the Loss

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i}$$

- How can I find that gradient for every weight in the network?

Backpropagation

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!
- Solution: error backpropagation (Rumelhart, Hinton, Williams, 1986)
- Backprop is a special case of backward differentiation which relies on **computation graphs**.

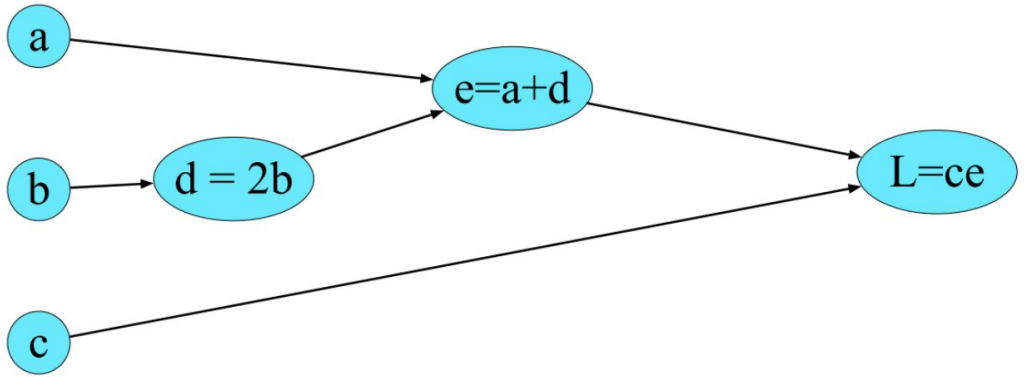
Computation Graphs

- A computation graph represents the process of computing a mathematical expression

Example: $L(a,b,c) = c(a+2b)$

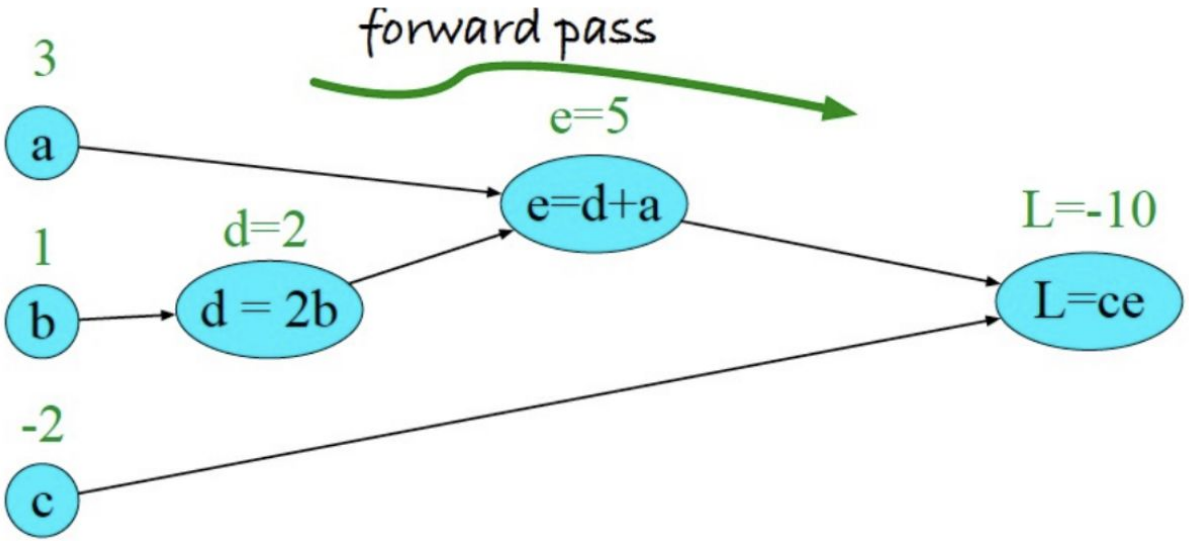
Computations:

$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$



Example: Forward Pass

$$L(a,b,c) = c(a+2b)$$



Backwards differentiation in computation graphs

Example $L(a,b,c) = c(a+2b)$

$$
\begin{aligned}
d &= 2*b \\
e &= a+d \\
L &= c*e
\end{aligned}
$$

We want: $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$

$$
\begin{aligned}
\frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}
\end{aligned}
$$

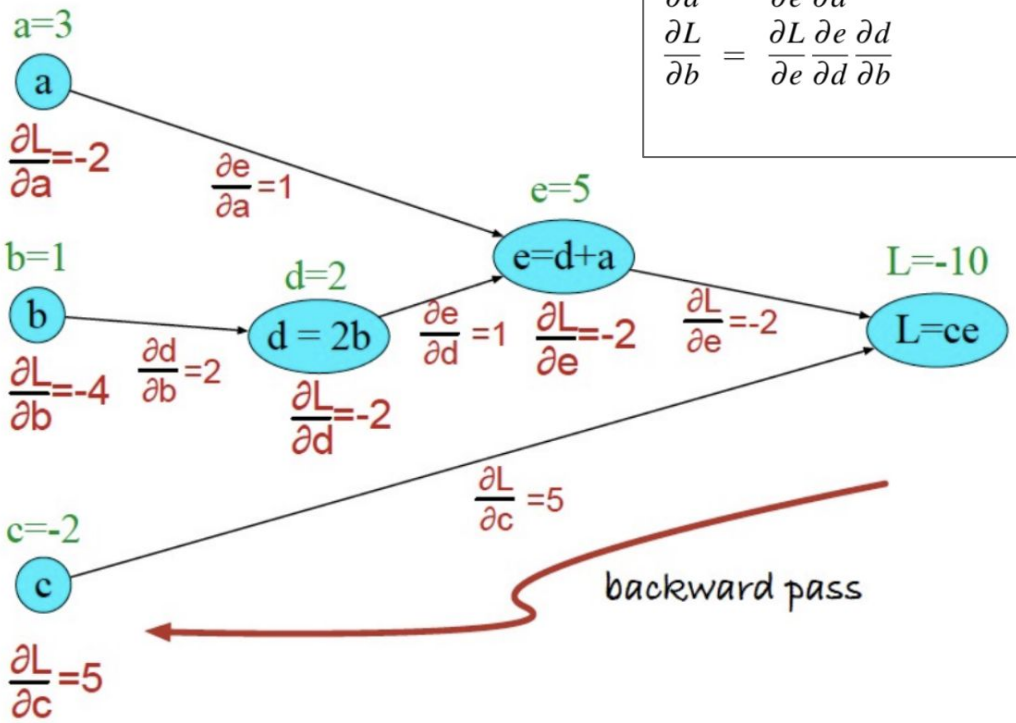
$$
\begin{aligned}
L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\
e = a+d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\
d = 2b &: \quad \frac{\partial d}{\partial b} = 2
\end{aligned}
$$

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in *a* affects *L*.

Example: Backward Pass

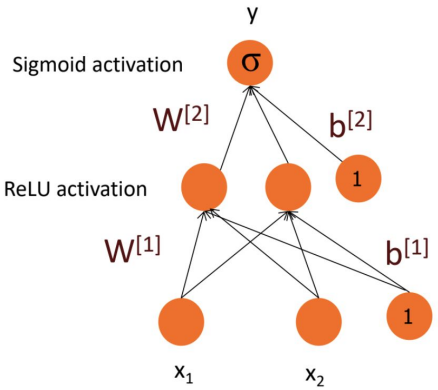
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$
$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$
$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$

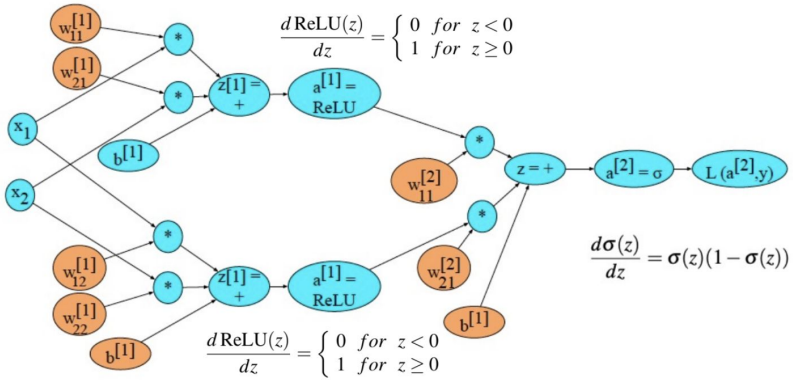


Backpropagation for a deeper network is harder!

- Backward differentiation on a two layer network



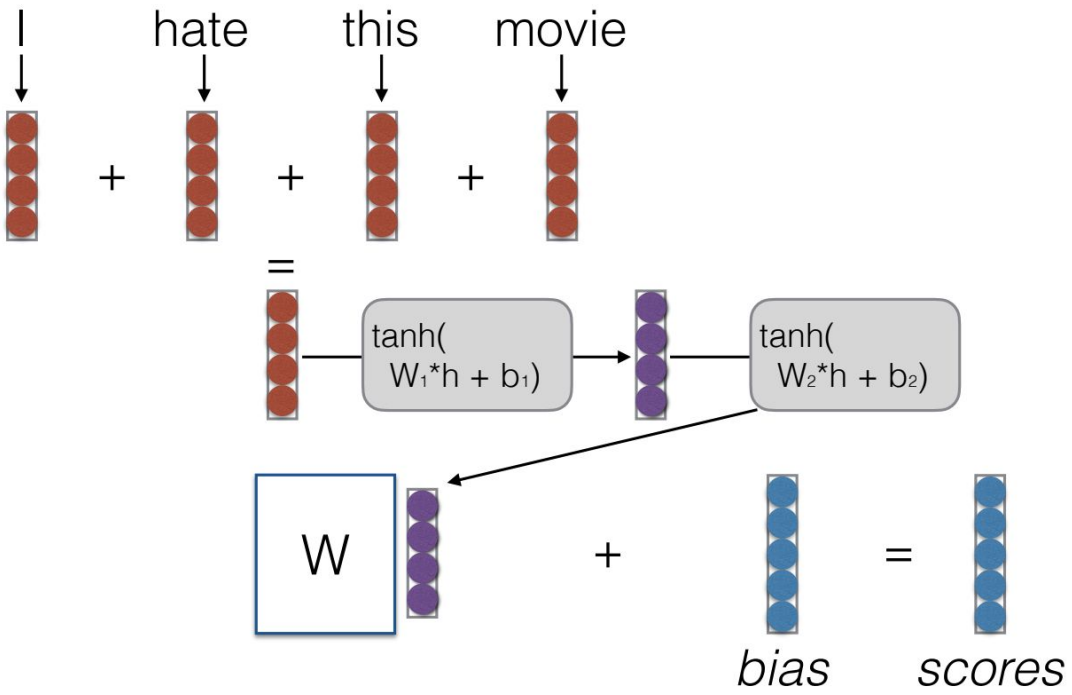
$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



Basic Process in Neural Network Frameworks

- Create a model
- For each example
  - **create a graph** that represents the computation you want
  - **calculate the result** of that computation
  - if training, perform **back propagation and update**

Deep CBOW



One More Important Concept

A Better Optimizer: Adam (Adaptive Moment Estimation)

- Most standard optimization option in NLP and beyond
- Considers momentum, moving average of gradient, and adaptive learning rate

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$ 1st momentum (direction)

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$ 2nd momentum (strength)

- Correction of bias early in training $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

- Final update

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

Any Questions?