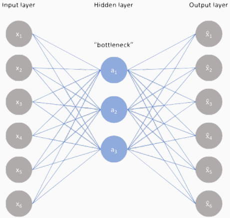
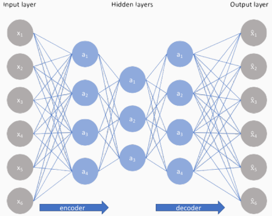
CS-GY 6923: Lecture 13
Semantic Embeddings, Beyond Autoencoders

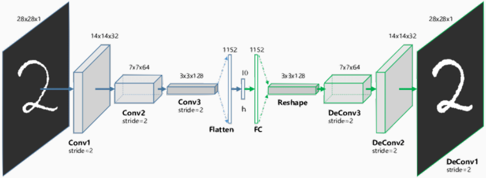
NYU Tandon School of Engineering, Prof. Christopher Musco

- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ be our model.
- Let $L_{\boldsymbol{\theta}}$ be a loss function. E.g. squared loss: $L_{\boldsymbol{\theta}}(\mathsf{x}) = \|\mathsf{x} - f_{\boldsymbol{\theta}}(\mathsf{x})\|_2^2$.
- Train model: $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \sum_{i=1}^{n} L_{\boldsymbol{\theta}}(\mathsf{x})$.

Important property of autoencoders: no matter what architecture is use, there must always be a **bottleneck** with fewer parameters than the input. The bottleneck ensures information is “distilled” from low-level features to high-level features.





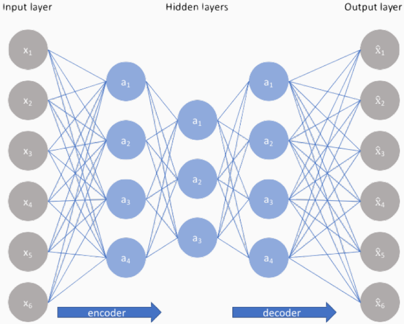


Separately name mapping from input to bottleneck, and from bottleneck to output.

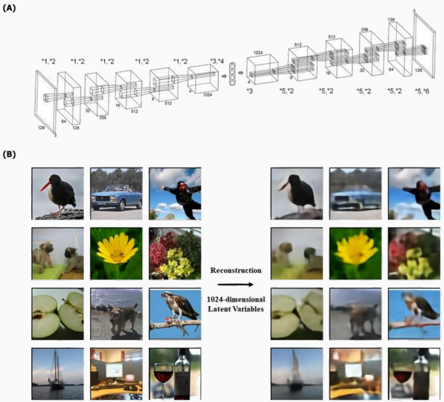
Encoder: $e : \mathbb{R}^d \to \mathbb{R}^k$

Decoder: $d : \mathbb{R}^d \to \mathbb{R}^k$

$$f(\mathbf{x}) = d(e(\mathbf{x}))$$



Example image reconstructions from autoencoder:



https://www.biorxiv.org/content/10.1101/214247v1.full.pdf

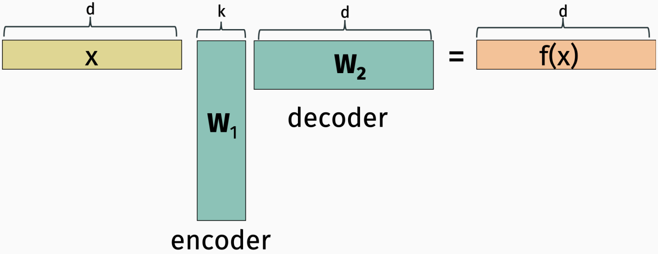
Input parameters: $d = 49152$.

Bottleneck "latent" parameters: $k = 1024$.

Lots of applications:

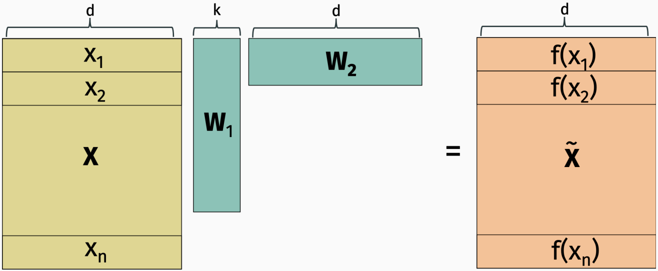
- Data compression.
- Data denoising and repair.
- Data synthesis.
- Feature learning.

Simple linear autoencoder:



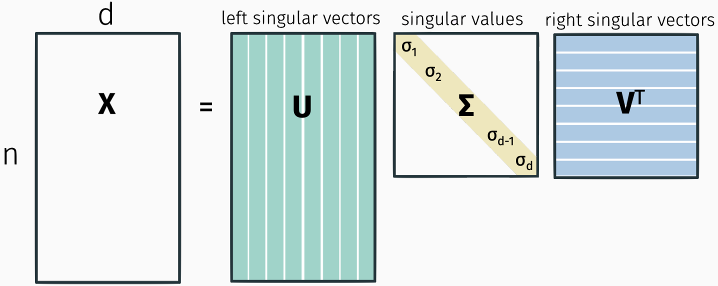
$$f(x)^T = x^T W_1 W_2$$

Given training data set $\mathbf{x}_1, \ldots, \mathbf{x}_n$, let $\mathbf{X}$ denote our data matrix. Let $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$ denote the autoencoded data. Want to minimize $\|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2$.



$\tilde{\mathbf{X}}$ is a low-rank (rank k) matrix. The optimal choice of $\mathbf{W}_1$ and $\mathbf{W}_2$ can be found using algorithms for optimal low-rank approximation.

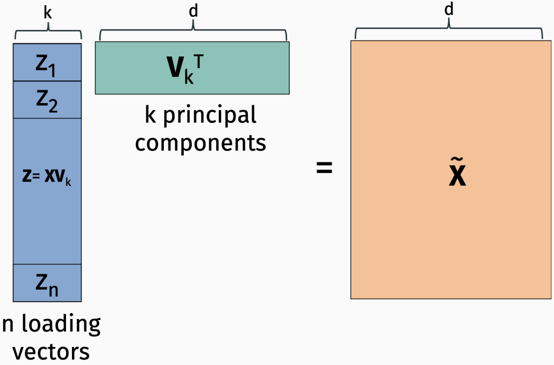
Any matrix **X** can be written using its **singular value decomposition**:



Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. **U** and **V** are orthogonal matrices.

Can be computed in $O(nd^2)$ time (faster with approximation algos).

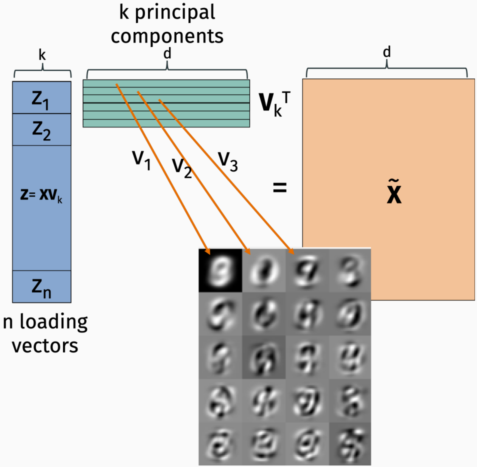
We obtain an optimal autoencoder by setting $\mathbf{W}_1 = \mathbf{V}_k$, $\mathbf{W}_2 = \mathbf{V}_k^T$. $f(\mathbf{x}) = \mathbf{x}\mathbf{V}_k\mathbf{V}_k^T$.



For many natural data sets, get a good approximation even when k is chosen to be relatively small compared to d.

What do principal components and loading vectors look like?

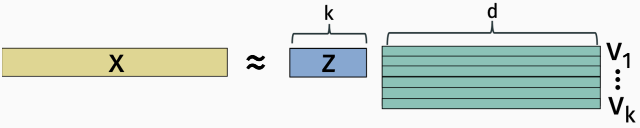
MNIST **principal components**:

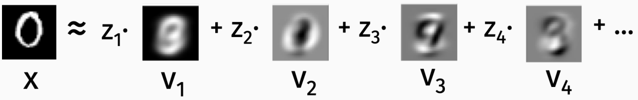


Often principal components are difficult to interpret.

What do the **loading vectors** looks like?

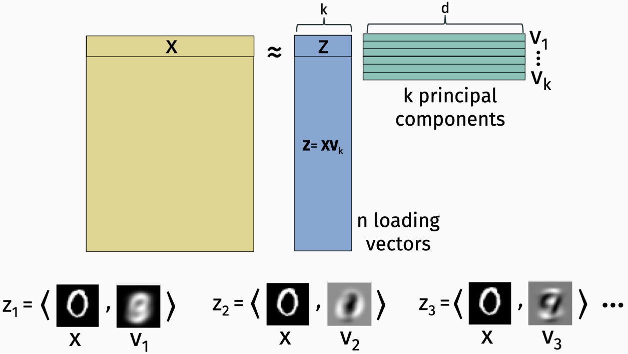
The loading vector $\mathbf{z}$ for an example $\mathbf{x}$ contains coefficients which recombine the top k principal components $\mathbf{v}_1, \ldots, \mathbf{v}_k$ to approximately reconstruct $\mathbf{x}$.



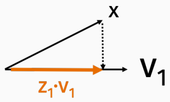


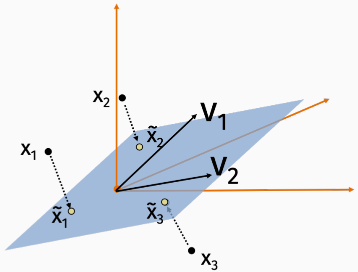
Provide a short “finger print” for any image $\mathbf{x}$ which can be used to reconstruct that image.

For any $\mathbf{x}$ with loading vector $\mathbf{z}$, $\mathbf{z}_i$ is the inner product similarity between $\mathbf{x}$ and the i^{th} principal component $\mathbf{v}_i$.



So we approximate $\mathbf{x} \approx \tilde{\mathbf{x}} = \langle \mathbf{x}, \mathbf{v}_1 \rangle \cdot \mathbf{v}_1 + \ldots + \langle \mathbf{x}, \mathbf{v}_k \rangle \cdot \mathbf{v}_k$.

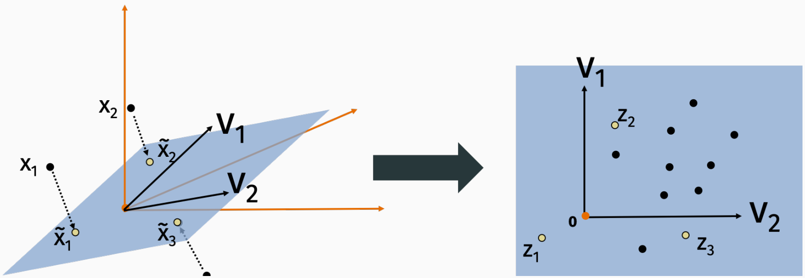




Since $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are orthonormal, this operation is a **projection** onto first k principal components.

I.e. we are projecting $\mathbf{x}$ onto the k-dimensional subspace spanned by $\mathbf{v}_1, \ldots, \mathbf{v}_k$.

For an example $\mathbf{x}_i$, the loading vector $\mathbf{z}_i$ contains the coordinates in the projection space:



Visual way of seeing what we argued last time: inner products and distances between loading vectors should approximate inner products and distances in the original space.

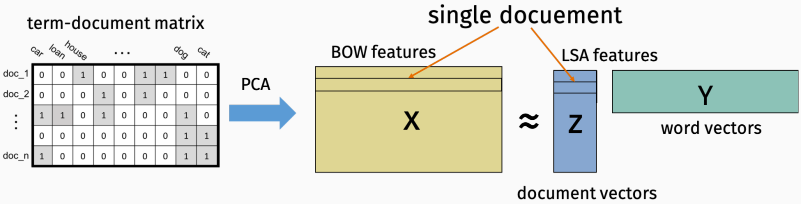
Word-document matrices tend to be low rank.

	car	loan	house	…				dog	cat
doc_1	0	0	1	0	0	1	1	0	0
doc_2	0	0	0	1	0	1	0	0	0
⋮	1	1	0	1	0	0	0	1	0
	0	0	0	0	0	0	0	1	1
doc_n	1	0	0	0	0	0	0	1	1

Documents tend to fall into a relatively small number of different categories, which use similar sets of words:

- **Financial news:** *markets, analysts, dow, rates, stocks*
- **US Politics:** *president, senate, pass, slams, twitter, media*
- **StackOverflow posts:** *python, help, convert, javascript*

Latent semantic analysis = PCA applied to a word-document matrix (usually from a large corpus). One of the most fundamental techniques in **natural language processing** (NLP).



Each column of **z** corresponds to a latent "category" or "topic". Corresponding row in **Y** corresponds to the "frequency" with which different words appear in documents on that topic.

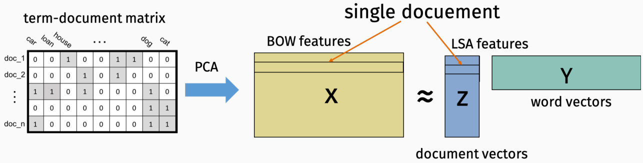
Similar documents have similar LSA document vectors. I.e. $\langle \mathbf{z}_i, \mathbf{z}_j \rangle$ is large.

- $\mathbf{z}_i$ provides a more compact "finger print" for documents than the long bag-of-words vectors. Useful for e.g search engines.
- Comparing document vectors is often more effective than comparing raw BOW features. Two documents can have $\langle \mathbf{z}_i, \mathbf{z}_j \rangle$ large even if they have no overlap in words. E.g. because both share a lot of words with words with another document k, or with a bunch of other documents.

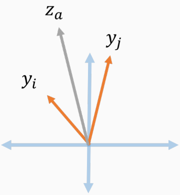
Same fingerprinting idea was also important in early facial recognition systems based on "eigenfaces":

Each image above is one of the principal components of a dataset containing images of faces.

WORD EMBEDDINGS

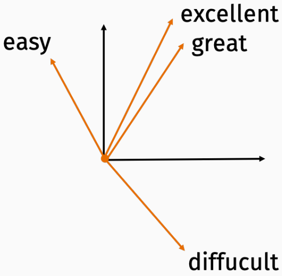


- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$ when doc_a contains $word_i$.
- If $word_i$ and $word_j$ both appear in doc_a, then $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$, so we expect $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ to be large.



If two words appear in the same document their, word vectors tend to point more in the same direction.

Result: Map words to numerical vectors in a semantically meaningful way. Similar words map to similar vectors. Dissimilar words to dissimilar vectors.



Extremely useful "side-effect" of LSA.

Capture e.g. the fact that "great" and "excellent" are near synonyms. Or that "difficult" and "easy" are antonyms.

Word embeddings are considered a type of semantic embedding.

They can be obtain by training on a very large corpus of text (e.g. Wikipedia, Twitter, news data sets) and then used for many different tasks in the future as an initial way to convert text data to numerical data.

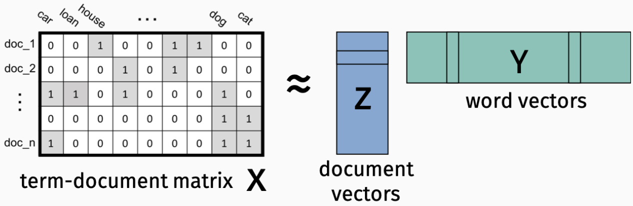
Review 1: *Very small and handy for traveling or camping. Excellent quality, operation, and appearance.*

Review 2: *So far this thing is great. Well designed, compact, and easy to use. I'll never use another can opener.*

Review 3: *Not entirely sure this was worth* \$20. *Mom couldn't figure out how to use it and it's fairly difficult to turn for someone with arthritis.*

Goal is to classify reviews as "positive" or "negative".

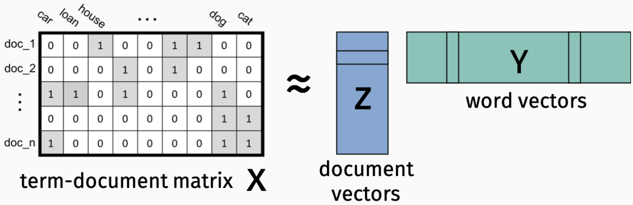
Another view on word embeddings from LSA:



We chose $\mathbf{Z}$ to equal $\mathbf{X}\mathbf{V}_k = \mathbf{U}_k\mathbf{\Sigma}_k$ and $\mathbf{Y} = \mathbf{V}_k^T$.

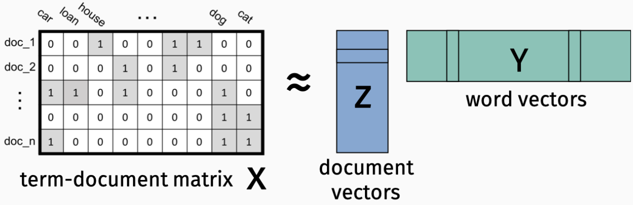
Could have juse as easily set $\mathbf{Z} = \mathbf{U}_k$ and $\mathbf{Y} = \mathbf{\Sigma}_k\mathbf{V}_k^T$, so $\mathbf{Z}$ has orthonormal columns.

Another view on word embeddings from LSA:



- $X \approx ZY$
- $X^T X \approx Y^T Z^T Z Y = Y^T Y$
- So for $word_i$ and $word_j$, $\langle y_i, y_j \rangle \approx [X^T X]_{i,j}$.

What does the i, j entry of $X^T X$ reprent?

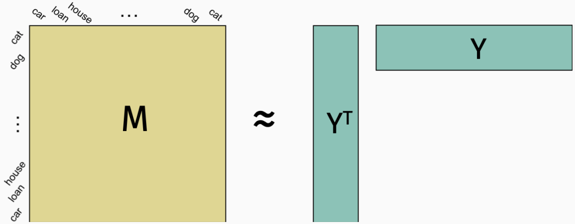


What does the i, j entry of $X^T X$ reprent?

$\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ is larger if $word_i$ and $word_j$ appear in more documents together (high value in **word-word co-occurrence matrix**, $\mathbf{X}^T\mathbf{X}$). Similarity of word embeddings mirrors similarity of word context.

General word embedding recipe:

1. Choose similarity metric $k(word_i, word_j)$ which can be computed for any pair of words.
2. Construct similarity matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ with $\mathbf{M}_{i,j} = k(word_i, word_j)$.
3. Find low rank approximation $\mathbf{M} \approx \mathbf{Y}^T\mathbf{Y}$ where $\mathbf{Y} \in \mathbb{R}^{k \times n}$.
4. Columns of $\mathbf{Y}$ are word embedding vectors.



How do current state-of-the-art methods differ from LSA?

- Similarity based on co-occurrence in smaller chunks of words. E.g. in sentences or in any consecutive sequences of 3, 4, or 10 words.
- Usually transformed in non-linear way. E.g. $k(word_i, word_j) = \frac{p(i,j)}{p(i)p(j)}$ where $p(i, j)$ is the frequency both i, j appeared together, and $p(i)$, $p(j)$ is the frequency either one appeared.

Computing word similarities for “window size” 4:

The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

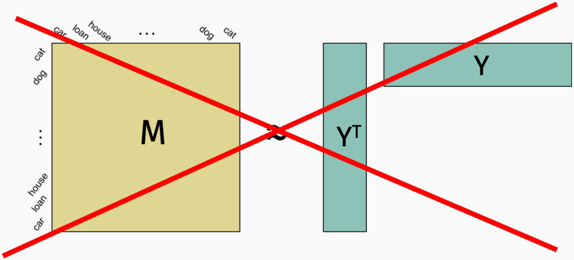
The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

	dog	park	crowded	the
dog	0	2	0	3
park	2	0	1	2
crowded	0	1	0	0
the	3	2	0	0

Current state of the art models: `GloVE`, `word2vec`.

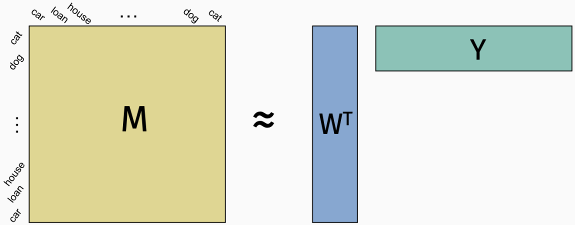
- `word2vec` was originally presented as a shallow neural network model, but it is equivalent to matrix factorization method (Levy, Goldberg 2014).
- For `word2vec`, similarity metric is the “point-wise mutual information”: $\log \frac{p(i,j)}{p(i)p(j)}$.



SVD will not return a symmetric factorization in general. In fact, if $\mathbf{M}$ is not positive semidefinite[1] then the optimal low-rank approximation does not have this form.

[1] I.e., κ is not a positive semidefinite kernel.

CAVEAT ABOUT FACTORIZATION

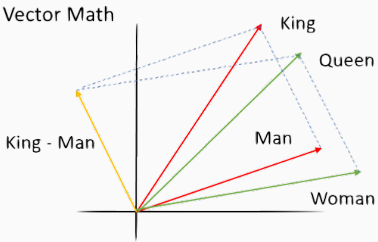


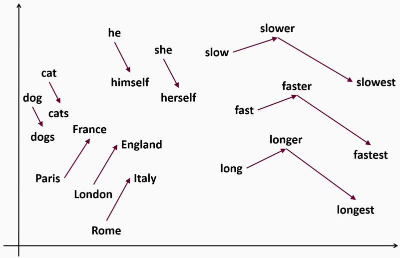
- For each word i we get a left and right embedding vector $\mathbf{w}_i$ and $\mathbf{y}_i$. It's reasonable to just use one or the other.
- If $\langle \mathbf{y}_i, \mathbf{y}_i \rangle$ is large and positive, we expect that $\mathbf{y}_i, \mathbf{y}_i$ have similar similarity scores with other words, so they typically are still related words.
- Another option is to use as your features for a word the concatenation $[\mathbf{w}_i, \mathbf{y}_i]$

If you want to use word embeddings for your project, the easiest approach is to download pre-trained word vectors:

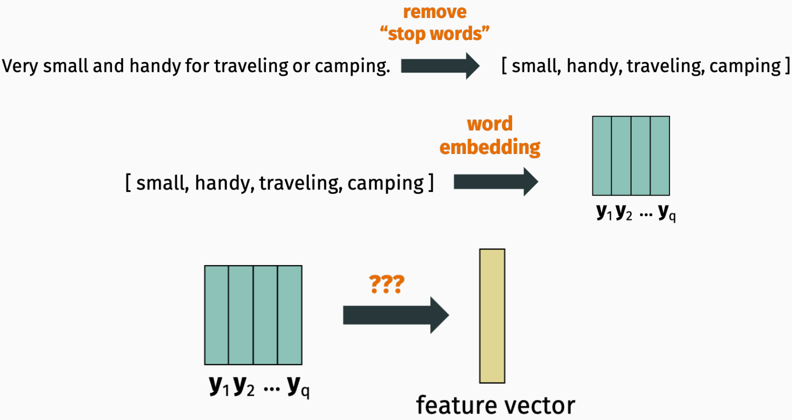
- Original gloVe website: `https://nlp.stanford.edu/projects/glove/`.
- Compilation of many sources: `https://github.com/3Top/word2vec-api`

Lots of cool demos online for what can be done with these embeddings. E.g. “vector math” to solve analogies.



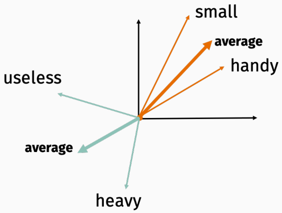


How to go from word embeddings to features for a whole sentence or chunk of text?

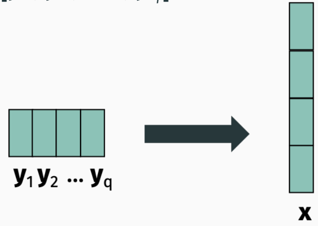


A few simple options:

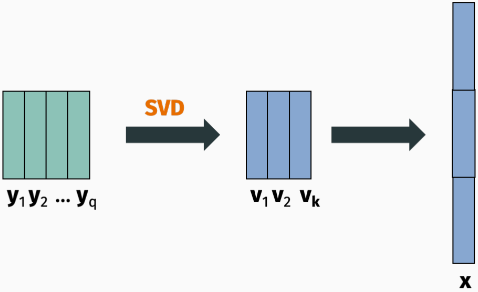
Feature vector $\mathbf{x} = \frac{1}{q}\sum_{i=1}^{q} \mathbf{y}_q$.



Feature vector $\mathbf{x} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_q]$.

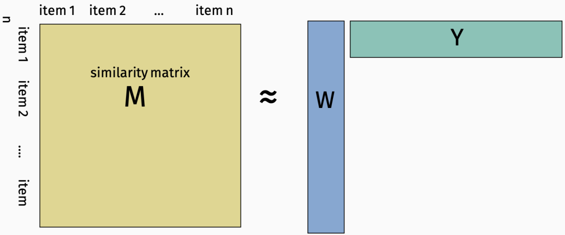


To avoid issues with inconsistent sentence length, word ordering, etc., can concatenate a fixed number of top principal components of the matrix of word vectors:



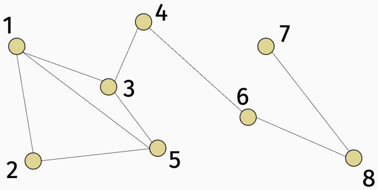
There are much more complicated approaches that account for word position in a sentence. Lots of pretrained libraries available (e.g. Facebook's `InferSent`).

The same approach used for word embeddings can be used to obtain meaningful numerical features for any other data where there is a natural notion of similarity.



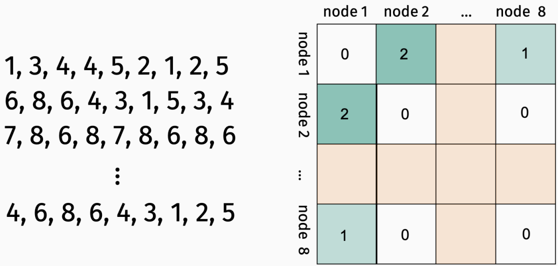
For example, the items could be nodes in a social network graph. Maybe be want to predict an individuals age, level of interest in a particular topic, political leaning, etc.

Generate random walks (e.g. “sentences” of nodes) and measure similarity by node co-occurence frequency.



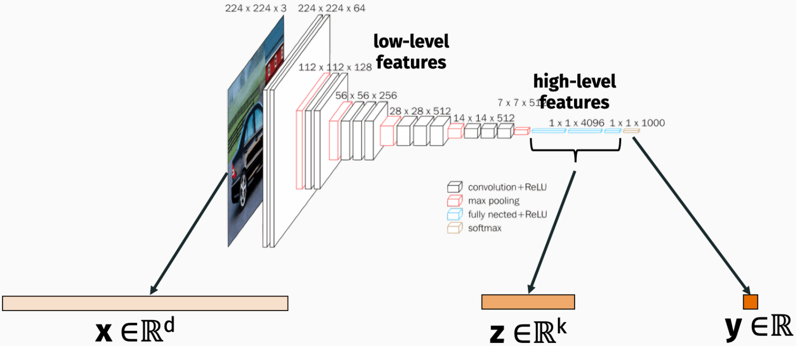
1, 3, 4, 4, 5, 2, 1, 2, 5
6, 8, 6, 4, 3, 1, 5, 3, 4
7, 8, 6, 8, 7, 8, 6, 8, 6
⋮
4, 6, 8, 6, 4, 3, 1, 2, 5

Again typically normalized and apply a non-linearity (e.g. log) as in word embeddings.



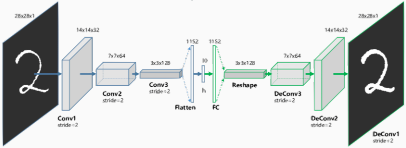
Popular implementations: `DeepWalk`, `Node2Vec`. Again initially derived as simple neural network models, but are equivalent to matrix-factorization (Qiu et al. 2018).

While they have many applications, one of the original goals of autoencoders was to learn "high level" features when we did not have substantial training data.



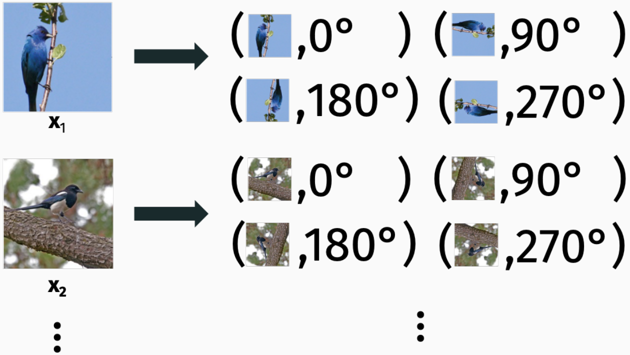
Typical supervised approach to transfer learning.

Growing realization: The types of features needed to precisely reconstruct an image (e.g. with small ℓ_2 error) don't exactly match up with the features required to understand an image.



Automatically create a supervised learning problem with a "simpler" task than image reconstruction.

Example: Rotation Learning.



4 class learning problem. Train a supervised neural network

Example: AlphaCode (released this year by Google/Deepmind). Can process a description of a task and output correct code to complete the task.

You are given two strings s and t, both consisting of lowercase English letters. You are going to type the string s character by character, from the first character to the last one.

When typing a character, instead of pressing the button corresponding to it, you can press the "Backspace" button. It deletes the last character you have typed among those that aren't deleted yet (or does nothing if there are no characters in the current string). For example, if s is "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (the first press of Backspace deletes no character, and the second press deletes the character 'c'). Another example, if s is "abcaa" and you press Backspace instead of the last two letters, then the resulting text is "a".

Your task is to determine whether you can obtain the string t, if you type the string s and press "Backspace" instead of typing several (maybe zero) characters of s.

Input

The first line contains a single integer q ($1 \le q \le 10^5$) — the number of test cases.

The first line of each test case contains the string s ($1 \le |s| \le 10^5$). Each character of s is a lowercase English letter.

The second line of each test case contains the string t ($1 \le |t| \le 10^5$). Each character of t is a lowercase English letter.

It is guaranteed that the total number of characters in the strings over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, print "YES" if you can obtain the string t by typing the string s and replacing some characters with presses of "Backspace" button, or "NO" if you cannot.

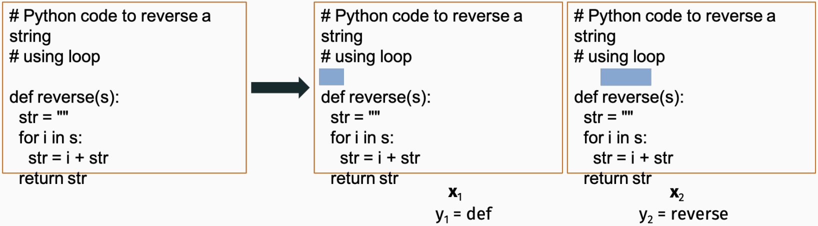
You may print each letter in any case (YES, yes, Yes will all be recognized as positive answer, NO, no and nO will all be recognized as negative answer).

Input
4 ababa ba ababa bb aaa aaaa aababa ababa
Output
YES NO NO YES

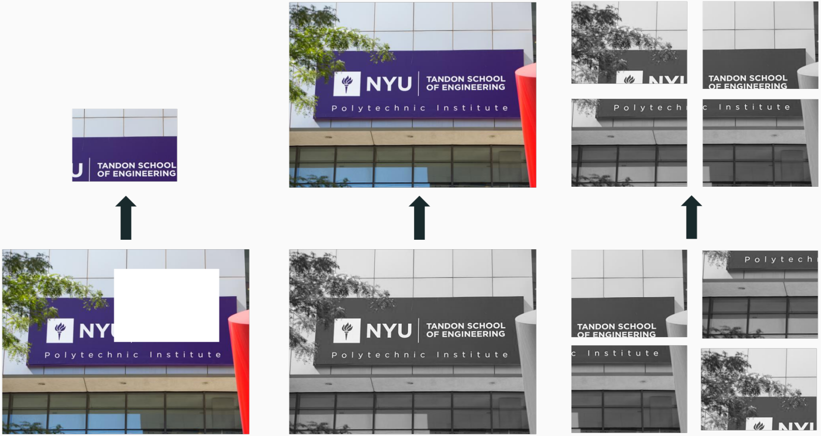
```python
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
        if a[0]==b[0]:
            c.append(b.pop(0))
            a.pop(0)
        elif a[0]!=b[0] and len(a)!=1:
            a.pop(0)
            a.pop(0)
        elif a[0]!=b[0] and len(a)==1:
            a.pop(0)
    if len(b)==0:
        print("YES")
    else:
        print("NO")
```

There is not a whole lot of training data for this sort of problem!

Key Component in AlphaCode: Semi-supervised learning using code on Github.



There are a lot of different possibilities!



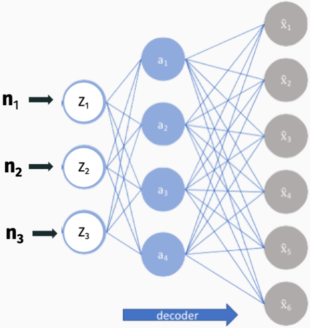
In-painting task

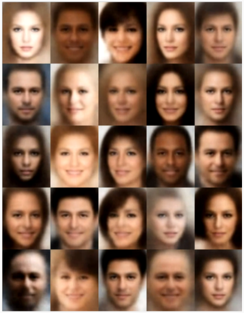
Colorization task

Puzzle task

- **Advantage:** Self-supervised learning tends to outperform autoencoders for feature learning (e.g. better performance in transfer learning tasks).
- **Disadvantage:** There is no “decoder” function, so no natural way to use these techniques for e.g. data compression, super resolution, or synthetic data generation.

Autoencoder approach to generative ML: Pretrain auto-encoder. Feed random inputs into decode to produce random realistic outputs.

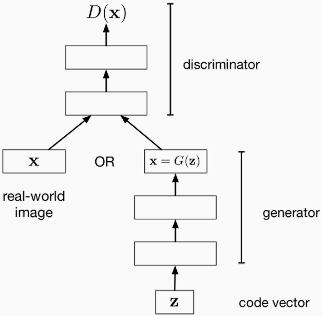




Pretty cool, but tends to produce images with immediately recognizable flaws (e.g. soft edges, high-frequency artifacts).

Lots of efforts to hand-design regularizers that penalize images that don't look realisitic to the human eye.

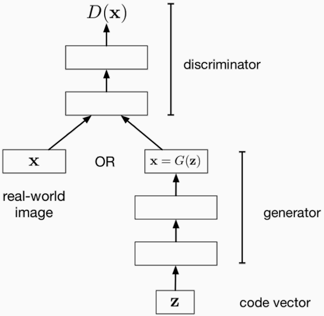
Main idea behind GANs: Use machine learning to automatically encourage realistic looking images.



Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be real images and let $\mathbf{z}_1, \ldots, \mathbf{z}_m$ be random code vectors. The goal of the discriminator is to output a number between $[0, 1]$ which is close to 0 if the image is fake, close to 1 if it's real.

Train weights of discriminator $D_{\boldsymbol{\theta}}$ to minimize:

$$\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} -\log\left(D_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}'}(\mathbf{z}_i))\right)$$



Goal of the generator $G_{\boldsymbol{\theta}'}$ is the opposite. We want to maximize:

$$\max_{\boldsymbol{\theta}'} \sum_{i=1}^{n} -\log\left(D_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}'}(\mathbf{z}_i)\right)$$

This is called an “adversarial loss function”. D is playing the role of the adversary.

$$\boldsymbol{\theta}^*, \boldsymbol{\theta}'^* \text{ solve } \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\theta}'} \sum_{i=1}^{n} -\log\left(D_{\boldsymbol{\theta}}(\mathbf{x}_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}'}(\mathbf{z}_i)\right)$$

This is called a minimax optimization problem. Really tricky to solve in practice.

- **Repeatedly play:** Fix one of $\boldsymbol{\theta}^*$ or $\boldsymbol{\theta}'^*$, train the other to convergence, repeat.
- **Simultaneous gradient descent:** Run a single gradient descent step for each of $\boldsymbol{\theta}^*, \boldsymbol{\theta}'^*$ and update D and G accordingly. Difficult to balance learning rates.
- Lots of tricks (e.g. slight different loss functions) can help.


ORIGINAL
DEEPFAKE