WEB INTELLIGENCE

Lecture 6: Language Modeling II

Russa Biswas

October 21, 2025

AALBORG UNIVERSITY

(Based on slides of Dan Jurafsky, Andrew Ng)

Language Modeling - Recap

Probability of a sentence = Probability of a sequence of the words in the sentence

$$P(S) = P(w_1 w_2 \ldots w_n)$$

Chain Rule

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

Markov Assumption

$$P(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx P(w_i \mid w_{i-1})$$

N-gram model

$$P(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

Maximum Likelihood Estimation

How to evaluate a Language Model?

Two ways to evaluate: **Extrinsic Evaluation** and **Intrinsic Evaluation**

- **Extrinsic evaluation** captures how useful the model is in a particular task (also known as **in-vivo** evaluation)
  - To compare models A and B
    - Put each model in a real task (such as Machine Translation, speech recognition, etc.)
    - Run the task, get a score for A and for B (such as How many words translated correctly, How many words transcribed correctly etc.)
    - Compare accuracy for A and B
- Limitations:
  - Expensive, time-consuming
  - Doesn't always generalize to other applications

How to evaluate a Language Model?

Intrinsic evaluation captures how well the model captures what it is supposed to capture (also known as in-vitro evaluation): ***Perplexity***

- **Perplexity**
  - Directly measures language model performance at predicting words.
  - Doesn't necessarily correspond with real application performance
  - But gives us a single general metric for language models
  - Useful for large language models (LLMs) as well as n-grams

How to evaluate a Language Model?

Divide the data/corpus into **training** set and **test** set (disjoint sets)

We train parameters of our model on a **training set**.

We test the model's performance on data we haven't seen.

- A **test set** is an unseen dataset; different from training set.
  - Intuition: we want to measure generalization to unseen data
- An **evaluation metric** (like **perplexity**) tells us how well our model does on the test set.
- If we're building an LM for a specific task
  - The test set should reflect the task language we want to use the model for (domain specific – medical, legal, etc.)
- If we're building a general-purpose model
  - We'll need lots of different kinds of training data (general data covering all domains)
  - We don't want the training set or the test set to be just from one domain or author or language.

How to evaluate a Language Model?

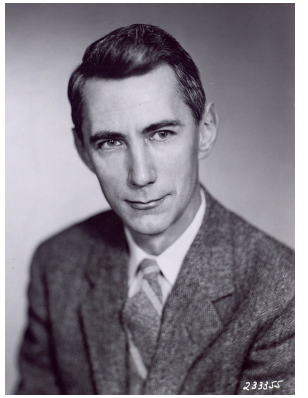
- ***Training on the Test set***: If sentences from the test set is present on the training data
  - Issues:
    - LM will assign that sentence an artificially high probability when we see it in the test set
    - hence assign the whole test set a falsely high probability.
    - the LM would look better than it really is
- Sentences from test set **should not be** in the training set
- Dev Set:
  - If we test on the test set many times we might implicitly tune to its characteristics (overfit)
  - Noticing which changes make the model better.
  - Therefore, we need a third dataset: **development test set** or **devset** or **validation set.**
    - We test our LM on the devset until the very end to adjust the LM
    - And then test our LM on the **test set** once

Perplexity

Intuition: A good LM prefers "real" sentences
- Assign higher probability to “real” or “frequently observed” sentences
- Assigns lower probability to “word salad” or “rarely observed” sentences?

The Shannon Game: **How well can we predict the next word**?
- Once upon a ____
- That is a picture of a ____

time 0.9
dream 0.03
midnight 0.02
…
and 1e-100



Claude Shannon

- Unigrams are terrible at this game (Why?)
- A good LM is one that assigns a higher probability to the next word that actually occurs

Picture credit: Historiska bildsamlingen
https://creativecommons.org/licenses/by/2.0/

Perplexity

The best LM assigns high probability to the entire test set

When comparing two LMs, A and B

- We compute P_A(test set) and P_B(test set)
- The better LM will give a higher probability to the test set than the other LM.
- Probability depends on size of test set
  - Probability gets smaller the longer the text
  - Better: a metric that is **per-word**, normalized by length
- **Perplexity is the inverse probability of the test set,**

normalized by the number of words

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Perplexity

- The inverse comes from the original definition of perplexity from cross-entropy rate in information theory)
- Probability range is [0,1], perplexity range is [1,∞]
- **Minimizing perplexity is the same as maximizing probability**

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Chain rule: $$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

Bigrams: $$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

Perplexity – Weighted Average Branching Factor

- The branching factor of a language is the number of possible next words that can follow any word
- The Perplexity (PP) is referred to as the **weighted average branching factor** and is the most common evaluation metric for N-gram language models

Example: Deterministic language L = {red, blue, green}

Branching factor = 3 (any word can be followed by red, blue, green)

Now assume LM **A** where each word follows any other word with **equal probability ⅓**

Given a test set T = "red red red red blue"

$Perplexity_A(T) = P_A(\text{red red red red blue})^{-1/5} = ((⅓)^5)^{-1/5} = (⅓)^{-1} = 3$

- But now suppose red was very likely in training set, such that for LM **B**: **P(red) = 0.8 p(green) = 0.1 p(blue) = 0.1**
- We would expect the probability to be higher, and hence the perplexity to be smaller:

$Perplexity_B(T) = P_B(\text{red red red red blue})^{-1/5} = (0.8 * 0.8 * 0.8 * 0.8 * 0.1)^{-1/5} = 0.04096^{-1/5} = 0.527^{-1} = 1.89$

Perplexity

Wall Street Journal (19,979 word vocabulary)

- Training set: 38 million words
- Test set: 1.5 million words

	Unigram	Bigram	Trigram
Perplexity	962	170	109

Lower perplexity = better language model

Since language model probabilities are very small, multiplying them together often yields to underflow. It is often better to use logarithms instead, so

$$PP(W) = P(w_1 w_2 ... w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}}$$

$$PP(S) = \exp\left(-\frac{1}{n}\sum_{i=1}^{n} \log_e P(w_i | w_1, ..., w_{i-1})\right)$$

Evaluation: Accuracy

How to objectively measure the quality of a (classification) experiment?

- Compare your achieved results with a ground truth (gold standard).
- How to achieve a ground truth?
  - Crowd sourcing
- How to compare achieved results with a ground truth?

Scenario:

Binary Classification task: Goal is to predict whether an email is **Spam** (Positive) or **Not Spam** (Negative).

Train set: 10,000 emails, test set: 100 emails and the results are as follows:

Evaluation: Accuracy

Scenario:

Binary Classification task: Goal is to predict whether an email is **Spam** (Positive) or **Not Spam** (Negative).

Train set: 10,000 emails, test set: 100 emails and the results are as follows:

The number of spam emails correctly predicted as spam - 40

The number of not spam emails correctly predicted as not spam - 45

The number of not spam emails incorrectly predicted as spam - 5 Type I error

The number of spam emails incorrectly predicted as not spam - 10 Type II error

	Predicted: Spam	Predicted: Not Spam
Actual/Ground Truth: Spam	40 True Positives (TP)	10 False Negatives (FN)
Actual/Ground Truth: Not Spam	5 False Positives (FP)	45 True Negatives (TN)

Evaluation: Accuracy

	Predicted: Spam	Predicted: Not Spam
Actual/Ground Truth: Spam	40 True Positives (TP)	10 False Negatives (FN)
Actual/Ground Truth: Not Spam	5 False Positives (FP)	45 True Negatives (TN)

Accuracy: Proportion of correctly classified instances out of all instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$= \frac{40 + 45}{40 + 10 + 5 + 45} \quad = \frac{85}{100} \quad = 0.85$$

Evaluation: Precision and Recall

	Predicted: Spam	Predicted: Not Spam
Actual/Ground Truth: Spam	40 True Positives (TP)	10 False Negatives (FN)
Actual/Ground Truth: Not Spam	5 False Positives (FP)	45 True Negatives (TN)

Precision is the **fraction of retrieved instances that are relevant**

$$precision = \frac{true\,positive}{true\,positive + false\,positive}$$

$$Precision = \frac{40}{40+45} = \frac{40}{85} = 0.89$$

Recall is the **fraction of relevant instances that are retrieved/predicted**

$$recall = \frac{true\,positive}{true\,positive + false\,negative}$$

$$Recall = \frac{40}{40+10} = \frac{40}{50} = 0.80$$

Evaluation: F1 score

	Predicted: Spam	Predicted: Not Spam
Actual/Ground Truth: Spam	40 True Positives (TP)	10 False Negatives (FN)
Actual/Ground Truth: Not Spam	5 False Positives (FP)	45 True Negatives (TN)

F1-score is the harmonic mean of precision and recall, and provides a balance between the two.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

$$= 2 * \frac{0.89 * 0.80}{0.89 + 0.80} = 0.84$$

Evaluation

	Predicted:Positive	Predicted: Negative
Actual/Ground Truth: Positive	30 True Positives (TP)	20 False Negatives (FN)
Actual/Ground Truth: Negative	10 False Positives (FP)	40 True Negatives (TN)

Compute Accuracy, Precision, Recall and F1 score

One-hot encoding

One-hot encoding is a method of representing data as binary vectors

Vocabulary: 10,000 unique words

Size of each word vector = 10,000

$$\boldsymbol{O}_{man} = [00000 \ldots\ldots 000 \ldots 1 \ldots 000 \ldots 000000000] = \boldsymbol{O}_{2345}$$
$$\boldsymbol{O}_{woman} = [00000 \ldots\ldots 000 \ldots 001 \ldots\ldots 000000000] = \boldsymbol{O}_{3679}$$
$$\boldsymbol{O}_{king} = [00000 \ldots\ldots 001 \ldots 000 \ldots 000 \ldots 000000000] = \boldsymbol{O}_{356}$$
$$\boldsymbol{O}_{queen} = [00000 \ldots\ldots 000 \ldots 000 \ldots 001 \ldots 000000000] = \boldsymbol{O}_{1892}$$
$$\boldsymbol{O}_{orange} = [00000 \ldots\ldots 000 \ldots 1 \ldots 000 \ldots 000000000] = \boldsymbol{O}_{489}$$
$$\boldsymbol{O}_{apple} = [00000 \ldots\ldots 000 \ldots 1 \ldots 000 \ldots 000000000] = \boldsymbol{O}_{4432}$$

I want a glass of apple juice

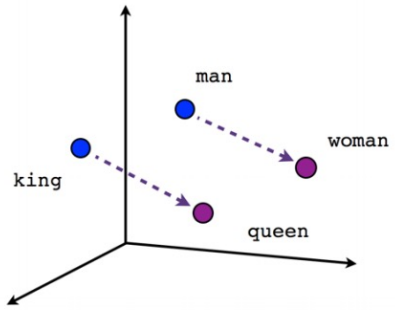
I want a glass of orange _____

One hot encoding

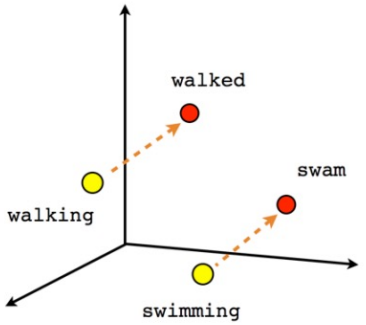
	Man	Woman	King	Queen	Apple	Orange
Gender	-1	1	-0.95	0.95	0.00	0.01
Royal	0.01	0.02	0.93	0.96	-0.01	0.00
Food	-0.001	0.001	-0.001	0.001	0.95	0.98
---size, color, alive, age, cost, etc.						

Features: 300

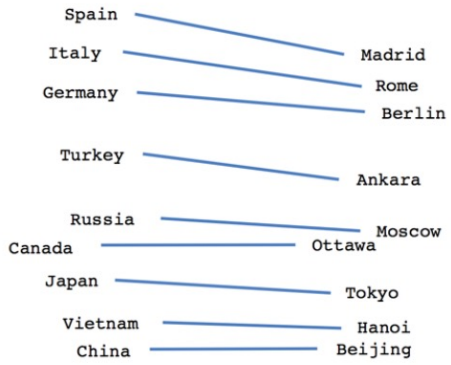
I want a glass of apple **juice**
I want a glass of orange **juice**



Male-Female



Verb tense



Country-Capital

Embedding Matrix

10000

300

w1	w2	.....	wn
0.1	-0.4	.....	0.8
-0.5	0.5	.....	-0.1
0.2	-0.9	.....	0.02
...	...	.....	....
0.1	0.2	.....	-0.9

$$\boldsymbol{O}_{orange} = [00000 \ldots \ldots .000 \ldots .1 \ldots .000 \ldots .000000000] = \boldsymbol{O}_{489}$$

Size of Matrix E is 300 x 10000

Size of $\boldsymbol{O}_{orange}$ is 10000 x 1

Matrix multiplication of E and O = embedding vector

Word Embeddings

One way to define "usage": words are defined by their environments (the words around them)

Zellig Harris (1954):

- **If A and B have almost identical environments we say that they are synonyms**.
- We define a word as a vector
- Called an "embedding" because it's embedded into a space: standard way to represent meaning in NLP
- **Every modern NLP algorithm uses embeddings as the representation of word meaning**
- Fine-grained model of meaning for similarity

Word Embeddings – Why vectors?

- 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 [34,21,14]
    - We can generalize to **similar but unseen** words!!!
- tf-idf
  - Information Retrieval!
  - A common baseline model
  - **Sparse** vectors
  - Words are represented by (a simple function of) the **counts** of nearby words
- Word2vec
  - **Dense** vectors
  - Representation is created by training a classifier to **predict** whether a word is likely to appear nearby
  - Later we'll discuss extensions called **contextual embeddings**

Word Embeddings

- Sparse versus dense vectors
- tf-idf vectors are
  - **long** (length |V|= 20,000 to 50,000)
  - **sparse** (most elements are zero)
- Alternative: learn vectors which are
  - **short** (length 50-1000)
  - **dense** (most elements are non-zero)
- Why dense vectors?
  - Short vectors may be easier to use as **features** in machine learning (fewer weights to tune)
  - Dense vectors may **generalize** better than explicit counts
- They may do better at capturing synonymy:
  - *car* and *automobile* are synonyms; but are distinct dimensions
  - a word with *car* as a neighbor and a word with *automobile* as a neighbor should be similar, but aren't
- **In practice, they work better**

Word Embeddings

- Neural Language Model”-inspired models
  - Word2vec (skipgram, 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)
    - Compute distinct embeddings for a word in its context
    - Separate embeddings for each token of a word
    - We'll return to this in a later chapter
    - All the new age LLMs

Word2vec

- Instead of **counting** how often each word *w* occurs near "*apricot*"
  - 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 idea: **self-supervision**:
    - A word c that occurs near apricot in the corpus asks as the gold "correct answer" for supervised learning
    - No need for human labels
    - Bengio et al. (2003); Collobert et al. (2011)
- Popular embedding method
- Very fast to train
- Code available on the web
- Idea: **predict** rather than **count**

Word2vec

Word2vec provides a variety of options.

skip-gram with negative sampling (SGNS)

Continuous bag of words approach (CBOW)

Approach: predict if candidate word *c* is a "neighbor" – Skip-gram

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

Word2vec

- Assume a +/- 2 word window, given training sentence:

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

c1 c2 [target] c3 c4

Goal: train a classifier that is given a candidate (**w**ord, **c**ontext) pair

(apricot, tablespoon)

(apricot, land)

…

And assigns each pair a probability:

$P(+|w, c)$

Word2vec

Remember: two vectors are similar if they have a high dot product

- Cosine is just a normalized dot product

So: Similarity(w,c) $\propto$ w · c

We'll need to normalize to get a probability

- (cosine isn't a probability either)

- **Turning dot products into probabilities**
  - Sim(w,c) ≈ w · *c*
  - To turn this into a probability
  - We'll use the sigmoid from logistic regression:

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

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

Word2vec

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

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

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

We'll assume independence and just multiply them:

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

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

Word2vec

...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

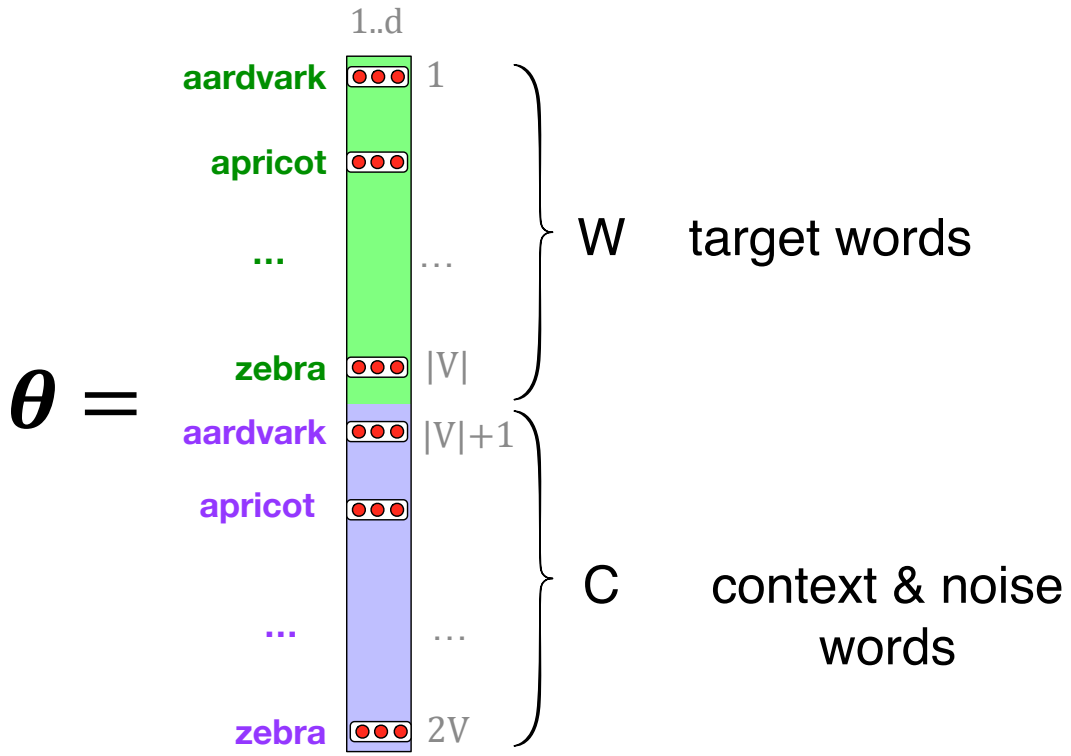
For each positive example we'll grab k negative examples, sampling by frequency

negative examples -

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

Word2vec

- A probabilistic classifier that,
  - given a test target word *w*
  - its context window of *L* words $c_{1:L}$,
- assigns a probability that w occurs in this window.
- To compute this, we just need embeddings for all the words.



Word2vec

- 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.

Word2vec

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

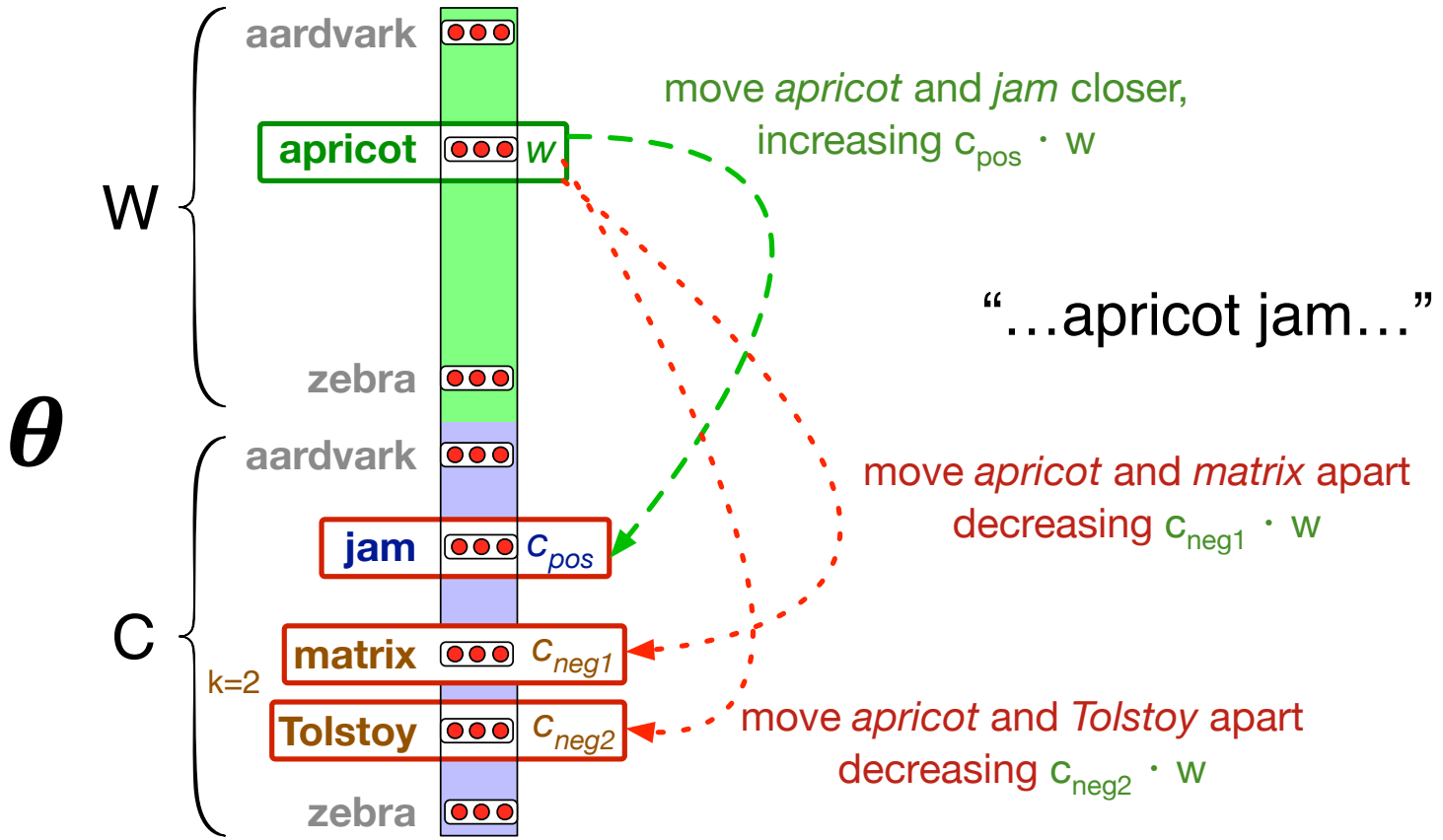
Maximize the dot product of the word with the actual context words, and

minimize the dot products of the word with the *k* negative sampled non-neighbor words.

$$
\begin{aligned}
L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\
&= -\left[\log P(+|w,c_{pos})+\sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\
&= -\left[\log \sigma(c_{pos}\cdot w)+\sum_{i=1}^{k}\log\sigma(-c_{neg_i}\cdot w)\right]
\end{aligned}
$$

Word2vec

- 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



Word2vec

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

Summarize Word2vec

- Start with V random d-dimensional vectors as initial embeddings
- Train a classifier based on embedding similarity
  - Take a corpus and take pairs of words that co-occur as positive examples
  - Take pairs of words that don't co-occur as negative examples
  - Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

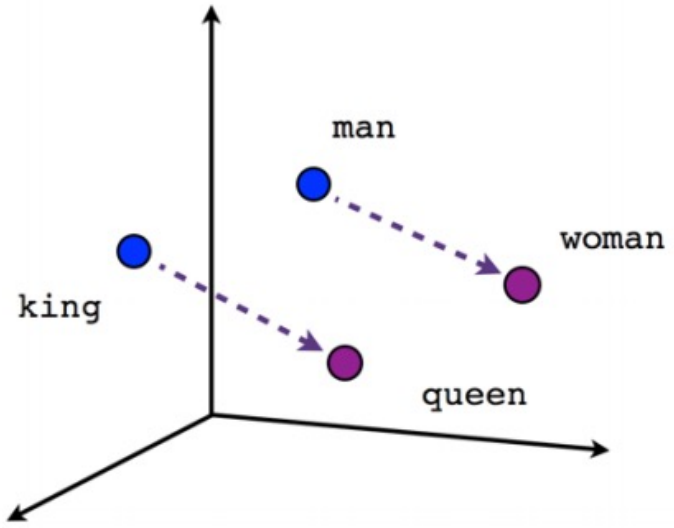
Word2vec – SGD

$$
\begin{aligned}
L_{CE} &= -\left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{neg_i} \cdot w)\right] \\
\frac{\partial L_{CE}}{\partial c_{pos}} &= [\sigma(c_{pos} \cdot w) - 1]w \\
\frac{\partial L_{CE}}{\partial c_{neg}} &= [\sigma(c_{neg} \cdot w)]w \\
\frac{\partial L_{CE}}{\partial w} &= [\sigma(c_{pos} \cdot w) - 1]c_{pos} + \sum_{i=1}^{k} [\sigma(c_{\mathrm{neg}_i} \cdot w)]c_{neg_i}
\end{aligned}
$$

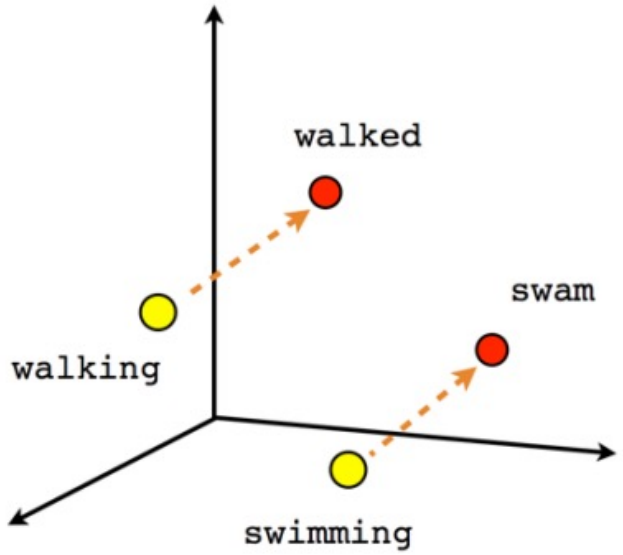
Update equation in SGD: Start with randomly initialised C and W matrices, then incrementally do updates

$$
\begin{aligned}
c_{pos}^{t+1} &= c_{pos}^{t} - \eta[\sigma(c_{pos}^{t} \cdot w) - 1]w \\
c_{neg}^{t+1} &= c_{neg}^{t} - \eta[\sigma(c_{neg}^{t} \cdot w)]w \\
w^{t+1} &= w^{t} - \eta[\sigma(c_{pos} \cdot w^{t}) - 1]c_{pos} + \sum_{i=1}^{k} [\sigma(c_{\mathrm{neg}_i} \cdot w^{t})]c_{neg_i}
\end{aligned}
$$

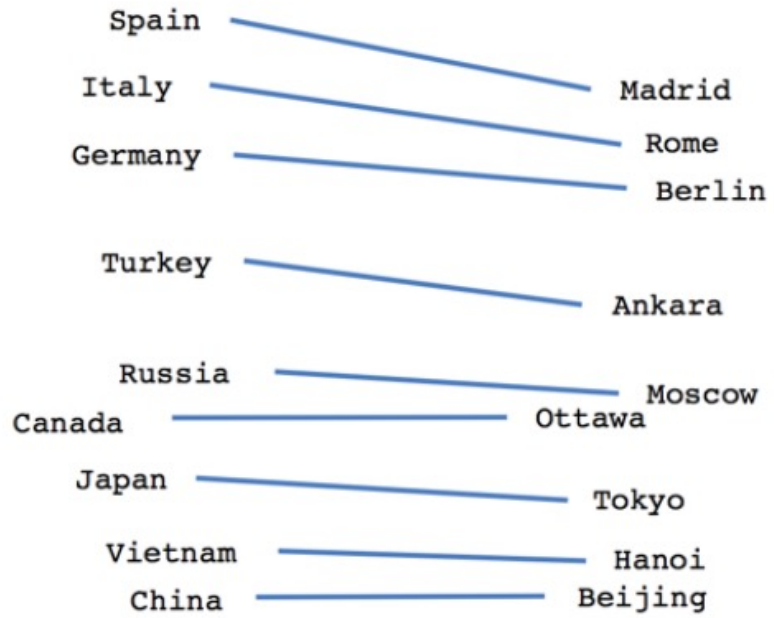
How does the vector space look like?



Male-Female



Verb tense



Country-Capital

Next lecture – do you know?

- Stochastic Gradient Descent ?
- Hierarchical Softmax ?

Literature

- Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*, Cambridge University Press. 2008.
- https://stanford-cs324.github.io/winter2022/lectures/introduction/
- https://stanford.edu/~jurafsky/slp3/slides/6_Vector_Jan13_2020.pdf
- Video Lectures of Andrew Ng.
- Papers:
  - Efficient Estimation of Word Representations in Vector Space
  - Distributed Representations of Words and Phrases and their Compositionality