CS-GY 6923: Lecture 12
Autoencoders, Principal Component Analysis

NYU Tandon School of Engineering, Prof. Christopher Musco

State-of-the-art supervised learning models like neural networks learn **very good features**.

But they require lots and lots of data. Imagenet has 14 million unlabeled images. Mostly of everyday objects.

What if you want to apply deep convolutional networks to a problem where you do not have a lot of **labeled data** in the first place?

Example: Classify images of different Quidditch balls.

Real example: Classify images of insects for use in agricultural applications in new localities.

Zero-Shot Insect Detection via Weak Language Supervision

Benjamin Feuer,[1] Ameya Joshi,[1] Minsu Cho,[1] Kewal Jani,[1] Shivani Chiranjeevi, [2] Zi Kang Deng, [3] Aditya Balu, [2] Asheesh K. Singh, [2] Soumik Sarkar, [2] Nirav Merchant, [3] Arti Singh, [2] Baskar Ganapathysubramanian, [2] Chinmay Hegde [1]

[1] New York University
[2] Iowa State University
[3] University of Arizona

Aedes Vexans

Creatonotos Gangis

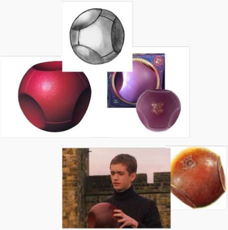
Daphnis Neril

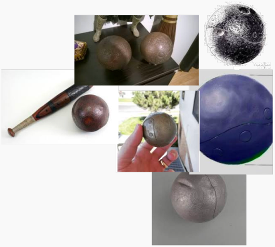
Hypena Deceptalis

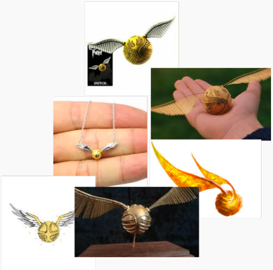
Pyralis Farinalis

ONE-SHOT LEARNING

A human could probably achieve near perfect classification accuracy even given access to a **single labeled example** from each class:



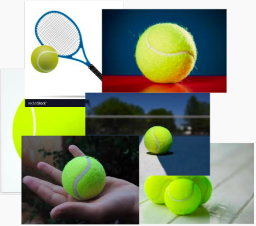




Major question in ML: How? Can we design ML algorithms which can do the same?

TRANSFER LEARNING

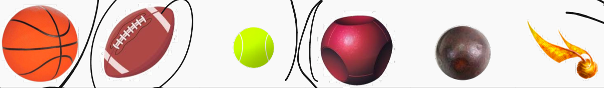
Transfer knowledge from one task we already know how to solve to another.



For example, we have learned from past experience that balls used in sports have consistent shapes, colors, and sizes. These features can be used to distinguish balls of different type.

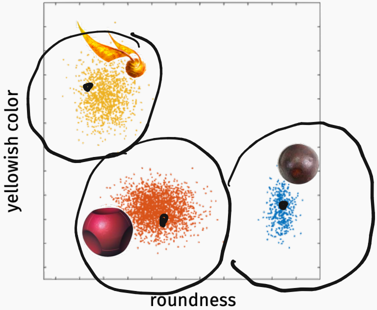
Examples of possible high-level features a human would learn:

Classes



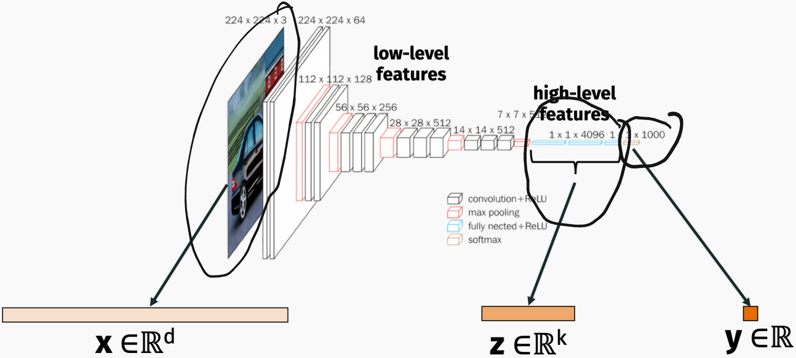
Features						
roundness	1	.1	1	.6	1	.4
size relative to human hand	10	7	2	7	5	1
yellowish color	.2	.1	1	.1	0	.9

If these features are highly informative (i.e. lead to highly separable data) few training examples are needed to learn.



Might suffice to classify ball using nearest training example in feature space, even if just a handful of training examples.

Empirical observation: Features learned when training models like deep neural nets seem to capture exactly these sorts of high-level properties.



Even if we can't put into words what each feature in **z** means...

This is now a common technique in computer vision:

1. Download network trained on large image classification dataset (e.g. Imagenet).
2. Extract features $\mathbf{z}$ for any new image $\mathbf{x}$ by running it through the network up until layer before last.
3. Use these features in a simpler machine learning algorithm that requires less data (nearest neighbor, logistic regression, etc.).

This approach has even been used on the quidditch problem:
`github.com/thatbrguy/Object-Detection-Quidditch`

Transfer learning: Lots of labeled data for one problem makes up for little labeled data for another.

But what if we don't even have labeled data for a sufficiently related problem?

How to extract features in a data-driven way from (unlabeled data) is one of the central problems in **unsupervised learning**.

- **Supervised learning:** All input data examples come with targets/labels. What machines have been really good at for the past 8 years.
- **Unsupervised learning:** No input data examples come with targets/labels. Interesting problems to solve include clustering, anomaly detection, semantic embedding, etc.
- **Semi-supervised learning:** Some (typically very few) input data examples come with targets/labels. What human babies are really good at, and we have recently made machines a lot better at.

Simple but clever idea: If we have inputs $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ but few or no targets $y_1, \ldots, y_n$, just make the inputs the targets.

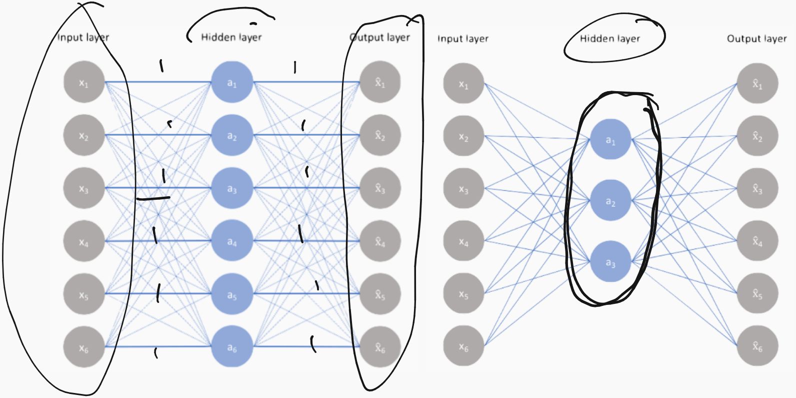
- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ be our model.
- Let $L_{\boldsymbol{\theta}}$ be a loss function. E.g. squared loss: $L_{\boldsymbol{\theta}}(\mathbf{x}) = \|\mathbf{x} - f_{\boldsymbol{\theta}}(\mathbf{x})\|_2^2$.
- Train model: $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}}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

If $f_{\boldsymbol{\theta}}$ is a model that incorporates feature learning, then these features can be used for supervised tasks.

$f_{\boldsymbol{\theta}}$ is called an autoencoder. It maps input space to input space (e.g. images to images, french to french, PDE solutions to PDE solutions).

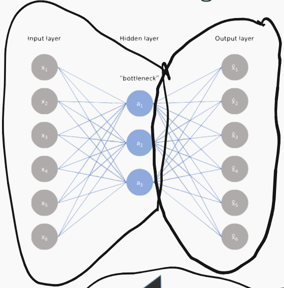
AUTOENCODER

Two examples of autoencoder architectures:

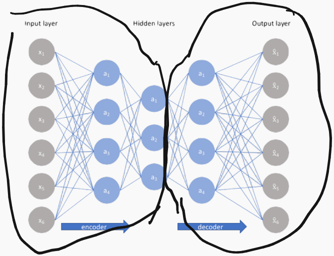


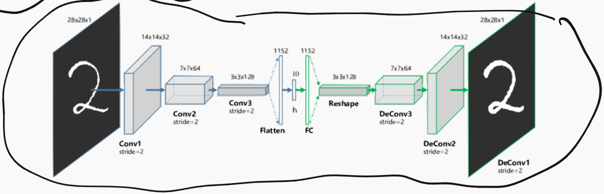
Which would lead to better feature learning?

Important property of autoencoders: no matter the architecture, 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.



Encoder Decoder

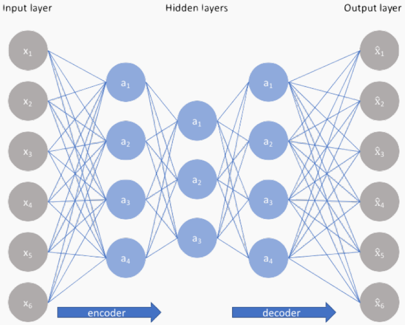




Separately name the mapping from input to bottleneck and from bottleneck to output.

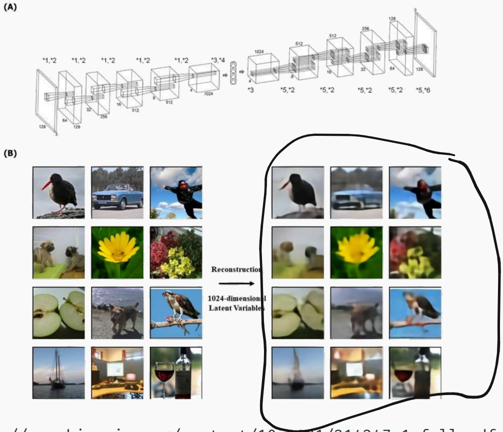
Encoder: $e : \mathbb{R}^d \rightarrow \mathbb{R}^k$ **Decoder:** $d : \mathbb{R}^d \rightarrow \mathbb{R}^k$

$$f(x) = d(e(x))$$



Often symmetric, but does not have to be.

Example image reconstructions from autoencoder:



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

Input parameters: $d = 49152$.

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

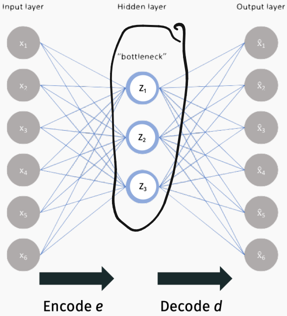
The best autoencoders do not work as well as supervised methods for feature extraction, but they require no labeled data.[1]

There are a lot of cool applications of autoencoders beyond feature learning!

- Learned data compression.
- Denoising and in-painting.
- Data/image synthesis.

[1] Recent progress on self-supervised learning achieves the best of both worlds – state-of-the-art feature learning with no labeled data.

Due to their bottleneck design, autoencoders perform **dimensionality reduction** and thus data compression.



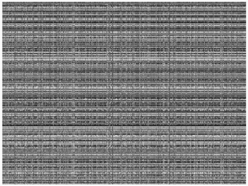
latent features z

Given input $\mathbf{x}$, we can completely recover $f(\mathbf{x})$ from $\mathbf{z} = e(\mathbf{x})$. $\mathbf{z}$ typically has many fewer dimensions than $\mathbf{x}$ and for a typical image $f(\mathbf{x})$ will closely approximate $\mathbf{x}$.

The best lossy compression algorithms are tailor made for specific types of data:

- JPEG 2000 for images
- MP3 for digital audio.
- MPEG-4 for video.

All of these algorithms take advantage of specific structure in these data sets. E.g. JPEG assumes images are locally "smooth".



With enough input data, autoencoders can be trained to find this structure on their own.

Proposed method, 5908 bytes (0.167 bit/px), PSNR: luma 23.38 dB/chroma 31.86 dB, MS-SSIM: 0.9219

Proposed method, 6021 bytes (0.170 bit/px), PSNR: 24.12 dB, MS-SSIM: 0.9292

JPEG 2000, 5908 bytes (0.167 bit/px), PSNR: luma 23.24 dB/chroma 31.04 dB, MS-SSIM: 0.8803

JPEG 2000, 6037 bytes (0.171 bit/px), PSNR: 23.47 dB, MS-SSIM: 0.9036

"End-to-end optimized image compression", Ballé, Laparra, Simoncelli

Need to be careful about how you choose loss function, design the network, etc. but can lead to much better image compression than "hand-tuned" algorithms like JPEG.

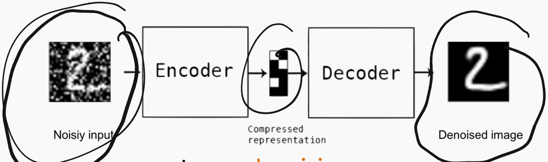


Image denoising

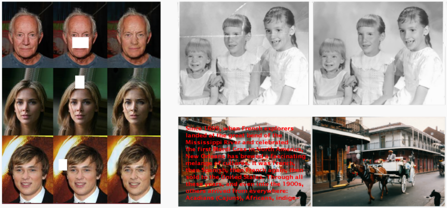
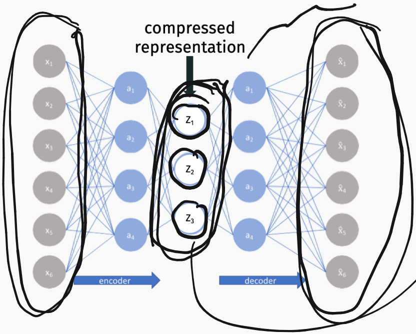


Image inpainting

Train autoencoder on <u>uncorrupted</u> images (unsupervised). Pass corrupted image **x** through autoencoder and return $f(\mathbf{x})$ as repaired result.

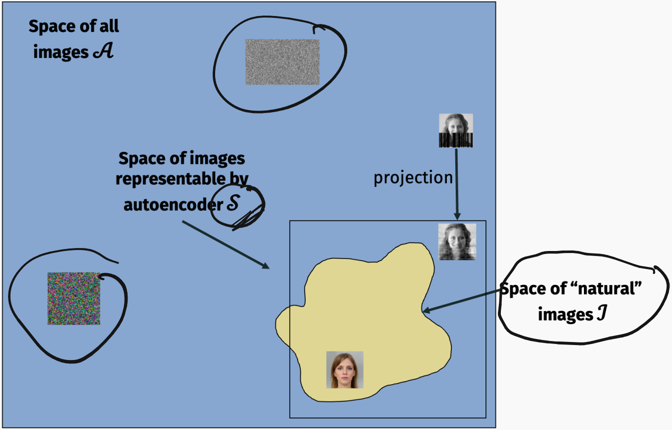
Why does this work?



Consider $128 \times 128 \times 3$ images with pixels values in $0, 1 \ldots, 255$. How many possible images are there?

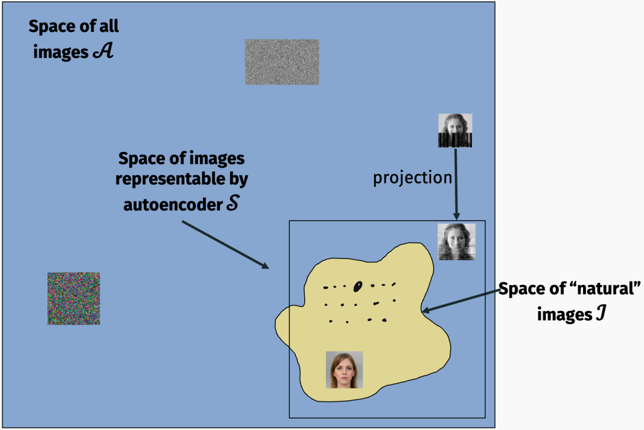
If $\mathbf{z}$ holds k values in $0, .1, .2, \ldots, 1$, how many unique images $\mathbf{w}$ can be output by the autoencoder function f?

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



For a good (accurate, small bottleneck) autoencoder, $\mathcal{S}$ will closely approximate $\mathcal{I}$. Both will be much smaller than $\mathcal{A}$.

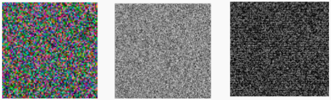
AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

Suppose we want to generate a random natural image. How might we do that?

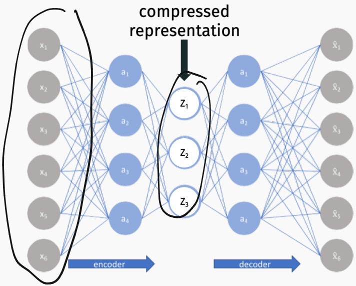
- **Option 1**: Draw each pixel value in $\mathbf{x}$ uniformly at random. Draws a random image from $\mathcal{A}$.



- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.

How do we randomly select an image from $\mathcal{S}$?

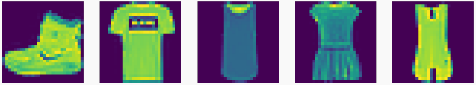
How do we randomly select an image **x** from $\mathcal{S}$?

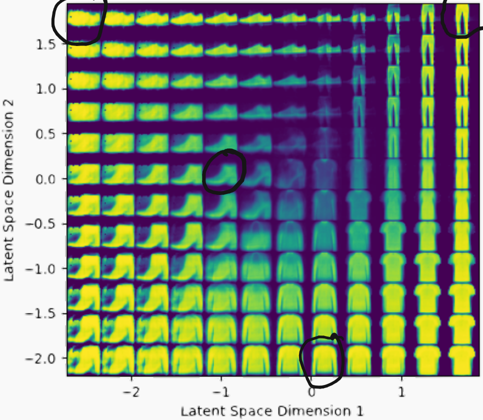


Randomly select code **z**, then set $\mathbf{x} = d(\mathbf{z})$.[2]

[2] Lots of details to think about here. In reality, people use "variational autoencoders" (VAEs), which are a natural modification of AEs.

Teal created a demo for the "Fashion MNIST" data set:





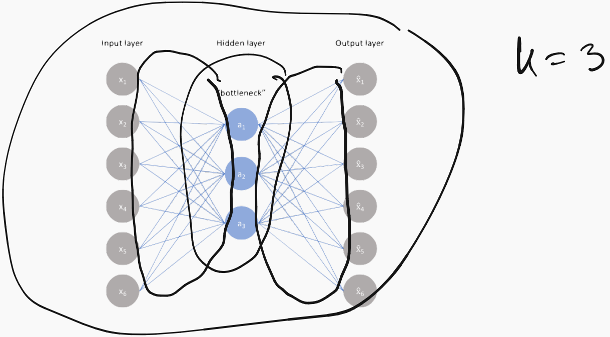
PRINCIPAL COMPONENT ANALYSIS

Rest of lecture: Deeper dive into understanding a simple, but powerful autoencoder architecture. Specifically we will view principal component analysis (PCA) as a type of autoencoder.

PCA is the "linear regression" of unsupervised learning: often the go-to baseline method for feature extraction and dimensionality reduction.

Very important outside machine learning as well.

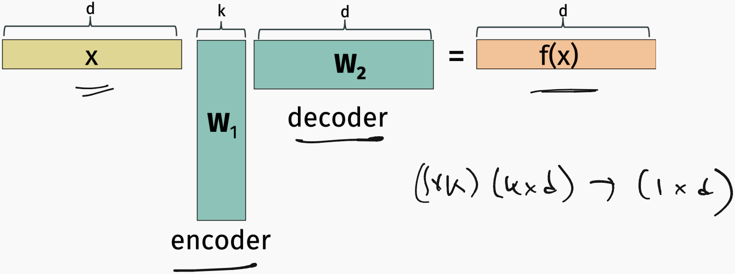
Consider the simplest possible autoencoder:



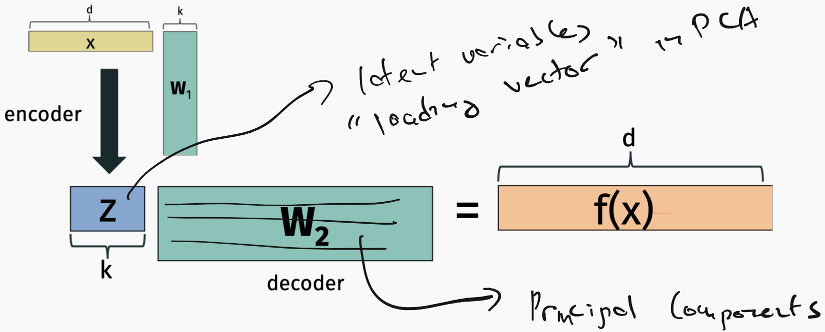
- One hidden layer. No non-linearity. No biases.
- Latent space of dimension k.
- Weight matrices are $\mathbf{W}_1 \in \mathbb{R}^{d \times k}$ and $\mathbf{W}_2 \in \mathbb{R}^{k \times d}$.

Given input $\mathbf{x} \in \mathbb{R}^d$, what is $f(\mathbf{x})$ expressed in linear algebraic terms?

(1xd)(dxk) → (1xk)

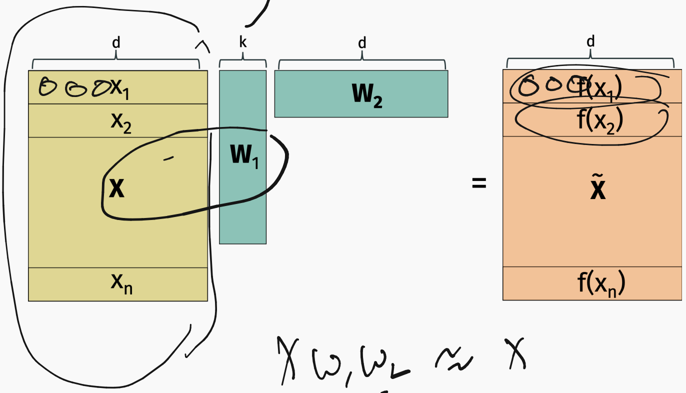


$$f(\mathbf{x})^T = \mathbf{x}^T \mathbf{W}_1 \mathbf{W}_2$$



Encoder: $e(\mathbf{x}) = \mathbf{x}^T \mathbf{W}_1$. Decoder: $d(\mathbf{z}) = \mathbf{z}\mathbf{W}_2$

Given training data set $x_1, \ldots, x_n$, let X denote our data matrix. Let $\tilde{X} = XW_1W_2$.



$\|M\|_F^2 = \sum_{i=1}^{n} \sum_{j=1}^{d} M_{ij}^2$

Natural squared autoencoder loss: Minimize $L(\mathbf{X}, \tilde{\mathbf{X}})$ where:

$$L(\mathbf{X}, \tilde{\mathbf{X}}) = \sum_{i=1}^{n} \|\mathbf{x}_i - f(\mathbf{x}_i)\|_2^2$$

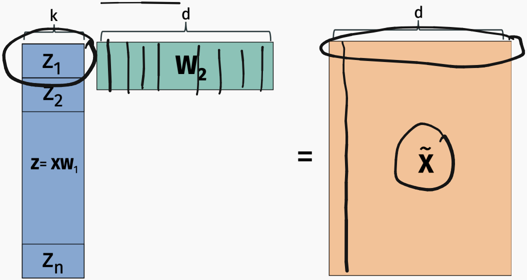
$$= \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{x}_i[j] - f(\mathbf{x}_i)[j])^2$$

$$= \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2$$

Goal: Find $\mathbf{W}_1, \mathbf{W}_2$ to minimize the Frobenius norm loss $\|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 = \|\mathbf{X} - \mathbf{X}\mathbf{W}_1\mathbf{W}_2\|_F^2$ (sum of squared entries).

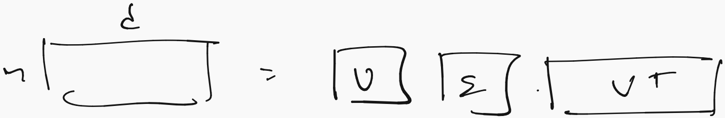
Rank in linear algebra:

- The columns of a matrix with column rank k can all be written as linear combinations of just k columns.
- The rows of a matrix with row rank k can all be written as linear combinations of k rows.
- Column rank = row rank = **rank**.



$\tilde{X}$ is a low-rank matrix. It only has rank k for $k \ll d$.

LOW-RANK APPROXIMATION

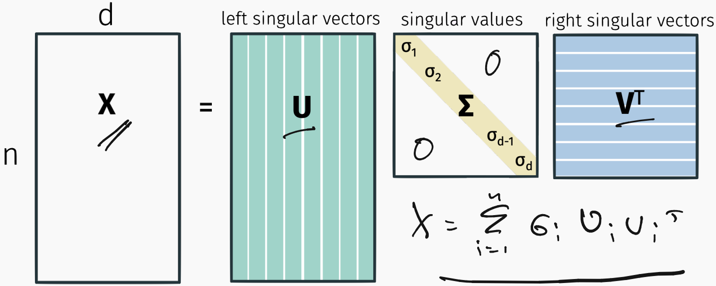


Principal component analysis is the task of finding $\mathbf{W}_1$, $\mathbf{W}_2$, which amounts to finding a rank k matrix $\tilde{\mathbf{X}}$ which approximates the data matrix $\mathbf{X}$ as closely as possible.

Finding the best $\mathbf{W}_1$ and $\mathbf{W}_2$ is a non-convex problem. We could try running an iterative method like gradient descent anyway. But there is also a direct algorithm!

SINGULAR VALUE DECOMPOSITION

Any matrix **X** can be written:

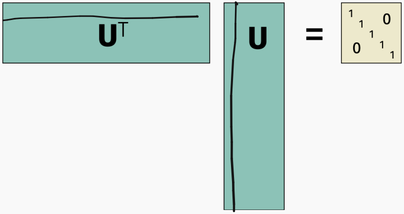


Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. **U** and **V** are orthogonal matrices.

This is called the **singular value decomposition.**

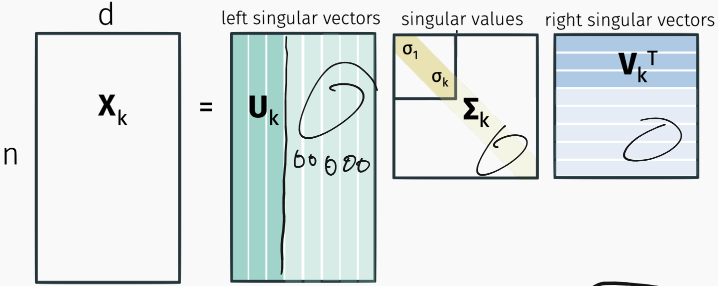
Can be computed in $O(nd^2)$ time (faster with approximation algos).

Let $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathbb{R}^n$ denote the columns of $\mathbf{U}$. I.e. the left singular vectors of $\mathbf{X}$.



$\|\mathbf{u}_i\|_2^2 = \langle u_i, u_i \rangle = (U^TU)_{ii} = 1$ $\quad$ $\mathbf{u}_i^T\mathbf{u}_j = (U^TU)_{ij} = 0$

Can read off optimal low-rank approximations from the SVD:



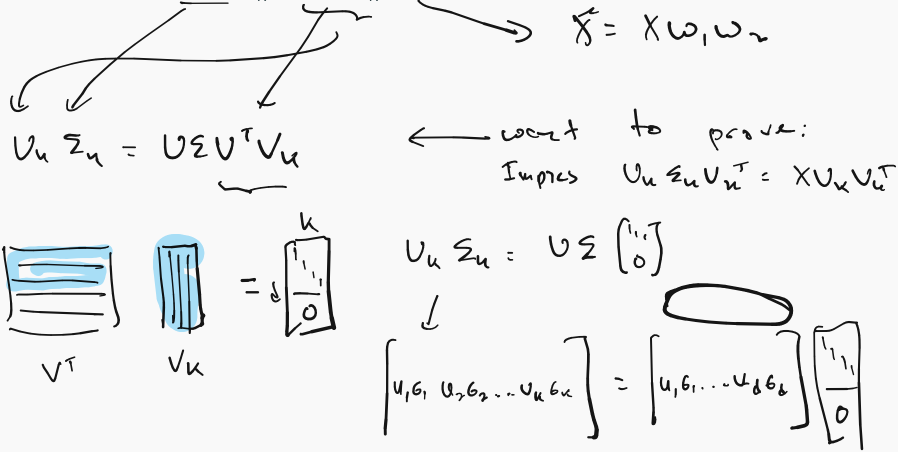
Eckart–Young–Mirsky Theorem: For any $k \leq d$, $\mathbf{X}_k = \mathbf{U}_k \boldsymbol{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

$$\mathbf{X}_k = \underset{\tilde{\mathbf{X}} \text{ with rank } \leq k}{\arg\min} \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2.$$

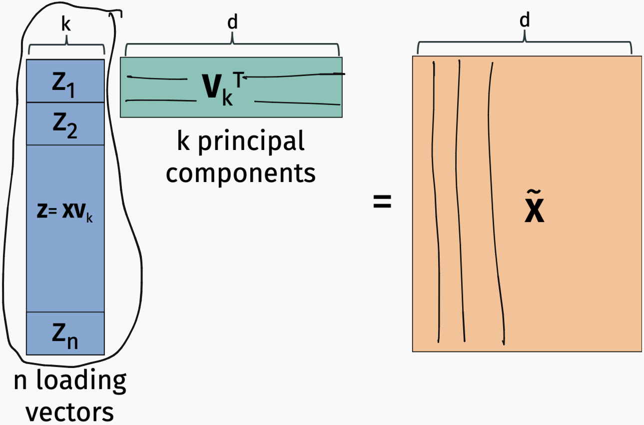
$W_1 = V_k \quad W_2 = V_k^T$

Claim: $X_k = U_k \Sigma_k V_k^T = X V_k V_k^T$.

$\tilde{X} = X W_1 W_2$



So for a model with k hidden variables, we obtain an optimal autoencoder by setting $W_1 = V_k$, $W_2 = V_k^T$. $f(x) = xV_kV_k^T$.



Usually x's columns (features) are mean centered and normalized to variance 1 before computing principal components.

Computing the SVD.

- Full SVD:
  `U,S,V = scipy.linalg.svd(X)`.
  Runs in $O(nd^2)$ time.
- Just the top k components:
  `U,S,V = scipy.sparse.linalg.svds(X, k)`.
  Runs in roughly $O(ndk)$ time.

CONNECTION TO EIGENDECOMPOSITION

Recall that for a matrix $\mathbf{M} \in \mathbb{R}^{p \times p}$, $\mathbf{q}$ is an eigenvector of $\mathbf{M}$ if $\lambda \mathbf{q} = \mathbf{M}\mathbf{q}$ for any scalar λ.

- $\mathbf{U}$'s columns (the left singular vectors) are the orthonormal eigenvectors of $\mathbf{X}\mathbf{X}^T$.

  (n×d)(d×n)
  (n×n)

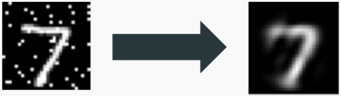
- $\mathbf{V}$'s columns (the right singular vectors) are the orthonormal eigenvectors of $\mathbf{X}^T\mathbf{X}$.
- $\sigma_i^2 = \lambda_i(\mathbf{X}\mathbf{X}^T) = \lambda_i(\mathbf{X}^T\mathbf{X})$

Exercise: Verify this directly. This means you can use any (symmetric) eigensolver for computing the SVD.

Like any autoencoder, PCA can be used for:

- Feature extraction
- Denoising and rectification
- Data generation
- Compression
- Visualization

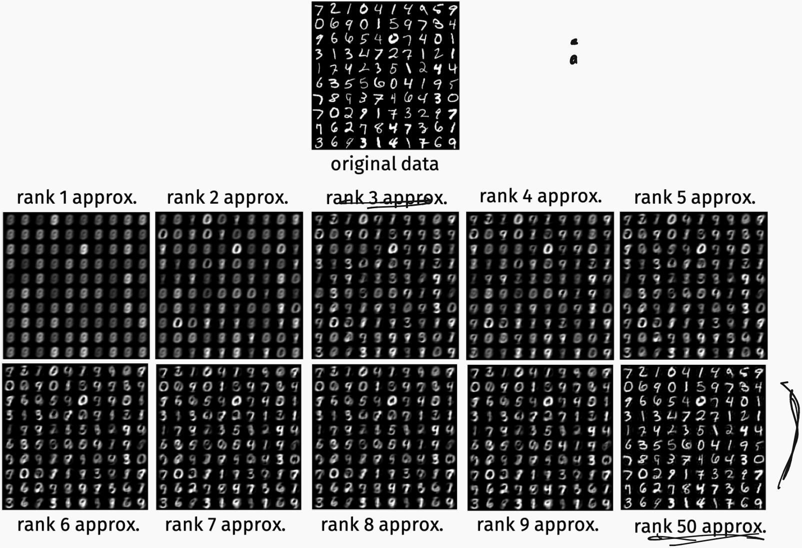
Return at 12:55



denoising

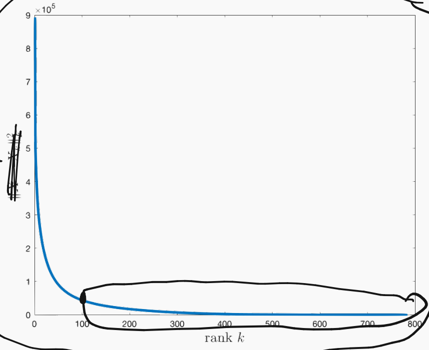
synthetic data generation

The larger we set k, the better approximation we get.



Error vs. k is dictated by X's singular values. The singular values are often called the **spectrum** of X.

$$\|\mathbf{X} - \mathbf{X}_k\|_F^2 = \sum_{i=k}^{d} \sigma_i^2.$$



COLUMN REDUNDANCY

Colinearity of data features leads to an approximately low-rank data matrix.

	bedrooms	bathrooms	sq.ft.	floors	list price	sale price
home 1	2	2	1800	2	200,000	195,000
home 2	4	2.5	2700	1	300,000	310,000
⋮	⋮	⋮	⋮	⋮	⋮	⋮
home n	5	3.5	3600	3	450,000	450,000

sale price $\approx 1.05 \cdot$ list price.

property tax $\approx .01 \cdot$ list price.

Sometimes these relationships are simple, other times more complex. But as long as there exists linear relationships between features, we will have a lower rank matrix.

$$\text{yard size} \approx \text{lot size} - \frac{1}{2} \cdot \text{square footage.}$$

$$\text{cumulative GPA} \approx \frac{1}{4} \cdot \text{year 1 GPA} + \frac{1}{4} \cdot \text{year 2 GPA} + \frac{1}{4} \cdot \text{year 3 GPA} + \frac{1}{4} \cdot \text{year 4 GPA.}$$

Two other examples of data with good low-rank approximations:

1. Genetic data:

single nucleotide polymorphisms (SNPs) loci

	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

2. “Term-document” matrix with bag-of-words data:

	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

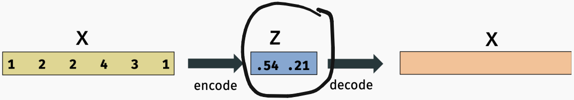
EXAMPLES OF LOW-RANK STRUCTURE

SNPs matrices tend to be very low-rank.

single nucleotide polymorphisms (SNPs) loci

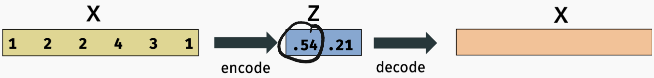
	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

Most of the information in **x** is explained by just a few **latent variable**.



EXAMPLES OF LOW-RANK STRUCTURE

"Genes Mirror Geography Within Europe" – Nature, 2008.



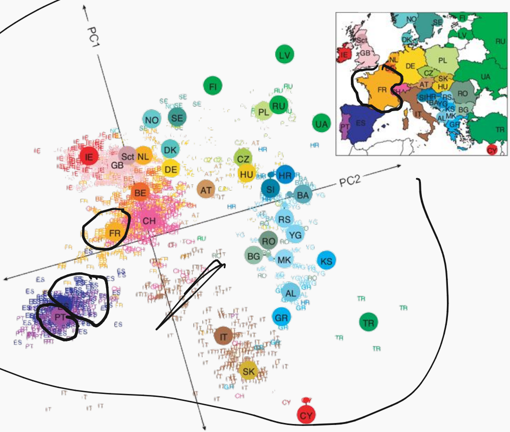
In data collected from European populations, latent variables capture information about geography.

$z[1] \approx$ relative north-south position of birth place

$z[2] \approx$ relative east-west position of birth place

Individuals born in similar places tend to have similar genes.

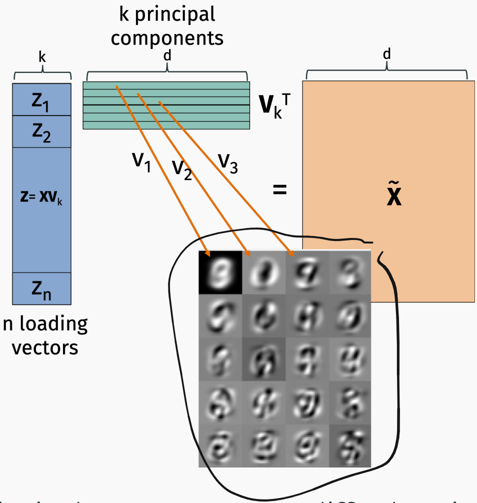
"Genes Mirror Geography Within Europe" – Nature, 2008.



Genetic data can be nicely visualized using PCA! Plot each data example **x** using two loading variables in **z**.

For more complex data, what do principal components and loading vectors look like?

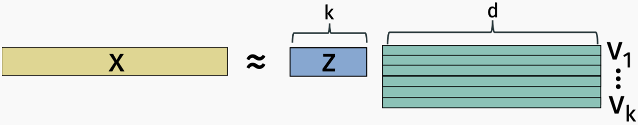
MNIST **principal components**:

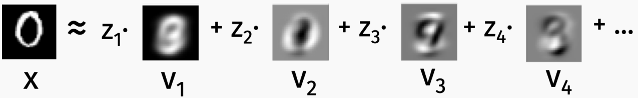


Often principal components are difficult to interpret.

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

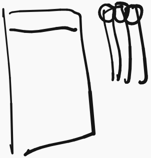
The loading vector $\mathbf{z}$ for an example $\mathbf{x}$ contains coefficients which recombine the top k principal components $\mathbf{v}_1, \ldots, \mathbf{v}_k$ to approximately reconstruct $\mathbf{x}$.

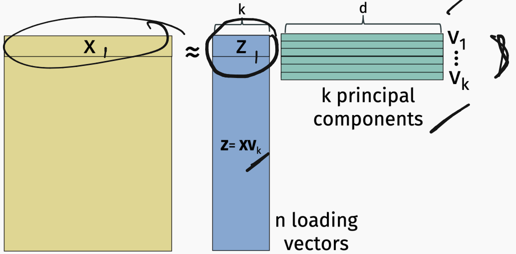


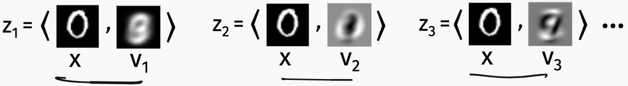


Provide a short “finger print” for any image $\mathbf{x}$ which can be used to reconstruct that image.

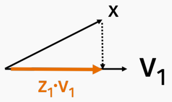
For any $\mathbf{x}$ with loading vector $\mathbf{z}$, $\mathbf{z}_i$ is the inner product similarity between $\mathbf{x}$ and the i^{th} principal component $\mathbf{v}_i$.

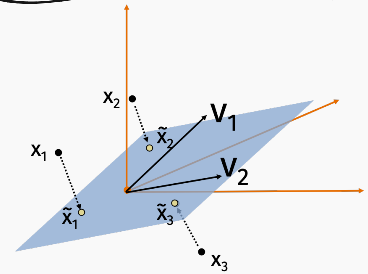






So we approximate $\mathbf{x} \approx \tilde{\mathbf{x}} = \langle \mathbf{x}, \mathbf{v}_1 \rangle \cdot \mathbf{v}_1 + \ldots + \langle \mathbf{x}, \mathbf{v}_k \rangle \cdot \mathbf{v}_k$.



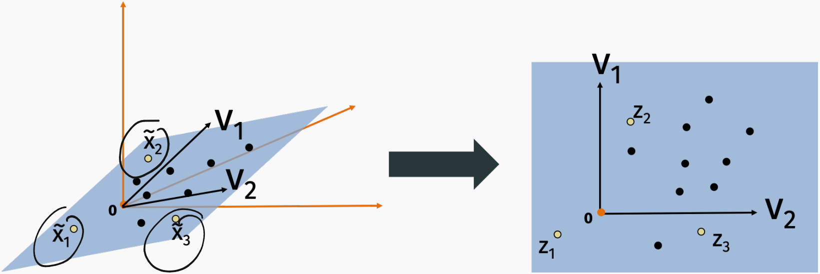


Since $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are orthonormal, this operation is a **projection** onto first k principal components.

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

LOADING VECTORS: PROJECTION VIEW

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



$$\|\tilde{x}_1 - \tilde{x}_2\|_2 = \|z_1 - z_2\|_2$$

$$\langle \tilde{x}_1, \tilde{x}_2 \rangle = \langle z_1, z_2 \rangle$$

Important takeaway for data visualization and more: Latent feature vectors preserve similarity and distance information in the original data.

Let $\mathbf{x}_1 \ldots, \mathbf{x}_n \in \mathbb{R}^d$ be our original data vectors, $\mathbf{z}_1 \ldots, \mathbf{z}_n \in \mathbb{R}^k$ be our loading vectors (encoding), and $\tilde{\mathbf{x}}_1 \ldots, \tilde{\mathbf{x}}_n \in \mathbb{R}^d$ be our low-rank approximated data.

We have:

$$\|\tilde{\mathbf{x}}_i\|_2^2 = \|\mathbf{z}_i\|_2^2$$

$$\langle x_i, x_j \rangle \approx \langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j \rangle = \underline{\langle \mathbf{z}_i, \mathbf{z}_j \rangle}$$

$$\|x_i - x_j\|_2^2 \approx \|\tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_j\|_2^2 = \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

$(\lambda q = XX^T q)$ power method

Conclusion: If our data had a good low rank approximation, we expect that:

$$\|\mathbf{x}_i\|_2^2 \approx \|\mathbf{z}_i\|_2^2$$

$$\langle \mathbf{x}_i, \mathbf{x}_j \rangle \approx \langle \mathbf{z}_i, \mathbf{z}_j \rangle$$

$$\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \approx \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

$X = U\Sigma U^T$

$U^T X U = \Sigma$

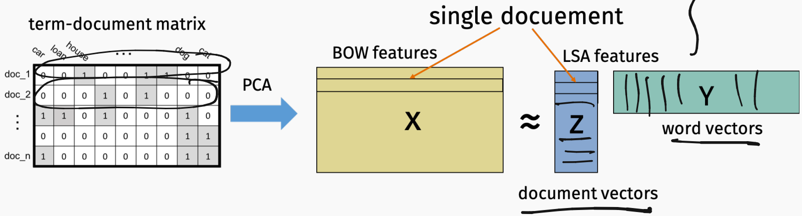
Word-document matrices tend to be low rank.

	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).



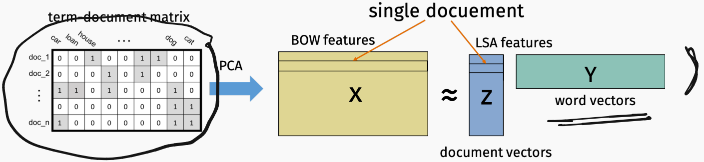
Each column of **z** corresponds to a latent "category" or "topic". Corresponding row in **Y** corresponds to the "frequency" with which different words appear in documents on that topic.

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.

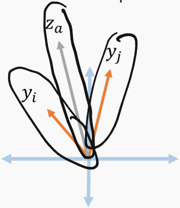
Same fingerprinting idea was also important in early facial recognition systems based on “eigenfaces”:

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



$\hat{X}_{ij} = \langle z_i, y_j \rangle$

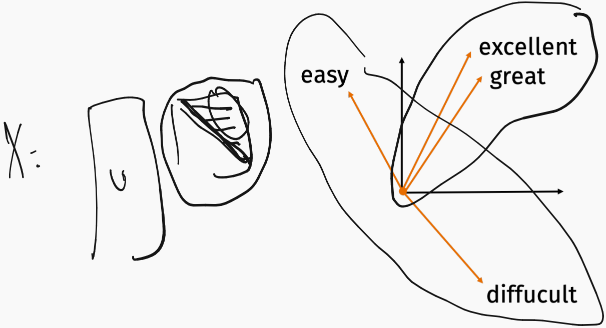
- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$ when doc_a contains $word_i$.
- If $word_i$ and $word_j$ both appear in doc_a, then $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$, so we expect $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$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



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

SEMANTIC EMBEDDINGS

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.

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".

Vocabulary: Small, handy, excellent, great, quality, compact, easy, difficult.

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.*

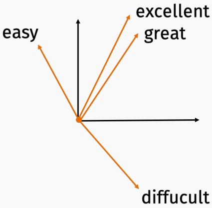
[, , , , , , ,]

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.*

[, , , , , , ,]

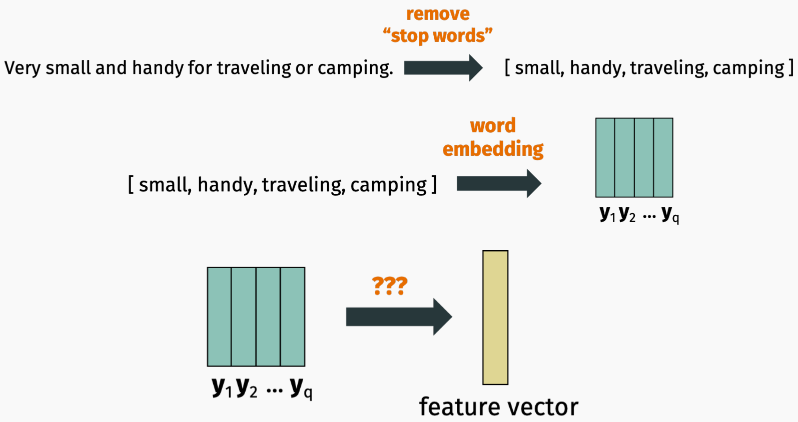
Bag-of-words approach typically only works for large data sets.

The features do not capture the fact that “great” and “excellent” are near synonyms. Or that “difficult” and “easy” are antonyms.



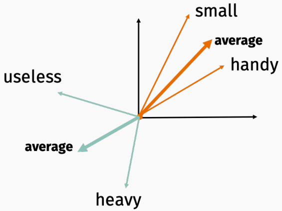
This can be addressed by first mapping words to semantically meaningful vectors. That mapping can be trained using a much large corpus of text than the data set you are working with (e.g. Wikipedia, Twitter, news data sets).

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

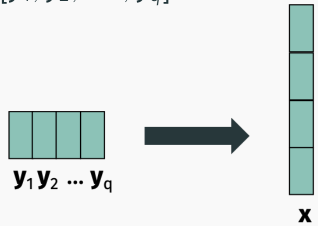


A few simple options:

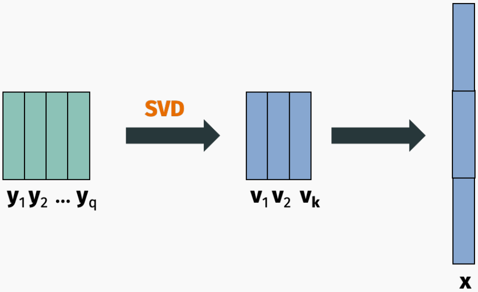
Feature vector $\mathbf{x} = \frac{1}{q}\sum_{i=1}^{q} \mathbf{y}_q$.



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

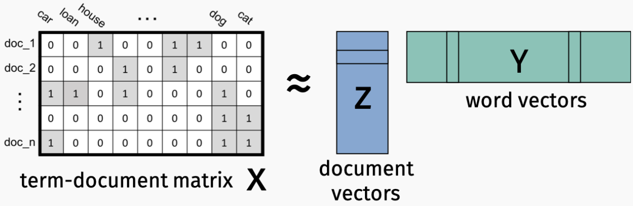


To avoid issues with inconsistent sentence length, word ordering, etc., can concatenate a fixed number of top principal components of the matrix of word vectors:



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

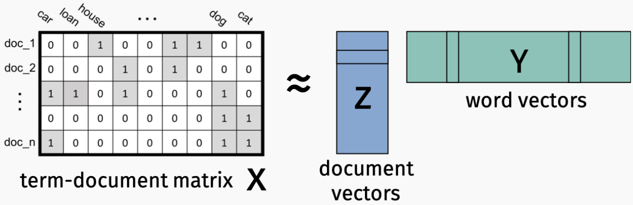
Another view on word embeddings from LSA:



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

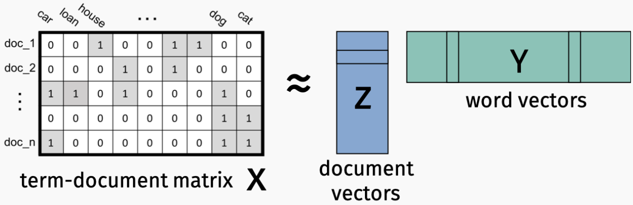
Could have just as easily set $\mathbf{Z} = \mathbf{U}_k$ and $\mathbf{Y} = \boldsymbol{\Sigma}_k\mathbf{V}_k^T$, so $\mathbf{Z}$ has orthonormal columns.

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?

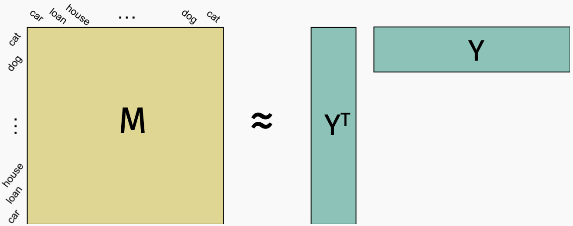


What does the i, j entry of X^TX reprent?

$\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ 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.

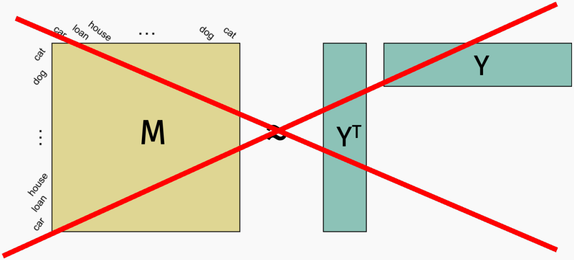
The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

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`.

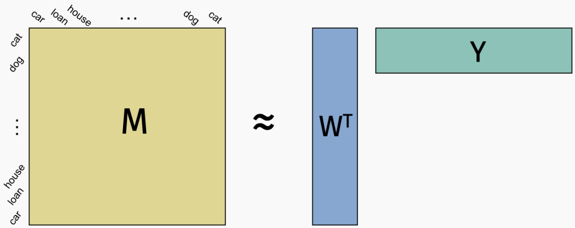
- `word2vec` was originally presented as a shallow neural network model, but it is equivalent to matrix factorization method (Levy, Goldberg 2014).
- For `word2vec`, similarity metric is the "point-wise mutual information": $\log \frac{p(i,j)}{p(i)p(j)}$.



SVD will not return a symmetric factorization in general. In fact, if $\mathbf{M}$ is not positive semidefinite[3] then the optimal low-rank approximation does not have this form.

[3] I.e., $k(word_i, word_j)$ is not a positive semidefinite kernel.

CAVEAT ABOUT FACTORIZATION



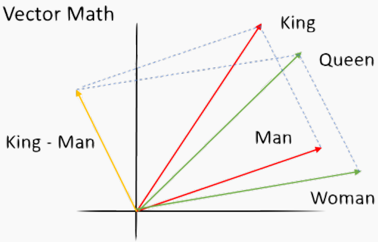
- For each word i we get a left and right embedding vector $\mathbf{w}_i$ and $\mathbf{y}_i$. It's reasonable to just use one or the other.
- If $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ is large and positive, we expect that $\mathbf{y}_i$ and $\mathbf{y}_j$ have similar similarity scores with other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

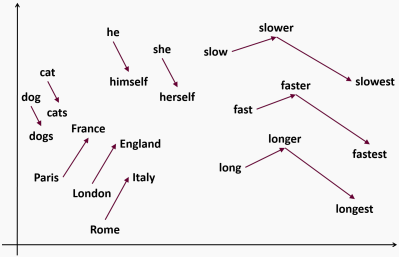
EASIEST WAY TO USE WORD EMBEDDINGS

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

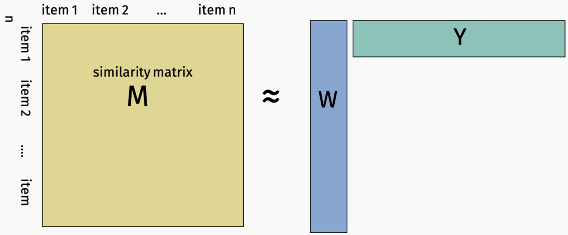
- Original gloVe website: `https://nlp.stanford.edu/projects/glove/`.
- Compilation of many sources: `https://github.com/3Top/word2vec-api`

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



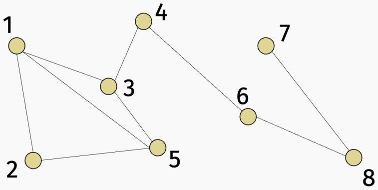


The same approach used for word embeddings can be used to obtain meaningful numerical features for any other data where there is a natural notion of similarity.



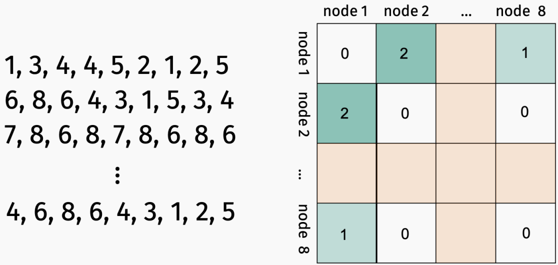
For example, the items could be nodes in a social network graph. Maybe be want to predict an individuals age, level of interest in a particular topic, political leaning, etc.

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).