CSE 156 | Lecture 4: Word Embeddings & Tokenization

Ndapa Nakashole

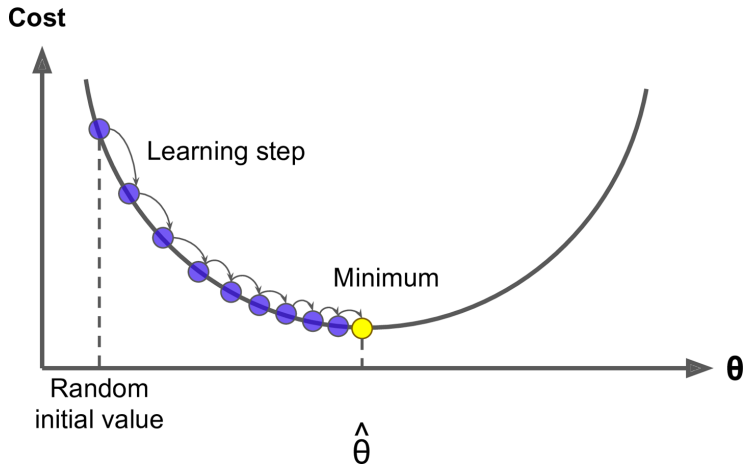
October 10, 2024

Today

1. Finish FeedForward Neural Networks and Text Classification
2. Word Embeddings
3. Tokenization

Recap: Gradient Descent Iterative Improvement of W

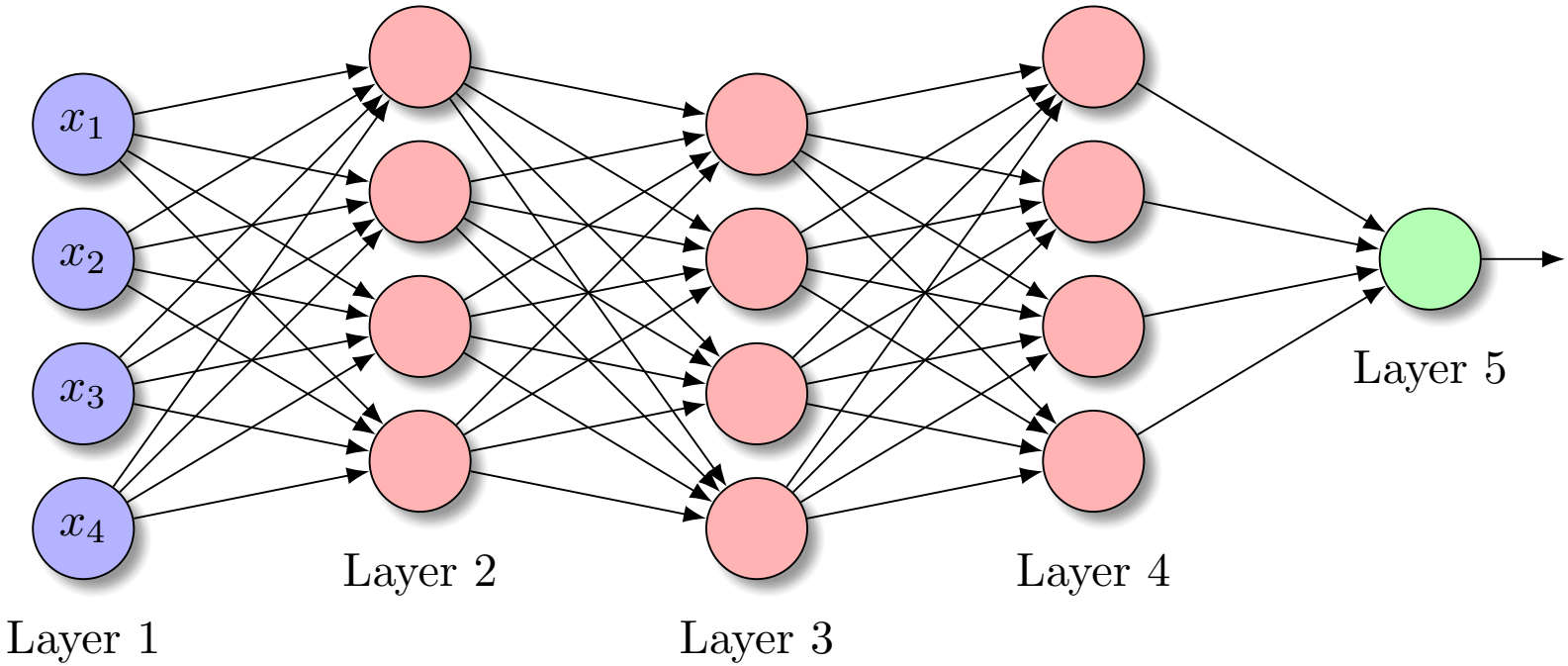
- Idea: for current value of $\mathbf{W}$, calculate gradient of $L(\mathbf{W})$, then take small step in the direction of the negative gradient. **Repeat until convergence**



$$W_{t+1} = W_t - \alpha \nabla L\left(W_t\right)$$ Gradient Descent Update Rule

- Three ways to compute the gradient
  1. Numerically (finite differences)
  2. Analytically (write down the gradient by hand)
  3. Backpropagation (algorithm for computing the gradient)

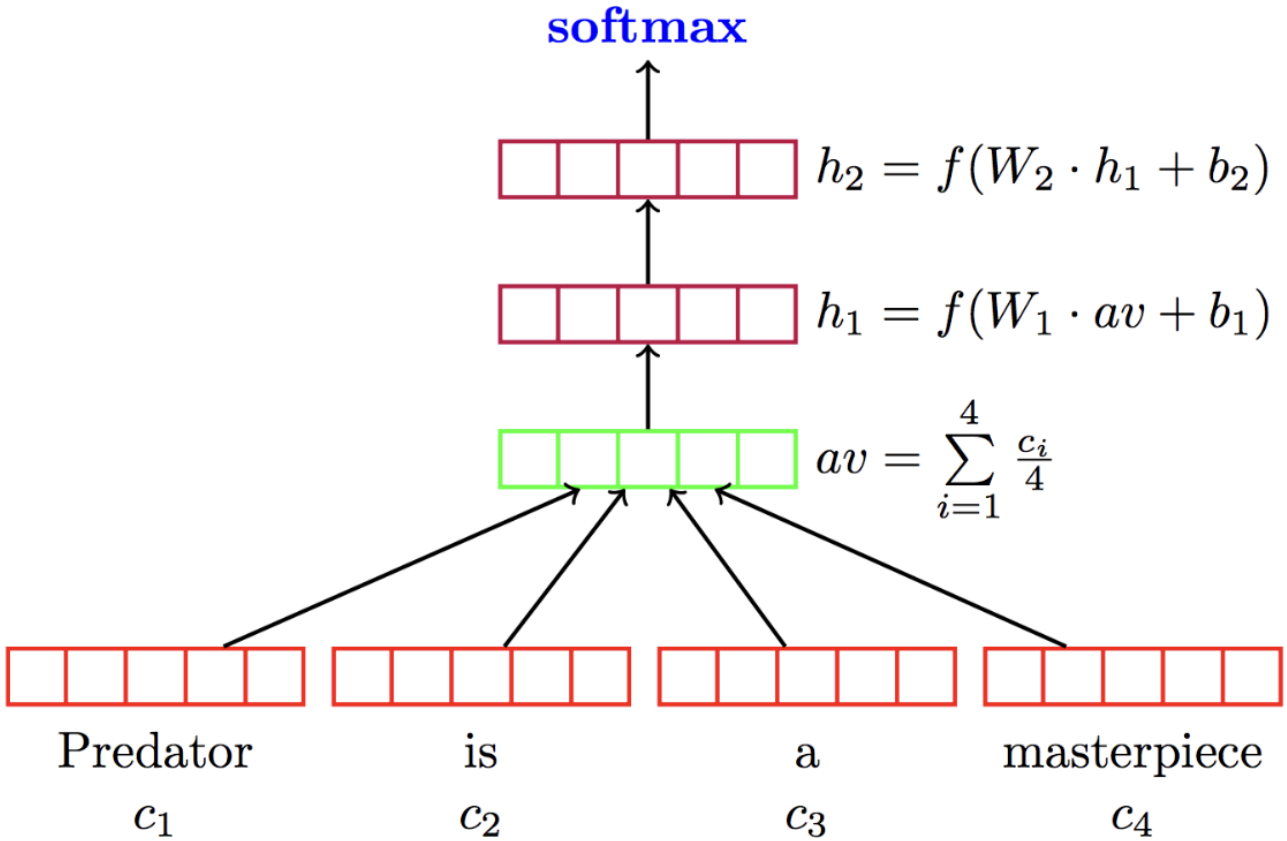
Recap: A feedforward network = running several logistic regressions at the same time



Allows us to **re-represent and compose our data multiple times and to learn a classifier that is highly non-linear in terms of the original inputs** (but, typically, is linear in terms of the pre-final layer representations)

Recap: Deep Averaging Networks (PA1)

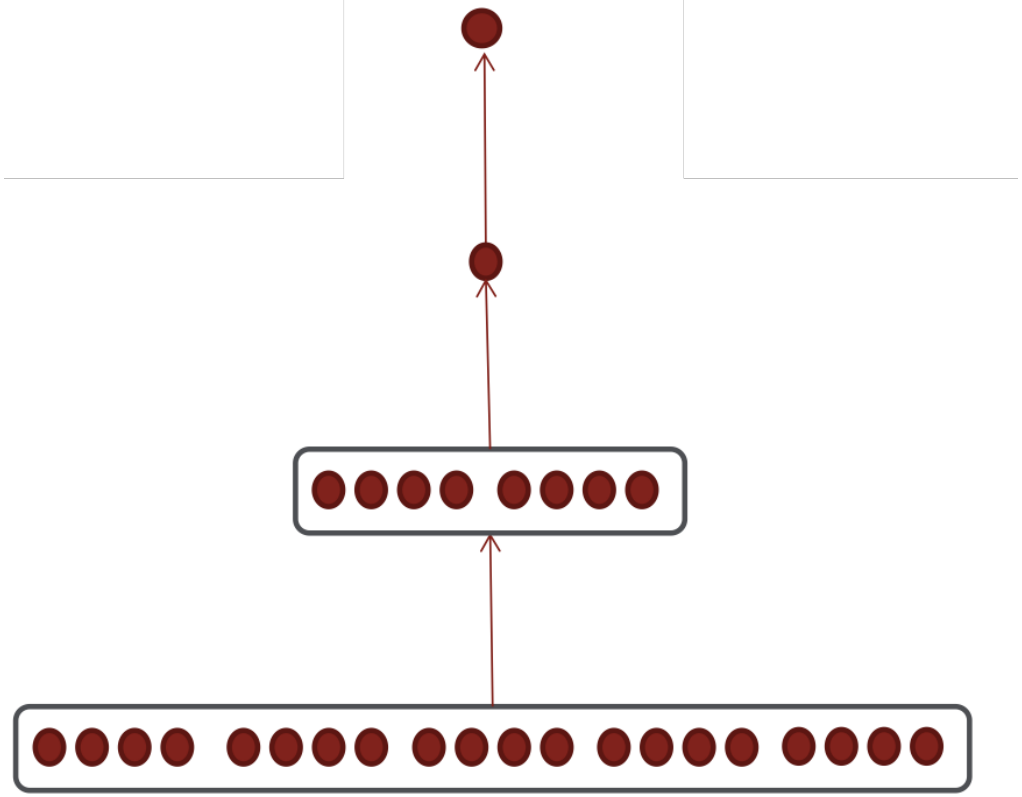
- Deep Averaging Networks: feedforward neural network on average of word embeddings from input text



Iyyer et al. (2015)

Text Classification with a FFN: Summary

Convert text to vector, apply FFN, pick label with highest score



$p = \text{softmax}(s)$

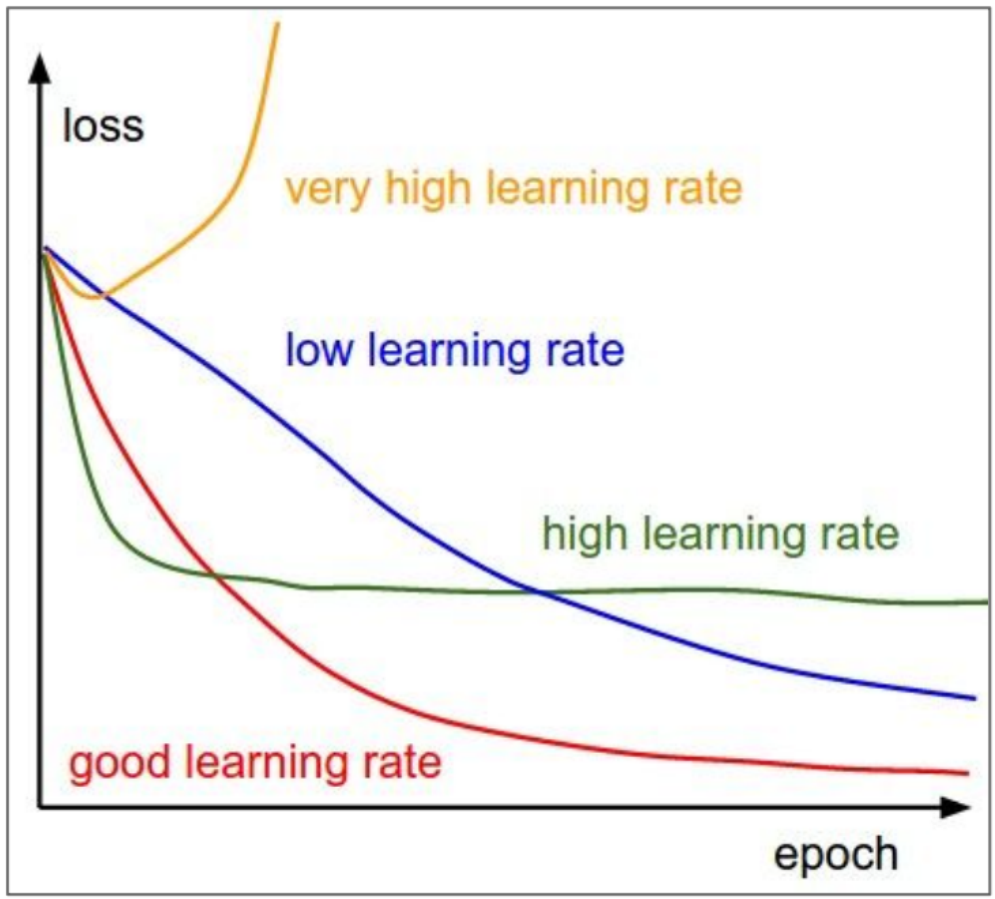
$s = \boldsymbol{W}_2 \boldsymbol{h}$

$\boldsymbol{h} = g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b})$

$\boldsymbol{x}$ (input)

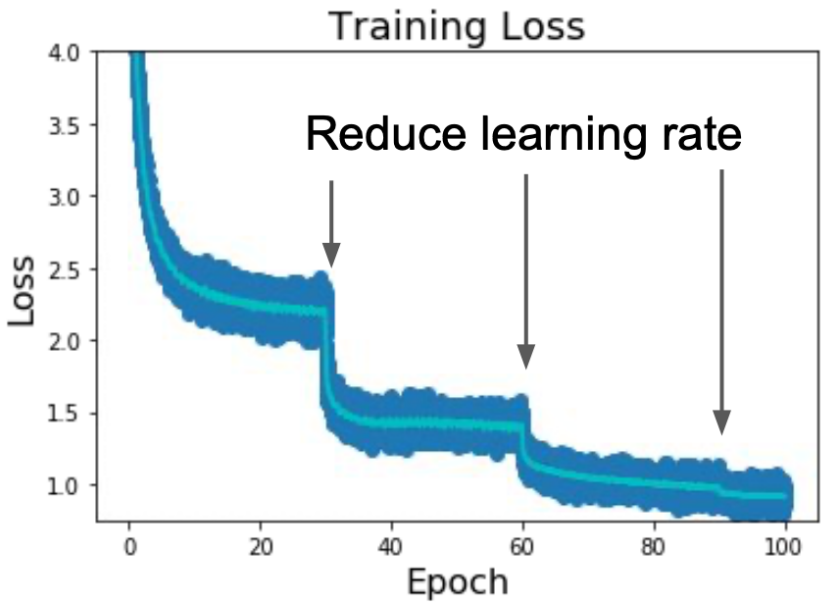
Learning rate is a crucial hyperparameter

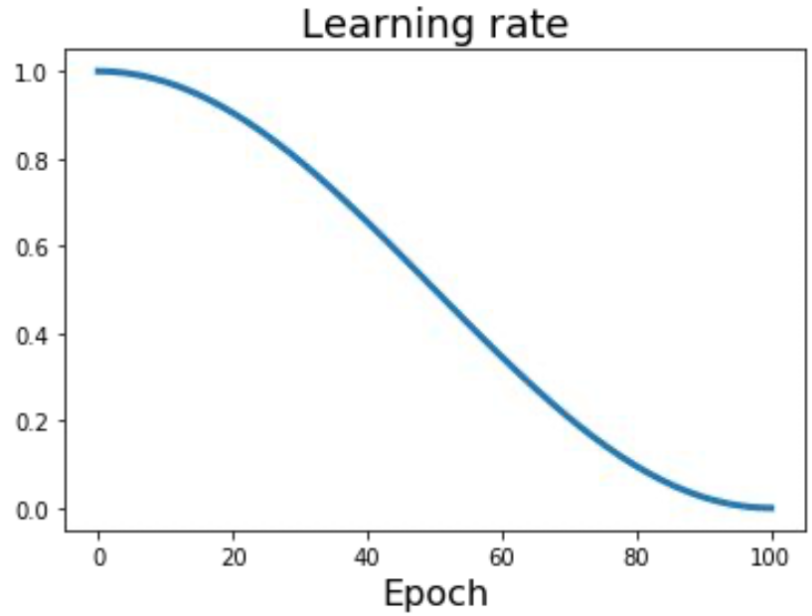
- **Learning rate**: Step size in gradient descent. Too small: slow convergence. Too large: overshoots the minimum



Learning Rate Schedules: change the learning rate over time

Can start with a high learning rate and then decrease it over time to improve convergence

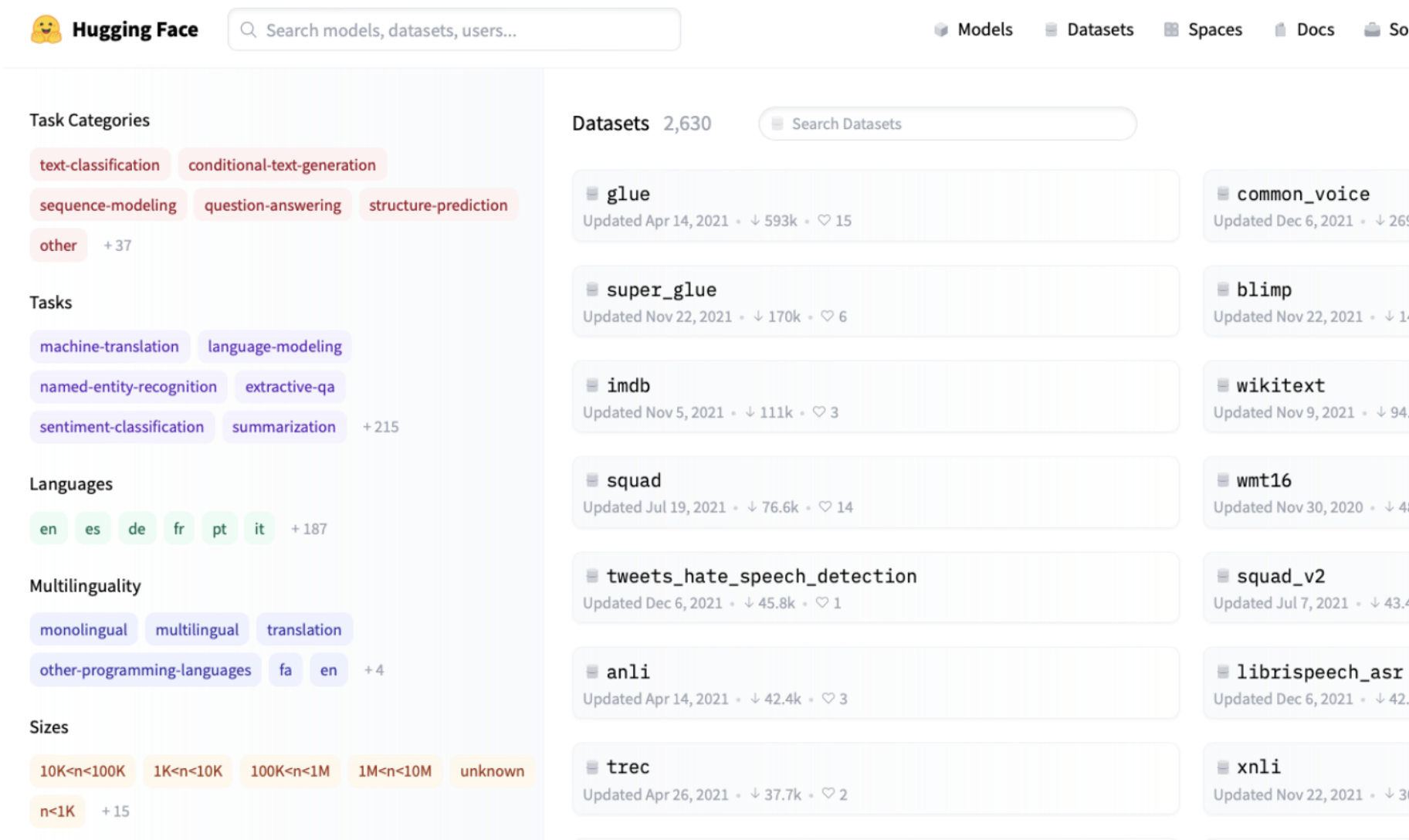




- **Step**: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30,60 , and 90 .

- **Cosine**:
  $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$
  - α_0 : Initial learning rate
  - α_t : Learning rate at epoch t
  - T : Total number of epochs

Where can I find text classification data?



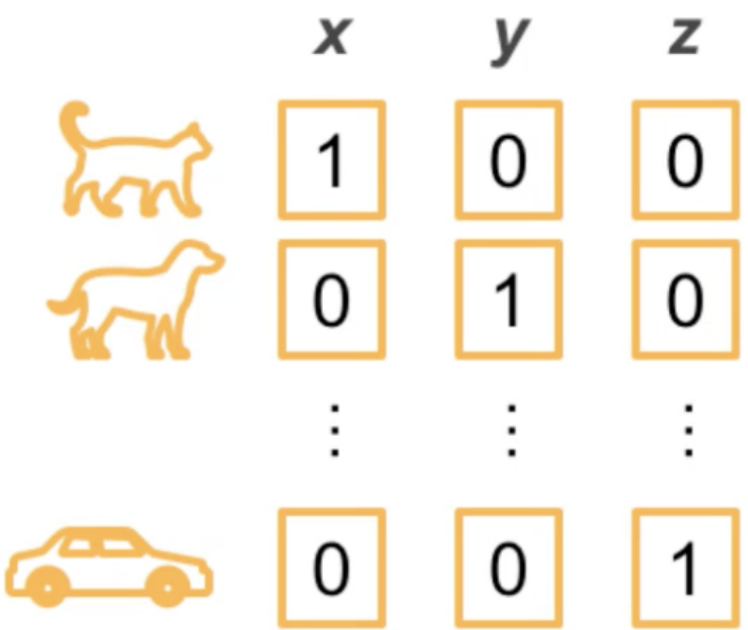
See: `https://huggingface.co/datasets`

Text Classification: we have looked at

1. The Text Classification Problem
2. Example Applications
3. Models: Linear Classifiers, Feedforward Neural Networks

Learning Word Representations: Word Embeddings

1-hot Vectors



- One-hot vectors map objects/ words into fixed-length vectors
- These vectors only contain the identity information of the object
- They do not contain any semantic information about the words, $\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{z}, \mathbf{y} \rangle = 0$
- Bag of Words: is a summation of 1-hot vectors, hence also lacks semantic information

Word Embeddings

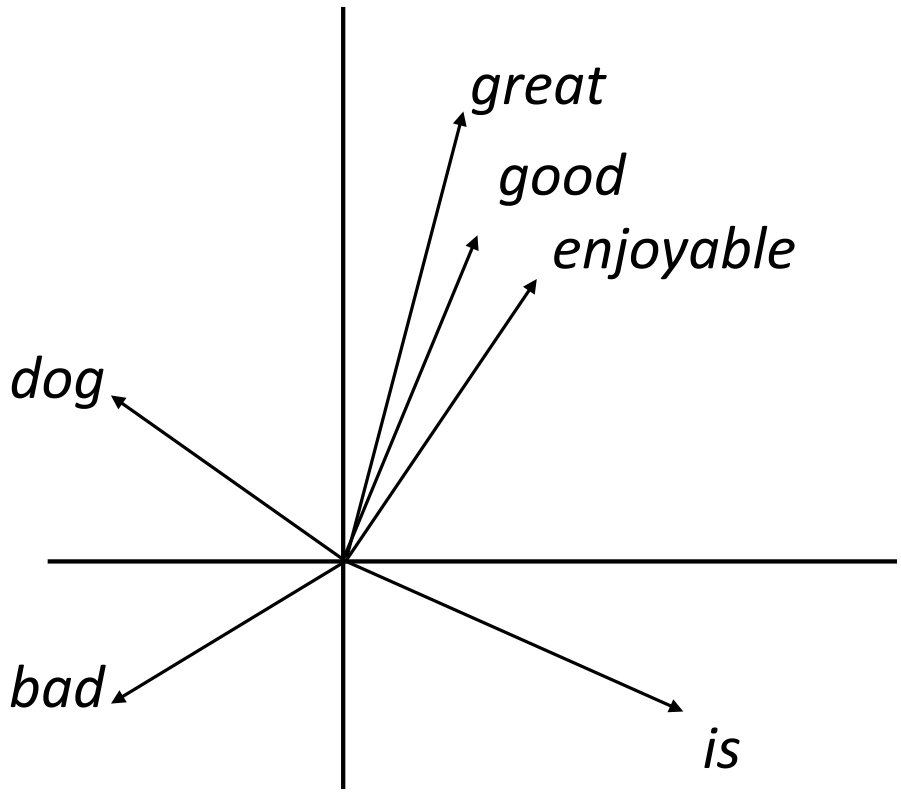
Interesting result: Word meaning can be represented by a vector of real numbers

Want a vector space where similar words have similar vectors

the movie was great

≈

the movie was good



- Goal: come up with a way to produce these embeddings
- For each word, want "medium" dimensional vector (50-300 dims)

Distributional Semantics

We will use the idea of distributional semantics to learn word embeddings

kombucha

I had a glass of kombucha at the farmers market

Children or pregnant women should not drink kombucha

- **Distributional semantics**: words that appear in similar contexts have similar meanings
  - Can substitute "kombucha" with "beer" or "wine" in the above sentences
- **Idea**: just find at a lot of usage of a word, and build up its meaning from there!
- "You shall know a word by the company it keeps" Firth (1957)

Kombucha

Word Vectors

- We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts, measuring similarity as the vector dot (scalar) product

$$\textit{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix} \qquad \textit{monetary} = \begin{pmatrix} 0.413 \\ 0.582 \\ -0.007 \\ 0.247 \\ 0.216 \\ -0.718 \\ 0.147 \\ 0.051 \end{pmatrix}$$

- Word2Vec (Mikolov et al., 2013): simpler and faster than previous models
  - Two algorithms: Skip-gram and Continuous Bag of Words (CBOW)
  - They are similar, we will focus on Skip-gram

Skip-gram

Skip Gram: Learning Word Embedding (Mikolov et al. 2013)

- **Input**: a corpus of text (e.g. all of Wikipedia, News articles, Books, etc.)
- **Output**: a set of embeddings: a real-valued vector for each word in the vocabulary
- We are going to learn these by setting up a fake prediction problem: predict a word's context from that word

the dog bit the man

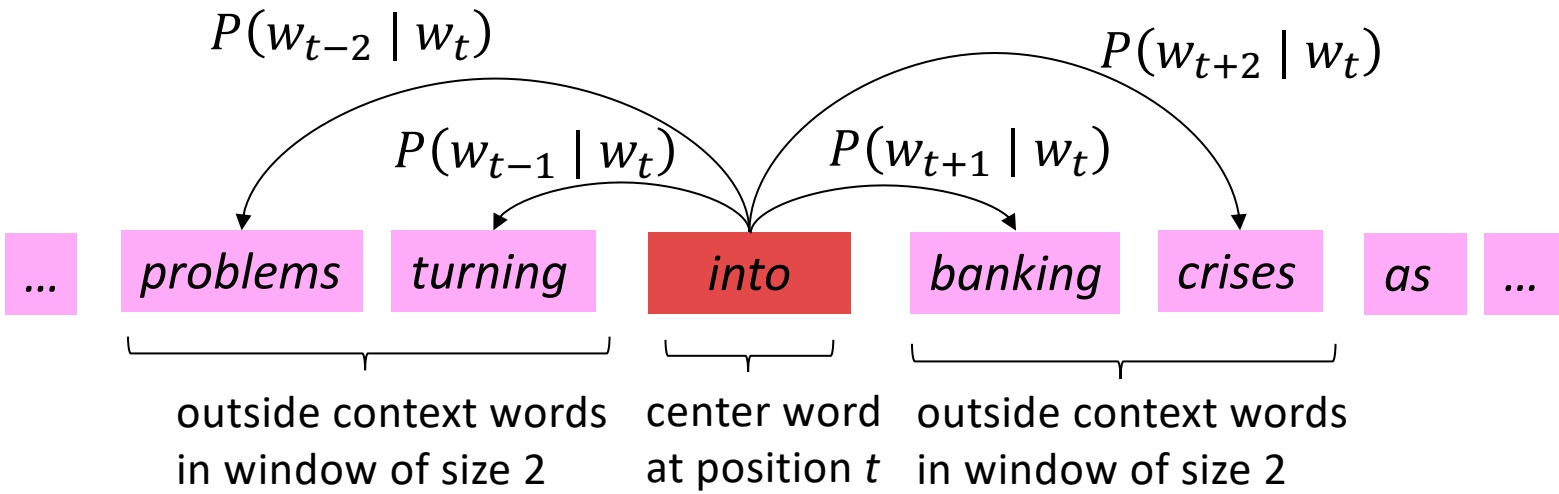
(word = *bit*, context = *dog*)

(word = *bit*, context = *the*)

Example: Center is "into", predict context words

Context Window Size = 2

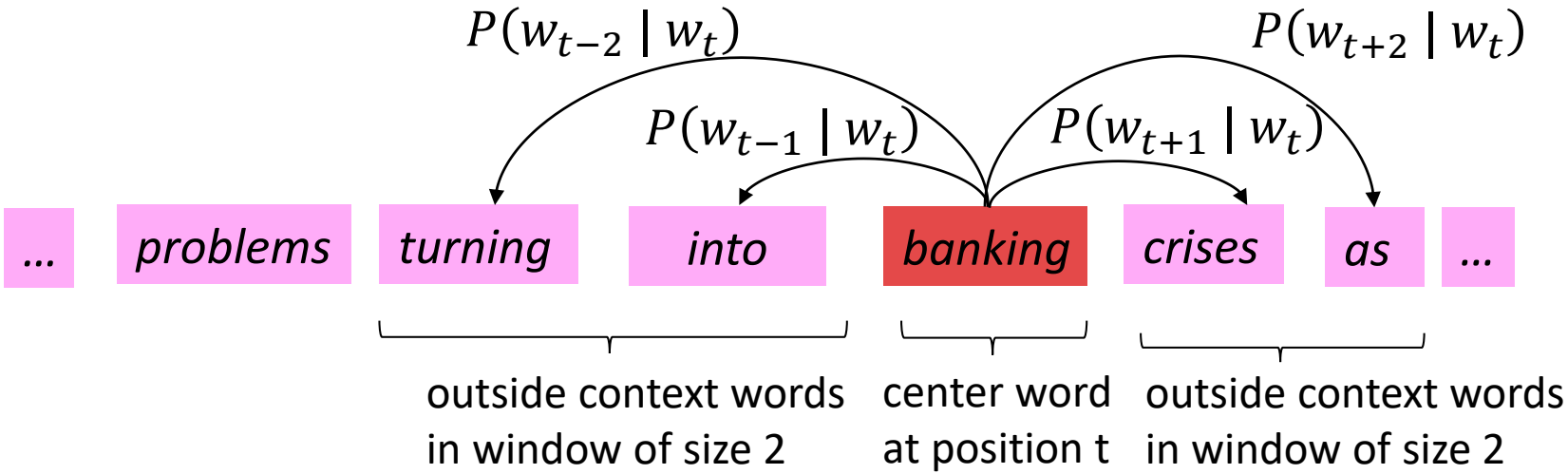
$$P\left(w_{t+j} \mid w_t\right)$$



Example: Center is "banking", predict context words

Context Window Size = 2

$$P\left(w_{t+j} \mid w_t\right)$$



- We do this for every word in the corpus, and learn the embeddings in the process

Skip-gram Loss Function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size m, given center word w_j.

$$\text{Data Likelihood} = \prod_{t=1}^{T} \prod_{\substack{m \leq j \leq m \\ j \neq 0}} P\left(w_{t+j} \mid w_t; \theta\right)$$

where θ are all the parameters of the model

We can write the loss as:

$$L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{m \leq j \leq m \\ j \neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

Skip-gram: loss function

We want to minimize the loss function:

$$L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{m\leq j\leq m \\ j\neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

Question: How to calculate $P\left(w_{t+j} \mid w_t; \theta\right)$?

Answer: Use the softmax function!

For a center word c and a context word o:

$$P(o \mid c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w\in V}\exp\left(u_w^T v_c\right)}$$

We will use two vectors per word w:

- v_w when w is a center word
- u_w when w is a context word

Training Word Vectors: minimize the loss with gradient descent

- Recall: θ represents all parameters: the word vectors and the weights of the neural network, in one big vector
- Each word has two vectors - works better for training

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

Computing the Gradients

$$L(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{m\leq j\leq m\\ j\neq 0}} \log P\left(w_{t+j} \mid w_t; \theta\right)$$

$P\left(w_{t+j} \mid w_t; \theta\right)$ is the softmax function:

For a center word c and a context word o:

$$P(o \mid c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w\in V} \exp\left(u_w^T v_c\right)}$$

$$\frac{\partial}{\partial v_c} \log \frac{\exp\left(u_0^\top v_c\right)}{\sum_{w=1}^{v} \exp\left(u_w v_c\right)}$$

Computing the Gradients wrt the center word vector

$$\frac{\partial}{\partial v_c} \log \frac{\exp\left(u_0^\top v_c\right)}{\sum_{w=1}^{V} \exp\left(u_w v_c\right)}$$

$$= \frac{\partial}{\partial v_c} \log \exp\left(u_o^\top v_c\right) - \frac{\partial}{\partial v_c} \log \sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)$$

Step 1: Differentiate the First Term

$$\frac{\partial}{\partial v_c}\left(u_o^\top v_c\right) = \frac{\partial}{\partial v_c} u_o^\top v_c = u_o$$

Gradient the gradient wrt the center word vector (2/4)

Step 2: Differentiate the Second Term

$$\log \sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)$$

Use chain rule, derivative of $\log(z)$ wrt z is $\frac{1}{z}$:

$$\frac{\partial}{\partial v_c} \log \sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)$$

$$= \frac{1}{\sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)} \frac{\partial}{\partial v_c} \sum_{x=1}^{V} \exp\left(u_x^\top v_c\right)$$

Gradient the gradient wrt the center word vector (3/4)

Step 2: Differentiate the Second Term (continued)

$$\frac{\partial}{\partial v_c} \sum_{x=1}^{V} \exp\left(u_x^\top v_c\right)$$

Move derivative inside the sum:

$$\begin{aligned}
&\frac{\partial}{\partial v_c} \sum_{x=1}^{V} \exp\left(u_x^\top v_c\right) \\
&= \sum_{x=1}^{V} \frac{\partial}{\partial v_c} \exp\left(u_x^\top v_c\right) \\
&= \sum_{x=1}^{V} \exp\left(u_x^\top v_c\right) u_x
\end{aligned}$$

Gradient the gradient wrt the center word vector (4/4)

Putting it all together

$$\frac{\partial}{\partial v_c} \log\left(p(o \mid c)\right) = u_o - \frac{1}{\sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)} . \sum_{x=1}^{V} \exp\left(u_x^\top v_c\right) u_x$$

Distribute term across the sum:

$$= u_o - \sum_{x=1}^{V} \frac{\exp\left(u_x^\top v_c\right)}{\sum_{w=1}^{V} \exp\left(u_w^\top v_c\right)} u_x$$

$$= u_o - \sum_{x=1}^{V} P(x \mid c) u_x$$

$$= \text{observed context vector} - \text{expected context vector}$$

Thus center word is pulled towards words that are observed in its context, and away from those that are not. i.e

$v_c^{\text{new}} = v_c^{\text{old}} +$ observed $-$ expected

Gradient wrt the context word vector

- It is similar to the gradient wrt the center word vectors
- At home: derive the gradient wrt the context word vector

Glove Embeddings

Key Limitation of Word2Vec: Capturing cooccurrences inefficiently

- Go through each word of the whole corpus
- Predict surrounding words of each (window's center) word
- This captures cooccurrence of words **one at a time**
- **Glove**: why not capture cooccurrence counts directly?

GloVe: Global Vectors for Word Representation" (Pennington et al. (2014)

Example Corpus:

I like deep learning .
I like NLP .
I enjoy flying .

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

GloVe: Global Vectors for Word Representation" (Pennington et al. (2014)

Glove loss function:

$$L(\theta) = \frac{1}{2} \sum_{i,j=1}^{\mathcal{V}} f\left(X_{ij}\right) \left(u_i^T v_j - \log X_{ij}\right)^2$$

- $\mathcal{V}$: vocabulary size
- X_{ij}: cooccurrence count of word i and j
- u_i, v_j: word vectors
- $f(X_{ij})$: weighting function to reduce the influence of very frequent or very rare word pairs

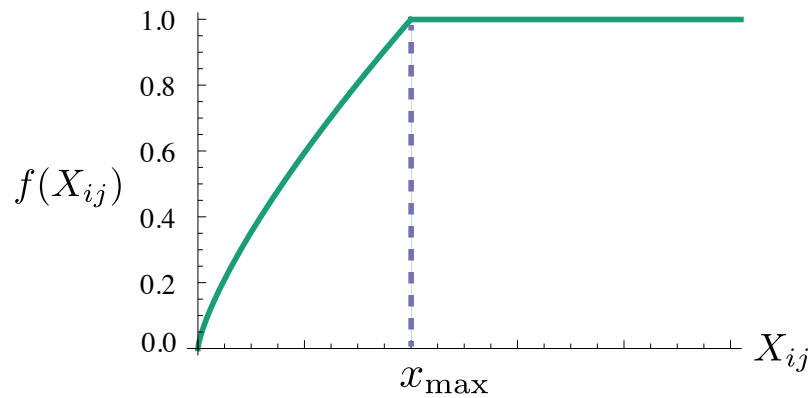


Figure: $f(X_{ij})$

Are my word embeddings any good? Evaluation of Word Embeddings

Evaluation Methods for Word Embeddings

Intrinsic:

- Directly evaluate the embeddings
- Fast to compute
- Not clear if really helpful unless **correlation to real task is established**

Extrinsic:

- Evaluation on a real task (MT, QA, Parsing, Summarization, etc.)
  - For every embedding method, retrain the model on the task
- Can take a long time to compute accuracy

Intrinsic word vector evaluation

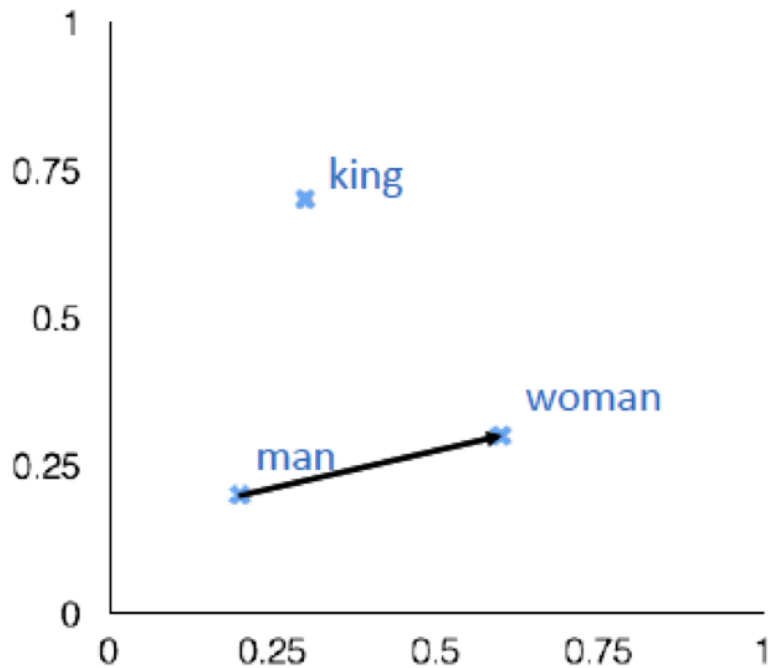
Word Vector Analogies:

a:b :: c:?

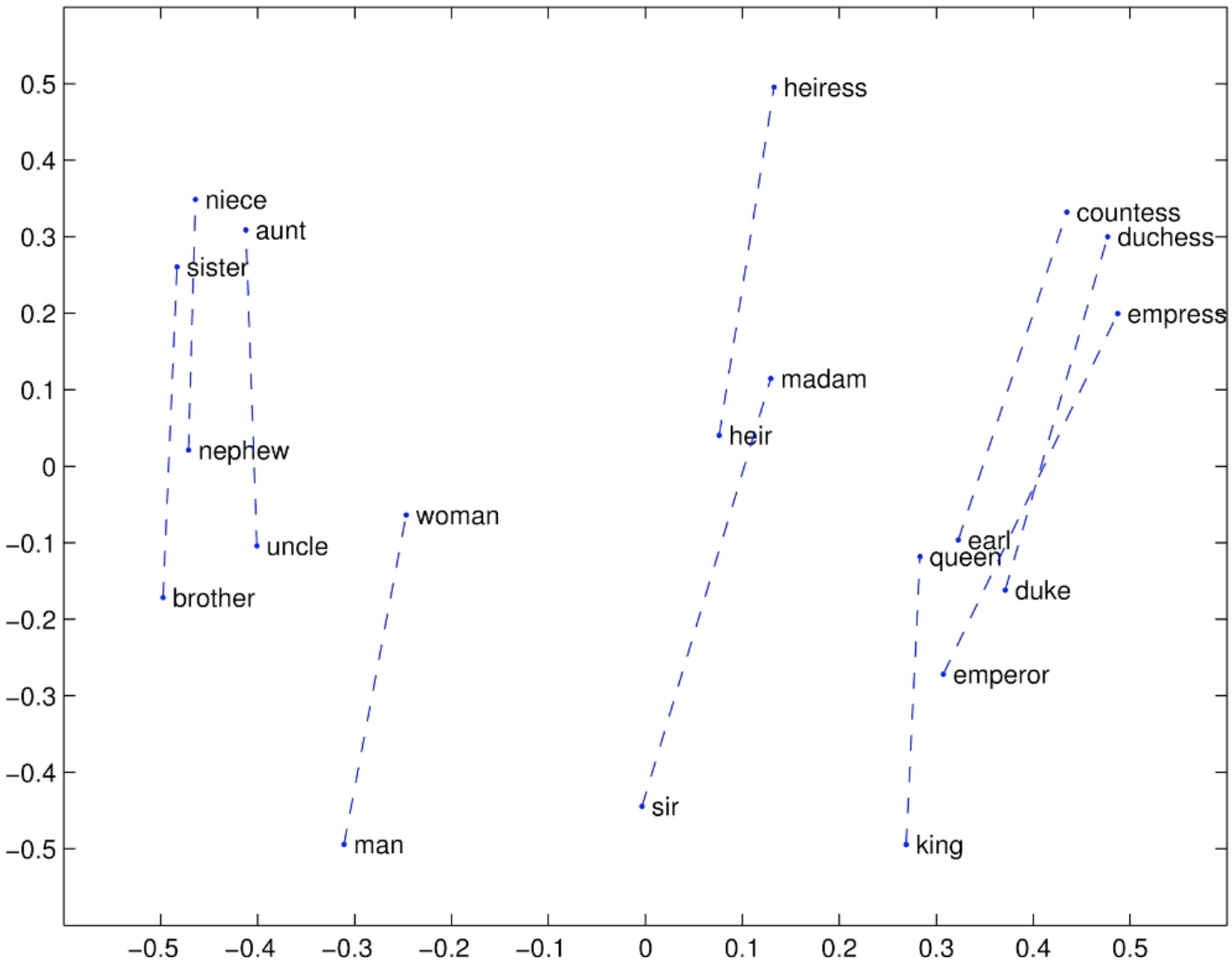
man:woman :: king:?

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}$$

- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search (!)
- Problem: What if the information is there but not linear?



GloVe Visualization



How can I use Word Embeddings for my task? (PA1)

Extrinsic Evaluation

- **Approach 1**: Learn embeddings from scratch as parameters from your data: Often works pretty well
- **Approach 2**: Initialize using pretrained word embeddings, keep fixed. Faster because no need to update these parameters
- Approach 3: initialize using pretrained word embeddings, fine-tune on task. Often works best

Hyperparameters in Word Embeddings

- Dimensionality: 50-300 dimensions
- Window size: 5-10 words
- Initialization of word embeddings
- Learning rate
- Training epochs

Limitation of Word Embeddings: conflated word senses

- One word, one vector: conflated word senses (polysemy)
- E.g., “bank” in “river bank” vs. “bank account” or “rock” in “rock music” vs. “rock climbing”, or “apple” in “apple fruit” vs. “Apple Inc.” or “bass” in “bass guitar” vs. “bass fish”, ...
- Solution: **contextual embeddings** coming later (ELMo, BERT, GPT-2)

Out-of-vocabulary (OOV) words, or rare words: how do we learn good embeddings for them?

Tokenization

What is tokenization?

- **Token**: basic processing unit in NLP models
- **Tokenization** is the process of breaking up text into tokens (words, or subwords, etc.), which then are converted to IDs through a look-up table.
  - So far in this class... the process has been simple: splitting a string into words based on spaces and punctuation
  - The quick brown fox jumps over the lazy dog
  - 11 298 34 567 432 13 11 49 305

This tokenization step requires an external tokenizer to detect word boundaries!

Tokenizer

- Tokenizer is independent of the model, it is a separate module with its own training data
- The tokenizer is a translation layer between the raw text and token IDs
  - **Encode**: raw text $\rightarrow$ token IDs
  - **Decode**: token IDs $\rightarrow$ raw text

Tokenization: source of some of the problems in LLMs

- **Python Processing** in GPT-2: affected significantly by tokenization approach
- **Math processing**: Tokenization can lead to unexpected results when numbers are split into multiple tokens
- Other sources of quirks: punctuation, contractions, hyphenated words, numbers, etc.

Problems with Whitespace Tokenization

*Mr.**O'Neill** thinks that the boys' stories about San Francisco **aren't** amusing.*

- Whitespace tokenizer failures
  - prize-winning ⇒ prize , - , winning
  - 1850s ⇒ 1850 , s
  - U.K. ⇒ U , . , K ,
  - 2.5 ⇒ 2 , . , 5
- Word tokenizers require lots of specialized rules about how to handle specific inputs
  - Check out spaCy's tokenizers! (https://spacy.io/)

GPT3 Tokenizer

https://platform.openai.com/tokenizer

Tokens	Characters
161	331

```
Tokenization is at the heart of much weirdness of LLMs. Do
not brush it off.
127 + 677 = 804
1275 + 6773 = 8048
Egg.
I have an Egg.
egg.
EGG.
for i in range(1, 101):
    if i% 3== i% and i = \
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

GPT3.5/4 Tokenizer

Groups more white space into a single token, densifies Python code, can attend to more code in a single pass

Tokens	Characters
120	331

```
Tokenization is at the heart of much weirdness of LLMs. Do
not brush it off.
127 + 677 = 804
1275 + 6773 = 8048
Egg.
I have an Egg.
egg.
EGG.
for i in range(1, 101):
    if i% 3== i% and i = \
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Big problem with word tokenization

Problem: What happens when we encounter a word at **test time** that we've never seen in our training data?

- We can't assign an index to it! We don't have a word embedding for that word!

Solution: replace low-frequency words in training data with a special <UNK> token, use this token to handle unseen words at test time too

Limitations of UNK

We lose information about the word that was replaced by <UNK>

E.g.

- The chapel is sometimes referred to as "Hen Gapel Lligwy" ("hen" being the Welsh word for "old" and "capel" meaning "chapel").
- The chapel is sometimes referred to as " Hen <UNK> <UNK> " (" hen " being the Welsh word for " old " and " <UNK> " meaning " chapel ").

- We don't want to generate UNK when generating text (imagine ChatGPT doing this)
- In languages with **productive morphology**, lots of long words are formed by composing smaller pieces, e.g., German, Finnish, Turkish removing rare words can lead significant loss of information

Other limitations

Word-level tokenization treats different forms of the same word (e.g., "open", "opened", "opens", "opening", etc) as separate types $->$ separate embeddings for each

- When words are related to each other, we'd like to share information between them
- Despite these limitations, word-level tokenization was the approach until recently (circa 2016)

An alternative: character tokenization

- Small vocabulary, just the number of unique characters in the training data!
- However, we end up with longer input sequences

...

- E.g., two sentences might be three tokens apart in a word model, but 30 tokens in a character model, e.g., "the children are happy" $\Rightarrow$ " vs "t h e", "c h i l d r e n", "a r e", "h a p p y"

Problem with character tokenization

- Character-level models **must discover that words exist and are delimited by spaces** — this information is built into word-based models

Character, Word, and Subword Tokenization

- **Character**: Learning a meaningful context-independent representation for the letter "t" is much harder than learning a context-independent representation for the word "today"
- **Word**: Big vocabulary size forces model to use huge embedding matrix at input & output layer $\rightarrow$ increased memory and compute time
- **Hybrid**: between word-level and character-level tokenization: **subword** tokenization
  - Allows the model to have a **reasonable vocabulary size while being able to learn meaningful context-independent representations**
  - Enables the model to process **words it has never seen before, by decomposing them into known subwords**

2016: subword tokenization

- Subword tokenization: break words into multiple **word pieces**
  - Generally want more frequent words to be represented by fewer tokens, e.g, "the" should be a single token
  - Can handle **rare** words better than word-level tokenization
  - Can **share parameters** between related words (e.g., "open", "opened", "opens", "opening")
  - Reduce vocabulary size $\rightarrow$ **reduce num parameters, compute+memory**
- Developed for machine translation by Sennrich et al., ACL 201, Later used in BERT, T5, RoBERTa, GPT, etc.
- Relies on a simple algorithm called Byte Pair Encoding (Gage, 1994)
  - Iteratively merges the most frequent pair of consecutive characters, until a fixed vocabulary size is reached

Work in MT led to Strong subword Tokenization

Neural Machine Translation of Rare Words with Subword Units

Rico Sennrich and **Barry Haddow** and **Alexandra Birch**
School of Informatics, University of Edinburgh
{rico.sennrich,a.birch}@ed.ac.uk, bhaddow@inf.ed.ac.uk

MT also led to strong sequence encoders, the attention-based models - Transformers!

Byte Pair Encoding

Form base vocabulary (all characters that occur in the training data)

word	frequency
hug	10
pug	5
pun	12
bun	4
hugs	5

Base vocab: b, g, h, n, p, s, u

Example from `https://huggingface.co/transformers/tokenizersummary.html`

Byte Pair encoding

Next, count the frequency of each character **pair** in the data, and choose the one that occurs most frequently

word	frequency	character pair	frequency
$\mathrm{h}+\mathrm{u}+\mathrm{g}$	10	ug	20
$\mathrm{p}+\mathrm{u}+\mathrm{g}$	5	pu	17
$\mathrm{p}+\mathrm{u}+\mathrm{n}$	12	un	16
$\mathrm{b}+\mathrm{u}+\mathrm{n}$	4	hu	15
$\mathrm{h}+\mathrm{u}+\mathrm{g}+\mathrm{s}$	5	gs	5

Byte Pair Encoding

Next, choose the most common pair (**ug**) and then merge the characters together into one symbol. Add this new symbol to the vocabulary. Then, retokenize the data

word	frequency	character pair	frequency
h+ug	10	un	16
p+ug	5	$h + ug$	15
p+u+n	12	pu	12
b+ u+ n	4	$p + ug$	5
h+ ug+s	5	$ug + \mathrm{s}$	5

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]

Byte pPir Encoding

Keep repeating this process! This time we choose **un** to merge, next time we choose $h + ug$, etc.

word	frequency	character pair	frequency
$\mathrm{h} + ug$	10	un	16
$\mathrm{p} + ug$	5	$h + ug$	15
$\mathrm{p} + \mathrm{u} + \mathrm{n}$	12	pu	12
$\mathrm{b} + \mathrm{u} + \mathrm{n}$	4	$p + ug$	5
$\mathrm{h} + ug + \mathrm{s}$	5	$ug + \mathrm{s}$	5

Byte pair encoding

Eventually, after a fixed number of merge steps, we stop

word	frequency
hug	10
p+ug	5
p+un	12
b+un	4
hug +s	5

new vocab: b, g, h, n, p, s, u, ug, un, hug

BPE

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)
```

- Initialize vocabulary with all characters in the training data
- While vocab size $<$ max vocab size:
  - Count frequency of all character pairs
  - Merge most frequent pair
  - Update vocabulary

BPE Tutorial with Implementation

Refer to this tutorial for implementing BPE tokenization, notice that it works on byte pairs, not character pairs but the idea is the same

Tutorial video on tokenization

`https://www.youtube.com/watch?v=zduSFxRajkE&t=20s&ab_channel=AndrejKarpathy`

BPE impact on Machine Translation of English to German, OOVs

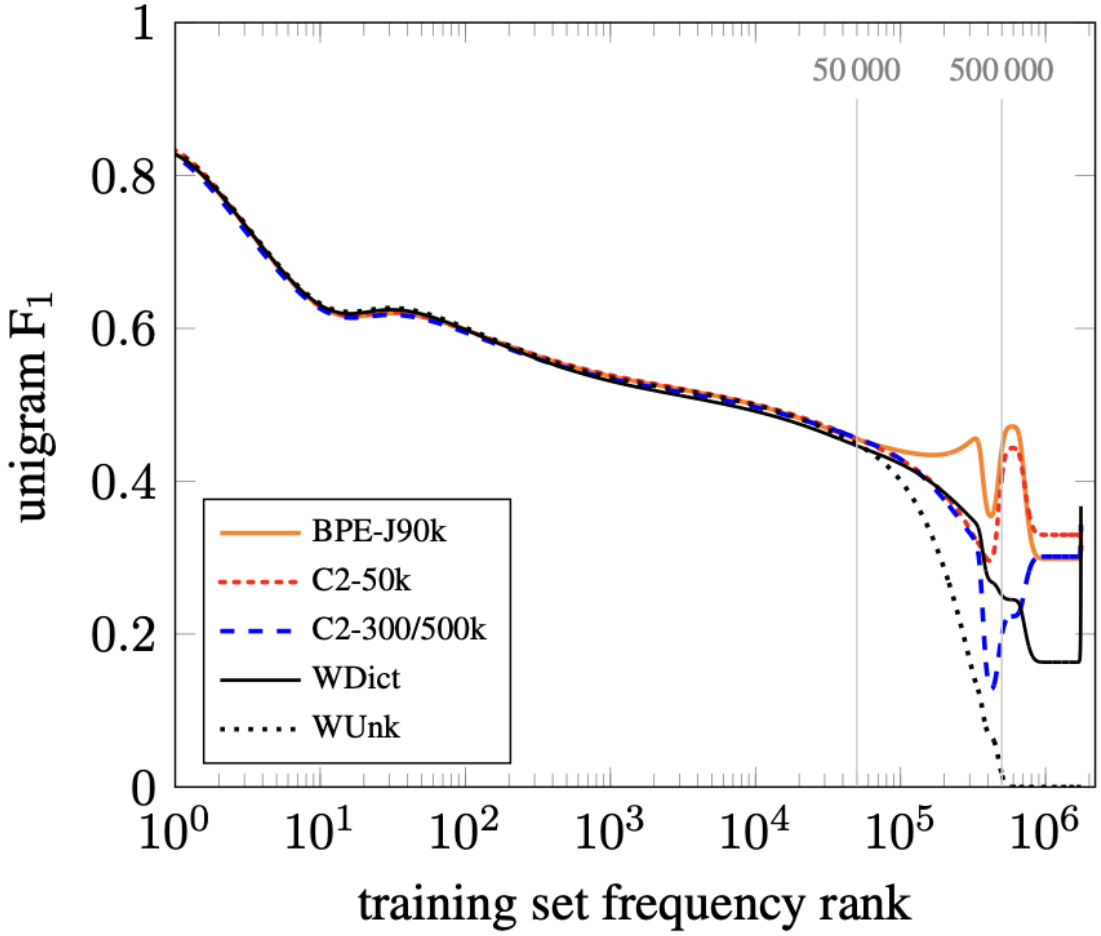


Figure 2: English→German unigram F_1 on newstest2015 plotted by training set frequency rank for different NMT systems.

Byte Pair Encoding

- To avoid <UNK>, all possible characters / symbols need to be included in the base vocab. This can be a lot if including all unicode characters (there are ~ 150 K unicode symbols)!
- GPT-2 uses bytes as the base vocabulary (size 256) and then applies BPE on top of this sequence (with some rules to prevent certain types of merges).
  - GPT-2 has a vocabulary size of 50,257, which corresponds to the 256 bytes base tokens, a special end-of-text token and the symbols learned with $50,000$ merges.
- Common vocabulary sizes: 32 K to 64 K tokens

Other subword encoding schemes

- WordPiece (Schuster et al., ICASSP 2012): merge by likelihood as measured by language model, not by frequency
- SentencePiece (Kudo et al., 2018): can do subword tokenization without pretokenization (good for languages that don't always separate words w/ spaces), although pretokenization usually improves performance
  - It works directly on text stream without notion of individual words

Subword Considerations

- **Multilingual Models**: Subword models are hard to use multilingually because they will over-segment less common languages naively (Ács 2019)

- Work-around: Upsample less represented languages in the training data, or use a language-specific tokenizer

Main takeaway: Tokenization

- **All pre-trained models use some kind of subword tokenization** with a tuned vocabulary; usually less 50k (but can be around 250k pieces for multilingual models)

- Still a number of **heuristics are needed** ...
  - Adjustments like restricting merges across categories are crucial for optimizing performance and vocabulary usage.

- Looking forward: learn tokenization jointly with the model (already some work on this, but still in early stages)

Questions?