Lecture 9: Introduction to Language

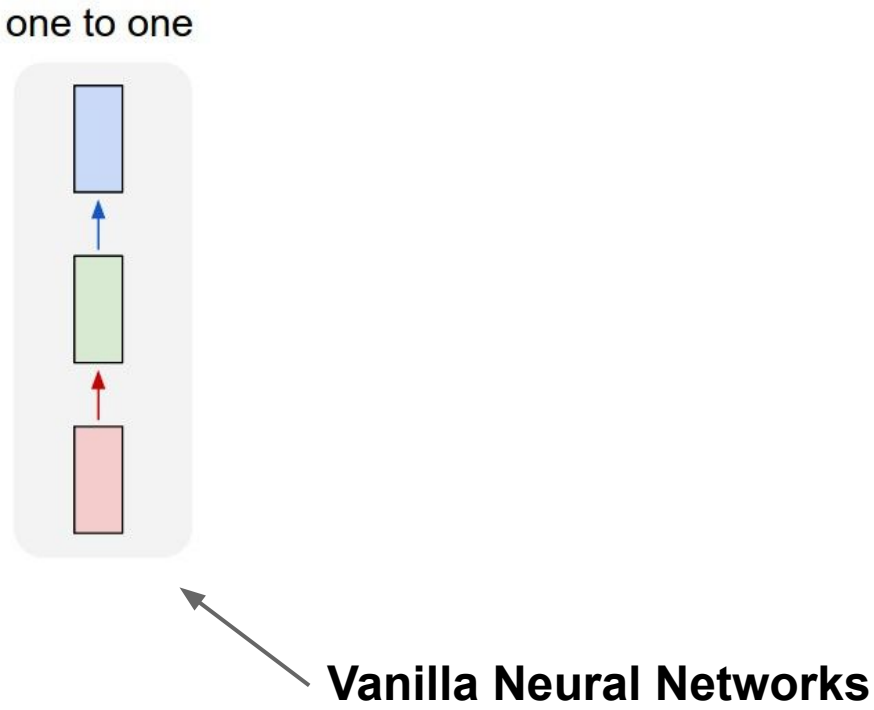
Administrative: Assignment 3

- Released! CNNs, Dropout, Norm Layers
- Due in a week

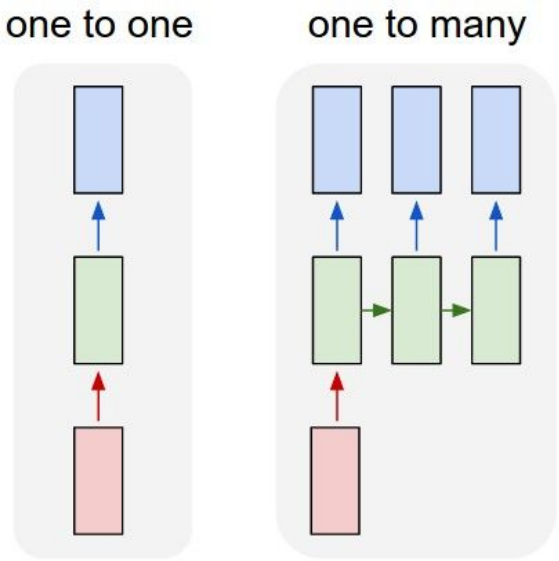
Recitation

- Efficient fine-tuning

“Vanilla” Neural Network

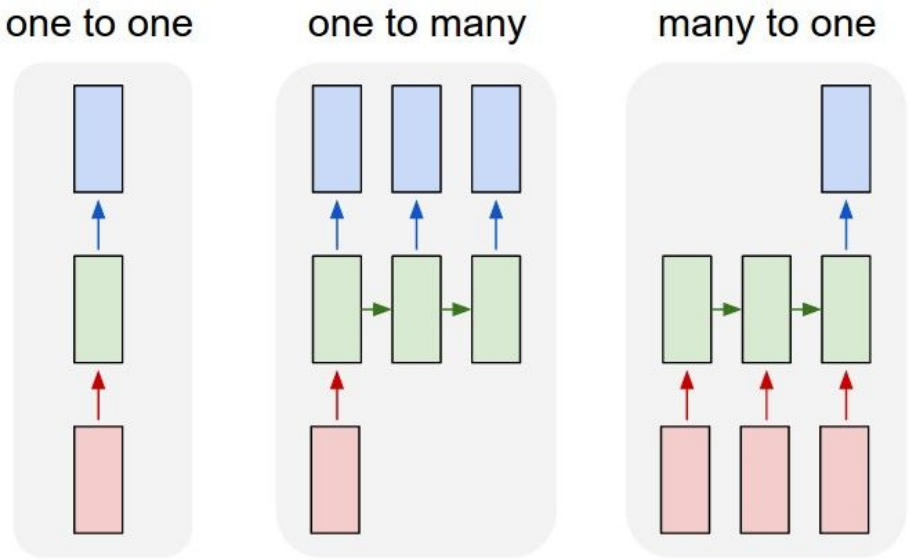


Recurrent Neural Networks: Process Sequences



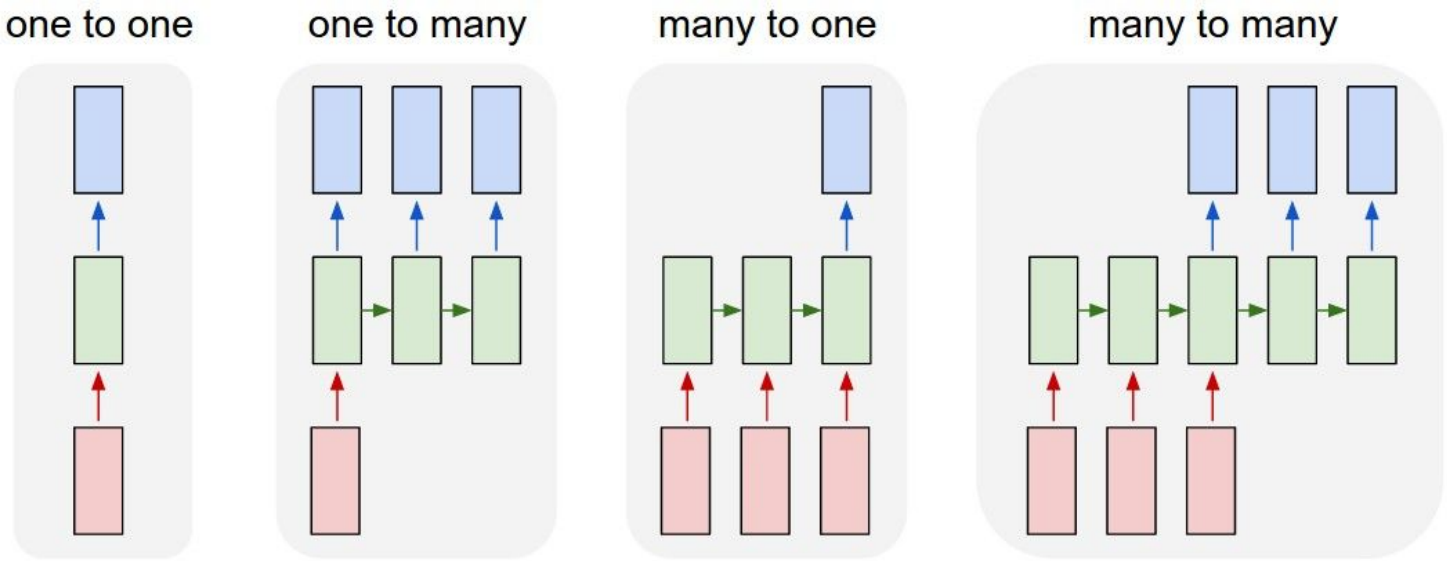
e.g. **Image Captioning**
image -> sequence of words

Recurrent Neural Networks: Process Sequences



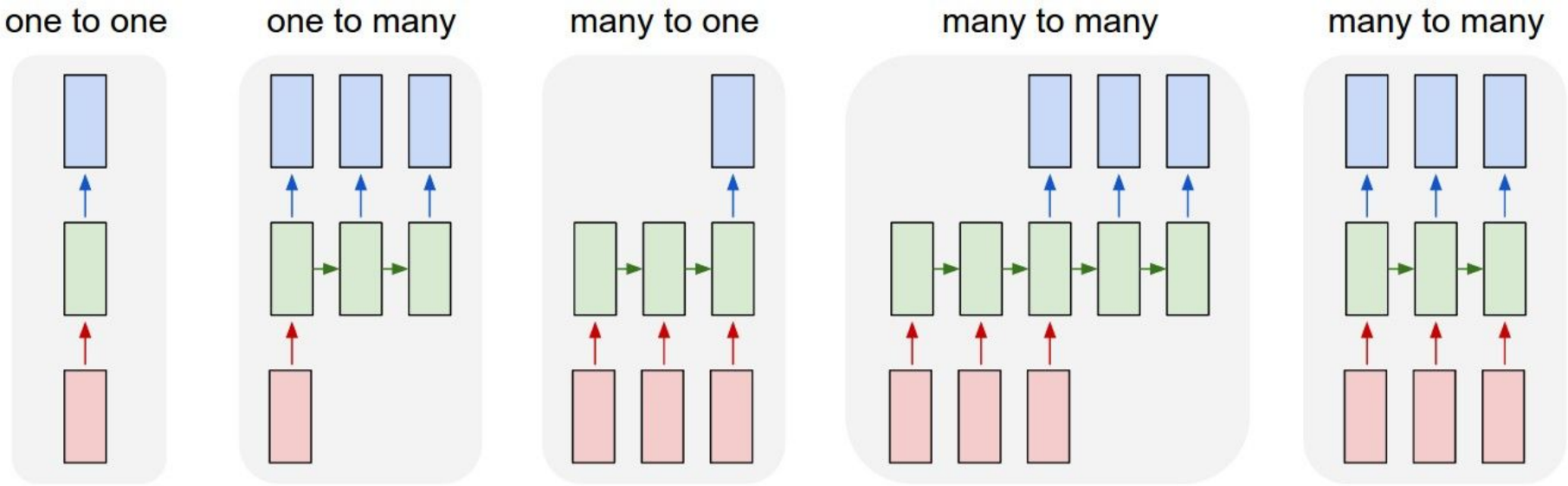
e.g. **action prediction, sentiment classification**
sequence of video frames -> action class

Recurrent Neural Networks: Process Sequences



E.g. **Video Captioning**
Sequence of video frames -> caption

Recurrent Neural Networks: Process Sequences



e.g. **Video classification on frame level**
Fill in the blanks with words

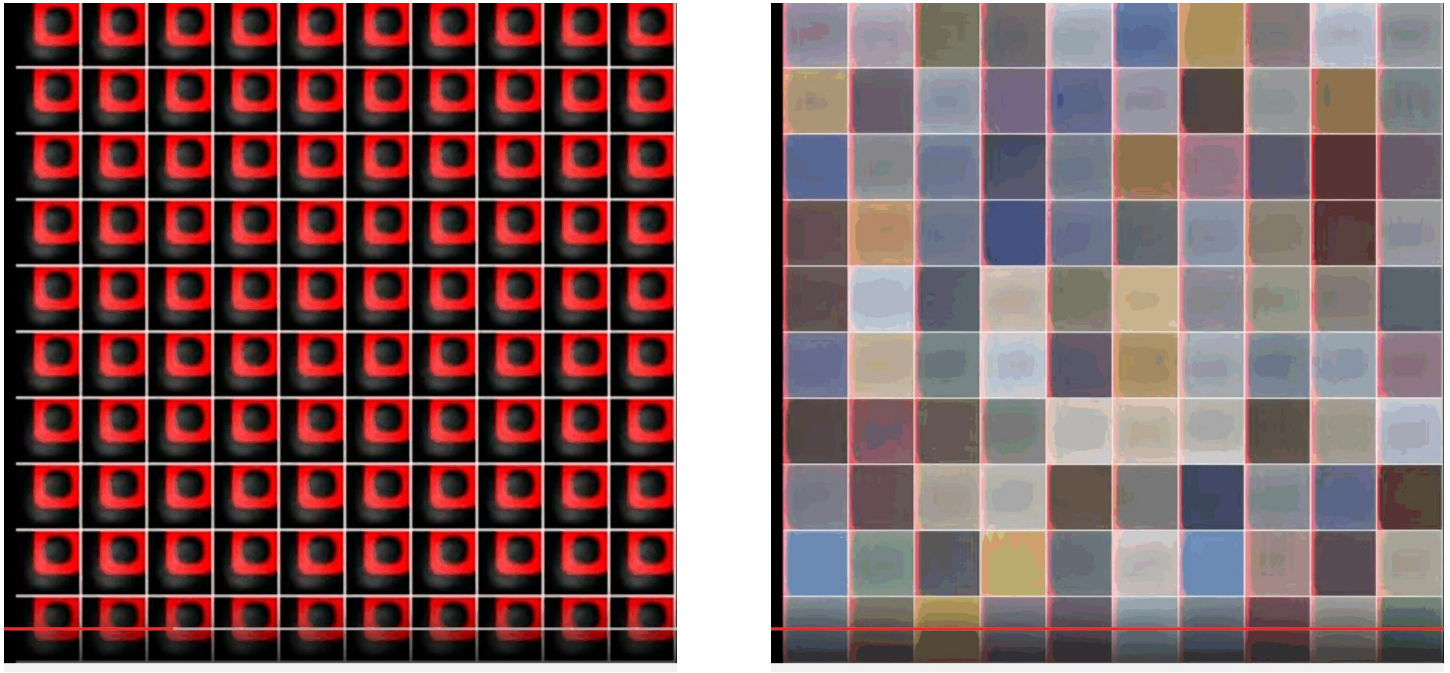
Sequential Processing of Non-Sequence Data

Classify images by taking a series of “glimpses”

Ba, Mnih, and Kavukcuoglu, “Multiple Object Recognition with Visual Attention”, ICLR 2015.
Gregor et al, “DRAW: A Recurrent Neural Network For Image Generation”, ICML 2015
Figure copyright Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra, 2015. Reproduced with permission.

Sequential Processing of Non-Sequence Data

Generate images one piece at a time!



Gregor et al, "DRAW: A Recurrent Neural Network For Image Generation", ICML 2015

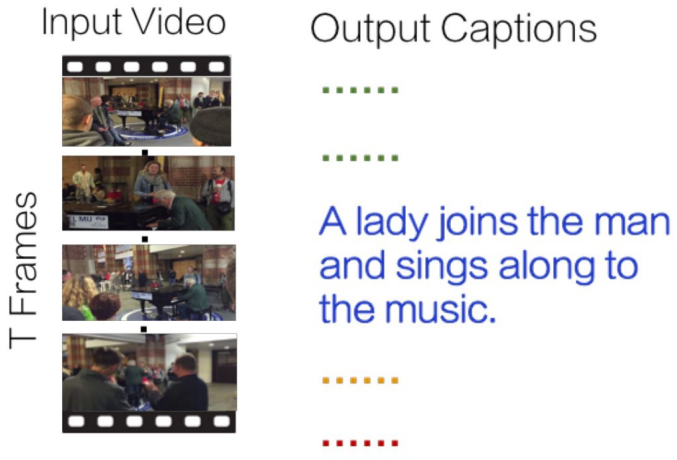
Why existing neural networks are insufficient?

Variable sequence length inputs and outputs!

Example task: video captioning

Input video can have variable number of frames

Output captions can be variable length.



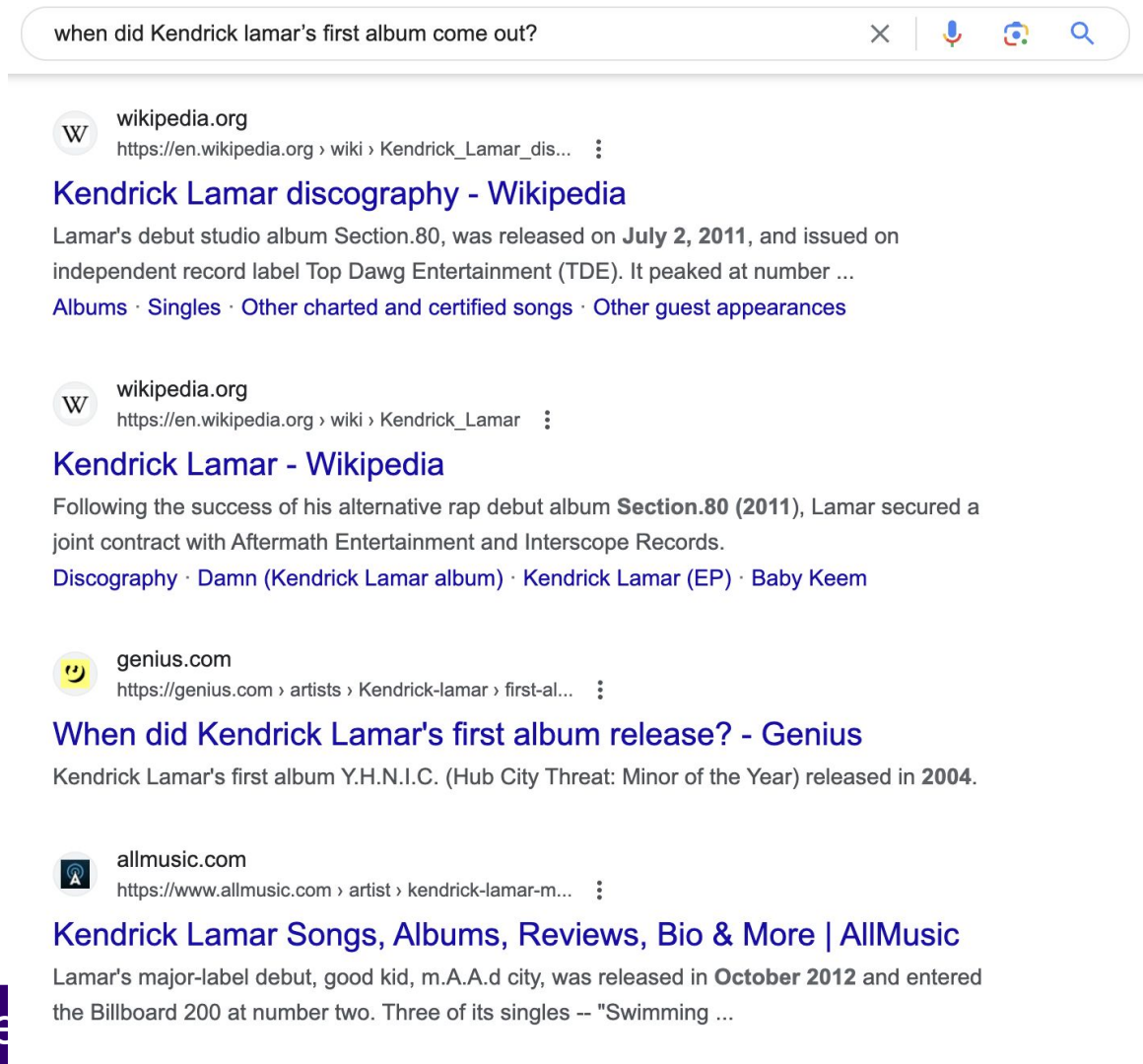
Krishna, et al. Dense captioning Events in Videos. ICCV 2019
Figure copyright Ranjay Krishna, Kenji Hata, Frederic Ren, Li Fei-Fei, and Juan Carlos Niebles, 2019. Reproduced with permission.

Today: Introduction to language

Language models are revolutionizing search

User: when did Kendrick lamar's first album come out?

GPT: July 2, 2011



Language models can learn to follow examples

S: I broke the window.
Q: What did I break?
S: I gracefully saved the day.
Q: What did I gracefully save?
S: I gave flowers to John.

What will GPT generate?

Language models can learn to follow examples

S: I broke the window.
Q: What did I break?
S: I gracefully saved the day.
Q: What did I gracefully save?
S: I gave flowers to John.
Q: Who did I give flowers to?

Language models can learn to follow examples

S: I broke the window.
Q: What did I break?
S: I gracefully saved the day.
Q: What did I gracefully save?
S: I gave John flowers.
Q: Who did I give flowers to?
S: I gave her a rose and a guitar.

What will GPT generate?

Language models can learn to follow examples

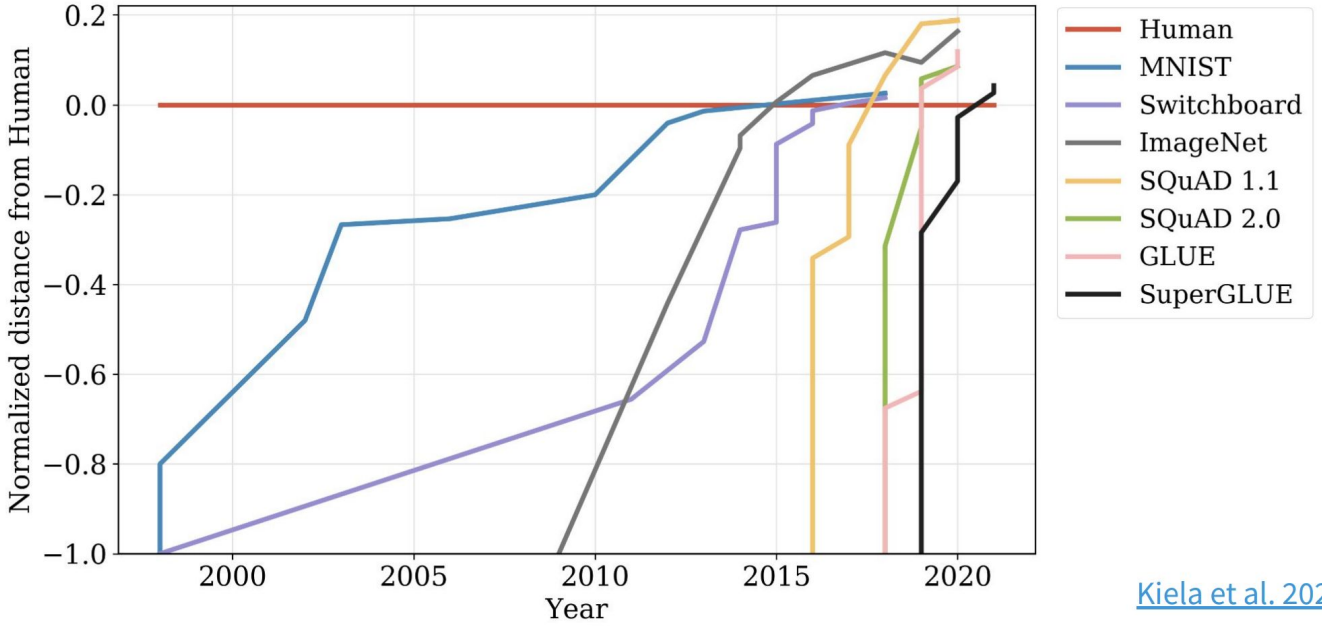
S: I broke the window.
Q: What did I break?
S: I gracefully saved the day.
Q: What did I gracefully save?
S: I gave John flowers.
Q: Who did I give flowers to?
S: I gave her a rose and a guitar.
Q: Who did I give a rose and a guitar to?

Language models can even write code / sql

How many users have signed up since the start of 2020?
SELECT count(id) FROM users WHERE created_at > '2020-01-01'

What is the average number of influencers each user is subscribed to?
SELECT avg(count) FROM (SELECT user_id, count(*) FROM subscribers GROUP BY user_id) AS avg_subscriptions_per_user

Compared to Vision (ImageNet, MNIST), language benchmarks reach super-human fast!



Kiela et al. 2021

Kiela et al. Dynabench: Rethinking Benchmarking in NLP NAACL 2021

How do we represent the meaning of a word?

Definition: meaning (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Common-est linguistic way of thinking of meaning:

signifier (symbol) ⇔ signified (idea or thing)

tree ⇔ {  ,  ,  , …}

How do we have usable meaning in a computer?

Previous solution by Linguists: Use a thesaurus (e.g., WordNet) containing lists of synonym sets and hypernyms ("is a" relationships).

e.g., synonym sets containing "good":

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
            ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
…
adverb: well, good
adverb: thoroughly, soundly, good
```

How do we have usable meaning in a computer?

Previous solution by Linguists: Use a thesaurus (e.g., WordNet) containing lists of synonym sets and hypernyms (“is a” relationships).

e.g., hypernyms of “panda”:

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

IMAGENET classes were derived from WordNet

22K categories and **14M** images

- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
- Food
- Materials
- Structures
- Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
- Sport Activities

Deng, Dong, Socher, Li, Li, & Fei-Fei, 2009

Problems with resources like WordNet

A useful resource but missing nuance:

- e.g., “proficient” is listed as a synonym for “good”
  This is only correct in some contexts
- Also, WordNet lists offensive synonyms without any coverage of the connotations or appropriateness of words

Missing new meanings of words:

- e.g., slop, cooked, cooking
- Impossible to keep up-to-date with Gen Z!

Subjective

- Requires human labor to create and adapt

Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols:

hotel, conference, motel – each has its own symbol.

This is a localist representation

Such symbols for words can be represented by one-hot vectors:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]

hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

Vector dimension = number of words in vocabulary (e.g., 500,000+)

Problem with words as discrete symbols

Example: in web search, if a user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”
But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]

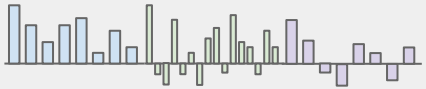
These two vectors are orthogonal
There is no natural notion of similarity for one-hot vectors!

New idea: learn better word representations that encode similarity in the vectors themselves

- Similar to deep learning representations for images!
- Can we learn representations for words?

Just as Neural networks extract linearly separate image features,
can we learn word features?

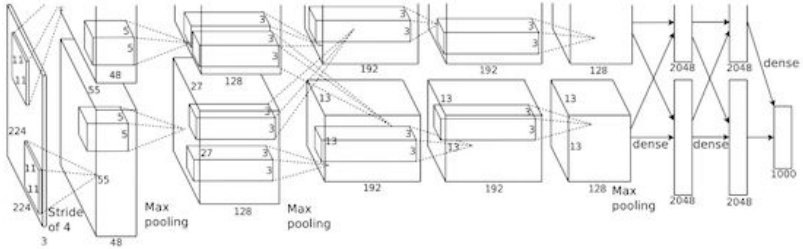
Feature Extraction



f

10 numbers giving scores for classes

training



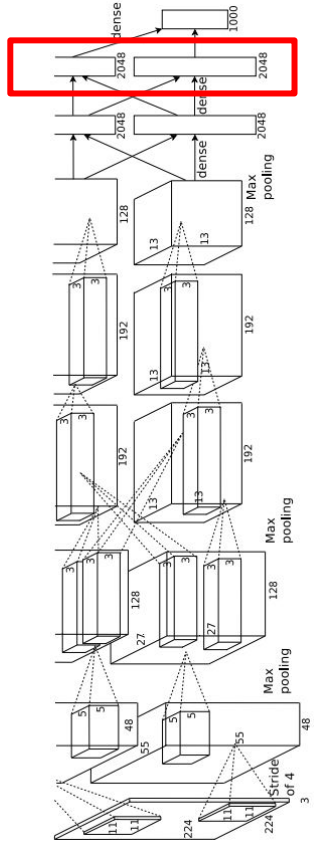
Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012. Figure copyright Krizhevsky, Sutskever, and Hinton, 2012. Reproduced with permission.

10 numbers giving scores for classes

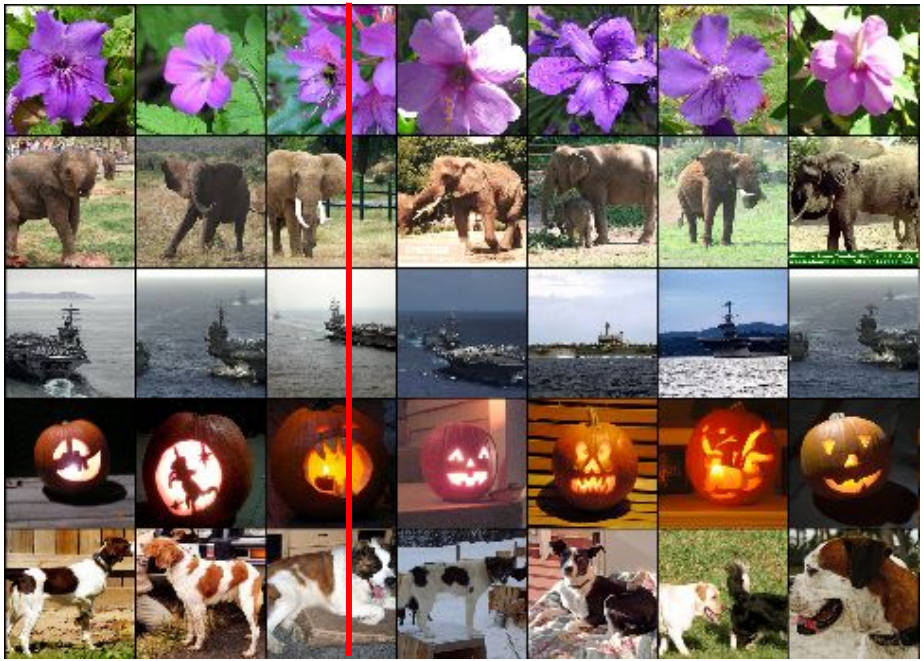
training

Last Layer: Nearest Neighbors

4096-dim vector



Test image L2 Nearest neighbors in feature space



Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012.
Figures reproduced with permission.

What we want in the end: word vectors

Want a dense vector for each word that captures the meaning of the word, similar words should have similar vectores

$$\text{word} = \begin{bmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{bmatrix}$$

Representing words by their context

Distributional semantics: A word's meaning is given by the words that frequently appear close-by

"You shall know a word by the company it keeps" (J. R. Firth 1957: 11)

One of the most successful ideas of modern statistical language modeling!

High level idea: a word *w* in text is a function of its contextual words.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a _____.

The _____ is a popular tourist attraction in Seattle.

I missed ___ bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The _____ is a popular tourist attraction in Seattle.

I missed ___ bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The _____ is a popular tourist attraction in Seattle.

I missed ___ bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The Space Needle is a popular tourist attraction in Seattle.

I missed ___ bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The Space Needle is a popular tourist attraction in Seattle.

I missed ___ bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The Space Needle is a popular tourist attraction in Seattle.

I missed the bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The Space Needle is a popular tourist attraction in Seattle.

I missed the bus.

I had 3 pencils and lost one so now I have _____ pencils.

Let’s try it: Fill in the blanks.

It’s cold today! Don’t forget to wear a jacket / coat / sweater.

The Space Needle is a popular tourist attraction in Seattle.

I missed the bus.

I had 3 pencils and lost one so now I have 2 / two pencils.

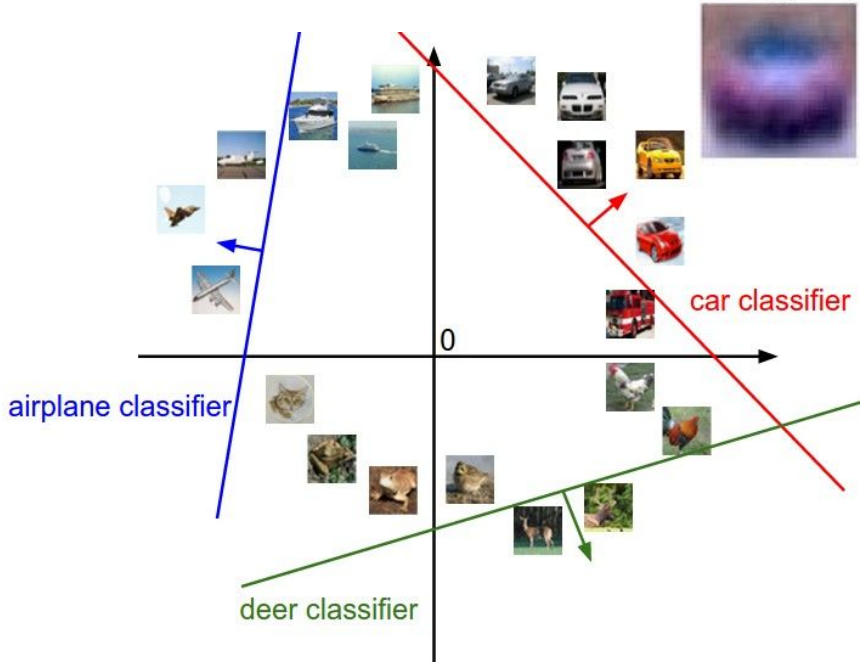
What we want in the end: word vectors

We will build a dense vector for each word,

- chosen so that it is similar to vectors of words that appear in similar contexts: e.g. jacket / coat / sweater.
- measuring similarity as the vector dot (scalar) product.
- Word vectors are also called (word) embeddings or (neural) word representations

$$\text{Jacket} = \begin{bmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{bmatrix}$$

Remember the **Geometric Viewpoint**

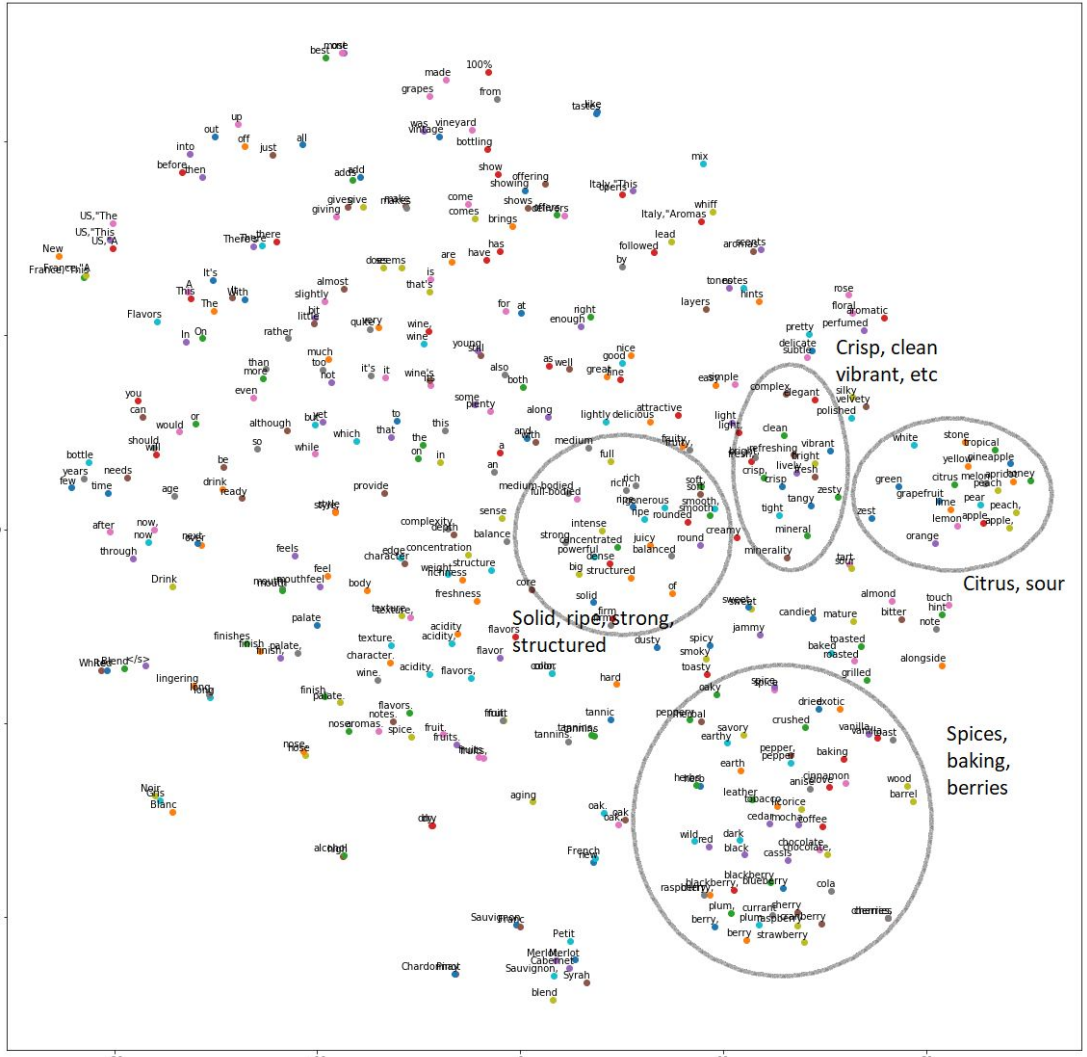


$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

Plot created using Wolfram Cloud

Visualizing word vectors in 2D (trained on wine reviews)



How do we train these word vectors: Word2vec

Jeff Dean's tweet

They won test of time award in December 2023!

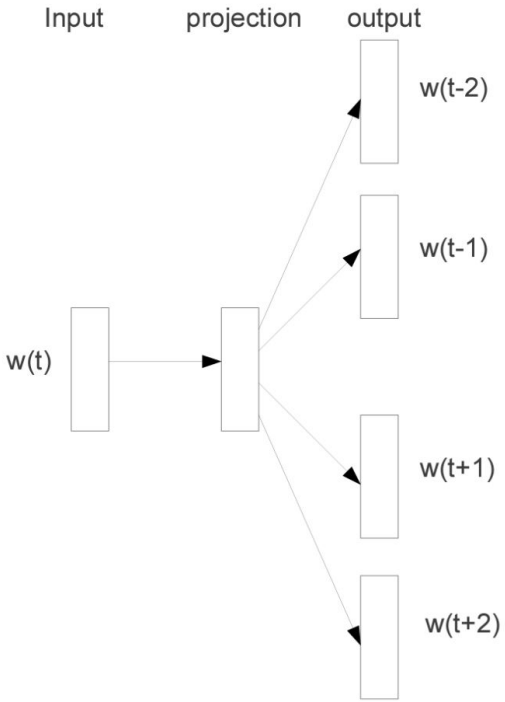
Original paper was rejected and never published (has 40K citations!)
Mikolov et al. Efficient Estimation of Word Representations in Vector Space" 2013

Next paper by same authors is what won the award and was published in NeurIPS 2013 (also 40K citations!)
Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems 26 (2013).

Word2vec is a framework for learning word vectors

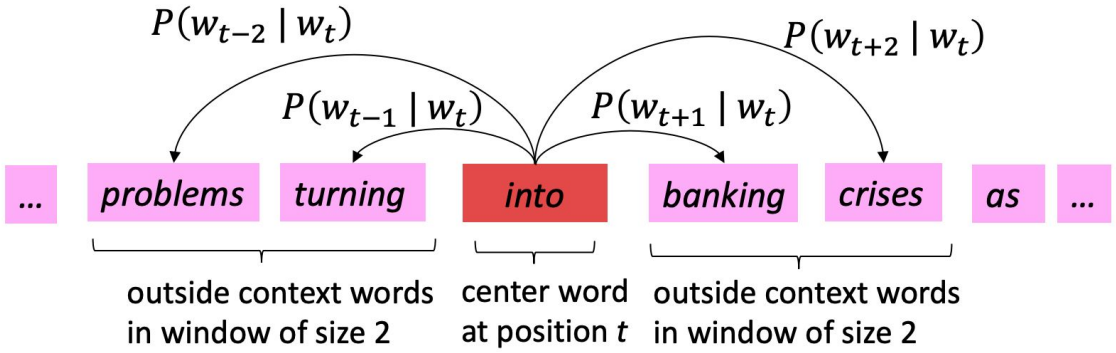
Idea:

1. Collect a large corpus (“body”) of text: a long list of words from the internet
2. Establish a fixed vocabulary of size V.
   a. Usually most frequent V words are used.
   b. All other words are mapped to an <unknown> or <unk> word.
3. Initialize every word by a random vector
4. Go through each position *t* in the text,
   a. Let the word in that position be *c*
   b. Let all (“outside”) words *o* be within context
   c. Calculate the probability of o given c (or vice versa)
   d. Maximize this probability



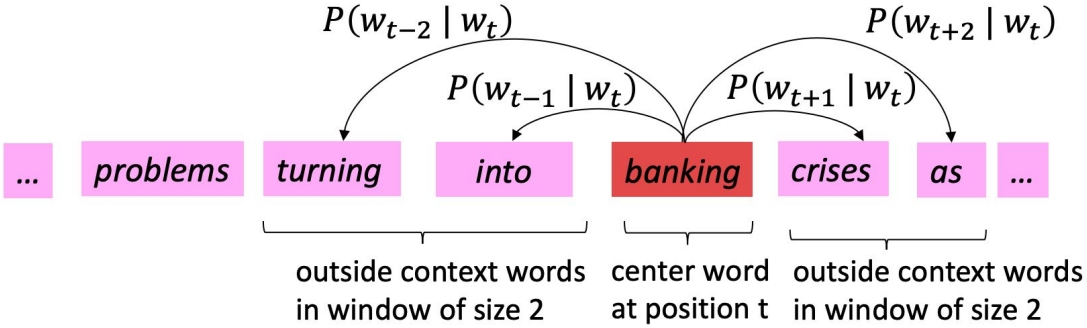
Using a context size of 2!

Example windows and process for computing $P(w_{t+j} \mid w_t)$



Move to the next word

Example windows and process for computing $P(w_{t+j} \mid w_t)$



The learning objective is to maximize the probabilities

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

$$\text{Likelihood} = \mathrm{M}(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

θ is all variables to be optimized

The loss function

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

$$\text{Likelihood} = \mathrm{M}(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$$

θ is all variables to be optimized

The loss function $L(\theta)$ is the (average) negative log likelihood:

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

Minimizing loss function ⇔ Maximizing predictive accuracy

How do we calculate this probability?

We want to minimize the objective 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 $P\left(w_{t+j} \mid w_t; \theta\right)$

How do we calculate this probability?

We want to minimize the objective function:

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

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

The best idea #1: Use a neural network whenever you need to model a function

How did Word2Vec design $f_j()$?

We will use two vectors per word *w*:

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

Let $f_1(w_t, w_{t+1}) = w_t^T w_{t+1}$ be a simple dot product

Then for a center word *c* and a context word *o*:

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

Remember the **Softmax (cross-entropy) Classifier**

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

$$L_i = -\log P(Y = y_i|X = x_i)$$

	Unnormalized log-probabilities / logits	exp	unnormalized probabilities	normalize	probabilities	compare	Correct probs
cat	**3.2**		**24.5**		**0.13**		**1.00**
car	5.1		164.0		0.87		0.00
frog	-1.7		0.18		0.00		0.00

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P||Q)$$

Understanding the calculation

② Exponentiation makes anything positive

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

① Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

③ Normalize over entire vocabulary to give probability distribution

To train the model: Optimize word vectors of all words to minimize loss using backprop

Recall: θ represents all the model parameters, in one long vector

- d-dimensional vectors
- V - many words,
- every word has two vectors

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

Use stochastic gradient descent with batches of N center words

Why did this algorithm use two vectors?

We will use two vectors per word *w*:

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

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

Hint: this paper came out in 2013, 1 year after AlexNet.

Why did this algorithm use two vectors?

We will use two vectors per word *w*:

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

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

Hint: this paper came out in 2013, 1 year after AlexNet.

Do you think the results would be better if they used 1 instead of 2 vectors?

What are the final word vectors?

It is the average of the two vectors:

$\frac{1}{2}(v_w + u_w)$

Variants of algorithm: The skip-gram model versus CBOW

Skip-gram predicts context words given center word.

CBOW (continuous bag of words) predicts center word given context words.

What's another issue with this calculation?

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

Solution: negative sampling

$$\log P(o|c) = -\log \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - \sum_{k \in \{K \text{ sampled indices}\}} \log \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)$$

Replace denominator with randomly sampled k vocabulary instances.
New hyperparameter: k

How should we sample?

- Rare words (aardvark) are unlikely to be helpful.
- You can sample a word *w* based on its probability of occurrence:
  - p(w) = U(w) / Z
  - where Z is total number of words
  - U(w) is the unigram frequency of word w (i.e., number of times it appears in the dataset).

Solution: negative sampling

$$\log P(o|c) = -\log \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - \sum_{k \in \{K \text{ sampled indices}\}} \log \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)$$

Replace denominator with randomly sampled k vocabulary instances.
New hyperparameter: k

How should we sample?

- Rare words (aardvark) are unlikely to be helpful.
- You can sample a word *w* based on its probability of occurrence:
  - p(w) = U(w) / Z
  - where Z is total number of words
  - U(w) is the unigram frequency of word w (i.e., number of times it appears in the dataset).

In practice, $p(w) = U(w)^{3/4} / Z$

Problem of sparse gradients

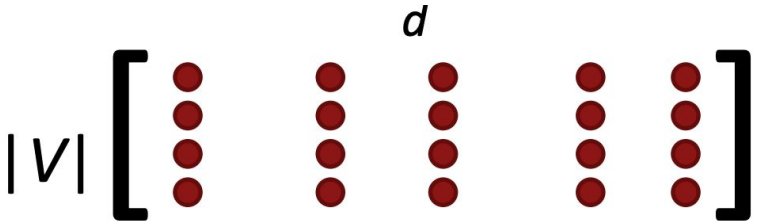
We iteratively take gradients at each *m* window for SGD

- In each window, we only have at most *2m + 1* words
- plus *2km* negative words with negative sampling
- so the gradient for each update is sparse for V of size 500K

$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Implementation detail

Most DL packages represent word vectors using a special embedding layer



Rows represent words

- even though we usually talk about words as column vectors
- In implementations, they are row vectors.
- It is a hash table with look up and write functions to avoid writing to the entire V x d matrix.

Indirectly, skip-gram is trying to calculate co-occurance of words

It does this using backprop and by iterating through the entire corpus of text data multiple times.

Can we do better?

Can we build a co-occurrence matrix directly?

- Calculate co-occurrence of words within a window.
- captures some syntactic and semantic information (“word space”)
- If window is too large (size of entire articles or documents):
  - Co-occurrence matrix will represent general topics
  - Example, all sports words will have similar entries

Example co-occurrence matrix

- Let's try an example with window length 1 (it is more common to use 5–10)
- Symmetric (irrelevant whether left or right context)

Example corpus:

- I like deep learning
- I like UW
- I enjoy class

counts	I	like	enjoy	deep	learning	UW	class	.
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
UW	0	1	0	0	0	0	0	1
class	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

Co-occurrence vectors

Problem with simple count co-occurrence vectors

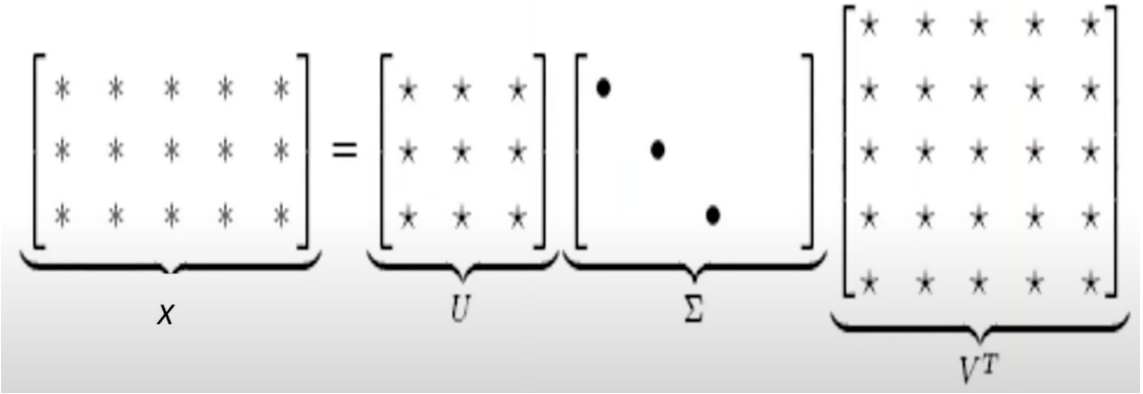
- Vectors increase in size with vocabulary
- Very high dimensional: require a lot of storage (though sparse)

Idea: Low-dimensional vectors

- Idea: store “most” of the important information in a fixed, small number of dimensions: a dense vector
- Usually 25–1000 dimensions, similar to word2vec
- **But how should we reduce the dimensionality from 500K to <1000?**

Classic Method: Dimensionality Reduction

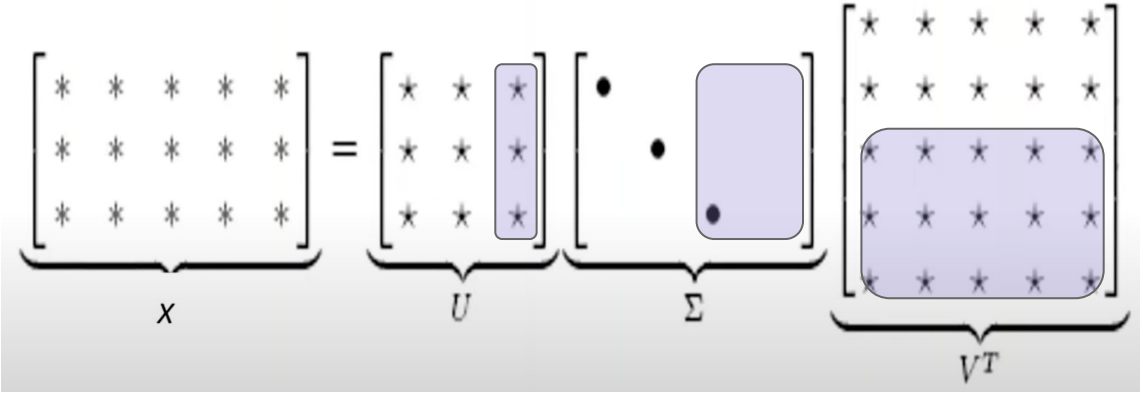
From linear algebra: Singular Value Decomposition of co-occurrence matrix X



U and V are orthogonal matrices
Σ is a diagonal matrix of singular values.

Classic Method: Dimensionality Reduction

From linear algebra: Singular Value Decomposition of co-occurrence matrix X



We can discard all except the largest d singular values and their corresponding multiplicative values in U and V
New d-dimensional word vector representations are: top-d(U) * top-d(Σ)

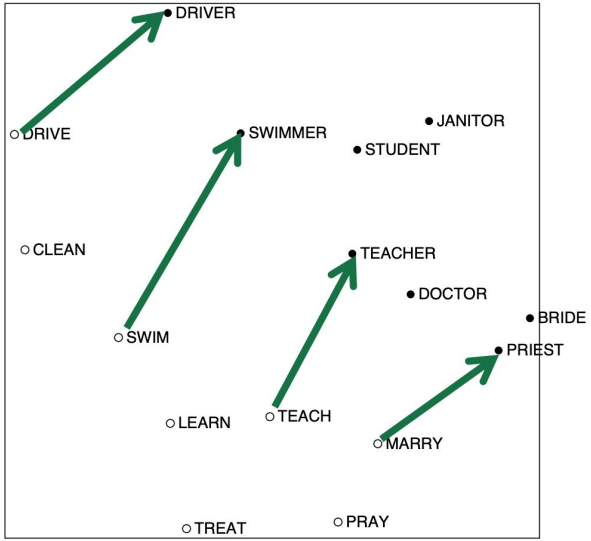
Making co-occurrence counts work

Running an SVD on raw counts doesn't work well.

Problem: function words (the, he, has) are too frequent à syntax has too much impact.
Some fixes:

- Use log the frequencies instead
- Limit the maximum values: min(X, t), with t ≈ 100
- Ignore the function words
- Ramped windows that count closer words more than further away words

Interesting semantic patterns emerge



COALS model from Rohde et al. ms., 2005. An Improved Model of Semantic Similarity Based on Lexical Co-Occurrence

Comparing the two methods:

Co-occurrence vectors

- Fast training
  - Single iteration over data
  - SVD is fast as long as vocabulary is reasonable.
- Good for capturing word similarities
- Needs hacks to work
- Not good for anything beyond similarities
- SVD is very slow for large vocabularies

Skip-gram algorithm

- Scales well with increasing vocabulary size
- Good for many other tasks
- Not as good for word similarities
- Needs multiple iterations across the dataset as backprop is slow

Can we combine the strengths of both methods?

Log bilinear model:

- Let every word be a d-dimensional vector
- Remember from skip-gram that dot product is the probability of one word given its context

$$w_i \cdot w_j = \log P(i|j)$$

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Can we combine the strengths of both methods?

Log bilinear model:

- Let every word be a d-dimensional vector
- Remember from skip-gram that dot product is the probability of one word given its context

$$w_i \cdot w_j = \log P(i|j)$$

Main idea: Similarity between two words should be proportional to their co-occurrence count.

- Log of count used as a hack

$$L(\theta) = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Can we combine the strengths of both methods?

Log bilinear model:

- Let every word be a d-dimensional vector
- Remember from skip-gram that dot product is the probability of one word given its context

$$w_i \cdot w_j = \log P(i|j)$$

Main idea: Similarity between two words should be proportional to their co-occurrence count.

- Log of count used as a hack

$$L(\theta) = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Can we combine the strengths of both methods?

Log bilinear model:

- Let every word be a d-dimensional vector
- Remember from skip-gram that dot product is the probability of one word given its context

$$w_i \cdot w_j = \log P(i|j)$$

Main idea: Similarity between two words should be proportional to their co-occurrence count.

- Log of count used as a hack

$$L(\theta) = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Can we combine the strengths of both methods?

Log bilinear model:

- Let every word be a d-dimensional vector
- Remember from skip-gram that dot product is the probability of one word given its context

$$w_i \cdot w_j = \log P(i|j)$$

Main idea: Similarity between two words should be proportional to their co-occurrence count.

- Log of count used as a hack
- f() is a threshold for large values

$$L(\theta) = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

GloVe [Pennington, Socher, and Manning, EMNLP 2014]: Encoding meaning components in vector differences

Word vectors are very good at analogies

a:b :: c:?

man:woman :: king:?

Word vectors are very good at analogies

a:b :: c:?

man:woman :: king:?

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

(man - woman + king)* w_i

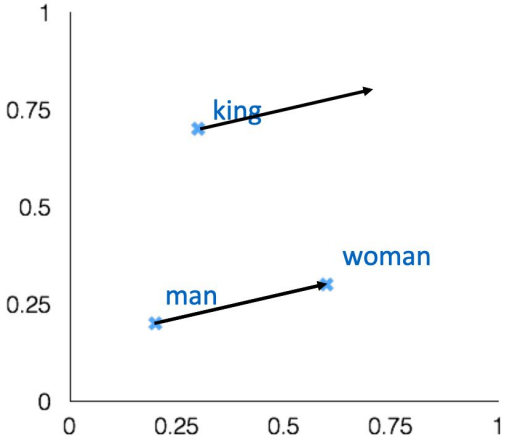
Word vectors are very good at analogies

a:b :: c:?

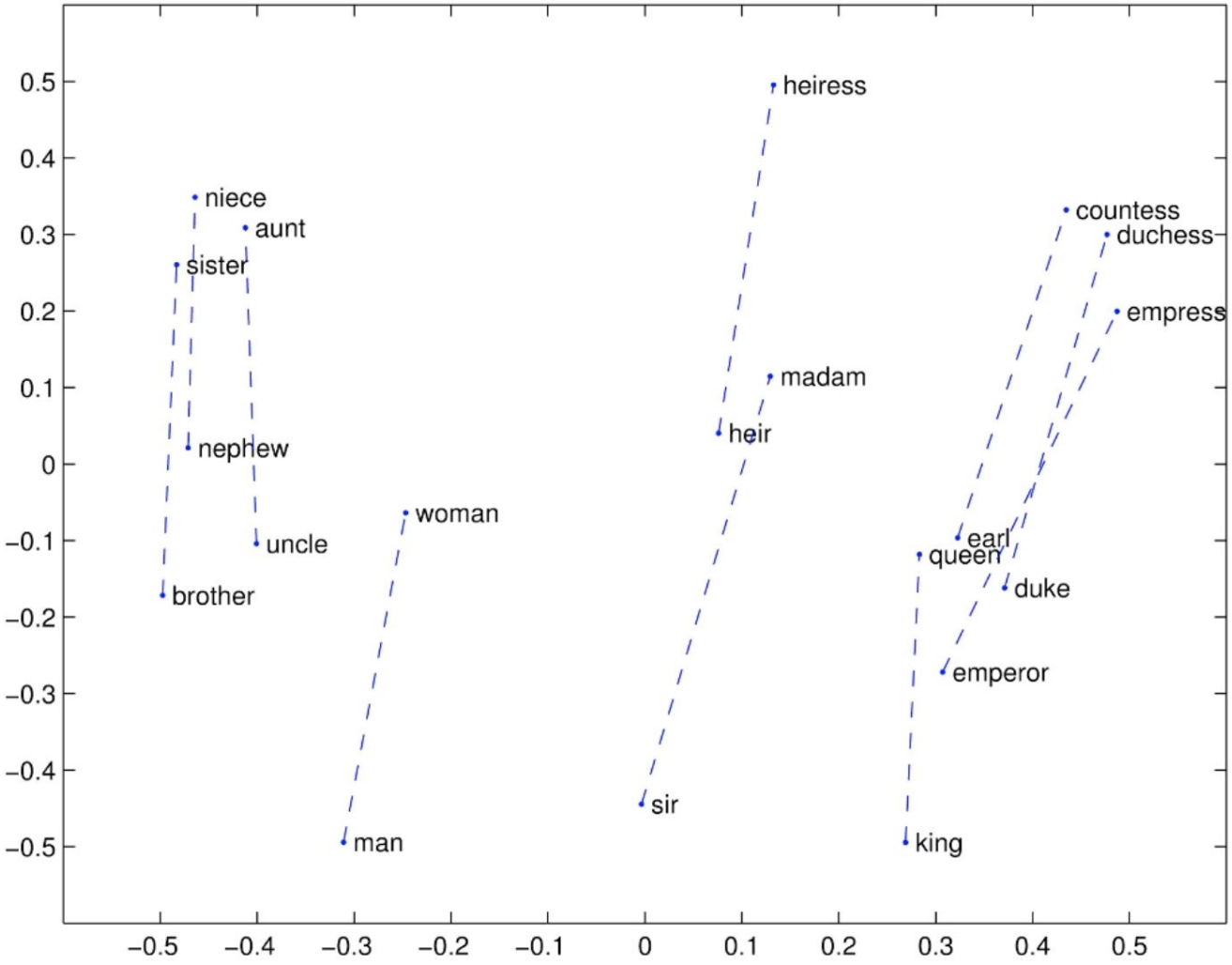
man:woman :: king:?

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

(man - woman + king)* w_i



More visualizations



Word vectors correlate with human judgement

Linguists have created a dataset of word similarity judgements

- Word vector distances and their correlation with human judgments
- Example dataset: WordSim353

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

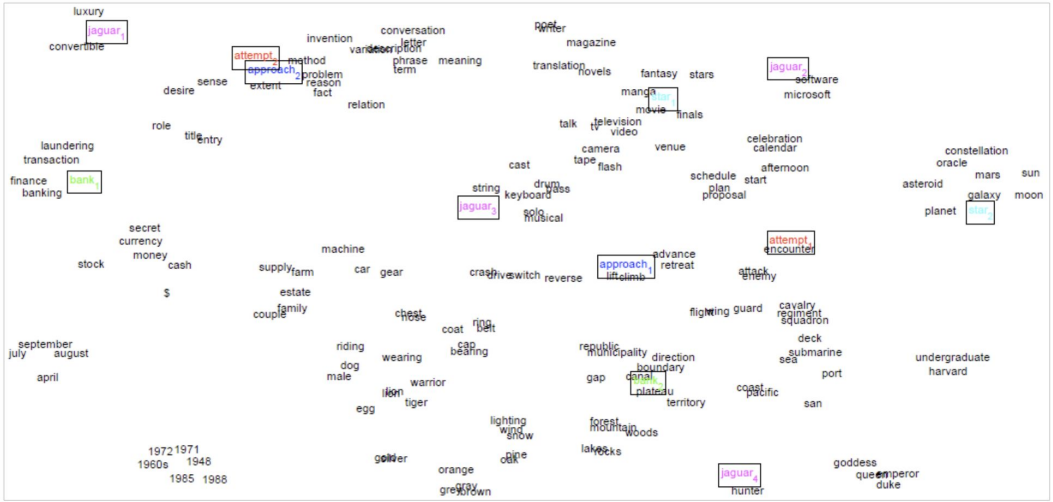
Problem of **polysemy**

Word senses and word sense ambiguity

Cap
- Using to only be similar to *Hat*
- But now, thanks to gen z, should also be closer to *Lying*

Can one vector capture all these meanings? Probably not!

Researchers have tried to segregate words into multiple vectors, each with its own meaning



But it doesn't work well. A word's usage in a sentence defines its meaning.

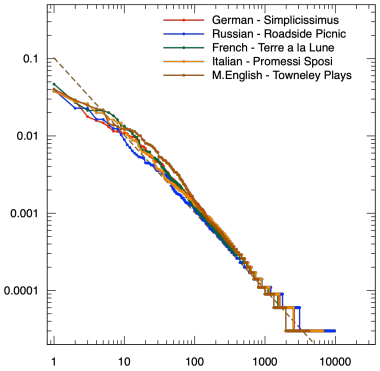
Words should be a function of not just its context but its position in the sentence

-> **Next lecture**

So far: vectors are associated with words

Our vocabulary was comprised of all of the **words in a language**
Problems:

- 500,000 words Webster's English Dictionary (3rd edition)
- Language is changing all of the time
  - 690 words were added to Merriam Webster's in September 2023 ("rizz", "goated", "mid")
- Long tail of infrequent words.
  - Zipf's law: word frequency is inversely proportional to word rank
- Some words may not appear in a training set of documents
- No modeled relationship between words - e.g., "run", "ran", "runs", "runner" are all separate entries despite being linked in meaning



Zipf's Law: Word Rank vs. Word Frequency for Several Languages

Character level vectors instead?

What about assigning a vector to every character instead?

(Maybe add capital letters, punctuation, spaces, …)
Pros:

- Small vocabulary size (for English)
- Complete coverage (unseen words are represented by letters)

Cons:

- Encoding a single sentence becomes very long!
  - # chars instead of # words
- Characters mean very different things in different words!
  - Even worse for representing multiple meanings

Subword tokenization!

How can we combine
1. the **high coverage** of character-level representations
2. with the efficiency of word-level representation?

Subword tokenization! (e.g., **Byte-Pair Encoding**)
- Start with character-level representations
- Build up representations from there

Original BPE Paper (Sennrich et al., 2016)

Example of how Byte-pair encoding works

Let's say our entire dataset contains only these 3 sentences:

$$\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$$

Example of how Byte-pair encoding works

Let's say our entire dataset contains only these 3 sentences:

$$\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$$

Initialize the vocabulary as all the individual characters. Current Vocab:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

Example of how Byte-pair encoding works

Let's say our entire dataset contains only these 3 sentences:

$$\mathcal{D} = \{\text{“i hug pugs”}, \text{“hugging pugs is fun”}, \text{“i make puns”}\}$$

Let's split it up into words by splitting right **before the whitespace:**

$$\mathcal{D} = \{\text{“i”}, \text{“ hug”}, \text{“ pugs”}, \text{“hugging”}, \text{“ pugs”}, \text{“ is”}, \text{“ fun”}, \text{“i”}, \text{“ make”}, \text{“ puns”}\}$$

Example of how Byte-pair encoding works

The vocabulary for reference:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

Let’s split it up into words by splitting right **before the whitespace:**

$$\mathcal{D} = \{\text{“i”}, \text{“ hug”}, \text{“ pugs”}, \text{“hugging”}, \text{“ pugs”}, \text{“ is”}, \text{“ fun”}, \text{“i”}, \text{“ make”}, \text{“ puns”}\}$$

Example of how Byte-pair encoding works

The vocabulary for reference:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

Let’s represent the dataset with only vocabulary elements:

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Example of how Byte-pair encoding works

The vocabulary for reference:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

Now, let’s find the most common bi-gram

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Example of how Byte-pair encoding works

The vocabulary for reference:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}\}, |\mathcal{V}| = 13$$

Now, let’s find the most common bi-gram

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘u’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘h’}, \text{‘u’}, \text{‘g’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘g’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Create new vocab:

$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$

Example of how Byte-pair encoding works

Update vocabulary with new vocab v_{14}:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}\}, |\mathcal{V}| = 14$$

Update dataset by replace bigram with new vocab v_{14}:

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Create new vocab:

$$v_{14} := \text{concat}(\text{‘u’}, \text{‘g’}) = \text{‘ug’}$$

Example of how Byte-pair encoding works

Current vocabulary:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}\}, |\mathcal{V}| = 14$$

Find the next common bigram:

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Example of how Byte-pair encoding works

Current vocabulary:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}\}, |\mathcal{V}| = 14$$

Find the next common bigram:

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ ’}, \text{‘p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ ’}, \text{‘p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Create new vocab:

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

Example of how Byte-pair encoding works

Update vocabulary with new vocab v_{15}:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’}, \text{‘ug’}, \text{‘ p’}\}, |\mathcal{V}| = 15$$

Update dataset by replace bigram with new vocab v_{15}:

$$\mathcal{D} = \{[\text{‘i’}], [\text{‘ ’}, \text{‘h’}, \text{‘ug’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}], [\text{‘h’}, \text{‘ug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ p’}, \text{‘ug’}, \text{‘s’}], [\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘u’}, \text{‘n’}], [\text{‘i’}], [\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘u’}, \text{‘n’}, \text{‘s’}]\}$$

Create new vocab:

$$v_{15} := \text{concat}(\text{‘ ’}, \text{‘p’}) = \text{‘ p’}$$

Repeat until vocab size reaches the amount you want (20 for example)

Final vocabulary:

$$\mathcal{V} = \{\text{‘ ’}, \text{‘a’}, \text{‘e’}, \text{‘f’}, \text{‘g’}, \text{‘h’}, \text{‘i’}, \text{‘k’}, \text{‘m’}, \text{‘n’}, \text{‘p’}, \text{‘s’}, \text{‘u’},$$
$$\text{‘ug’}, \text{‘ p’}, \text{‘hug’}, \text{‘ pug’}, \text{‘ pugs’}, \text{‘un’}, \text{‘ hug’}\},$$

Final dataset:

$$\mathcal{D} = \{ [\text{‘i’}], [\text{‘ hug’}], [\text{‘ pugs’}],$$
$$[\text{‘hug’}, \text{‘g’}, \text{‘i’}, \text{‘n’}, \text{‘g’}], [\text{‘ pugs’}],$$
$$[\text{‘ ’}, \text{‘i’}, \text{‘s’}], [\text{‘ ’}, \text{‘f’}, \text{‘un’}], [\text{‘i’}],$$
$$[\text{‘ ’}, \text{‘m’}, \text{‘a’}, \text{‘k’}, \text{‘e’}], [\text{‘ p’}, \text{‘un’}, \text{‘s’}]\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

Q: “map”?

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

Q: “map”?

Yes - [9,2,11]

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

Q: “map”?
Yes - [9,2,11]

Q: “huge”?

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

Q: “map”?
Yes - [9,2,11]

Q: “huge”?
Yes - [16, 3] or [6,14,3] or [6,13,5,3]

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

$$\mathcal{V} = \{1 : \text{‘ ’}, 2 : \text{‘a’}, 3 : \text{‘e’}, 4 : \text{‘f’}, 5 : \text{‘g’}, 6 : \text{‘h’}, 7 : \text{‘i’},$$
$$8 : \text{‘k’}, 9 : \text{‘m’}, 10 : \text{‘n’}, 11 : \text{‘p’}, 12 : \text{‘s’}, 13 : \text{‘u’},$$
$$14 : \text{‘ug’}, 15 : \text{‘ p’}, 16 : \text{‘hug’}, 17 : \text{‘ pug’}, 18 : \text{‘ pugs’},$$
$$19 : \text{‘un’}, 20 : \text{‘ hug’}\}$$

Q: “map”?
Yes - [9,2,11]

Q: “huge”?
Yes - [16, 3] or [6,14,3] or [6,13,5,3]

Q: “ huge” with a space in the front?

With this vocabulary, can you represent (or, tokenize/encode):

Q: Can you encode “apple”?

- No, there is no ‘l’ in the vocabulary

$$\mathcal{V} = \{1 : \text{`\ '}, 2 : \text{`a'}, 3 : \text{`e'}, 4 : \text{`f'}, 5 : \text{`g'}, 6 : \text{`h'}, 7 : \text{`i'},$$
$$8 : \text{`k'}, 9 : \text{`m'}, 10 : \text{`n'}, 11 : \text{`p'}, 12 : \text{`s'}, 13 : \text{`u'},$$
$$14 : \text{`ug'}, 15 : \text{`\ p'}, 16 : \text{`hug'}, 17 : \text{`\ pug'}, 18 : \text{`\ pugs'},$$
$$19 : \text{`un'}, 20 : \text{`\ hug'}\}$$

Q: “map”?
Yes - [9,2,11]

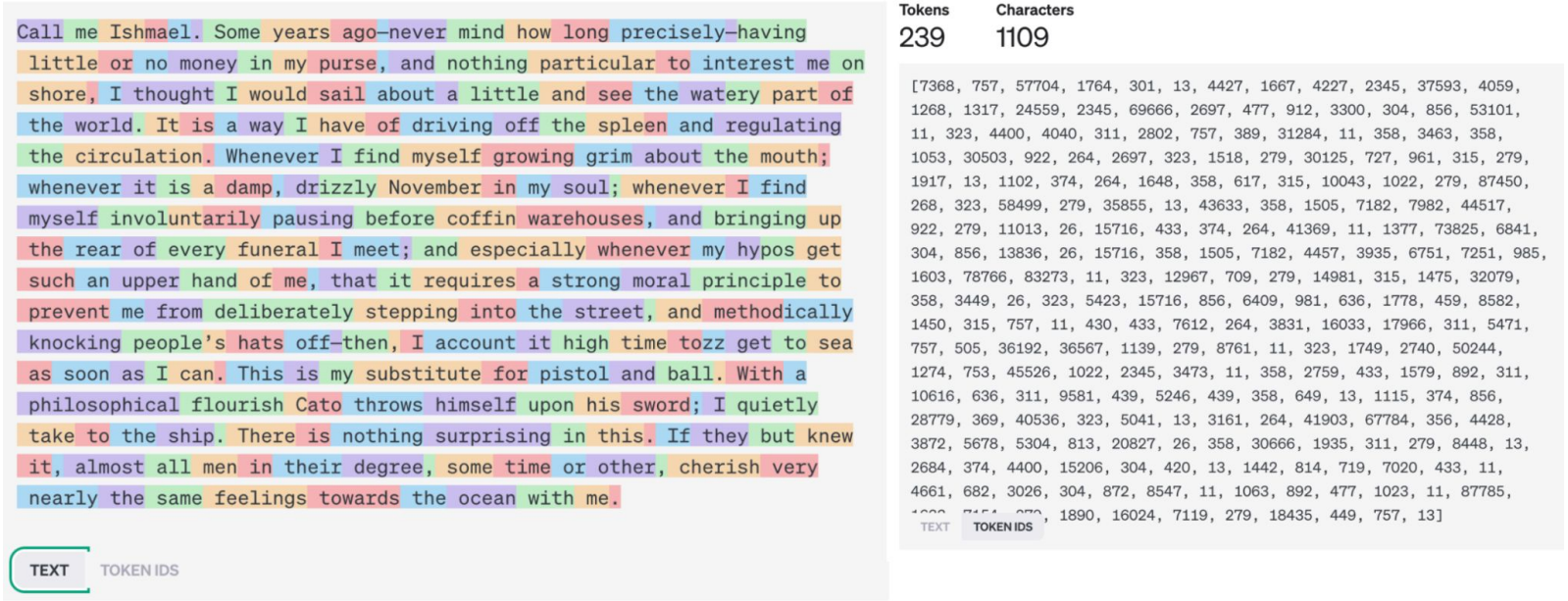
Q: “huge”?
Yes - [16, 3] or [6,14,3] or [6,13,5,3]

Q: “ huge” with a space in the front?
Yes - [20, 3]

Benefits of Byte-pair encoding

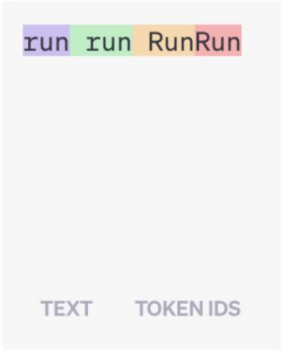
1. Efficient to run (greedy vs. global optimization)
2. Lossless compression
3. Potentially some shared representations
    a. e.g., the token “hug” could be used both in “hug” and “hugging”

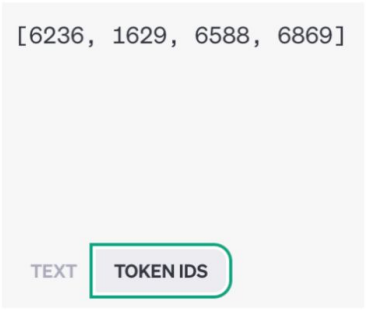
Byte-pair encoding - ChatGPT Example



Weird properties of tokenizers

Token != word
- Spaces are part of token
- “run” is a different token than “ run”
- Not invariant to case changes
- “Run” is a different token than “run”





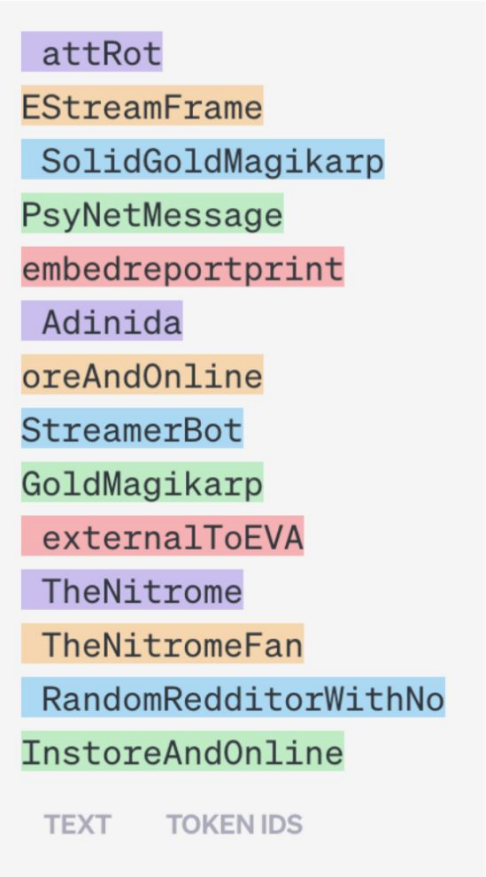
Weird properties of tokenizers

Token != word
- Spaces are part of token
- “run” is a different token than “ run”
- Not invariant to case changes
- “Run” is a different token than “run”
- Tokenization fits statistics of your data
  - e.g., while these words are multiple tokens…

Weird properties of tokenizers

Token != word
- Spaces are part of token
- “run” is a different token than “ run”
- Not invariant to case changes
- “Run” is a different token than “run”
- Tokenization fits statistics of your data
  - e.g., while these words are multiple tokens…

These words are all 1 token in GPT-3’s tokenizer!
Does anyone know why?



Next time:
RNNs & LSTMs