Natural Language Processing

2.1 Distributional Semantics, Latent Semantic Analysis and Word2Vec

Prof. Iacopo Masi and Prof. Stefano Faralli

In [1]:
```
import matplotlib.pyplot as plt
import scipy
import random
import numpy as np
import pandas as pd
pd.set_option('display.colheader_justify', 'center')
```

In [2]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
#plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]

def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

My own latex definitions

Very Brief Introduction

About Me

- **Associate Professor** with Sapienza since late 2020
- **Adjunct Research Assistant Professor** with University of Southern California (USC), Los Angeles **till August 2022**
- Worked as Research Scientist on big DARPA projects (Dept. of Defense) of USA.
- My Background:
  - Computer Vision
  - Machine Learning

I do research in AI

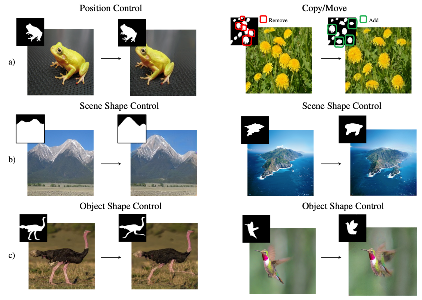
Biometrics, Face Recognition, Adversarial Robustness, Generative Models

Mainly visual domain (images) but I am broadening my range (e.g. NLP!)

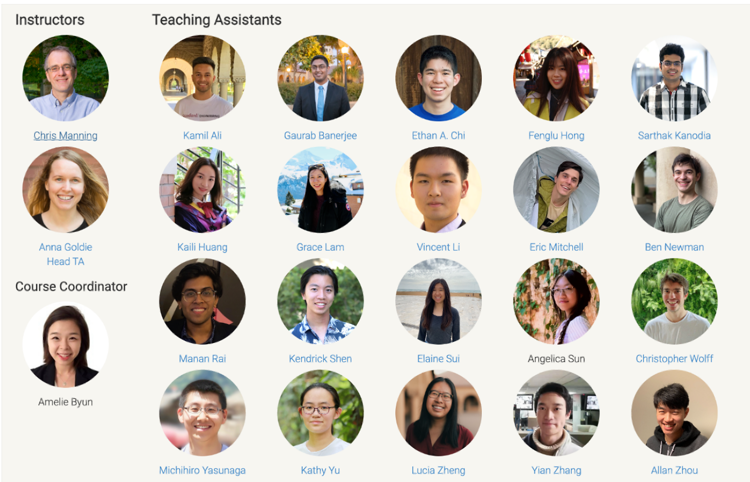
My Path

Latest Research Effort (w/ USC, UCLA, Sapienza)

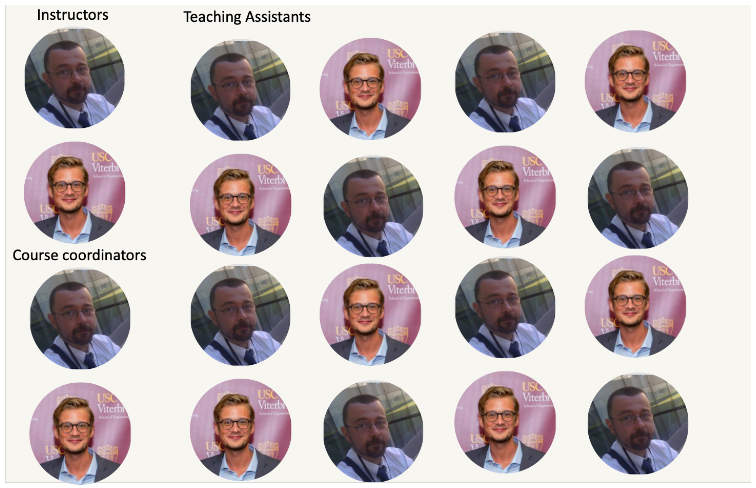
Image Synthesis by Inverting a Quasi-Robust Classifier in AAAI-23



Stanford NLP Course



Sapienza NLP Course



Today's lecture

- From Sparse to Distributed Representations

- Latent Semantic Analysis (LSA)

- Introduction to Word2Vec

This lecture material is taken from

📘 **Chapter 5 and 6 Jurafsky Book**

📘 **Chapter 14.5 Eisenstein Book**

- Stanford Slide Word2Vec
- Stanford Lecture Word2Vec
- Stanford Notes on Word2Vec

LSA part is taken from **Introduction to Information Retrieval by Manning**

Truncated SVD from sklearn

📝 Research papers on word2vec:

- First paper: word2vec + hierarchical softmax
- Negative Sampling paper

Word2vec and Distributed Representations

- Word2vec is a method for learning **distributed representations** of words (also called **word embeddings**).
- These representations capture the **meanings of words** in a continuous vector space, where similar words are mapped to nearby points.
- This allows words to be used in natural language processing tasks like **language translation** and **text classification**, in a way that reflects their meanings.

Word2vec and Distributed Representations

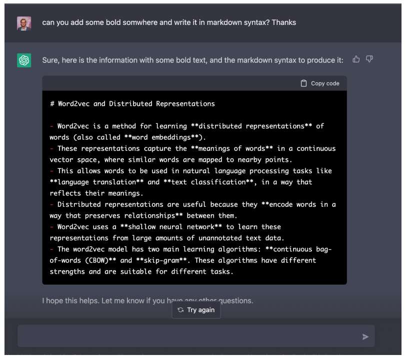
- Distributed representations are useful because they **encode words in a way that preserves relationships** between them.
- Word2vec uses a **shallow neural network** to learn these representations from large amounts of unannotated text data.
- The word2vec model has two main learning algorithms: **continuous bag-of-words (CBOW)** and **skip-gram**. These algorithms have different strengths and are suitable for different tasks.

...but there is a problem with the slides before

...does anyone spot any problem?

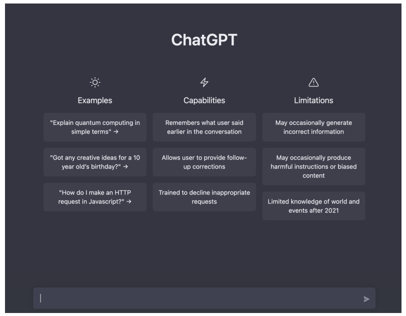
well, they have been generated by a **large NLP language model optimized for dialogue**

ChatGPT, if you do not believe me



To keep you motivated, let me entertain you...

Let's try latest NLP success ChatGPT **by OpenAI**



NLP

- Human (natural) language is different than Computer (Artificial) languages
- Human language is a system specifically constructed to **convey meaning**
- Vision is produced by a **physical manifestation of a signal**, **NLP is not**
- Human language uses symbols to indicate extra-linguistic entities

Word : Signifier = Idea : Signified

"tree" = {🌲,🌳,🌴,🌲,🏝️}

Stat rosa pristina nomine, nomina nuda tenemus

NLP models discrete symbols

- Vision and Audio are modeled as continuous signals that are then discretized (e.g. images)
- NLP instead is a **discrete/symbolic/categorical** domain.
  - This is why so far you have seen a lot of `count()`.
  - All the math involved in NLP is mostly **discrete math**.

Let's take a quick grasp of the difference between NLP and Vision

In [4]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('data/mario.png')
plt.figure(figsize=(7,7));
imgplot = plt.imshow(img)
plt.axis('off');
print(f'Image shape is {img.shape} --> HxWx(RGBA)');
```

```
Image shape is (120, 120, 4) --> HxWx(RGBA)
```

Let's sample from a English Corpus

In [3]:

```
# downloaded from https://www.nltk.org/nltk_data/
# https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/words.zip
with open('data/en-basic.txt') as fr: words = fr.read().split('\n')
```

```
print(*random.choices(words, k=100), sep=' ')
```

"Unigram" model (no use of frequency)

Sampling from a corpus of **#851 words**

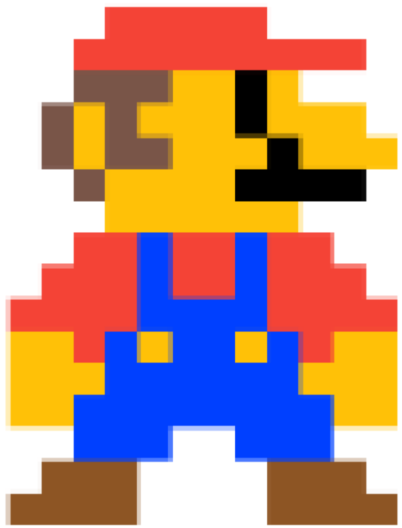
```
pot and enough deep secret rat thunder black industry answer death material angle crime probable
nation debt organization spade acid soup reward free circle west forward board bone substance
parcel south scissors move window hanging needle sticky pipe table old river attack design expert
between different bottle error poor grass group cut education voice prison peace comfort science
coat hanging writing pot news receipt street sky desire any first education damage warm the fire
property help tray lead east do normal early opposite tin history death you measure blue
attention breath pleasure so industry basket harbour yesterday moon will system
```

The text **globally does not make sense** but we still understand each word.

At times, **a few parts seem almost meaningful**:

- `and enough deep secret`
- `warm the fire property`
- `so industry basket`

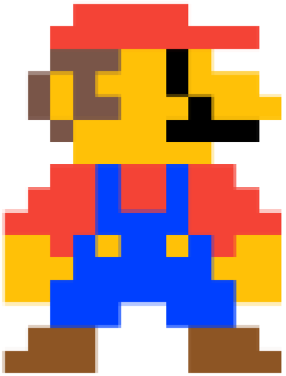
now with Vision



Vision

Mario below is an instance in space of $\mathbb{Z}^{120\times120\times3}_{[0,255]}$, if we do not consider the alpha channel.

- each pixel takes value in $[0, 255]$
- we have HxWx3 cell grid to fill

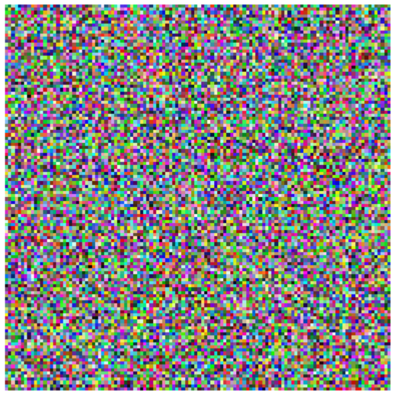


Let's sample randomly in the "visual" space

```
import numpy as np
rand_im = random.choices(range(0, 256), k=120*120*3)
rand_im = np.array(rand_im).reshape(120, 120, 3)
imgplot = plt.imshow(rand_im)
plt.axis('off');
```

What would you expect?

```
import numpy as np
rand_im = random.choices(range(0, 256), k=120*120*3)
rand_im = np.array(rand_im).reshape(120, 120, 3)
plt.figure(figsize=(5,5))
imgplot=plt.imshow(rand_im)
_=plt.axis('off')
```



Much more non-sense than with text, right?

Vision

- No notion of discrete symbols. What is the semantic of a pixel?
- Continuous signal that is discretized on a grid

NLP meets Vision will return at the end of the course

A bit of Linguistics to start

Definition of mouse from a dictionary:

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

- **mouse** is called a **lemma or citation form**; for verbs usually the **lemma** is the infinite form
- **mice** is a **wordform** (in this case the, irregular, plural form of mouse)

Word Sense

Multiple aspects of the meaning of a word

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

without **context** it is difficult to say if `mouse` refers to `1.` or `2.`

Word Sense Disambiguation

Synonym

- One important component of **word meaning** is the relationship between word senses.
- For example when **one word has a sense whose meaning** is *identical* to a **sense of another word** we say the two senses of those two words are **synonyms**.

Word A	Word B
couch	sofa
car	automobile
hazelnut	filbert

Word Similarity: what this lecture is about

- While words do not have many synonyms, most words do have lots of similar words
- Cat is not a synonym of dog, but **cats and dogs are certainly similar words** (for sure more than **cat vs airplane**)
- The notion of **word similarity** is very useful in larger semantic tasks

---

vanish | disappear | 9.8 belief | impression | 5.95 muscle | bone | 3.65 modest | flexible | 0.98 hole | agreement | 0.3

Representing Words as Discrete Symbols: One-Hot Encoding

In traditional NLP, we regard words as discrete symbols.

`hotel, conference, motel` can be represented with a vector $\mathbf{x}$:

- of dimension $|V|$
- each component indexes a word in the vocabulary V
- if we are dealing with a word present in V at index i then:

$$\mathbf{x}_j = \begin{cases} 1, & \text{if } i = j \\ 0 & \forall j \neq i \end{cases}$$

visually this is something as:

0	0	0	0	1	...	0
cat	airplane	dog	beer	hotel	....	garden

Representing Words as Discrete Symbols: One-Hot Encoding

In traditional NLP, we regard words as discrete symbols.

`hotel, conference, motel` can be represented with a vector $\mathbf{x}$:

- of dimension $|V|$
- each component indexes a word in the vocabulary V
- if we are dealing with a word present in V at index i then:

$$\mathbf{x}_j = \begin{cases} 1, & \text{if } i = j \\ 0 & \forall j \neq i \end{cases}$$

More formally:

$$\mathbf{x}_{\text{hotel}} = \underbrace{[0\,0\,0\,0\,1\,0\,0 \ldots 0\,0\,0\,0\,0\,0]}_{|V|}$$

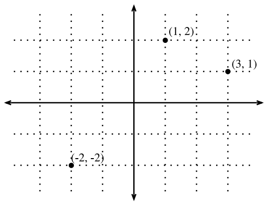
Is the same as saying that the word `hotel` appears at index 4 in the vocabulary (starting from zero)

Brief recap on Linear Algebra

Vectors: Geometric Interpretation 1

Point in space

- Given a vector, the first interpretation that we should give it is as a **point in space.**
- In two or three dimensions, we can visualize these points by using the components of the vectors to define the location of the points in space compared to a fixed reference called the *origin*.



Formalizing problems

This geometric point of view allows us to consider the problem on a more abstract level. No longer faced with some insurmountable seeming problem like classifying pictures as either cats or dogs but separate points in space.

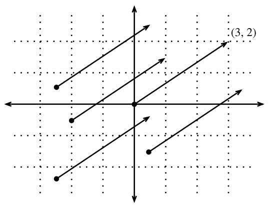
`Problem -> Formalization -> Math -> Computational System`

Taken from d2l.ai

Vectors: Geometric Interpretation 2

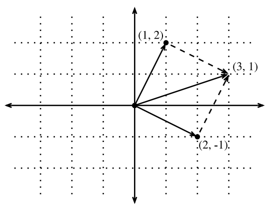
Direction in space

In parallel, there is a second point of view that people often take of vectors: **as directions in space.** Not only can we think of the vector $\mathbf{v} = [3, 2]^\top$ as the location 3 units to the right and 2 units up from the origin, we can also think of it as the direction itself to take 3 steps to the right and 2 steps up. In this way, we consider all the vectors in figure the same.



Direction in space

One of the benefits of this shift is that we can make visual sense of the act of vector addition. In particular, we follow the directions given by one vector, and then follow the directions given by the other, as is seen below.



Difference is just $\mathbf{A} - \mathbf{B} = \mathbf{A} + (-\mathbf{B})$

Taken from [d2l.ai]

Inner Product (Dot Product)

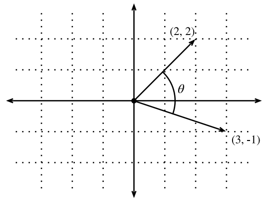
$$\mathbf{x}, \mathbf{y} \in \mathbb{R}^D \qquad \mathbf{x}^T\mathbf{y} = \langle \mathbf{x}, \mathbf{y} \rangle = \sum_i^D \mathbf{x}_i \cdot \mathbf{y}_i \tag{1}$$

```
x1 x2 x3 x4
           y1
           y2   =    result (dot_product)
           y3
           y4


dot_product = x1y1 + x2y2 + x3y3 + x4y4
```

- The result is a **scalar** (not a vector anymore).
- $\mathbf{x}, \mathbf{y}$ must live in the same dimension
- It is **commutative**
- The data is **paired**: just muliply elementwise and sum across axis.

Inner product: Geometric Interpretation

- The dot product also admits a geometric interpretation: **it is closely related to the angle between two vectors.**
- It is biased by the norm ("length") of the vectors (it favors "long" vectors)



$$\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\|\|\mathbf{w}\| \cos(\theta).$$

With some simple algebraic manipulation, we can rearrange terms to obtain

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right).$$

This is a nice result since nothing in the computation references two-dimensions.

Indeed, we can use this in three or three million dimensions without issue.

Cosine Similarity

In ML contexts where the **angle is employed to measure the closeness of two vectors**, practitioners adopt the term **cosine similarity** to refer to the portion

$$\cos(\theta) = \underbrace{\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}}_{\text{cosine similarity}}.$$

- What happens if cosine similarity is 1?
- Notice any similarity with some concept we saw in previous lecture?

The problem of Dot Product and One-Hot Encoding

Assume a vocabulary of words V

$$\mathbf{x}_{\text{motel}} = \begin{bmatrix} 0 \\ 1 \\ \ldots, \\ 0 \end{bmatrix} \quad \mathbf{x}_{\text{hotel}} = \begin{bmatrix} 0 \\ 1 \\ \ldots, \\ 0 \end{bmatrix} \quad \ldots \quad \mathbf{x}_{\text{car}} = \begin{bmatrix} 0 \\ 1 \\ \ldots, \\ 0 \end{bmatrix}$$

$$\mathbf{x}_{\text{motel}}^T \mathbf{x}_{\text{hotel}} = \mathbf{x}_{\text{motel}}^T \mathbf{x}_{\text{car}} = 0$$

There is no notion of similarity and vectors are very long--size of $|V|$.

The Distributional Hypothesis

Stated by J. R. Firth (1957) as:

> "You shall know a word by the company it keeps"

Distributional statistics have a striking ability to capture lexical semantic relationships such as **analogies**.

Distributional statistics and unsupervised learning

The distributional hypothesis has stood the test of time:

- distributional statistics are a core part of language technology today
- they make it possible to leverage large amounts of **unlabeled data** to learn about rare words that do not appear in labeled training data

The meaning of the word `tezguino`

1 Introduction

The meaning of an unknown word can often be inferred from its context. Consider the following (slightly modified) example in (Nida, 1975, p.167):

(1) A bottle of *tezgüino* is on the table.
Everyone likes *tezgüino*.
Tezgüino makes you drunk.
We make *tezgüino* out of corn.

The context plays a big role

We evince that `tezguino` may be an **alcoholic beverage made from corn mash.**

1 Introduction

The meaning of an unknown word can often be inferred from its context. Consider the following (slightly modified) example in (Nida, 1975, p.167):

(1) A bottle of *tezgüino* is on the table.
Everyone likes *tezgüino*.
Tezgüino makes you drunk.
We make *tezgüino* out of corn.

Long, Sparse

- One-Hot Encoding
- TF-IDF
- Term-Document matrix
- PPMI (Point-wise Mutual Information)

Short, Dense

- Latent Semantic Analysis (LSA)
- word2vec

Long, Sparse: Term-Document Matrix

```
corpus = [
    'This document is the second document.', #document 0
    'And this is the third one.',            #document 1
    'Is this the first document?',           #document 2
]
```

In [6]:

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = [
    'This document is the second document.', #0
    'And this is the third one.',            #1
    'Is this the first document?',           #2
]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
td_matrix=pd.DataFrame(data=X.todense(),
             index=[f'doc-{i}' for i in range(X.shape[0])],
             columns=vectorizer.get_feature_names_out(),)
```

Term-Document Matrix

$|V| = 9$ is the vocabulary size and $|D| = 3$ is the number of documents

$$\text{count}(t, d) \quad t \in V$$

	and	document	first	is	one	second	the	third	this
doc-0	0	2	0	1	0	1	1	0	1
doc-1	1	0	0	1	1	0	1	1	1
doc-2	0	1	1	1	0	0	1	0	1

You can interpret it two ways:

1. $|D| = 3$ documents vectors that lives in a $|V|$ dimensional space (row-wise)
   - An axis of the space indicates a **word**
2. V word vectors that lives in a $|D| = 3$ dimensional space (column-wise)
   - An axis of the space indicates a **document**

Long, Sparse: TF-IDF Matrix

In [7]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = [
    'This document is the second document.', #0
    'And this is the third one.',            #1
    'Is this the first document?',           #2
]
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
tfidf_matrix=pd.DataFrame(data=X.todense(),
             index=[f'doc-{i}' for i in range(X.shape[0])],
             columns=vectorizer.get_feature_names_out(),)
```

TF-IDF Matrix

$$\text{tf}(t,d) = \log_{10}\left(\text{count}(t,d)+1\right) \quad \text{idf}(t) = \log_{10}\left(\frac{|D|}{\text{df(t)}}\right) \quad df(t) = \sum_{d=1}^{|D|} 1[t \in d]$$

$$\text{w}(t,d) \doteq \text{tf}(t,d) \cdot \text{idf}(t)$$

	and	document	first	is	one	second	the	third	this
doc-0	0.00000	0.728445	0.000000	0.282851	0.00000	0.478909	0.282851	0.00000	0.282851
doc-1	0.49712	0.000000	0.000000	0.293607	0.49712	0.000000	0.293607	0.49712	0.293607
doc-2	0.00000	0.469417	0.617227	0.364544	0.00000	0.000000	0.364544	0.00000	0.364544

How do we go Short and Dense?

How do we go Short and Dense?

> "You shall know a word by the company it keeps"

Word embeddings (dense) are estimated by **optimizing some objective**: the reconstruction of a matrix of context count or the likelihood of a set of unlabeled data.

1. **[Latent Semantic Analysis (LSA)]** Factorize your count matrix $\mathbf{X}$ by minimizing its reconstruction $||\mathbf{X} - \mathbf{X}(\mathbf{u}, \mathbf{v})||_F$ and learning $\mathbf{u}$ document embedding and $\mathbf{v}$ word embedding
2. **[Word2vec]** we optimize the likelihood-based optimization of a model that encodes word embeddings over a corpus of training data with self-supervision.

Latent Semantic Analysis (LSA)

Sometimes it is called *Latent Semantic Indexing (LSI)*

Latent Semantic Analysis (LSA)

It is related to different concept of machine learning and unsupervised learning

- Eigendecomposition and Singular Value Decomposition (SVD)
- Dimensionality reduction (I have features in N dimensional space I want to reduce it to M dimensions $M \ll N$)
- **Finding latent topics in text in an unsupervised way**
- From long, sparse representation to **short, dense**

Decompose a matrix

Assume $\mathbf{X}$ is your term-document or tf-idf matrix computed over a corpus.

$\mathbf{X}$ dimension is $|D| \times |V|$ so it captures some form of correlation between:

- term vs term in a vector space defined by documents
- document vs document in a vector space define by terms

Key idea 1: Assume there are k hidden topics that we do not know about, but seek for discovering them in unsupervised way.

Key idea 2: Do not look at $\mathbf{X}$ simply as raw data yet decompose it in a way that **the decomposition gives you information and imposes constraints.**

Decompose with SVD (Singular Value Decomposition)

For now we simply seek to decompose $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$

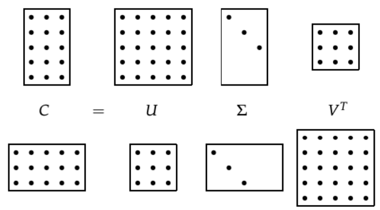
$\mathbf{X} \in \mathbb{R}^{|D|\times|V|} \quad = \quad \mathbf{U} \in \mathbb{R}^{|D|\times|D|},\ \ \mathbf{S} \in \mathbb{R}^{|D|\times|V|},\ \ \mathbf{V} \in \mathbb{R}^{|V|\times|V|}$

$$\begin{aligned} &\arg\min_{\mathbf{U},\mathbf{S},\mathbf{V}} \left\| \mathbf{X} - \mathbf{U}\mathbf{S}\mathbf{V}^\top \right\|_F \\ &\text{s.t. } \mathbf{U}^\top\mathbf{U} = \mathrm{Id} \quad \mathbf{V}^\top\mathbf{V} = \mathrm{Id} \\ &\quad\ \ \mathbf{S} = \mathrm{diag}(\lambda_1, \ldots . \lambda_s) \\ &\quad\ \ \text{we seek } \mathbf{U}\mathbf{S}\mathbf{V}^\top \text{to be low-rank} \end{aligned} \tag{2}$$

Property

- $\mathbf{U}^\top\mathbf{U} = \mathrm{Id}$ **orthonormality constraints** that all pairs of dimensions in U and V are uncorrelated, so that each dimension conveys unique information
- $\mathbf{S}$ is a diagonal matrix containing the ordered singular values (denote the **importance** of each axis in the new space)

SVD Visually



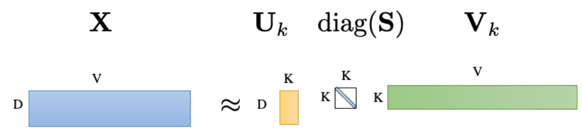
► **Figure 18.1** Illustration of the singular-value decomposition. In this schematic illustration of (18.9), we see two cases illustrated. In the top half of the figure, we have a matrix C for which $M > N$. The lower half illustrates the case $M < N$.

Low-rank approximation of the data with Truncated SVD

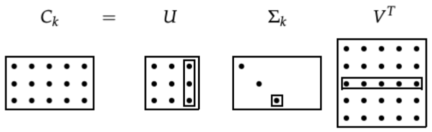
Key idea: We assume there might be $k \ll D$ latent topics in your data matrix $\mathbf{X}$.

- We use **SVD** but now approximate $\mathbf{X}$ by **just considering the highest singular values**
- In some sense, it is a way of saying we consider most of the variations in the data;
- small variations may be related to noise and so we "throw them away".

$$\mathbf{X} \approx \mathbf{X}_k = \mathbf{U_k}\mathbf{S_k}\mathbf{V_k}^\top$$



Truncated SVD - Low-rank approximation



▶ **Figure 18.2** Illustration of low rank approximation using the singular-value decomposition. The dashed boxes indicate the matrix entries affected by "zeroing out" the smallest singular values.

In [8]:
```
from sklearn.datasets import fetch_20newsgroups
categories = ['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
remove = ('headers', 'footers', 'quotes')
newsgroups_train = fetch_20newsgroups(subset='train', categories=categories, remove=remove)
newsgroups_test = fetch_20newsgroups(subset='test', categories=categories, remove=remove)
```

```
Hi,

I've noticed that if you only save a model (with all your mapping planes
positioned carefully) to a .3DS file that when you reload it after restarting
3DS, they are given a default position and orientation.  But if you save
to a .PRJ file their positions/orientation are preserved.  Does anyone
know why this information is not stored in the .3DS file?  Nothing is
explicitly said in the manual about saving texture rules in the .PRJ file.
I'd like to be able to read the texture rule information, does anyone have
the format for the .PRJ file?

Is the .CEL file format available from somewhere?

Rych
==================================================================================================


Seems to be, barring evidence to the contrary, that Koresh was simply
another deranged fanatic who thought it neccessary to take a whole bunch of
folks with him, children and all, to satisfy his delusional mania. Jim
Jones, circa 1993.


Nope - fruitcakes like Koresh have been demonstrating such evil corruption
for centuries.
==================================================================================================

 >In article <1993Apr19.020359.26996@sq.sq.com>, msb@sq.sq.com (Mark Brader)

MB>                                                             So the
MB> 1970 figure seems unlikely to actually be anything but a perijove.

JG>Sorry, _perijoves_...I'm not used to talking this language.

Couldn't we just say periapsis or apoapsis?
```

From raw text to Term-Document Matrix

Let's look at the data

- Newsgroup text from the '90s (internet was not there yet)
- We have the **categories** but we set them aside from the algorithm
- We assume **we do NOT know the categories of the text**
  - We seek to recover them in an unsupervised way Categories are `['alt.atheism', 'comp.graphics', 'sci.space', 'talk.religion.misc']`

```
from sklearn.datasets import fetch_20newsgroups
categories = ['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
remove = ('headers', 'footers', 'quotes')
newsgroups_train = fetch_20newsgroups(subset='train', categories=categories, remove=remove)
newsgroups_test = fetch_20newsgroups(subset='test', categories=categories, remove=remove)
```

In [9]:
```
print(*newsgroups_train.data[:3], sep='\n'+'=='*50+'\n')
```

From raw text to Term-Document Matrix

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer(stop_words='english')
X_train_counts = count_vect.fit_transform(newsgroups_train.data)
X_train_counts.shape # (documents, vocab)
```

In [10]:
```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer(stop_words='english')
X_train_counts = count_vect.fit_transform(newsgroups_train.data)
dims = X_train_counts.shape # (documents, vocab)
print(f'We have {dims[0]} documents and a vocabulary of size {dims[1]}')
```

```
We have 2034 documents and a vocabulary of size 26576
```

Term-Document Matrix Shape

X_train_counts shape is (2034, 26576)

- 2034 individual documents (in this case newsgroup posts)
- 26576 the number of terms (i.e. vocabulary size $|V|$)
- Note that `X_train_counts` is modeled as a sparse matrix `scipy.sparse.csr.csr_matrix`

Let us see a few $k = 10$ random words from the vocabulary:

```
random.choices(count_vect.get_feature_names_out(), k=10)
```

['specs', 'regularity', 'gillespie', 'convienent', 'ite', 'friends', 'rom', 'devotion', 'imaging', 'mayer']

In [11]:
```
td_matrix = pd.DataFrame(data=X_train_counts.todense(),
                         index=range(X_train_counts.shape[0]),
                         columns=count_vect.get_feature_names_out(),)
```

	00	000	0000	00000	000000	000005102000	000062david42	0001	000100255pixel	00041032	...	zurich	zurvanism	zus
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2029	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2030	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2031	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2032	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2033	0	0	0	0	0	0	0	0	0	0	...	0	0	0

2034 rows × 26576 columns

Sanity check and see the most frequent word

```
r, c = np.unravel_index(np.argmax(X_train_counts), X_train_counts.shape)
print(count_vect.get_feature_names_out()[c],
      'occurred', X_train_counts[r, c], 'times')
```

In [12]:
```
r, c = np.unravel_index(np.argmax(X_train_counts), X_train_counts.shape)
print(count_vect.get_feature_names_out()[c], 'occurred', X_train_counts[r, c], 'times')
```

```
jpeg occurred 232 times
```

In [13]:
```
#%time U, S, V = np.linalg.svd(X_train_counts.todense(), full_matrices=False)
```

In [14]:
```
#%time Us, Ss, Vts = scipy.sparse.linalg.svds(X_train_counts.asfptype(), k=X_train_counts.shape[0]-1)
```

2. The embedding space ($k = 2$)

- Unlike before, it is now **continuous, short and dense**
- The space encodes some notion of similarities between words
- Cosine similarity makes more sense

In [17]:
```
fig, ax = plt.subplots(figsize=(10,10))
clip = 1
ax.scatter(*Vts);
vocab = count_vect.get_feature_names_out()
for count, ((x,y), txt) in enumerate(zip(Vts.T, vocab)):
    if np.linalg.norm([x,y]) > .5e-1:
        ax.annotate(txt, (x+0.01, y))
if clip:
    #plt.ylim(-0.175, -0.030)
    plt.ylim(0.05, 0.15)
plt.grid('off')
```

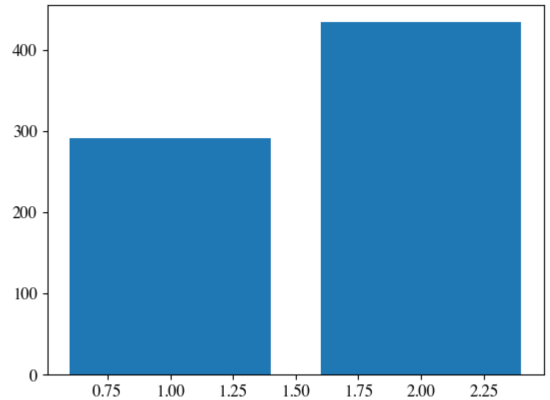
In [15]:
```
%time Us, Ss, Vts = scipy.sparse.linalg.svds(X_train_counts.asfptype(), k=2)
```

```
CPU times: user 83 ms, sys: 4.91 ms, total: 87.9 ms
Wall time: 43.7 ms
```

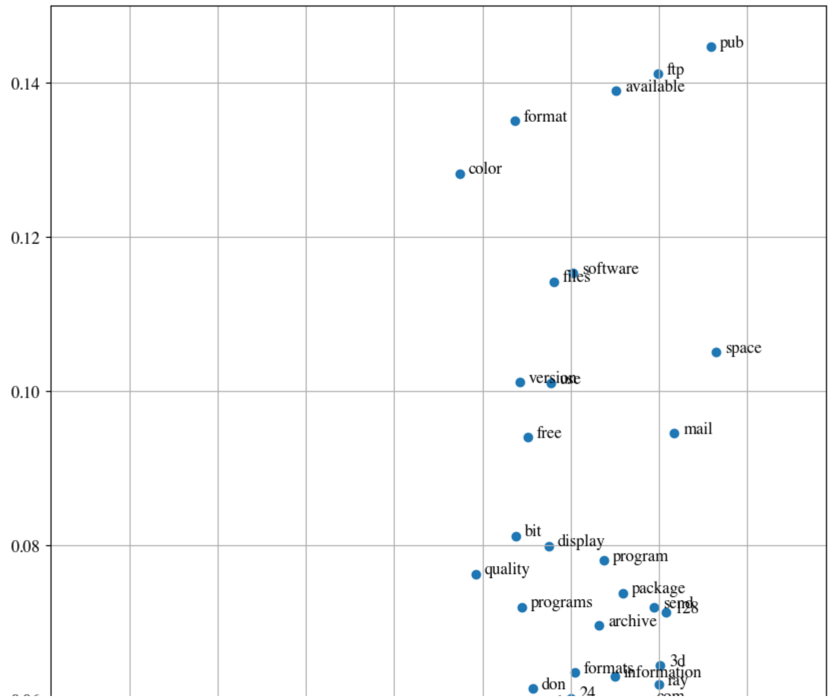
In [16]:
```
print("U = ", (Us.shape), "S = ",  (Ss.shape), "V = ",  (Vts.shape))
```

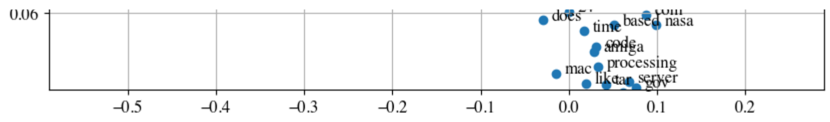
```
U =  (2034, 2) S =  (2,) V =  (2, 26576)
```

1. Singular Values give you the "topic" importance



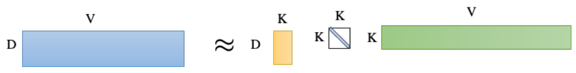
S =array([291.51012741, 433.92698542])





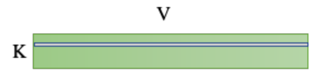
3. We can associate each new k component (topic) with most important words

$$\mathbf{X} \approx \mathbf{U}_k \quad \mathrm{diag}(\mathbf{S}) \quad \mathbf{V}_k$$



3. We can associate each new k component (topic) with most important words using $\mathbf{V}_k$

$$\mathbf{V}_k$$




```python
k=0 #first component, first topic
word_per_topic = 10
topic = [vocab[idx_word] for idx_word in np.argsort(Vts[k,:])[:-word_per_topic-1:-1]]
#topic
```

Topic $k=0$ associated with singular value λ_0 = 292

```python
word_per_topic = 10
topic = [vocab[idx_word] for idx_word in np.argsort(Vts[k,:])[:-word_per_topic-1:-1]]
```

	0	1	2	3	4	5	6	7	8	9
0	jpeg	image	edu	file	graphics	images	gif	data	pub	ftp

```python
k=1 #second component, first topic
topic = [vocab[idx_word] for idx_word in np.argsort(Vts[k,:])[:-word_per_topic-1:-1]]
#topic
```

Topic $k=1$ associated with singular value λ_1 = 434

```python
word_per_topic = 10
k=1;topic = [vocab[idx_word] for idx_word in np.argsort(Vts[k,:])[:-word_per_topic-1:-1]]
```

	0	1	2	3	4	5	6	7	8	9
0	jpeg	image	edu	file	graphics	images	gif	data	pub	ftp

How to think about the new embedding space

If you reduce the dimension from 3 to 2, you may think that the embedding is learning something like:

$$[\text{car}, \text{truck}, \text{flower}] \longrightarrow [(1.3452 * \text{car} + 0.2828 * \text{truck}), \text{flower}]$$

4. Dimensionality Reduction:

Projecting a document into lower-dimensional space

We treat the document as a vector $\mathbf{d}$ that lives in a V dimensional space.

- We want to use the output of SVD to project $\mathbf{d}$ from V dimension to k where $k \ll V$.
- Usually k is the order of hundreds.

We take the matrix $\mathbf{V}_k$ of the SVD decomposition that has dimension $k \times V$ (this makes sense because $\mathbf{d}$ lives in the same V-D space too).

$$\mathbf{d}_k = \underbrace{\mathbf{V}_k^\top}_{k \times V} \underbrace{\mathbf{d}}_{V}$$

Why SVD?

Consider X has `term by document` matrix.

$$\begin{array}{cc} & \begin{array}{ccccc} & & \mathbf{d}_j & & \\ & & \downarrow & & \end{array} \\ \mathbf{t}_i^T \rightarrow & \begin{bmatrix} x_{1,1} & \dots & x_{1,j} & \dots & x_{1,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{i,1} & \dots & x_{i,j} & \dots & x_{i,n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{m,1} & \dots & x_{m,j} & \dots & x_{m,n} \end{bmatrix} \end{array}$$

- Assuming that you have ℓ_2 normalized vectors, you can compute the **similarity** between two terms by evaluating the dot product $\mathbf{t}_i^T \mathbf{t}_j$
- If you want to compute it for all pairs just compute the covariance matrix: $\mathbf{X}\mathbf{X}^\top$ which measure all the **correlations between terms in the document space.**

Why SVD: decompose correlation of the data

$\mathbf{X}\mathbf{X}^\top$ which measures all the **correlations between terms in the document space.** What if we decompose this correlation?

$$\mathbf{X}\mathbf{X}^T \quad = \quad (\mathbf{U}\mathbf{S}\mathbf{V}^T)(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T = (\mathbf{U}\mathbf{S}\mathbf{V}^T)(\mathbf{V}^{T^T}\mathbf{S}^T\mathbf{U}^T) = \mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{V}\mathbf{S}^T\mathbf{U}^T = \mathbf{U}\underbrace{\mathbf{S}\mathbf{S}^T}_{\Sigma}\mathbf{U}^T$$

Singular Value Decomposition of $\mathbf{X}$ is related to **Eigendecomposition** of the covariance of $\mathbf{X}$.

Eigenvalue $\sigma_i = \lambda_i^2 \quad \longrightarrow \quad$ Eigenvalue = squared singular value

Eigendecomposition and Singular Value Decomposition

Method	X	Decomposition
SVD	any	$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$
Eigen	square	$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^{-1}$

Decomposition as a Geometric Pipeline

$$\mathbf{A}\mathbf{x} = \underbrace{(\mathbf{U}\underbrace{(\mathbf{\Sigma}\ \underbrace{(\mathbf{U}^{-1}\mathbf{x})}_{\text{1st step/rotate}}\ )}_{\text{2nd step/scale}})}_{\text{3rd step/rotate}}$$

Method	Step 1	Step 2	Step 3
Geometry	rotate	scale/reflect axis	rotate
SVD	$\mathbf{V}^T$	$\mathbf{\Sigma}$	$\mathbf{U}$
Geometry	rotate	scale/reflect axis	rotate
Eig	$\mathbf{V}^{-1}$	$\mathbf{\Sigma}$	$\mathbf{U}$

Gensim library

- Free python library for topic modeling
- It contains **LSA** and also **word2vec**
- No deeplearning yet!

- Latent Semantic Indexing, LSI (or sometimes LSA) transforms documents from either bag-of-words or (preferrably) TfIdf-weighted space into a latent space of a lower dimensionality. For the toy corpus above we used only 2 latent dimensions, but on real corpora, target dimensionality of 200–500 is recommended as a "golden standard" [1].

```
model = models.LsiModel(tfidf_corpus, id2word=dictionary, num_topics=300)
```

LSI training is unique in that we can continue "training" at any point, simply by providing more training documents. This is done by incremental updates to the underlying model, in a process called *online training*. Because of this feature, the input document stream may even be infinite – just keep feeding LSI new documents as they arrive, while using the computed transformation model as read-only in the meanwhile!

```
model.add_documents(another_tfidf_corpus)  # now LSI has been trained on tfidf_corpus + anot
lsi_vec = model[tfidf_vec]  # convert some new document into the LSI space, without affectin

model.add_documents(more_documents)  # tfidf_corpus + another_tfidf_corpus + more_documents
lsi_vec = model[tfidf_vec]
```

See the `gensim.models.lsimodel` documentation for details on how to make LSI gradually "forget" old observations in infinite streams. If you want to get dirty, there are also parameters you can tweak that affect speed vs. memory footprint vs. numerical precision of the LSI algorithm.

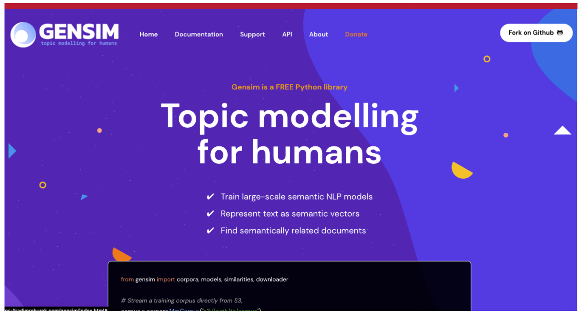
Introduction to word2vec

Problems with LSA and SVD

The dimensions of the matrix change very often (new words are added very frequently and corpus changes in size). SVD based methods do not scale well for big matrices and it is hard to incorporate new words or documents.

- The matrix is extremely sparse since most words do not co-occur.
- The matrix is very high dimensional in general ($\approx 10^6 \times 10^6$)
- Quadratic cost to train (i.e. to perform SVD)
- Requires the incorporation of some hacks on X to account for the drastic imbalance in word frequency

Applications: Topic modeling with Gensim



Introduction to word2vec

Word2vec [Mikolov et al. 2013] is a framework for **learning word vectors**.

The objective is the same to what we saw in **LSA**.

Introduction to word2vec

- Still learn short, dense and static embeddings.
- Static embeddings: one fixed embedding for each word in the vocabulary

Idea: apply **machine learning** and use the text as **self-supervision**

> The intuition of word2vec is that instead of counting how often each word w occurs near, say, apricot, we'll instead train a classifier on a binary prediction task: "Is word w likely to show up near apricot?" We don't actually care about this prediction task; instead we'll take the learned classifier weights as the word embeddings.

Recap of concepts to understand word2vec

- Self-Supervision
- Binary Logistic Regression
- Softmax classifier (Multinomial Logistic Regression)
- Optimization with Gradient Descent

Self-Supervision

Let's play a game and fill the missing word:

```
Yesterday, it was a hot summer day in Rome, so I went to a nice
gelateria and I bought a delicious yet melting ____________
```

word1	word2	word3	word4	sum
strolley	ice-cream	parrot	gelato	
prob=?	prob=?	prob=?	prob=?	1

Masking the input: with self-supervision you have the ground-truth word because it is in the text itself. You simple remove a word and ask the machine to predict the removed word, given some context.

Logistic Regression

- The name is a bit *misleading*: it is used as a **discriminative <u>classifier</u>**
- Discriminative means we model $p(y|x)$ *what is the probability for the label* $\text{prob}(Y=y)$ *given* x*?*
- Very commonly used algorithm

Logistic Regression

- $\mathbf{x} \in \mathbb{R}^d \quad y \in \{0,1\}$

$$\begin{cases} y=1 & \text{positive example} \\ y=0 & \text{negative example} \end{cases}$$

Logistic Regression - Probabilistic View

We model **conditional probability of $y|x$:**

$$\begin{cases} p(y=1|\mathbf{x};\boldsymbol{\theta}) = f_{\boldsymbol{\theta}} \\ p(y=0|\mathbf{x};\boldsymbol{\theta}) = ? \end{cases}$$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) \doteq \sigma\left(\boldsymbol{\theta}^T\mathbf{x}\right)$$

where:

$$\sigma(z) = \frac{1}{1+\exp^{-z}} \quad \text{sigmoid or logistic function}$$

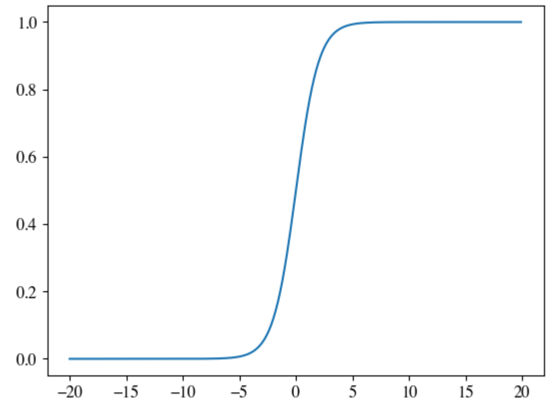
Logistic Regression (also called Logit)

$$f_{\boldsymbol{\theta}}(\mathbf{x}) \doteq \sigma\left(\boldsymbol{\theta}^T\mathbf{x}\right)$$

where:

$$\sigma(z) = \frac{1}{1+\exp^{-z}} \quad \text{sigmoid or logistic function}$$

Smooth and Differentiable alternative to sign



Logistic Regression - Probabilistic View

We model **conditional probability of $y|x$:**

$$\begin{cases} p(y=1|\mathbf{x};\boldsymbol{\theta}) = f_{\boldsymbol{\theta}} \\ p(y=0|\mathbf{x};\boldsymbol{\theta}) = 1 - f_{\boldsymbol{\theta}} \end{cases}$$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) \doteq \sigma\left(\boldsymbol{\theta}^T\mathbf{x}\right)$$

where:

$$\sigma(z) = \frac{1}{1+\exp^{-z}} \quad \text{sigmoid or logistic function}$$

$$\lim_{z\to\infty} \sigma(z) = 1$$

$$\lim_{z\to-\infty} \sigma(z) = 0$$

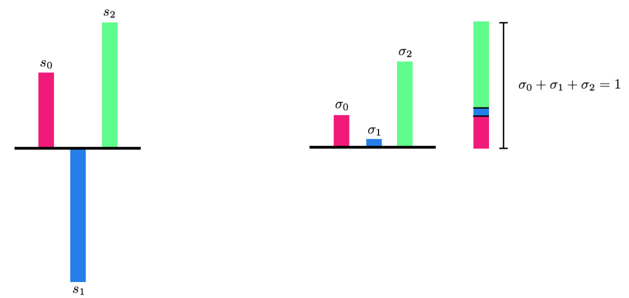
Logistic Regression - Probabilistic View

We model **conditional probability of $y|x$:**

$$\begin{cases} p(y=1|\mathbf{x};\boldsymbol{\theta}) = f_{\boldsymbol{\theta}} \\ p(y=0|\mathbf{x};\boldsymbol{\theta}) = 1 - f_{\boldsymbol{\theta}} \end{cases}$$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) \doteq \frac{1}{1+\exp^{-\boldsymbol{\theta}^T\mathbf{x}}}$$

How SoftMax transforms logits into probability



Multinomial Logistic Regression with SoftMax

We model **conditional probability of $y|x$:**

$$\begin{cases} p(y=1|\mathbf{x};\mathbf{W},\mathbf{b}) = p_1 = \boldsymbol{\sigma}_1\left(\mathbf{W}\mathbf{x}+\mathbf{b}\right) \\ p(y=2|\mathbf{x};\mathbf{W},\mathbf{b}) = p_2 = \boldsymbol{\sigma}_2\left(\mathbf{W}\mathbf{x}+\mathbf{b}\right) \\ \dots \\ p(y=K|\mathbf{x};\mathbf{W},\mathbf{b}) = p_K = \boldsymbol{\sigma}_K\left(\mathbf{W}\mathbf{x}+\mathbf{b}\right) \end{cases}$$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) \doteq \mathbf{z} \doteq \boldsymbol{\sigma}\left(\mathbf{W}\mathbf{x}+\mathbf{b}\right)$$

where:

$$\boldsymbol{\sigma}_i(z) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{Softmax function}$$

SoftMax Regression, More Compact Form

you can think $\mathbf{z}=\mathbf{W}\mathbf{x}+\mathbf{b}$ as **unormalized log-probability** of each class.

$$p(\mathbf{y}|\mathbf{x};\mathbf{W},\mathbf{b}) = \frac{\exp(\mathbf{W}\mathbf{x}+\mathbf{b})}{\sum_{k=1}^{K}\exp(\mathbf{W}_k\mathbf{x}+\mathbf{b}_k)}$$

Learning weights through Stochastic Gradient Descent (SGD)

We need to minimize

$$\mathcal{J}(\theta;\mathbf{x},y) = \frac{1}{2}\sum_{i=1}^{n}\mathcal{L}\big(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\big)$$

so to find:

$$\theta^* = \arg\min_{\theta} \mathcal{J}(\theta;\mathbf{x},y)$$

Gradient Descent Algorithm as an Iterative Method

<u>**Idea: make little step so that locally after each step the cost is lower than before**</u>
Input: Training set $\{\mathbf{x}_i, y_i\}$, learning rate γ, a small value in $\{0.1,\dots,\texttt{1e-6}\}$.

1. **Initialization - Very Important if the function is not strictly convex**

$$\boldsymbol{\theta} \doteq \mathbf{0}^T$$

   Set it to all zeros or random initialization from a distribution.
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt to the parameters $\boldsymbol{\theta}$ given **all the training set**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}(\theta;\mathbf{x},y)$$

3. When convergence is reached, you final estimate is in $\boldsymbol{\theta}$

Convergence

$$\{\boldsymbol{\theta}_{t=0}, \boldsymbol{\theta}_{t=1}, \dots, \boldsymbol{\theta}_{t=100}\}$$

0) <u>Always:</u> validation loss/metric *(early stopping)* (required)

1) No significant decrease in the loss function (preferred)

$$|\mathcal{J}(\theta;\mathbf{x},y)_t - \mathcal{J}(\theta;\mathbf{x},y)_{t-1}|$$

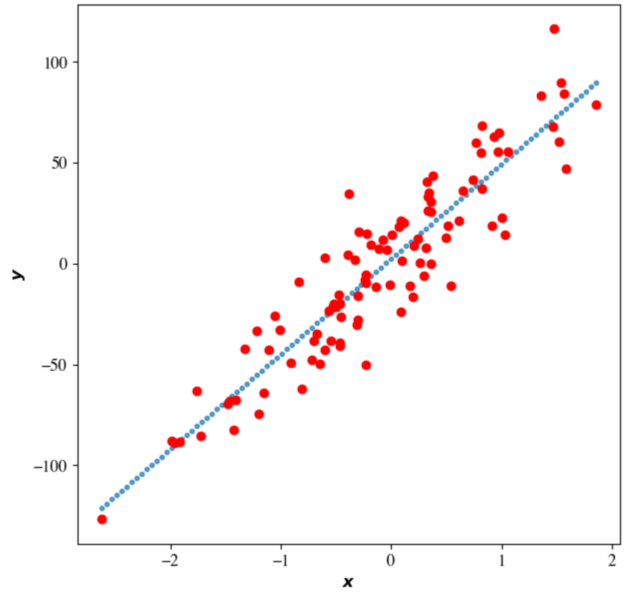
1) No variations in the parameters

$$||\ \theta_t - \theta_{t-1}\ ||$$

2) Gradient Norm goes to zero

$$||\ \boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}(\theta;\mathbf{x},y)\ || \rightarrow 0$$

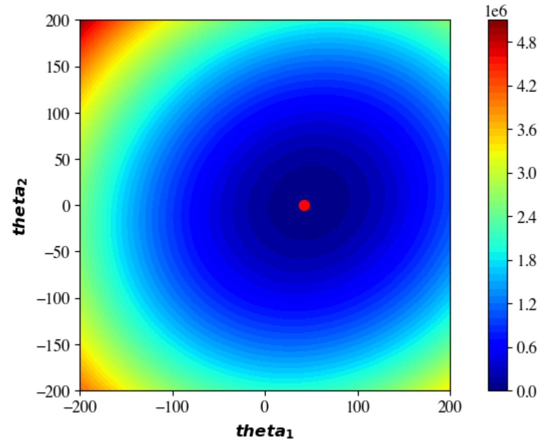
In [20]:
```python
%matplotlib inline
from sklearn import linear_model, datasets
from matplotlib import pyplot as plt
import numpy as np

n_samples = 100
size = 7

X, y, coef_gt = datasets.make_regression(
    n_samples=n_samples,
    n_features=1,
    n_informative=1,
    noise=20,
    coef=True,
    random_state=42,
)
fig = plt.figure(figsize=(size, size))
ax = fig.add_subplot()
# Linear Regression
bias = np.ones((X.shape[0], 1))
X = np.hstack((X, bias))
theta = np.linalg.inv(X.T@X)@X.T@y
# Now MeshGrid
Xmin, Xmax = X.min(), X.max()
x_interp = np.linspace(Xmin, Xmax, 100)
x_interp = x_interp.reshape(-1,1)
x_interp = np.c_[x_interp,np.ones_like(x_interp)]
y_interp = np.dot(theta, x_interp.T)
ax.scatter(x_interp[:,0], y_interp, alpha=0.7, marker='.')
ax.scatter(X[..., 0], y, c='red', marker='o')
ax.set_xlabel('$x$');
ax.set_ylabel('$y$');
```



In [21]:
```python
# do mesh grid on possible theta, evaluate the loss and plot the countf
theta_sampling = 50
theta_space = np.linspace(-200, 200, theta_sampling)
xxt, yyt = np.meshgrid(theta_space, theta_space)
xxt = xxt.flatten()
yyt = yyt.flatten()
all_coeff = np.stack((xxt, yyt), axis=1)
loss = 0.5*1
```

In [22]:
```python
losses = []
for coeff in all_coeff:
    diff = np.dot(X, coeff.T) - y
    losses.append(0.5*np.dot(diff.T, diff))
losses = np.array(losses)
losses = losses.reshape(theta_sampling, theta_sampling)
xxt = xxt.reshape(theta_sampling, theta_sampling)
yyt = yyt.reshape(theta_sampling, theta_sampling)
```

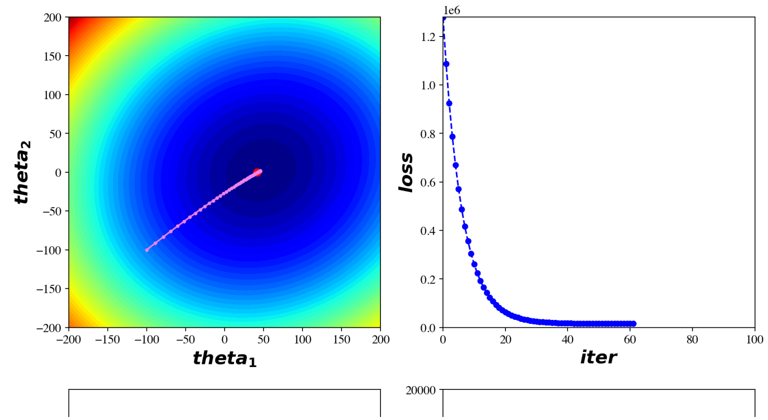
In [23]:
```python
plt.rcParams['axes.grid'] = False
plt.contourf(xxt, yyt, losses, levels=50, cmap='jet')
plt.colorbar()
plt.scatter(coef_gt, 0, color='red', marker='o', s=50)
plt.axis('scaled')
plt.xlabel('$theta_1$')
plt.ylabel('$theta_2$');
```

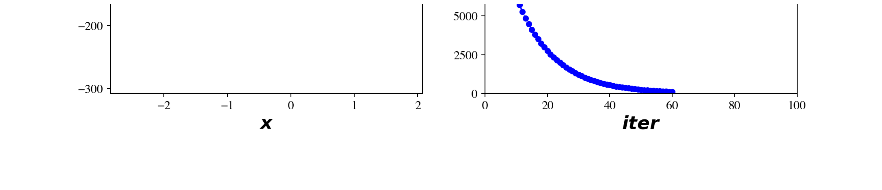


Ready for an awesome demo?

In [24]:

```python
## Implementation of Gradient Descent for Logistic Regression

import time
%matplotlib notebook


def get_diff(X, theta, y):
    return X@theta - y[..., np.newaxis]


def get_loss(diff):
    return 0.5*np.dot(diff.T, diff)


def plot_line(plot3, theta):
    x_interp = np.linspace(Xmin, Xmax, 100)
    x_interp = x_interp.reshape(-1, 1)
    x_interp = np.c_[x_interp, np.ones_like(x_interp)]
    y_interp = np.dot(x_interp, theta)
    if plot3:
        plot3.set_xdata(x_interp[:, 0])
        plot3.set_ydata(y_interp)
    else:
        return x_interp, y_interp

Xmin, Xmax = X.min(), X.max()
plt.ion()
figure, (axes_1, axes_2) = plt.subplots(2, 2, figsize=(12, 12))
plt.rcParams['axes.grid'] = False
ax0, ax1 = axes_1
ax2, ax3 = axes_2
ax0.contourf(xxt, yyt, losses, levels=50, cmap='jet')
ax0.scatter(coef_gt, 0, color='red', marker='o', s=50)
ax0.set_xlabel('$theta_1$',fontsize=18)
ax0.set_ylabel('$theta_2$',fontsize=18)
ax1.set_ylabel('$loss$',fontsize=18)
ax1.set_xlabel('$iter$',fontsize=18)
ax1.set(xlim=(0, 100), ylim=(0, 1.28e6))
ax2.scatter(X[..., 0], y, c='red', marker='o')
ax2.set_xlabel('$x$',fontsize=18)
ax2.set_ylabel('$y$',fontsize=18)
ax3.set_ylabel('$Grad. Norm.$',fontsize=18)
ax3.set_xlabel('$iter$',fontsize=18)
ax3.set(xlim=(0, 100), ylim=(0, 20000))

theta_curr = np.array([[-100, -100]]).T
losses_track = [get_loss(get_diff(X, theta_curr, y))]
grad_norm_track = [1000]

theta_track = np.array(theta_curr)
lr = 1e-3
loss_tol = 10

plot1, = ax0.plot(*theta_curr, color='violet',
                  marker='.', markersize=5, linestyle='-')
plot2, = ax1.plot(*losses_track, color='blue',
                  marker='.', markersize=10, linestyle='--')
xi, yi = plot_line(None, theta_curr)
plot3a,plot3b = ax2.plot(xi, yi, color='blue', marker='.',
                  markersize=3, linestyle='--')
plot4, = ax3.plot(1000, color='blue',
                  marker='.', markersize=10, linestyle='--')
while True:
    diff = get_diff(X, theta_curr, y)
    grad = (diff * X).sum(axis=0, keepdims=True).T
    theta_curr = theta_curr - lr*grad
    theta_track = np.append(theta_track, theta_curr, axis=1)
    diff = get_diff(X, theta_curr, y)
    losses_track.append(get_loss(diff))
    grad_norm_track.append(np.linalg.norm(grad,2))
    if abs(losses_track[-2]-losses_track[-1]) < loss_tol:
        break
    plot1.set_xdata(theta_track[0, :])
    plot1.set_ydata(theta_track[1, :])
    plot2.set_xdata(range(len(losses_track)))
    plot2.set_ydata(losses_track)
    plot4.set_xdata(range(len(grad_norm_track[1:])))
    plot4.set_ydata(grad_norm_track[1:])
    plot_line(plot3a, theta_curr)
    plot_line(plot3b, theta_curr)
    figure.canvas.draw()
    figure.canvas.flush_events()
    time.sleep(0.1)
print(*theta_curr)
plt.show()
```






```
[46.09610615] [1.80913084]
```

Let's go back to word2vec

Introduction to word2vec

- We have a very **large corpus** on which we apply **self-supervision**
  - The corpus defines the vocabulary V.
- Every word in V is represented as a vector in a D dimensional vector space
  - e.g. D = 100.
- The parametric model $\boldsymbol{\theta}$ that we learn **depends on all vectors for all the words**

word2vec is a generic framework

`word2vec` presents two algorithms:

1. **Skip-Gram** *(we see it today!)*
2. Continuous Bag-of-Word (CBOW)

Also it offers different training methods:

- **with softmax** *(we see it today!)*
- negative sampling from [Mikolov et al. 2013]
- hierarchical softmax

word2vec: Skip-Gram with softmax

Parameters to learn:

$$\boldsymbol{\theta} = [\boldsymbol{\theta}_W; \boldsymbol{\theta}_C]$$

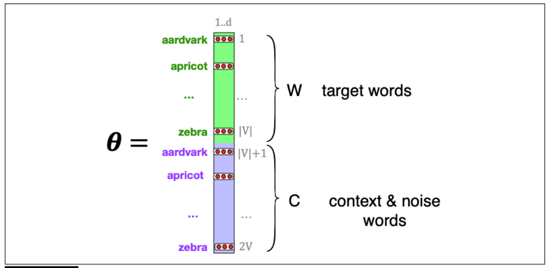


Figure 6.13 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter θ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings **W** and the context+noise embeddings **C**.

word2vec: Skip-Gram with softmax

Parameters to learn: $\boldsymbol{\theta} = [\boldsymbol{\theta}_W; \boldsymbol{\theta}_C]$

- $\boldsymbol{\theta}_W$ is a matrix $|V| \times D$ where we store a vector in D dimensions for each word of the vocabulary V. We store vectors when the word w works as a $\underline{center}$.
- $\boldsymbol{\theta}_C$ is a matrix $|V| \times D$ where we store a vector in D dimensions for each word of the vocabulary V. We store vectors when the word w works as the $\underline{context}$.

Thus the number of parameters to learn $\boldsymbol{\theta}$ is $\quad 2 \cdot |V| \cdot D$

Sample question you may find in the exam

Given a **word2vec** model with **skip-gram** that learns word embedding $\boldsymbol{\theta}_i \in \mathbb{R}^{10}$ with a vocabulary of 100 words, how many parameters word2vec needs to be learned? What is the dimension of the matrix learned?

Answer

word2vec learns two vectors for each word in the vocabulary. One vector in case the word is considered as target; one vector in case the word functions as context. Given that the vectors are in 10-D space, the number of params is $2 \cdot 10 \cdot 100 = 2000$. The matrix $\boldsymbol{\theta}$ shape is $2 \cdot 100 \times 10$, where each row indicates a word in the vocabulary.

word2vec: Skip-Gram Self-Supervision

We start from a large corpus of text:

`... lemon, a tablespoon of apricot jam, a pinch ...`

and a window that defines the context (let's assume window size is $m = 2$ and centered at the index t).

word2vec: Skip-Gram Self-Supervision

Given a window size of $m = 2$ words as **context**, we select:

- the center word w_t, apricot in this case.
- 2 words as context on the left of $w_t \longrightarrow (w_{t-2}, w_{t-1})$
- 2 word as context on the right of $w_t \longrightarrow (w_{t+1}, w_{t+2})$

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

word2vec: Skip-Gram Self-Supervision

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

Let's assume the vocabulary is made of $|V|$ words:

word1	word2	word3	...	wordN
lemon	tablespoon	gelato	...	jam

Note: $|V|$ is possibly in the order of millions.

word2vec: Skip-Gram Self-Supervision

⚠️With **strong naive conditional independence assumption**

$$p(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}|w_t;\boldsymbol{\theta}) \approx \prod_{-m\leq j\leq m} p(c_{t+j}|w_t;\boldsymbol{\theta})$$

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

word2vec: Skip-Gram Self-Supervision

$$p(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}|w_t = \text{apricot};\boldsymbol{\theta})$$

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

word2vec: Skip-Gram Self-Supervision

Given the example below, we have to compute:

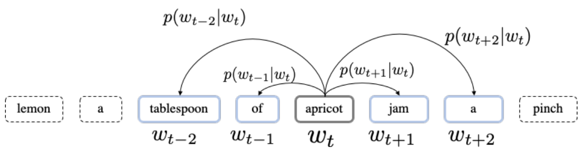
$$p(w_{t-2}|w_t)\cdot p(w_{t-1}|w_t)\cdot p(w_{t+1}|w_t)\cdot p(w_{t+2}|w_t)$$

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

word2vec: Skip-Gram Self-Supervision

Given the example below, we have to compute:

$$p(w_{t-2}|w_t)\cdot p(w_{t-1}|w_t)\cdot p(w_{t+1}|w_t)\cdot p(w_{t+2}|w_t)$$



How to compute $p(w_{t-1}|w_t)$?

$$p(c_1|w = \text{apricot})$$

We have to predict w_{t-1} given the center word w_t. Note w_{t-1} can takes values based on the words in V.

Think w_{t-1} as a **categorical variable** where its possible state spans V.

word1	word2	word3	...	wordN
lemone	tablespoon	gelato	...	jam

How to compute $p(w_{t-1}|w_t)$?

1. i is the index of the center word w in V and also $\boldsymbol{\theta}_W$. Compare the embedding of w as center word $\boldsymbol{\theta}_W[i]$ vs embeddings of all context words $\boldsymbol{\theta}_C$ with matrix to vector product.

$$\underbrace{\mathbf{z}}_{|V|\times 1} = \underbrace{\overbrace{\boldsymbol{\theta}_C}^{\text{as context}}}_{|V|\times D} \cdot \underbrace{\overbrace{\boldsymbol{\theta}_W[t]^T}^{\text{as center}}}_{D\times 1}$$

2. $\mathbf{z}$ is logits and encodes the similarity via dot product of the center word embedding $\boldsymbol{\theta}_W[i]$ **against all vocabulary words** taken as context $\boldsymbol{\theta}_C$

3. We pass $\mathbf{z}$ through softmax operator to get a distribution over $|V|$ as:

$$\mathbf{p} = \text{softmax}(\mathbf{z})$$

You can think $\mathbf{p}$ as of this form:

	lemon	tablespoon	gelato	...	jam
p	0.001	0.1	0.03	...	0.15

How to compute $p(w_{t-1}|w_t)$?

$$\mathbf{p} = \text{softmax}\left(\boldsymbol{\theta}_C \cdot \boldsymbol{\theta}_W[i]^T\right)$$

You can think $\mathbf{p}$ as of this form:

	lemon	tablespoon	gelato	...	jam
p	0.001	0.1	0.03	...	0.15

💡Note: until now we did not used the self-supervised label! We did not use the **ground-truth index for** w_{t-1}. (i.e. the fact that w_{t-1} is actually `tablespoon`!

~~lemon~~,	~~a~~	[tablespoon	of	apricot	jam	a]	~~pinch~~
		w_{t-2}	w_{t-1}	w_t	w_{t+1}	w_{t+2}	

Loss function: compare two discrete distributions

$$\mathbf{p} = \text{softmax}\left(\boldsymbol{\theta}_C \cdot \boldsymbol{\theta}_W[i]^T\right)$$

You can think $\mathbf{p}$ as of this form:

	lemon	tablespoon	gelato	...	jam
p (word2vec prediction)	0.001	0.1	0.03	...	0.15

Let's consider the label as a one-hot encoding vector where 1 is over the ground-truth word given by the text.

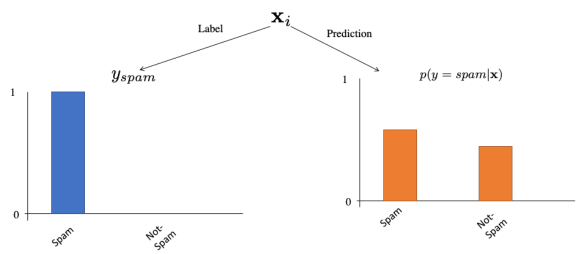
	lemon	tablespoon	gelato	...	jam
y label (ground-truth w_{t-1})	0	1	0	...	0

We want to adjust the weights $\boldsymbol{\theta}$ so that $\mathbf{p}$ matches the label!

Loss function: compare two discrete distributions

In this example V has two words

Who can tell me how we can compare two *[discrete]* probability distributions? We use **cross-entropy.**



Loss function: Cross-entropy (CE)

- $\mathbf{y} = [0, 0, 0, 1, 0]$ is the **one-hot encoding of the labels** $\rightarrow$ This is self-supervised by the text itself!
- $\mathbf{p}(w_{t-1}|w_t) = [0.1, 0.1, 0.1, 0.35, 0.35]$ is the output from `word2vec`. We want $\mathbf{p}$ to match $\mathbf{y}$.

$$\mathcal{L}(w_{t-1}, w_t; \theta) = \mathbf{y}^\top \log\left(\mathbf{p}(w_{t-1}|w_t)\right)$$

$$-0 \cdot \ln(0.1) - 0 \cdot \ln(0.1) - 0 \cdot \ln(0.1) \underbrace{-1 \cdot \ln(0.35)}_{\text{only this matter}} - 0 \cdot \ln(0.35) = -1 \cdot \ln(0.35) \approx 1.04$$

One-hot encoding is a selector!

- $\mathbf{y} = [0, 0, 0, 1, 0] \rightarrow$ works as a selector of probability of the actual ground-truth word that we removed!
- Of the probabilites returned by `word2vec` select that for which the index gt corresponds to the `1` in the label $\mathbf{y}$
- in our case, gt is the index of `tablespoon` in $|V|$.

$$\mathcal{L}(w_{t-1}, w_t; \theta) = -\log \mathbf{p}\left(w_{t-1}|w_t\right)[gt]$$

Loss function simplified

We can select immediately $[gt]$ in the numerator.

$$\mathcal{L}(w_{t-1}, w_t; \theta) = -\log\left(\frac{\exp\left(\boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T\right)}{\sum_{v=1}^{V} \exp\left(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[i]^T\right)}\right)$$

Sometimes is shown as:

$$\mathcal{L}(w_{t-1}, w_t; \theta) = \underbrace{-\boldsymbol{\theta}_C[gt] \cdot \boldsymbol{\theta}_W[i]^T}_{\text{similarity center vs context}} + \underbrace{\log\left(\sum_{v=1}^{V} \exp\left(\boldsymbol{\theta}_C[v] \cdot \boldsymbol{\theta}_W[i]^T\right)\right)}_{\text{make sure it is a probability}}$$

Exercise A: compute the loss for a center word in word2vec

$$\mathbf{p} = \operatorname{softmax}\left(\boldsymbol{\theta}_C \cdot \boldsymbol{\theta}_W[i]^T\right)$$

You can think $\mathbf{p}$ as of this form:

	lemon	tablespoon	gelato	...	jam
p (word2vec prediction)	0.001	0.1	0.03	...	0.15

Let's consider the label as a one-hot encoding vector where 1 is over the ground-truth word given by the text.

	lemon	tablespoon	gelato	...	jam
y label (ground-truth)	0	1	0	...	0

$$\mathcal{L}(w_{t-1}, w_t; \theta) = -\log \mathbf{p}\big(w_{t-1}|w_t\big)[gt] = -\ln(0.1) \approx 2.30$$

Exercise B: similar but more complex

We give 2D matrix of θ_W and θ_C, the position of the center word and context word, compute the loss.

Likelihood over a window

Likelihood of the data given the parameters.

$$L(\boldsymbol{\theta}) = p\big(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}|w; \boldsymbol{\theta}\big) \approx \prod_{-m \le j \le m, j \ne 0} p(w_{t+j}|w_t; \boldsymbol{\theta})$$

We seek the parameters $\boldsymbol{\theta}$ that maximizes the likelihood of the data.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = p\big(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}|w; \boldsymbol{\theta}\big)$$

Loss function over a window

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \prod_{-m \le j \le m, j \ne 0} p(w_{t+j}|w_t; \boldsymbol{\theta})$$

Logarithm is a strictly monotonic function so we can flip the sign, apply the $\log()$ and **minimize the loss instead:**

Minimize the cross-entropy loss $\leftrightarrow$ maximizing the likelihood of the data given the parameters

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = -\sum_{-m \le j \le m, j \ne 0} \log\left(p(w_{t+j}|w_t; \boldsymbol{\theta})\right)$$

Loss over a window, over all possible windows

We iterate over all possibile center words with $t \in \{1 \dots T\}$ maximizing the likelihood:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} p(w_{t+j}|w_t; \boldsymbol{\theta})$$

In practice, we minimize the average negative log-likelihood (loss):

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{T}\sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} \log\left(p(w_{t+j}|w_t; \boldsymbol{\theta})\right)$$

Minimize the cross-entropy loss is $\leftrightarrow$ maximizing the likelihood of the data given the parameters

We optimize the loss with SGD

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{T}\sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} \log\left(p(w_{t+j}|w_t; \boldsymbol{\theta})\right)$$

To optimize our $\boldsymbol{\theta}$ we have to make a small step in the negative direction of the gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\theta; w_{t-1}, w_t)$$

Pseudo-code of the training:

```
gamma = 1e-3
while not converged:
    loss = word2vec(corpus, theta)
    theta_grad = evaluate_gradient(loss, corpus, theta)
    theta = theta - gamma*theta_grad
```

Example: compute the gradient of the loss wrt center word embedding

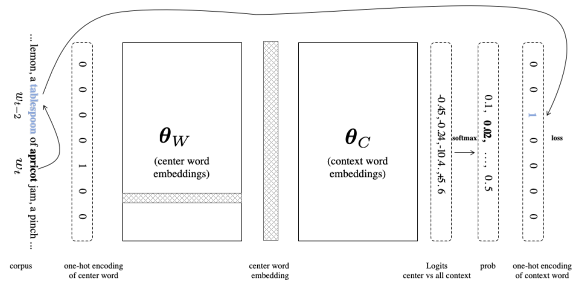
Define $\mathbf{u}_c \doteq \boldsymbol{\theta}_W[i]$ and $\mathbf{u}_o \doteq \boldsymbol{\theta}_C[gt]$. Compute the gradient of the loss wrt. to the embedding of the center word $\mathbf{u}_c$

$$\nabla_{\mathbf{u}_c}\mathcal{L}(\mathbf{u}_c, \mathbf{u}_o) = \nabla_{\mathbf{u}_c} - \log\left(\frac{\exp\left(\mathbf{u}_o \cdot \mathbf{u}_c^T\right)}{\sum_{v=1}^{V} \exp\left(\mathbf{u}_o \cdot \mathbf{u}_c^T\right)}\right)$$

Continue on the whiteboard

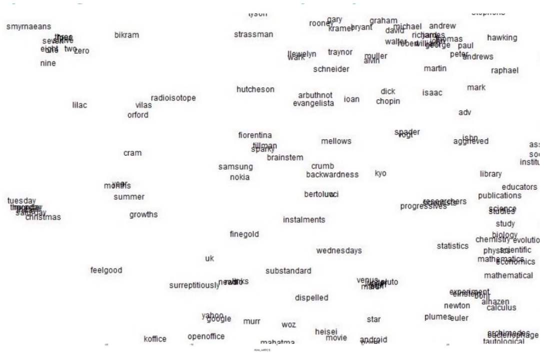
word2vec with Skip-Gram at a glance

... and why it can be seen as a tiny neural net.



word2vec learned space

word2vec minimizes the loss function by putting similar words (words that happen in same context) nearby in space



Taken from Stanford slide

Let's play with word2vec

Show available models in Gensim

You need Gensim.

In [25]:

```python
import gensim.downloader
# Show all available models in gensim-data
list(gensim.downloader.info()['models'].keys())
```

Out[25]:

```
['fasttext-wiki-news-subwords-300',
 'conceptnet-numberbatch-17-06-300',
 'word2vec-ruscorpora-300',
 'word2vec-google-news-300',
 'glove-wiki-gigaword-50',
 'glove-wiki-gigaword-100',
 'glove-wiki-gigaword-200',
 'glove-wiki-gigaword-300',
 'glove-twitter-25',
 'glove-twitter-50',
 'glove-twitter-100',
 'glove-twitter-200',
 '__testing_word2vec-matrix-synopsis']
```

Let's play with word2vec

This is a word2vec trained on entire Google News dataset, of about **100 billion words.**

`word2vec-google-news-300`

Model is 1.6GB so it will take time to download (it will show progress bar)!

In [26]:

```python
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
# [==================================================] 100.0% 1662.8/1662.8MB downloaded
```

The vocabulary

In [27]:

```python
N = len(wv.index_to_key)
print(f'vocabulary size is {N}')

for index, word in enumerate(wv.index_to_key):
    if index == 10:
        break
    print(f"word #{index} over {N} is {word}")
```

```
vocabulary size is 3000000
word #0 over 3000000 is </s>
word #1 over 3000000 is in
word #2 over 3000000 is for
word #3 over 3000000 is that
word #4 over 3000000 is is
word #5 over 3000000 is on
word #6 over 3000000 is ##
word #7 over 3000000 is The
word #8 over 3000000 is with
word #9 over 3000000 is said
```

Accessing a word embedding

In [28]:

```python
vec_king = wv['king']
print(f'vector shape is {vec_king.shape}')
```

```
vector shape is (300,)
```

In [29]:

```python
import pandas as pd
pd.DataFrame(dict(king=vec_king.tolist())).T
```

Out[29]:

	0	1	2	3	4	5	6	7	8	9	...	290	291
king	0.125977	0.029785	0.008606	0.139648	-0.025635	-0.036133	0.111816	-0.198242	0.05127	0.363281	...	-0.004669	-0.244141	-0.20

1 rows × 300 columns

Watch out if you do not pass before through the vocabulary before asking the model

In [30]:

```python
try:
    vec_cameroon = wv['cameroon']
except KeyError:
    print("The word 'cameroon' does not appear in this model")
```

```
The word 'cameroon' does not appear in this model
```

Word similarities!

In [31]:

```python
pairs = [
    ('car', 'minivan'),   # a minivan is a kind of car
    ('car', 'bicycle'),   # still a wheeled vehicle
    ('car', 'airplane'),  # ok, no wheels, but still a vehicle
    ('car', 'cereal'),    # ... and so on
    ('car', 'communism'),
    ('car', 'cartoon'),
]
for w1, w2 in pairs:
    print('%r\t%r\t%.2f' % (w1, w2, wv.similarity(w1, w2)))
```

```
'car'   'minivan'       0.69
'car'   'bicycle'       0.54
'car'   'airplane'      0.42
'car'   'cereal'        0.14
'car'   'communism'     0.06
'car'   'cartoon'       0.03
```

Most similar

In [32]:

```python
wv.most_similar(positive=['car', 'minivan'], topn=5)
```

Out[32]:

```
[('SUV', 0.8532192707061768),
 ('vehicle', 0.8175783753395081),
 ('pickup_truck', 0.7763688564300537),
 ('Jeep', 0.7567334175109863),
 ('Ford_Explorer', 0.7565720081329346)]
```

Outlier detection

```
wv.doesnt_match(['fire', 'water', 'land', 'sea', 'air', 'car'])
```

```
'car'
```

Analogy using the learned embedding space

- word2vec learns relationships between words automatically
- Arithmetic of the vectors has surprising properties!

`a is to b as a* is to what?`

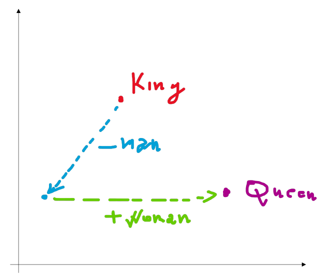
`Man is to King as Woman is to ______`

Algebric way

$$\boldsymbol{\theta}_W(\text{king}) - \boldsymbol{\theta}_W(\text{man}) + \boldsymbol{\theta}_W(\text{woman}) \approx \boldsymbol{\theta}_W(\text{queen})$$

Analogy using the learned embedding space

Geometric way (parallelo gram model)

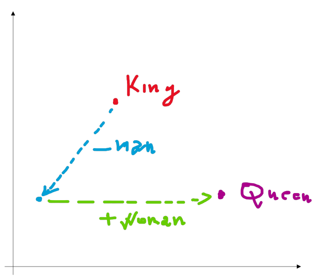


Analogy using the learned embedding space

Optimization

$$\mathbf{a} : \mathbf{b} = \mathbf{a}^* : \mathbf{b}^*?$$

$$\mathbf{b}^* = \arg\max_{\mathbf{x}} \text{sim}(\mathbf{x}, \mathbf{b} - \mathbf{a} + \mathbf{a}^*)$$



Analogy

```
def analogy(x1, x2, y1):
    result = wv.most_similar(positive=[y1, x2], negative=[x1])
    return result[0][0]
```

```
analogy('man', 'king', 'woman')
```

```
'queen'
```

```
analogy('australia', 'beer', 'france')# does not work well with Glove should give you champagne
```

```
'beers'
```

```
analogy('obama', 'clinton', 'reagan')
```

```
'kerry'
```

```
analogy('tall', 'tallest', 'long')
```

```
'longest'
```

```
analogy('good', 'fantastic', 'bad')
```

```
'horrible'
```