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

NYU Tandon School of Engineering, Prof. Christopher Musco

FINAL PROJECT

Optional. Can be completed in place of last assignment, which will contain a mix of written problems and coding problems.

1. Concept Based Project
2. Data set Based Project

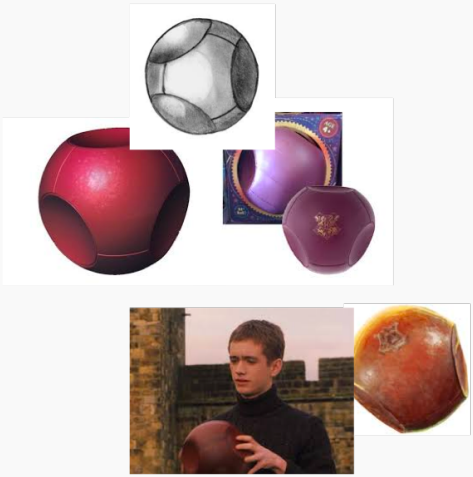
Please email me if you want to complete a project and who your team will be (1-3 students allowed). We can also discuss topic ideas.

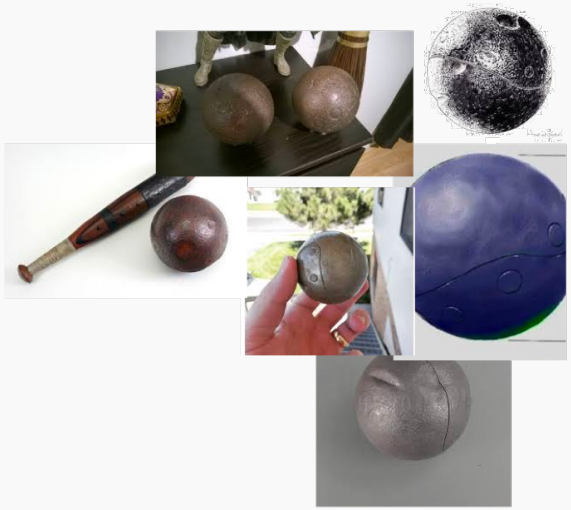
TRANSFER LEARNING

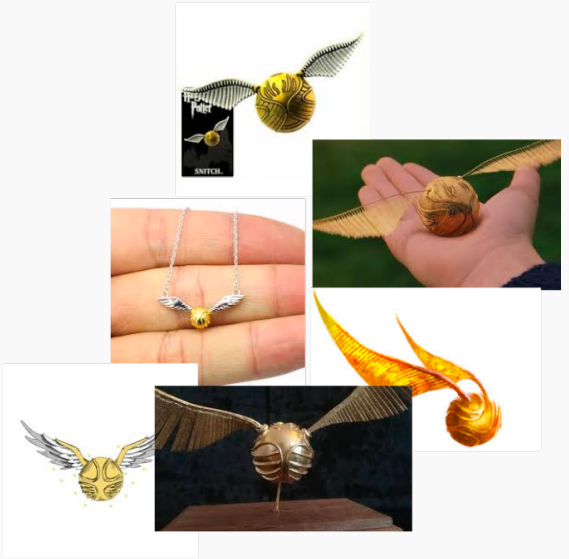
State-of-the-art supervised learning models like neural networks learn very good features.

Often these features are useful for tasks other than what the model was trained on. They can be used for other problems where we do not have a lot of training data.

Example: Classify images of different Quidditch balls.

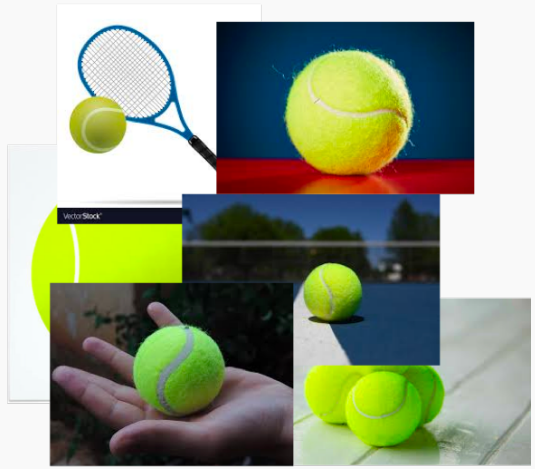






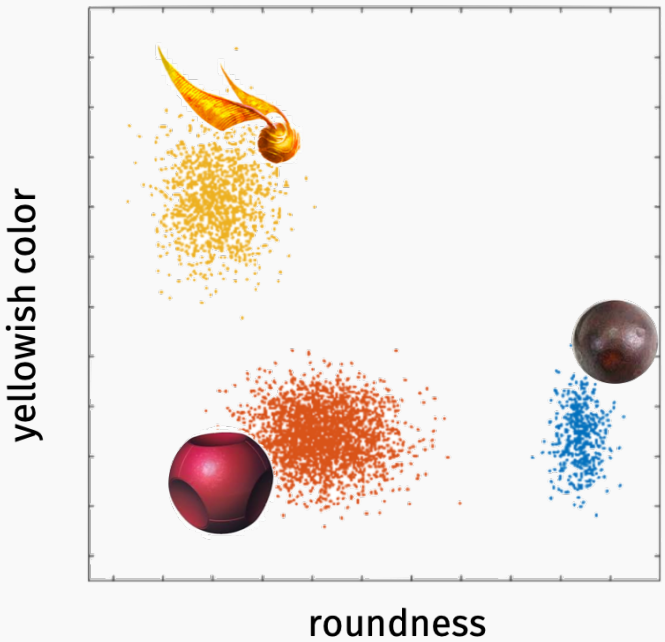
TRANSFER LEARNING

Based on features learned for muggle objects:



FEATURE LEARNING

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

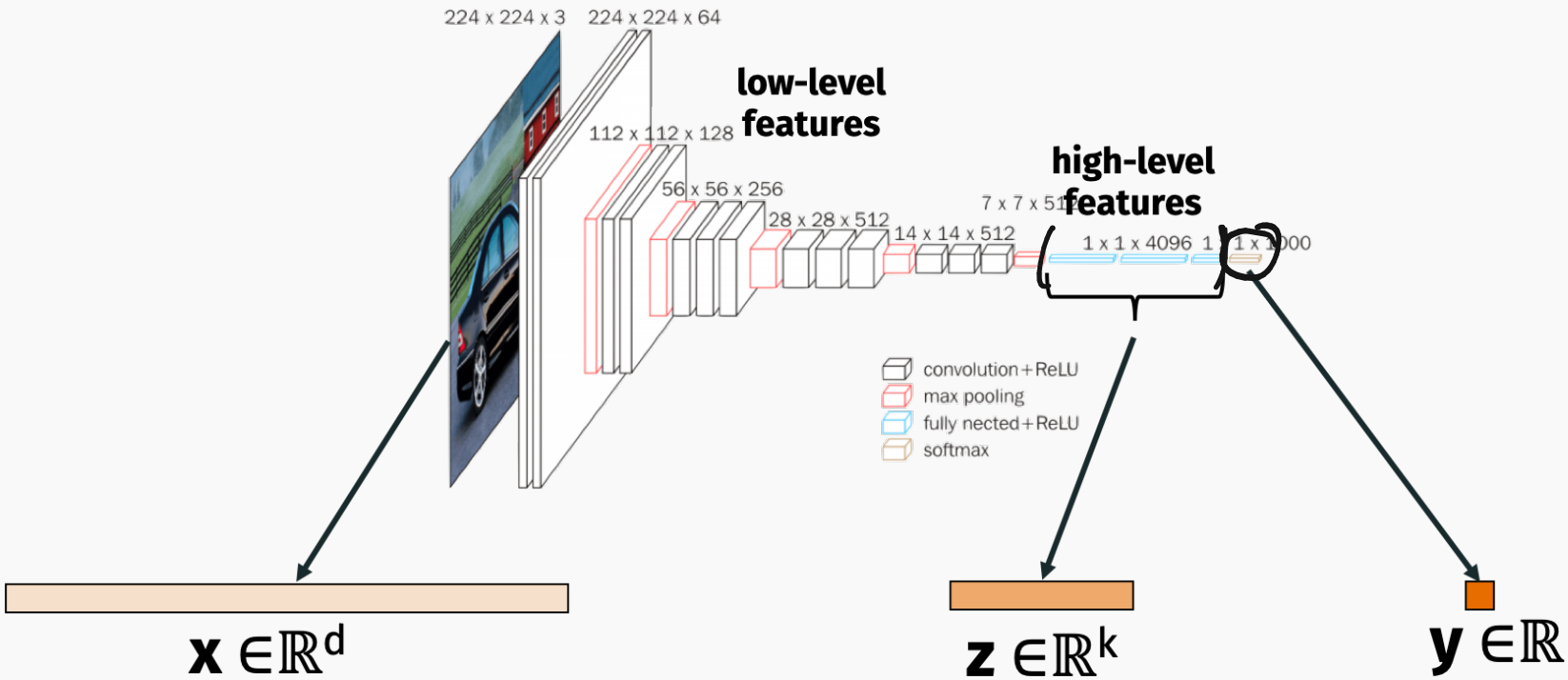


TRANSFER LEARNING

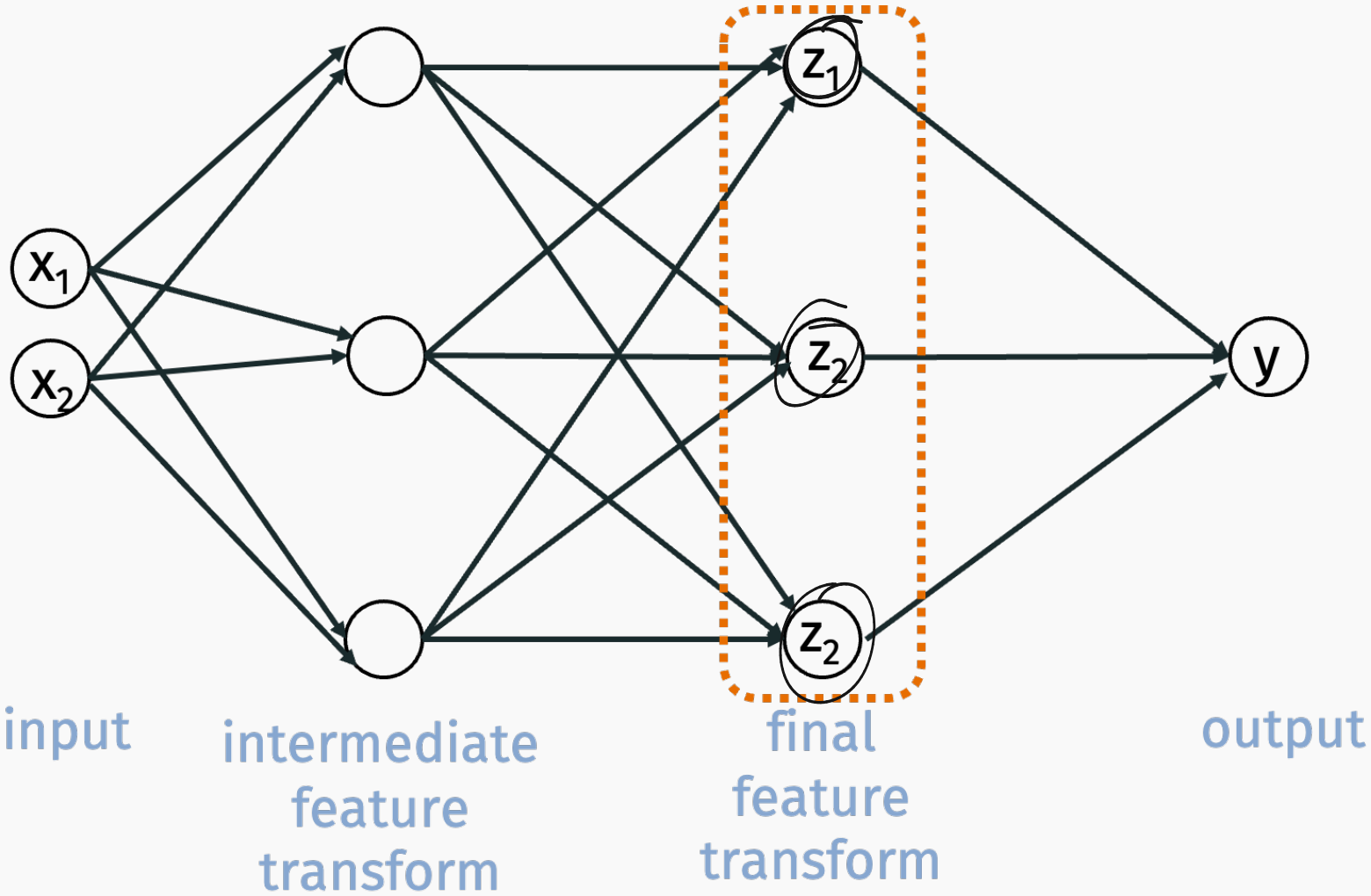
Common technique:

1. Download network trained e.g. on a 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.).

TRANSFER LEARNING



TRANSFER LEARNING



UNSUPERVISED FEATURE LEARNING

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.

AUTOENCODER

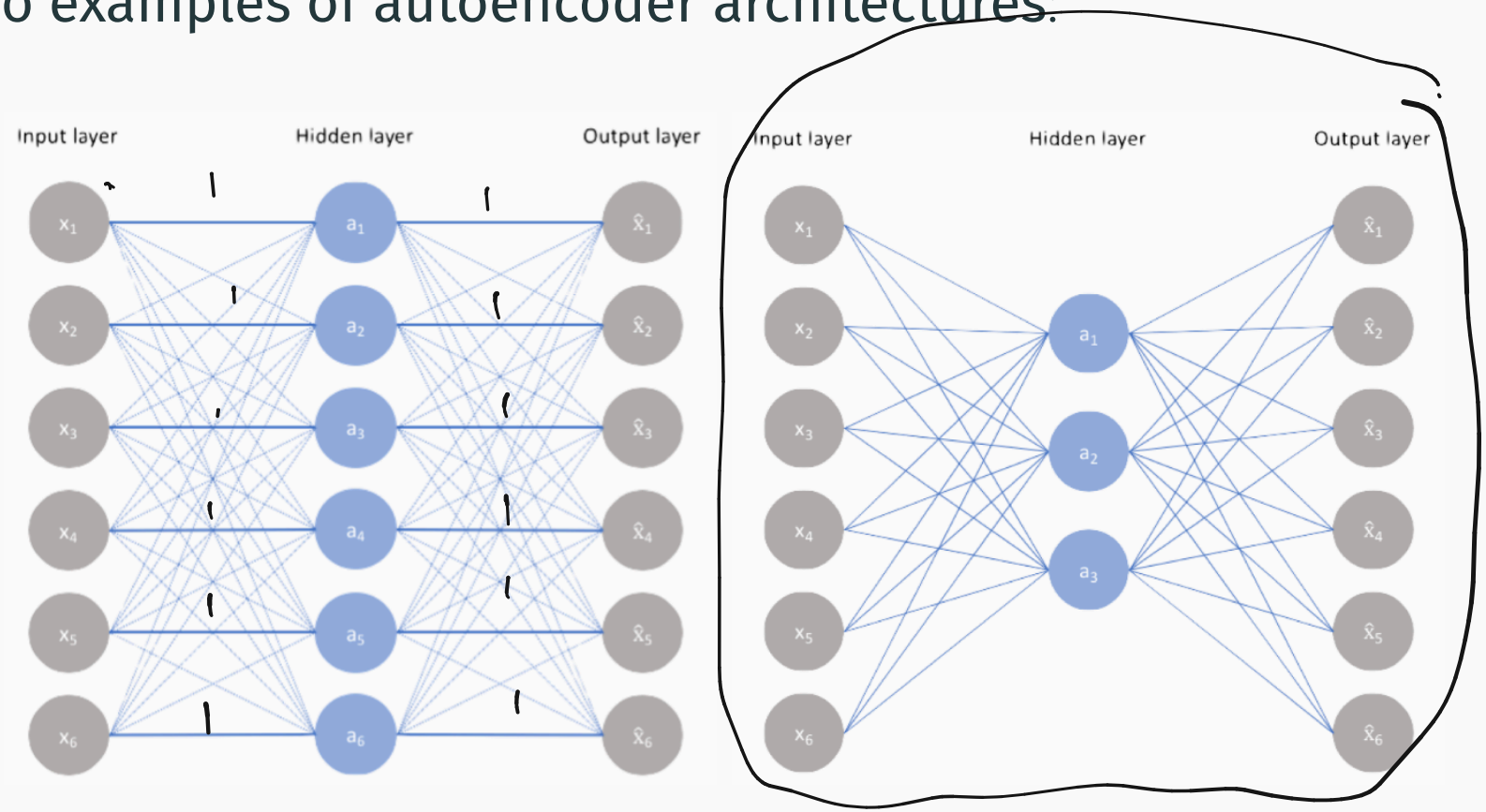
Simple but clever idea: If we have inputs $\mathbf{x}_1 \dots \mathbf{x}_n \in \mathbb{R}^d$ but few or no targets $y_1, \dots, y_n$, just make the inputs the targets.

- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$ 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).

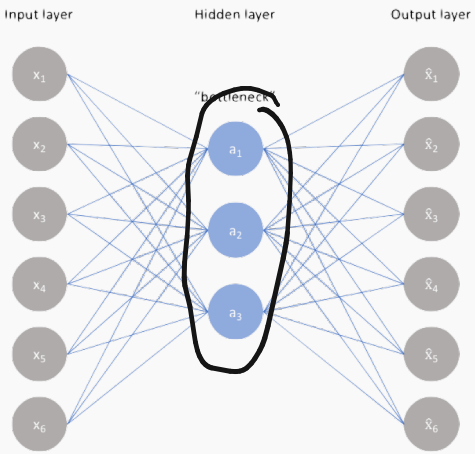
Two examples of autoencoder architectures:

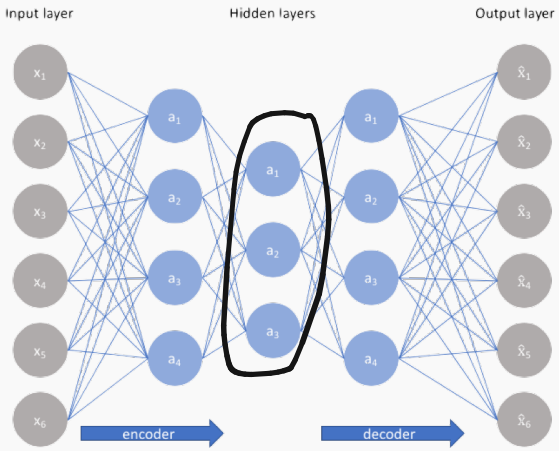


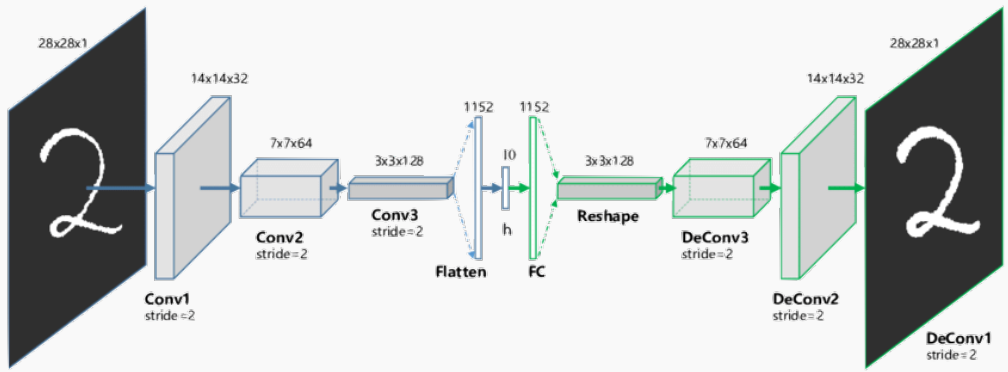
Which would lead to better feature learning?

AUTOENCODER

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.





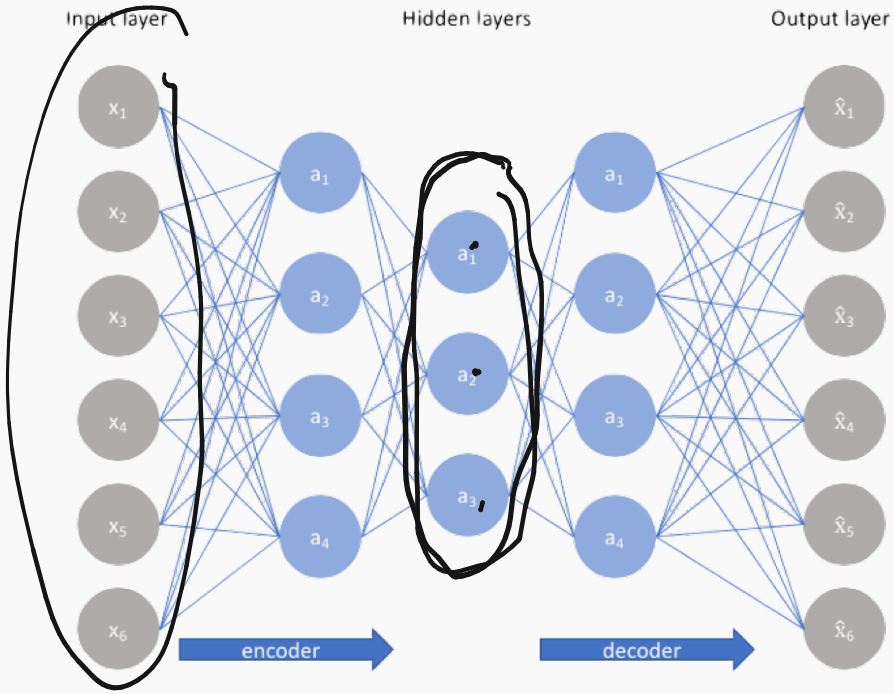


Architecture typically split into two parts:

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

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

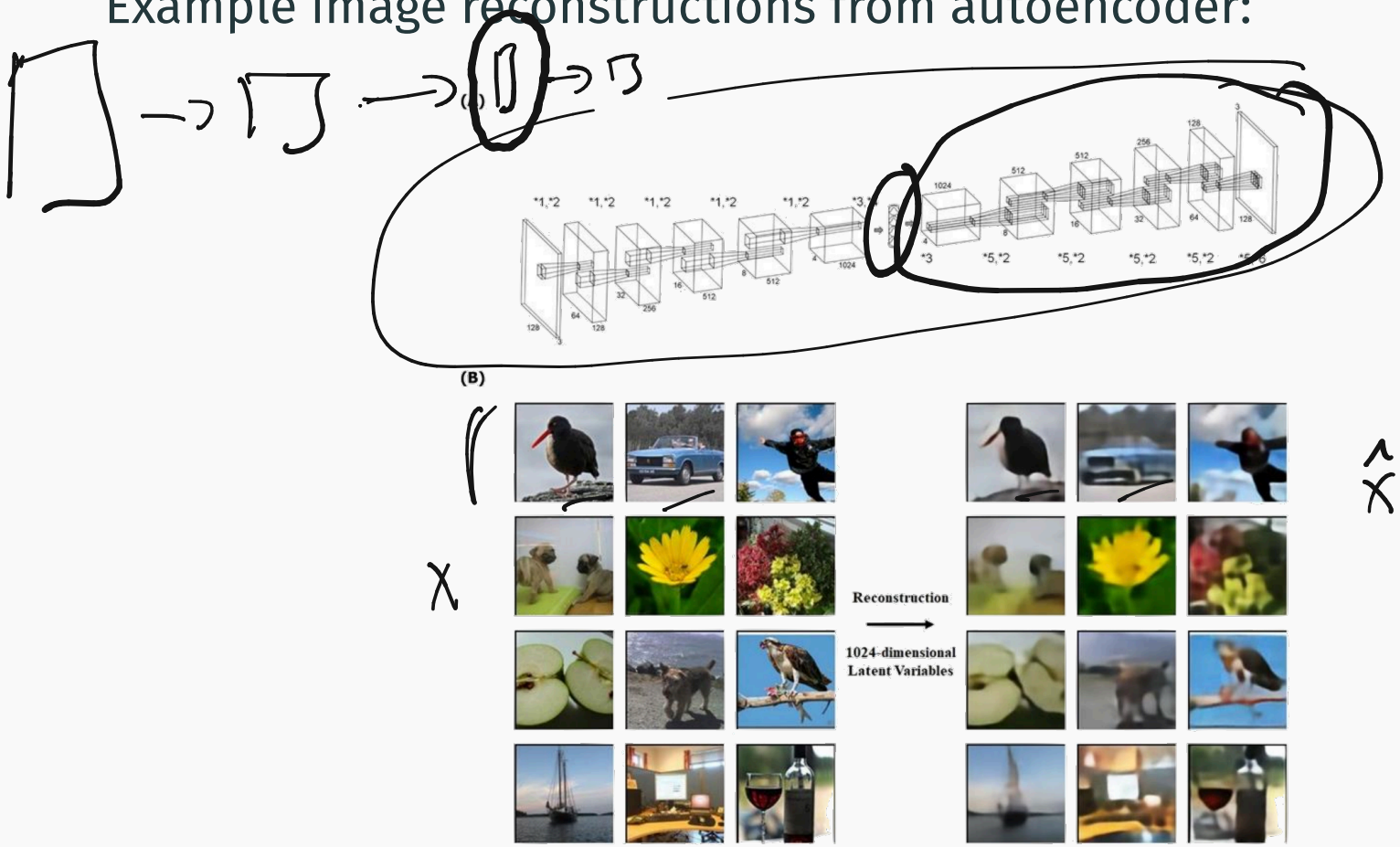
$$f(\mathbf{x}) = d(e(\mathbf{x}))$$



Often symmetric, but does not have to be.

AUTOENCODER RECONSTRUCTION

Example image reconstructions from autoencoder:



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

Input parameters: $d = 49152$.

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

AUTOENCODERS FOR FEATURE EXTRACTION

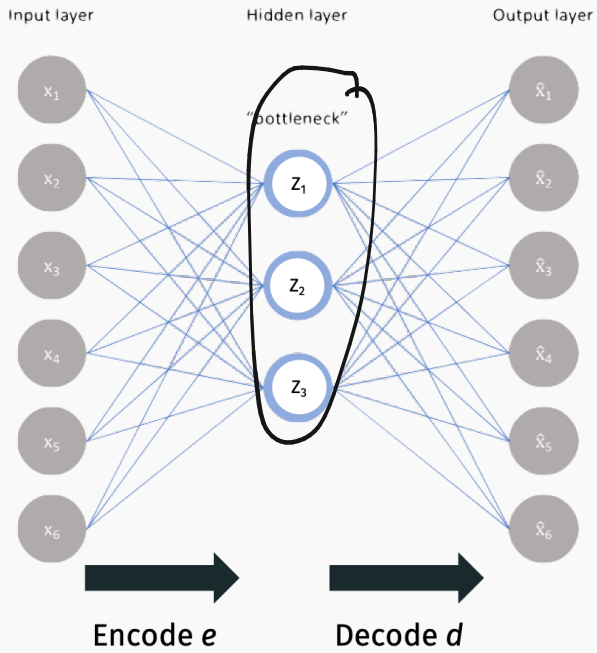
The best autoencoders do not work as well as supervised methods for feature extraction, but they require less labeled data. Recent progress on self-supervised learning gets closer to supervised feature learning. Might discuss a bit later.

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

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

AUTOENCODERS FOR IMAGE COMPRESSION

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



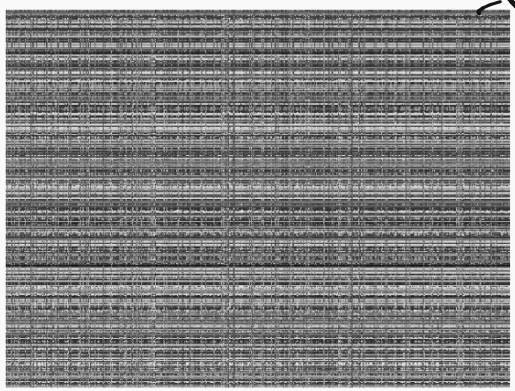
Given input $\mathbf{x}$, we can completely recover $f(\mathbf{x})$ from $\mathbf{z} = e(\mathbf{x})$. $\mathbf{z}$ typically has many fewer dimensions than $\mathbf{x}$ and for a typical image $f(\mathbf{x})$ will closely approximate $\mathbf{x}$.

AUTOENCODERS FOR IMAGE COMPRESSION

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”.



AUTOENCODERS FOR IMAGE COMPRESSION

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

"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.

AUTOENCODERS FOR IMAGE RESTORATION

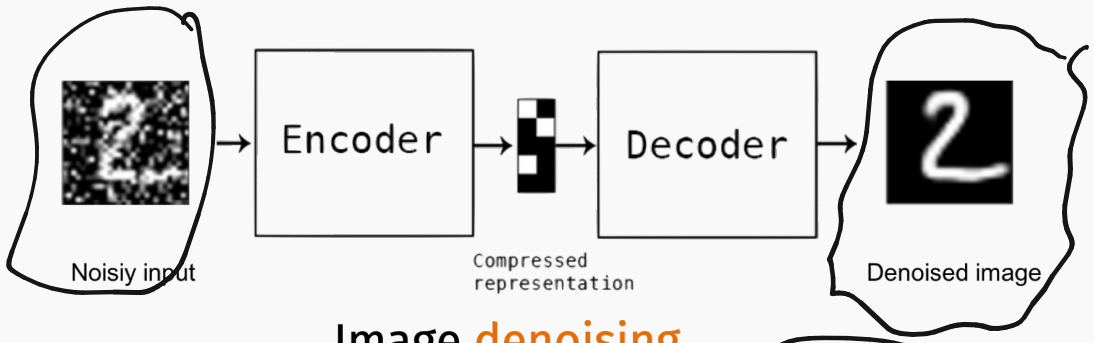


Image denoising

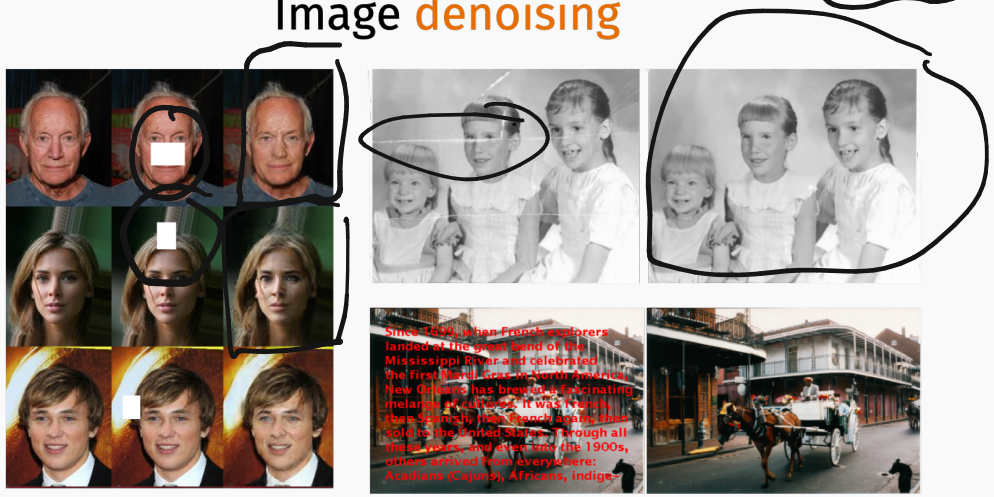
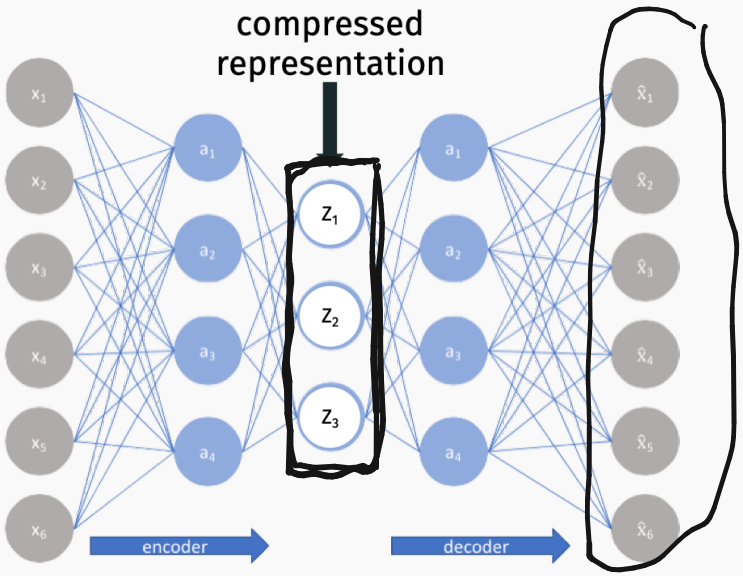


Image inpainting

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

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

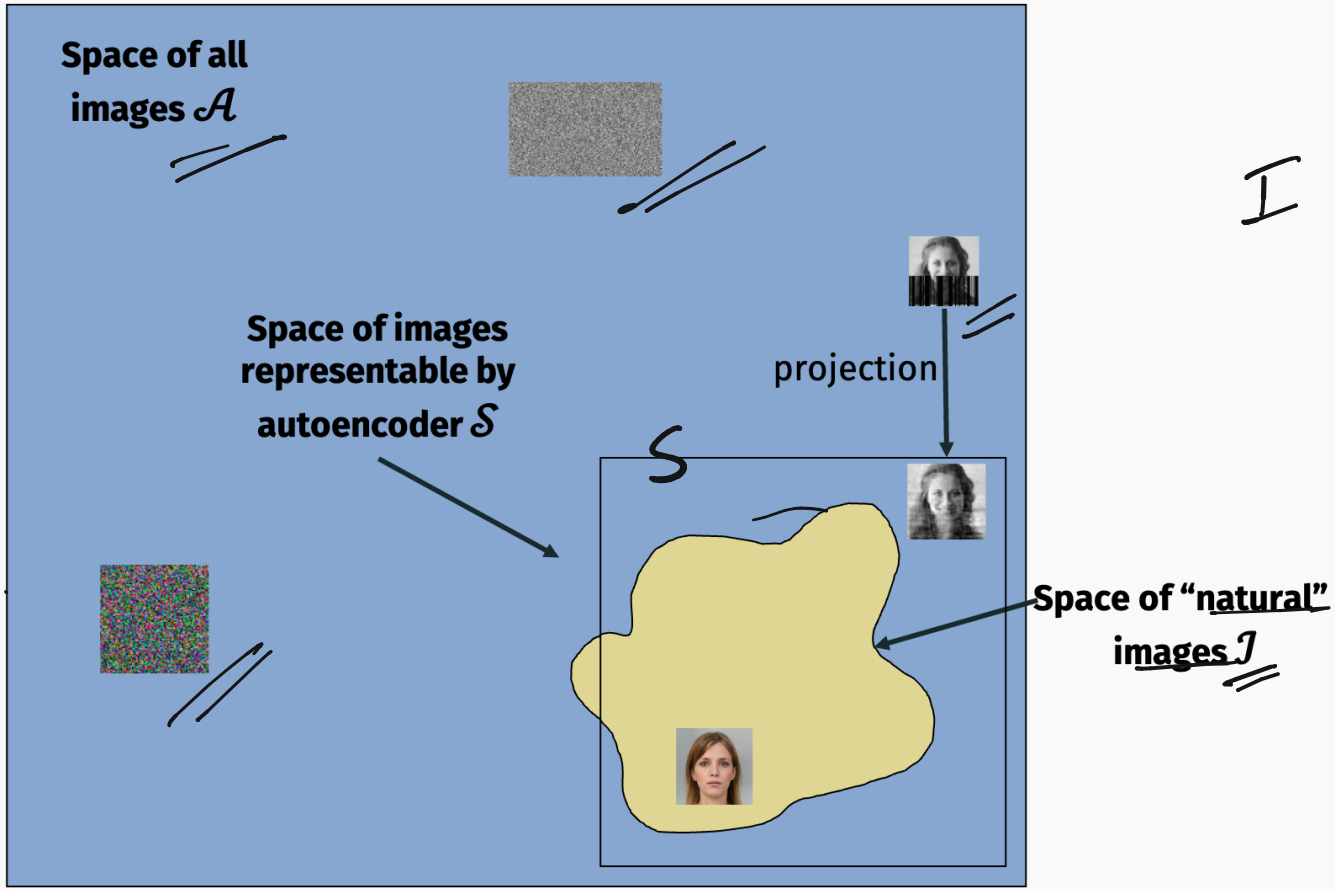
Why does this work?



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

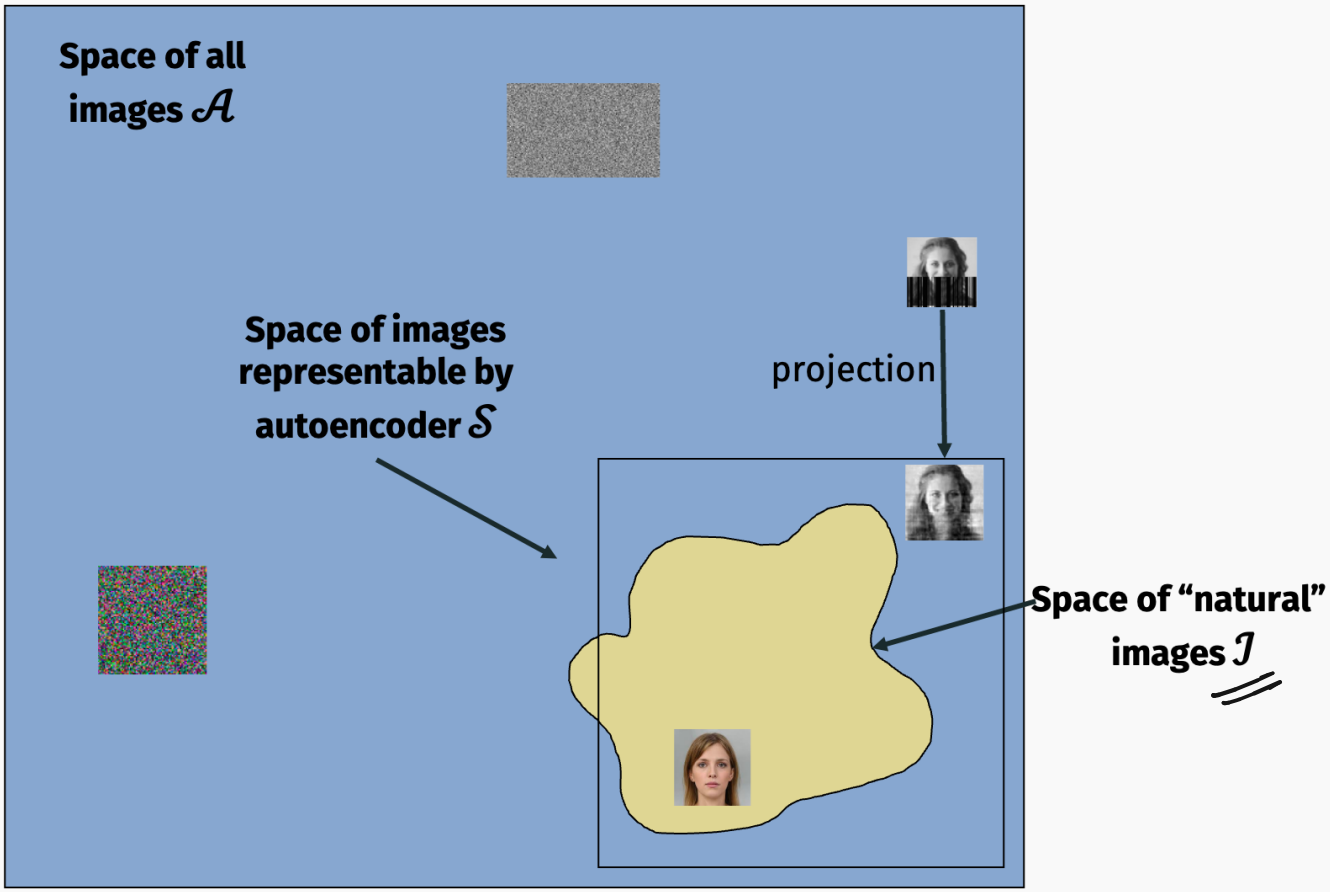
If $\mathbf{z}$ holds k values between in $0, .1, .2, \ldots, 1$, how many unique images $\mathbf{w}$ can be output by the autoencoder function f?

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

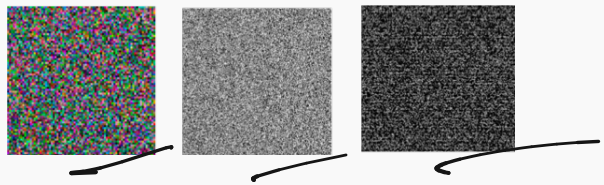


$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

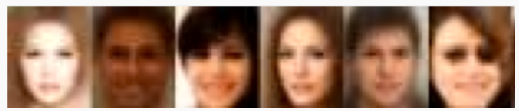
AUTOENCODERS FOR DATA GENERATION

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 $\mathcal{A}$.

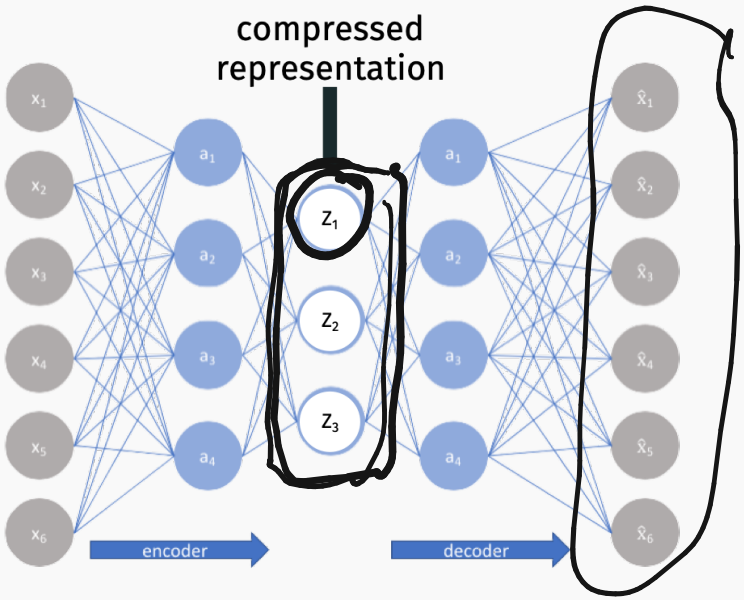


- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.



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

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



Randomly select code $\mathbf{z}$, then set $\mathbf{x} = d(\mathbf{z})$.[1]

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

AUTOENCODERS FOR DATA GENERATION

Generative models are a growing area of machine learning, drive by a lot of interesting new ideas. Generative Adversarial Networks in particular are now a major competitor with variational autoencoders.

SIDE NOTE ON AUTOENCODER ARCHITECTURES

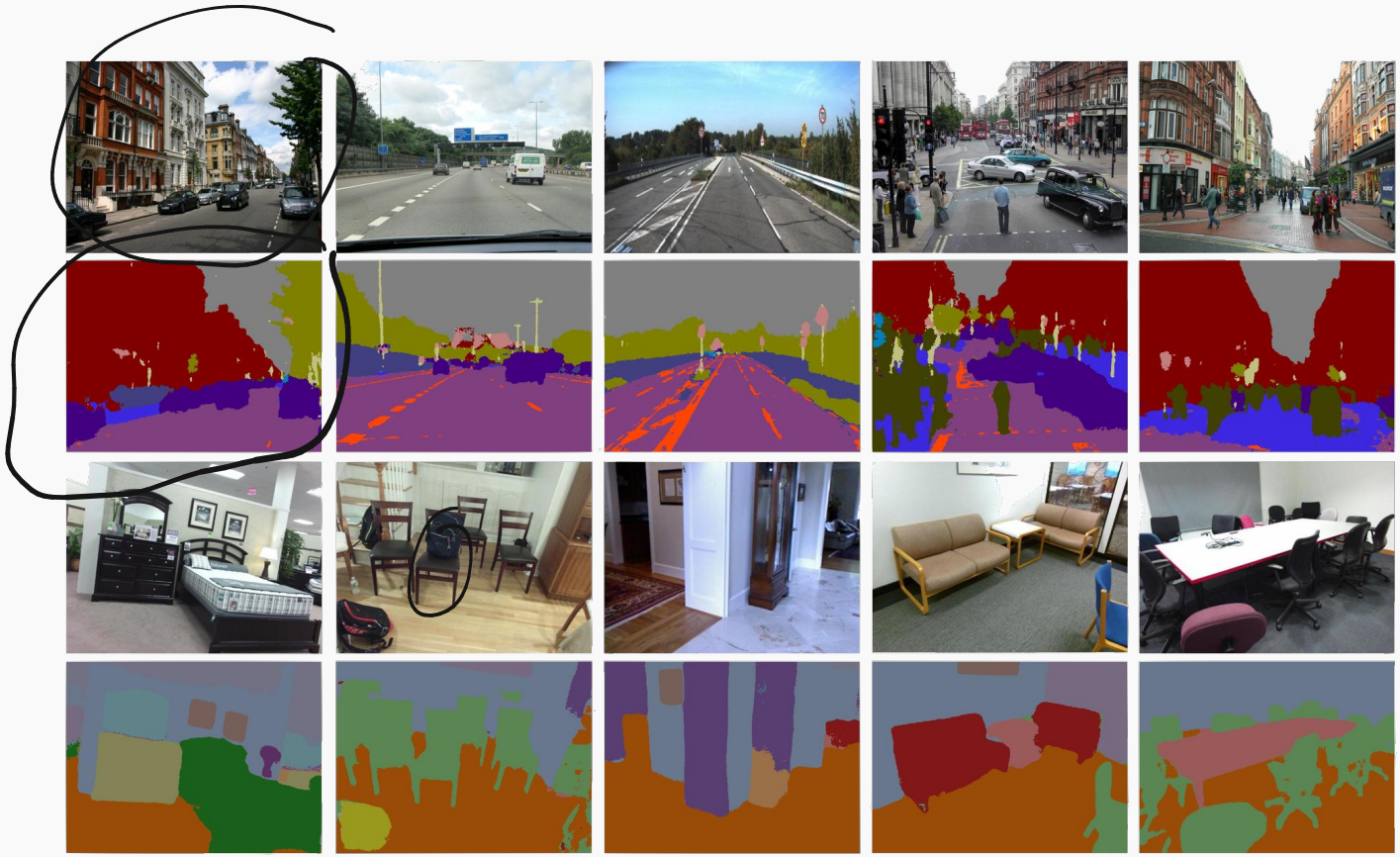
An autoencoder is a model $f : \mathbb{R}^d \to \mathbb{R}^d$. In other words, the output is the same dimension as the input:

- Image $\to$ Image
- Video $\to$ Video
- Audio clip $\to$ Audio clip

This structure is also useful for some supervised machine learning problems.

IMAGE SEGMENTATION

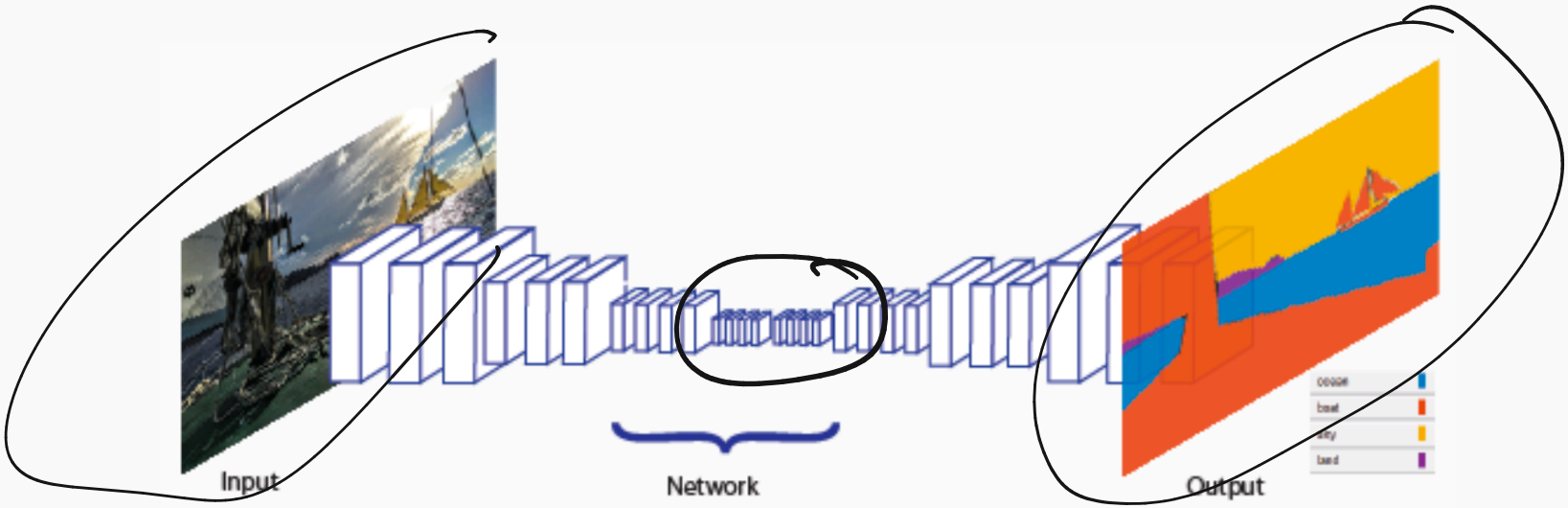
Goal: Learn mask which separates image pixels by what object (foreground or background) that they belong to.



First step in multi-objects classification and scene understanding. Harder than classifying single objects.

END-TO-END IMAGE SEGMENTATION

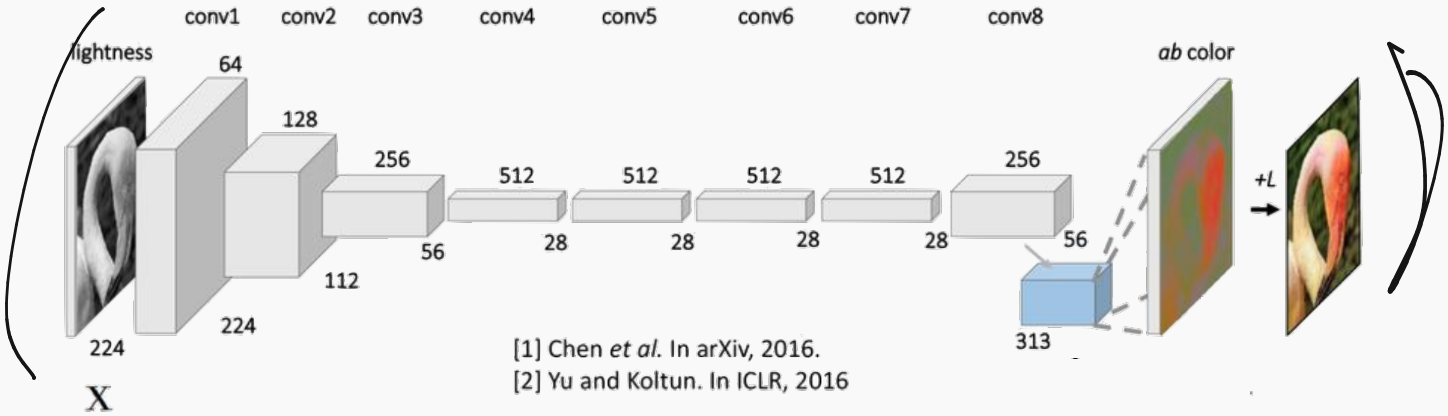
Model: Input is image **x**, output is image **m** that has the same size as **x**, but each pixel value is a label for a segmented region.

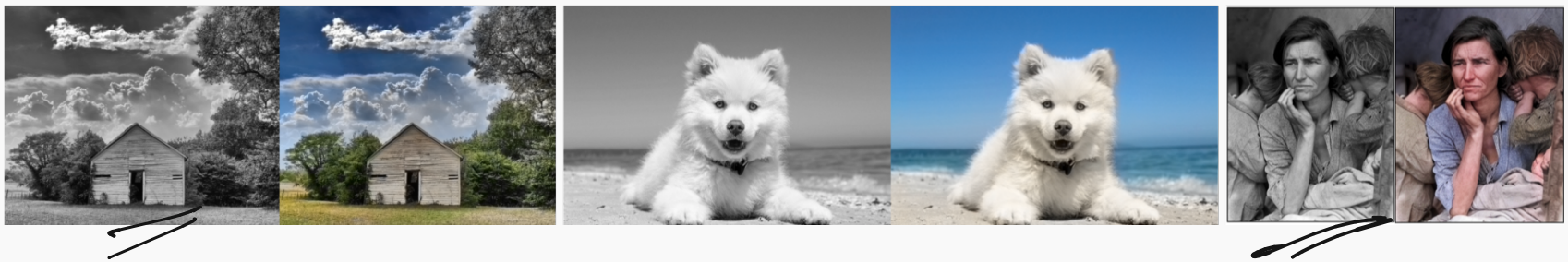


Now our training process is actually supervised, but uses the same structure as an autoencoder.

END-TO-END IMAGE COLORATION

Model: Input is black and white image **x**, output is colorized image **m**.





END-TO-END SUPER RESOLUTION

Model: Input is pixelated or blurred image **x**, output is full-resolution image **m**.

PRINCIPAL COMPONENT ANALYSIS

PRINCIPAL COMPONENT ANALYSIS

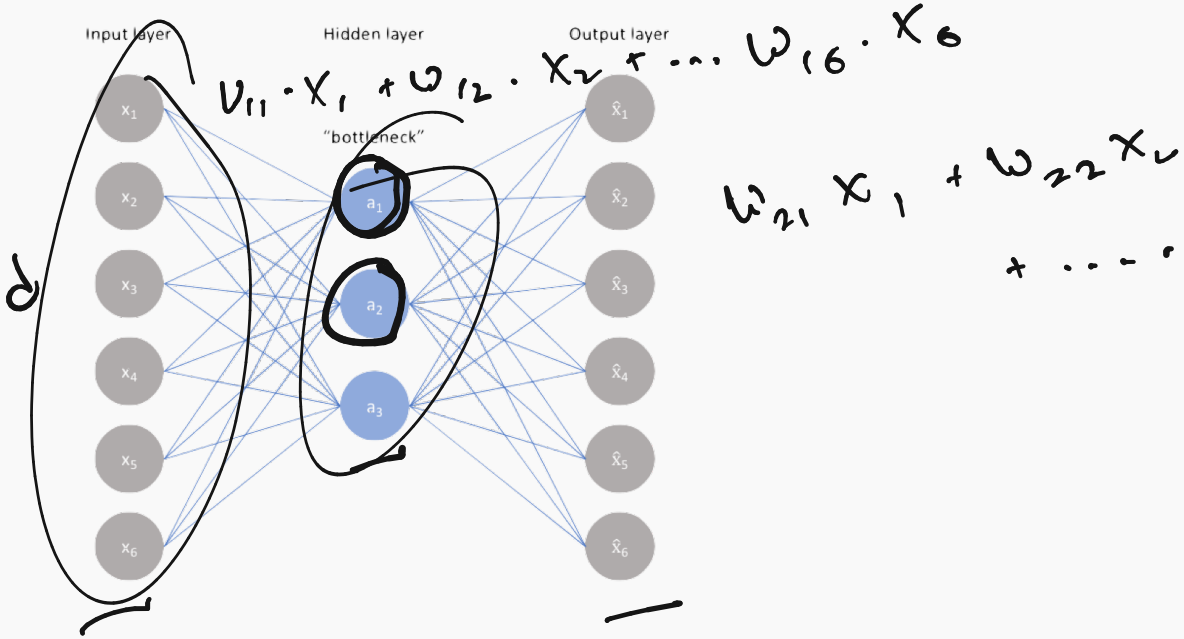
Rest of lecture: Deeper dive into understanding a simple, but powerful autoencoder architecture. Specifically we will view principal component analysis (PCA) as a type of autoencoder.

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.

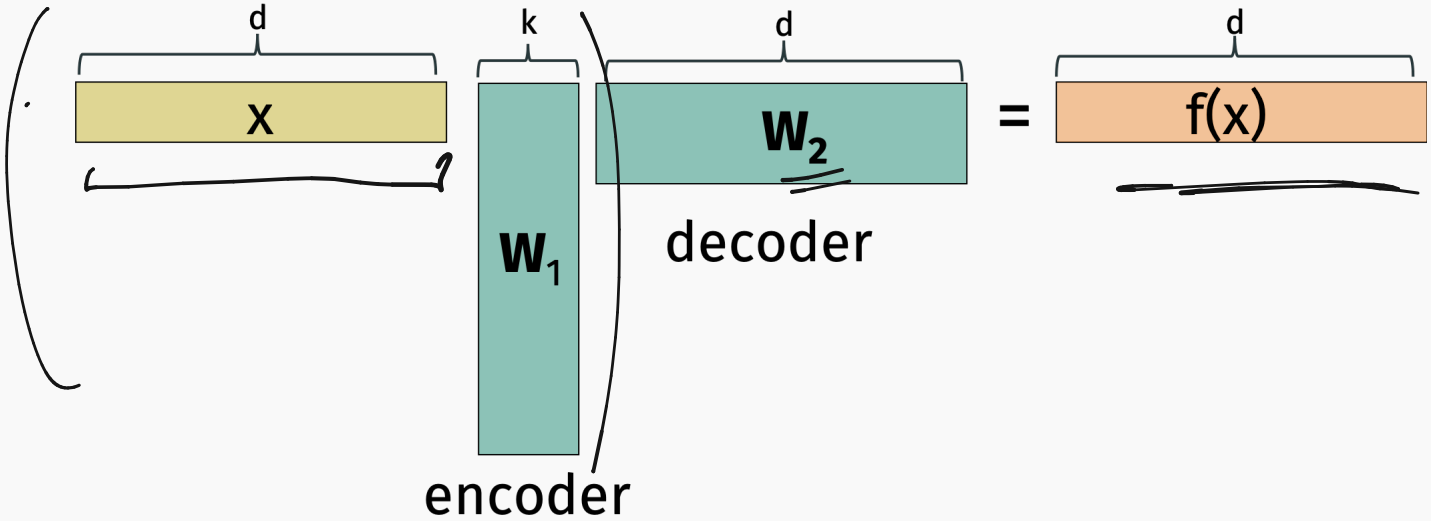
PRINCIPAL COMPONENT ANALYSIS

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?

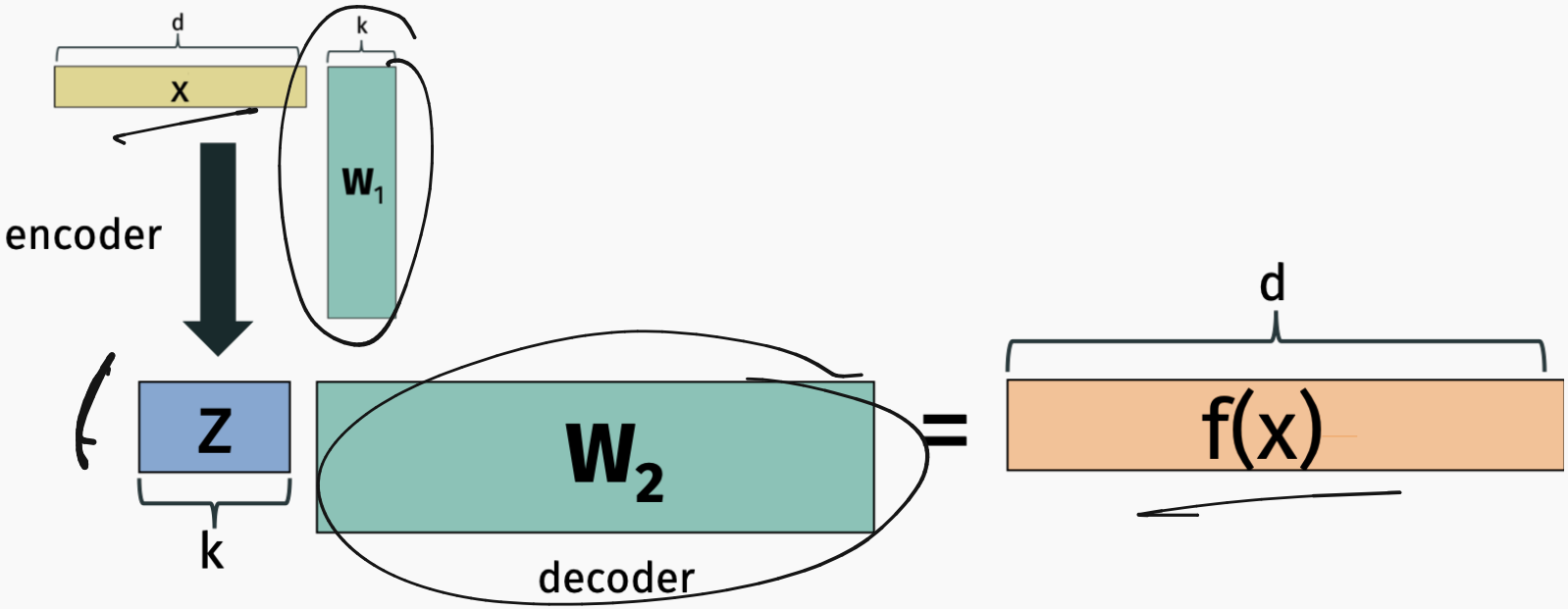


$(1 \times d)(d \times k) \rightarrow (1 \times k)$

$(1 \times k)(k \times d) \rightarrow (1 \times d)$

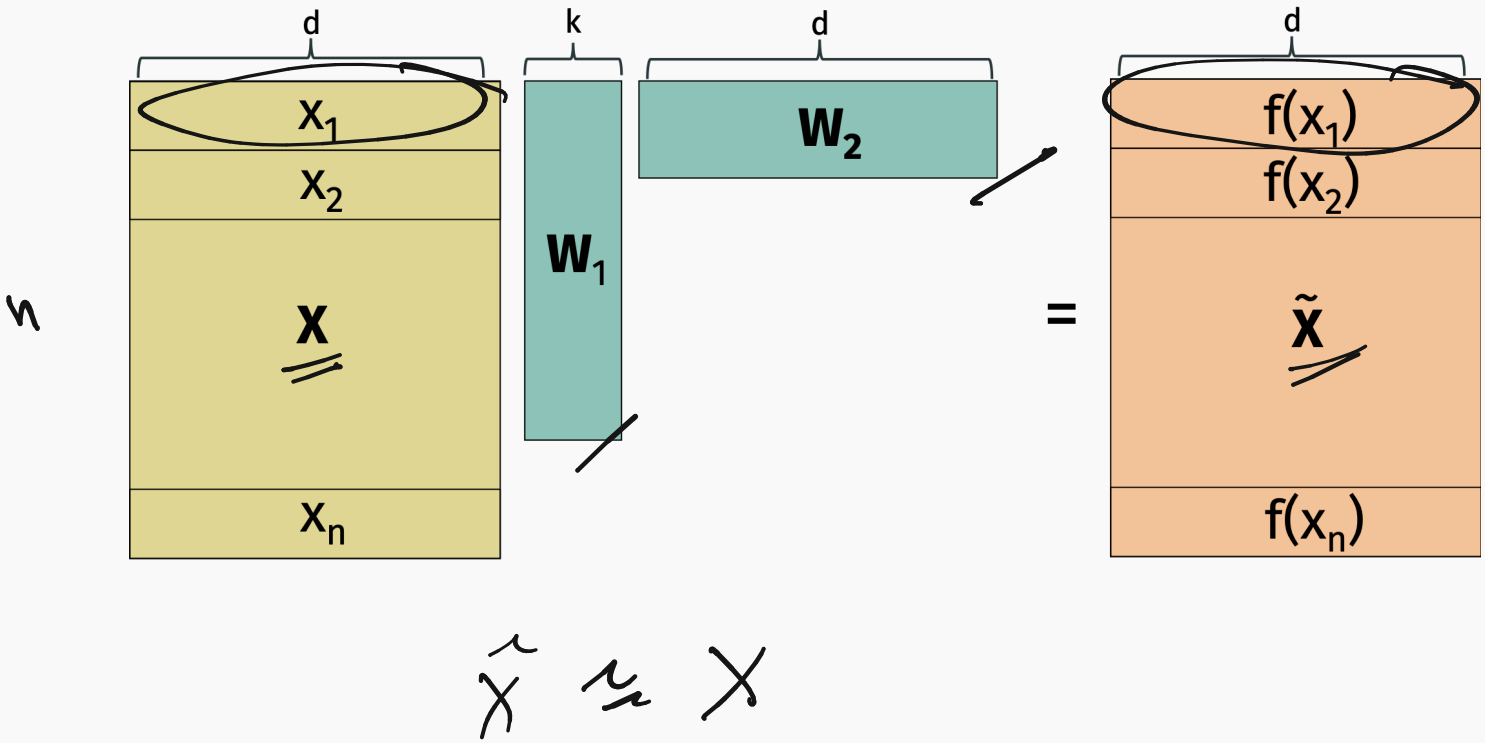
$$f(\mathbf{x}) = \mathbf{x} W_1 W_2$$

PRINCIPAL COMPONENT ANALYSIS



Encoder: $e(x) = xW_1$. $= z$ Decoder: $d(z) = zW_2$

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



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

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

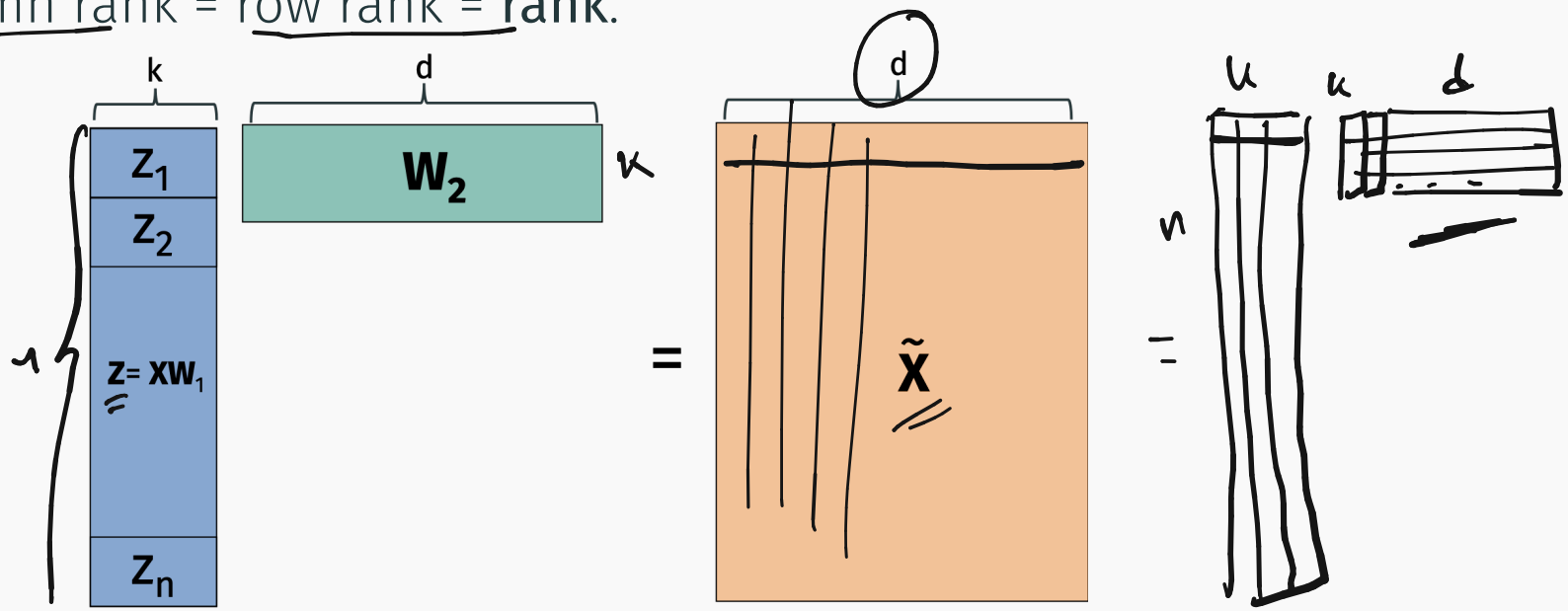
$$= \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{x}_i[j] - f(\mathbf{x}_i)[j])^2$$

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

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).

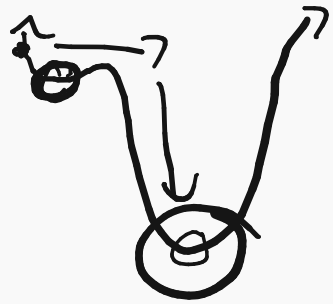
Rank in linear algebra:

- 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

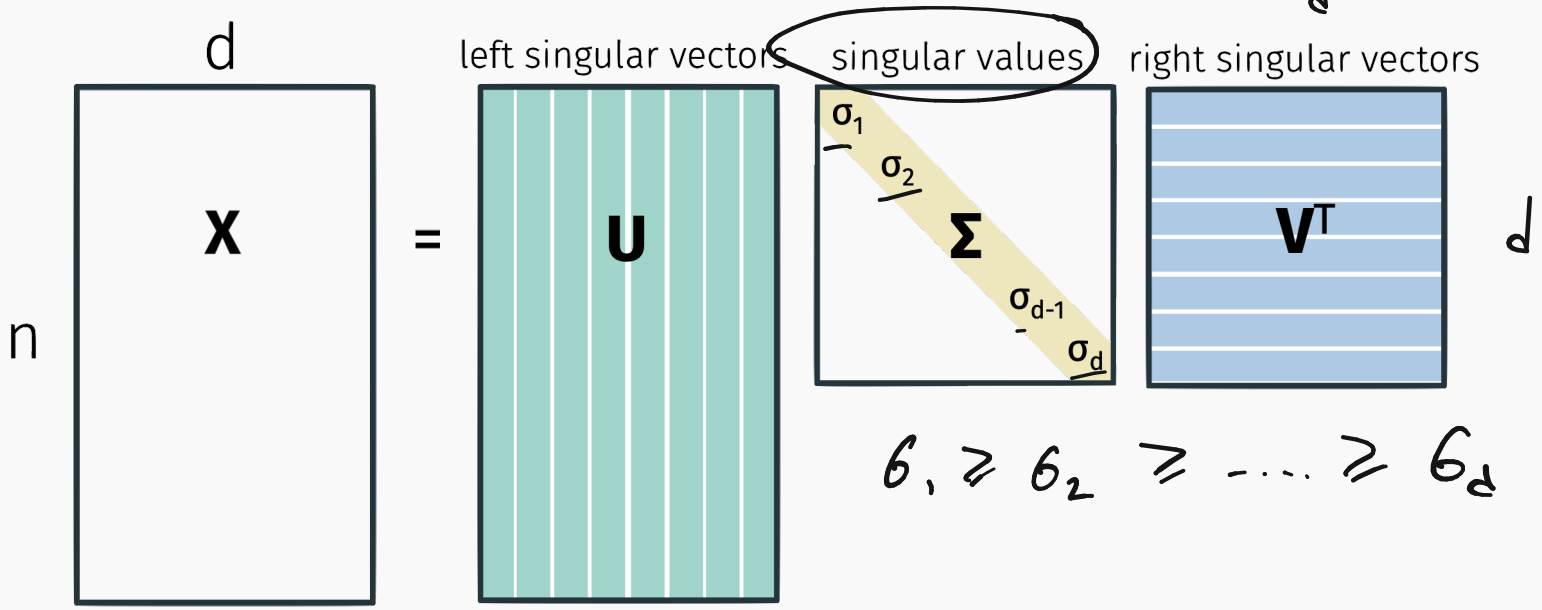


Principal component analysis is the task of finding $\mathbf{W}_1$, $\mathbf{W}_2$, which amounts to finding a rank k matrix $\tilde{\mathbf{X}}$ which approximates the data matrix $\mathbf{X}$ as closely as possible.

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 $\mathbf{X}$ can be written:

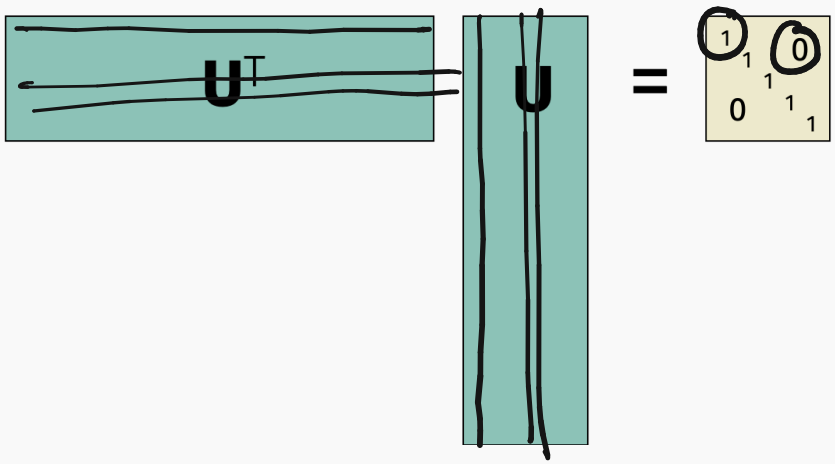


Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices.

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 $\mathbf{X}$.

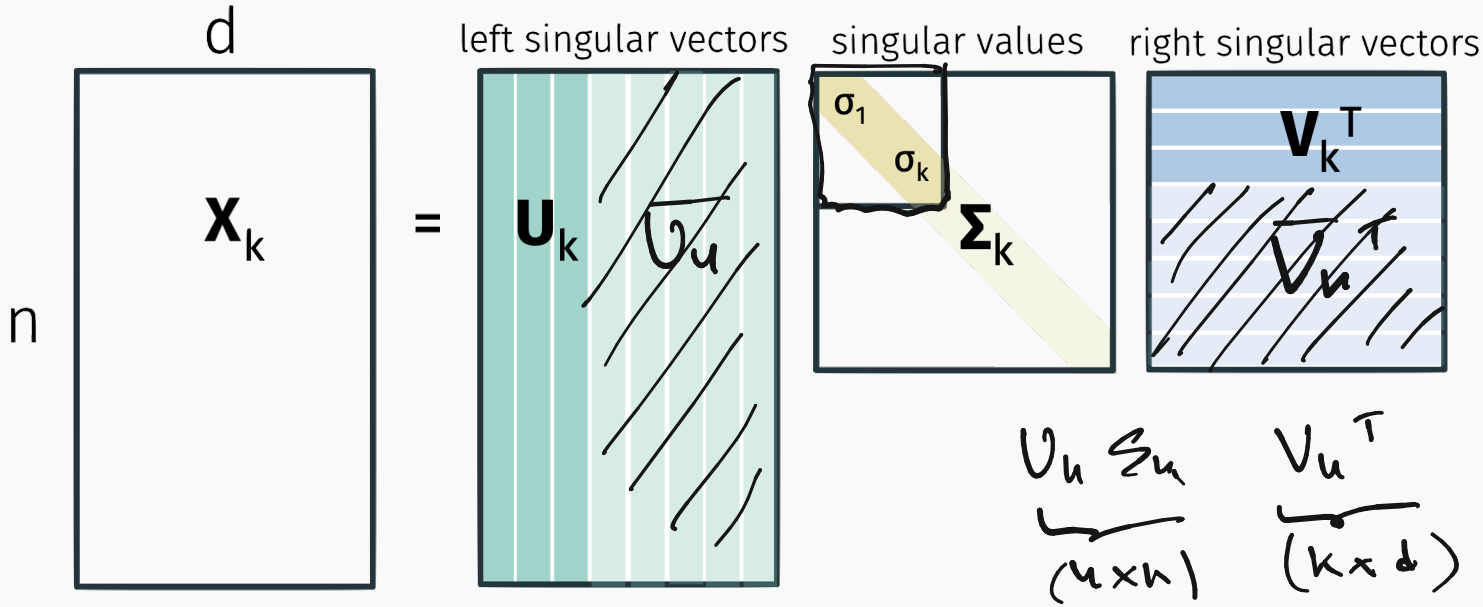


$\|\mathbf{u}_i\|_2^2 = 1$

$\mathbf{u}_i^T \mathbf{u}_j = 0$

$v_1^T v_1 = \|v_1\|_2^2 = 1$

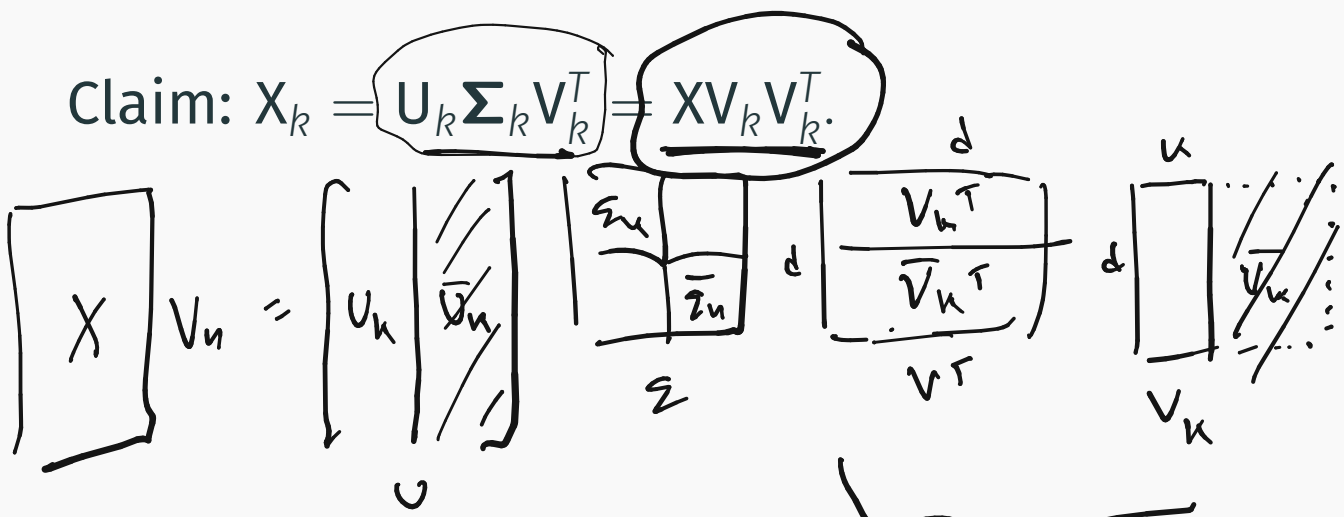
Can read off optimal low-rank approximations from the SVD:

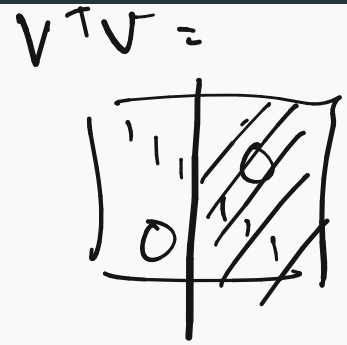


Eckart–Young–Mirsky Theorem: For any $k \leq d$, $\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

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

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





$XV_k = U_k \Sigma_k$

$XU_k V_k^T = U_k \Sigma_k V_k^T$

$U \leq V^T V_k$

$= \begin{bmatrix} 6_1 \\ \ddots \\ 6_k \\ \hline 0 \end{bmatrix}$

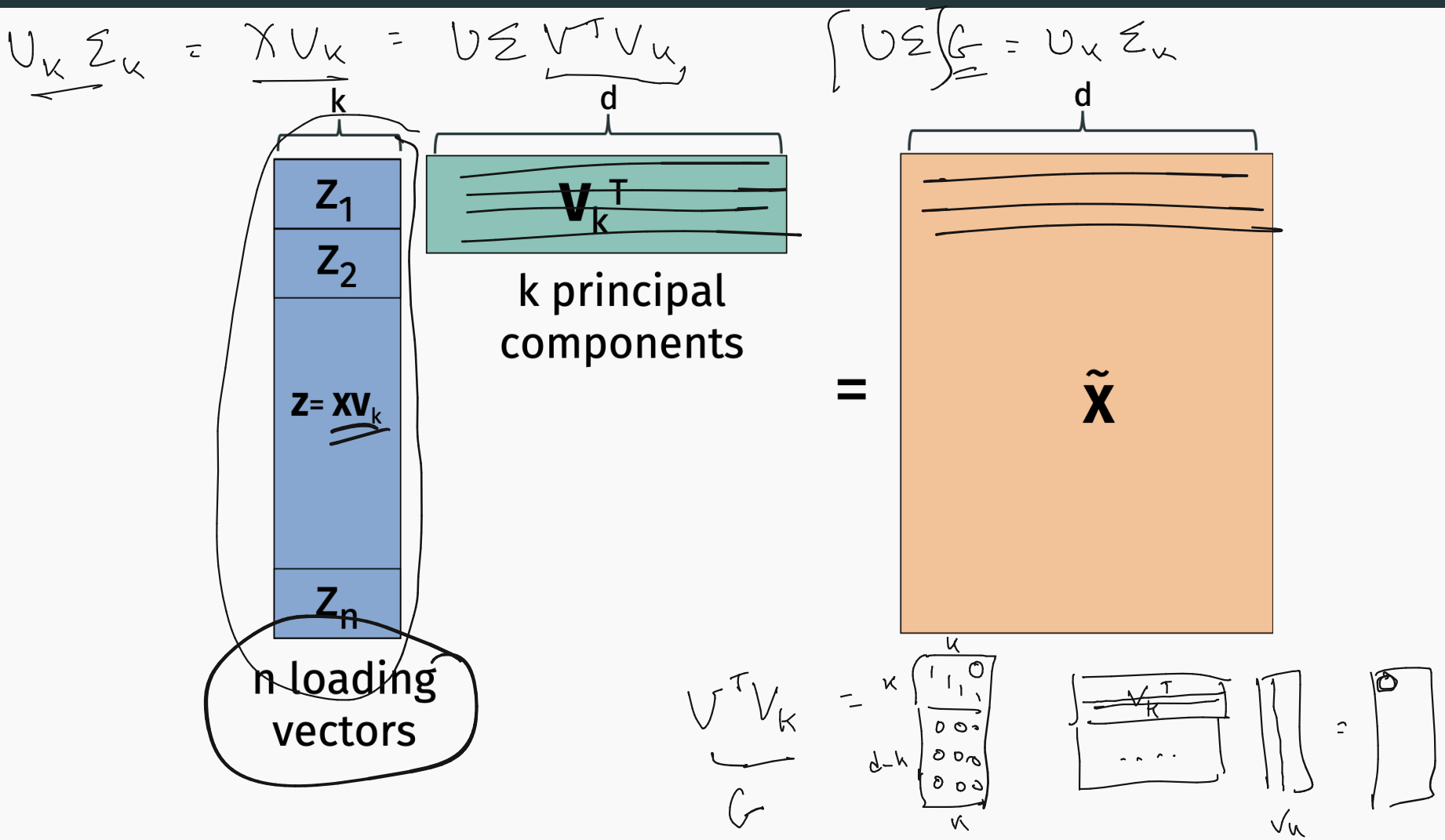
$U_k \Sigma_k$

$\begin{bmatrix} 1 \\ 1 \\ 1 \\ \hline 0 \end{bmatrix} = V^T V_k$

$XW_1 W_2$ is the best rank-k approximation to X.

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

PRINCIPAL COMPONENT ANALYSIS



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.

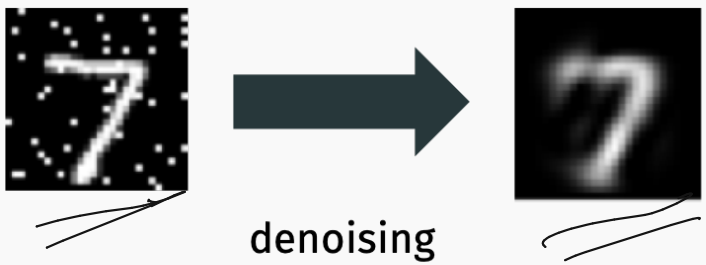
- $\mathbf{U}$'s columns (the left singular vectors) are the orthonormal eigenvectors of $\mathbf{X}\mathbf{X}^T$.
- $\mathbf{V}$'s columns (the right singular vectors) are the orthonormal eigenvectors of $\mathbf{X}^T\mathbf{X}$.
- $\sigma_i^2 = \lambda_i(\mathbf{X}\mathbf{X}^T) = \lambda_i(\mathbf{X}^T\mathbf{X})$

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

PCA APPLICATIONS

Like any autoencoder, PCA can be used for:

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



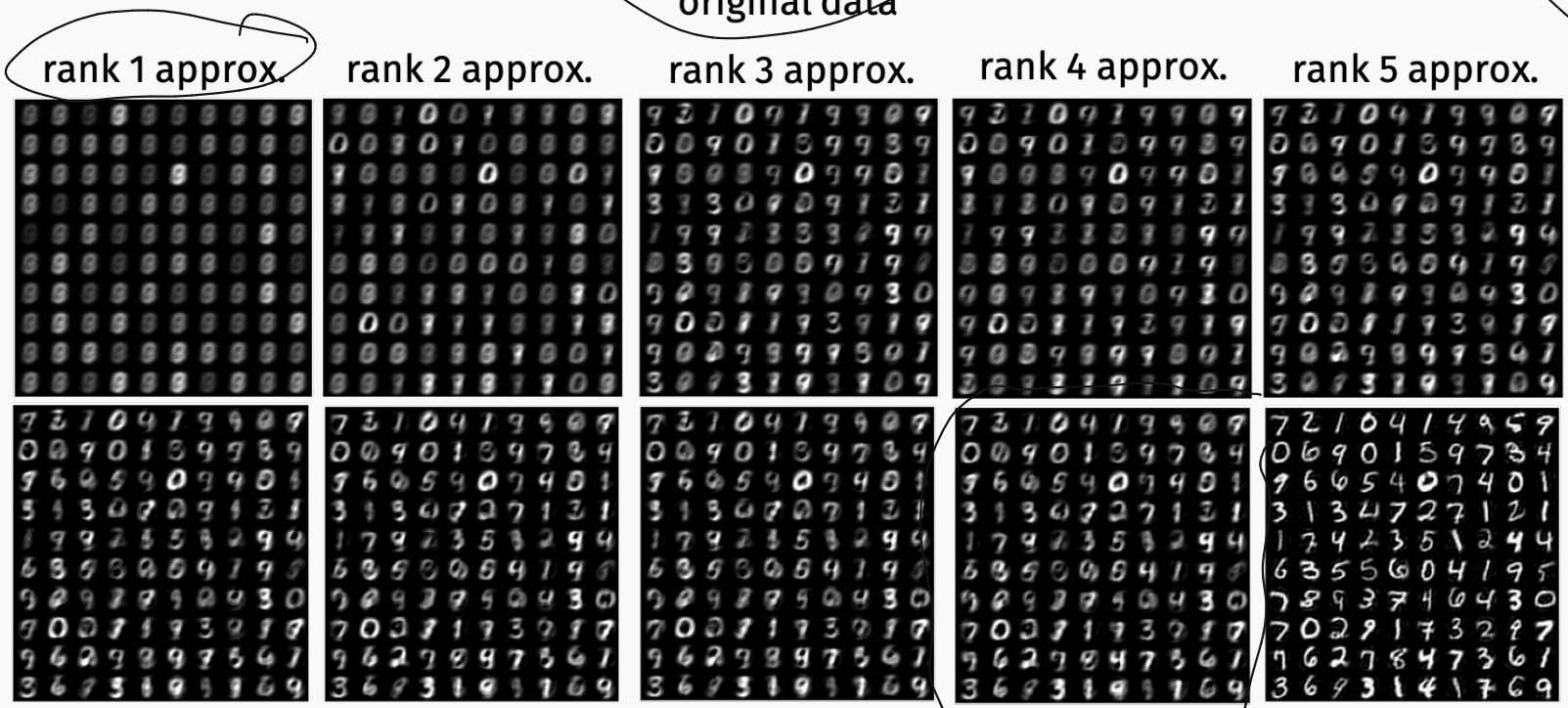
denoising

synthetic data generation

LOW-RANK APPROXIMATION

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

original data

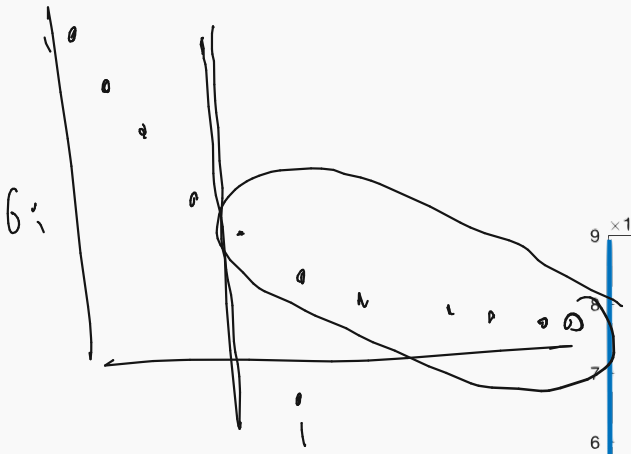


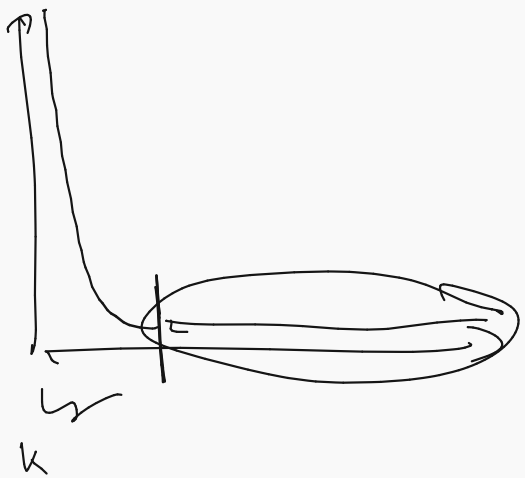
rank 1 approx. rank 2 approx. rank 3 approx. rank 4 approx. rank 5 approx.

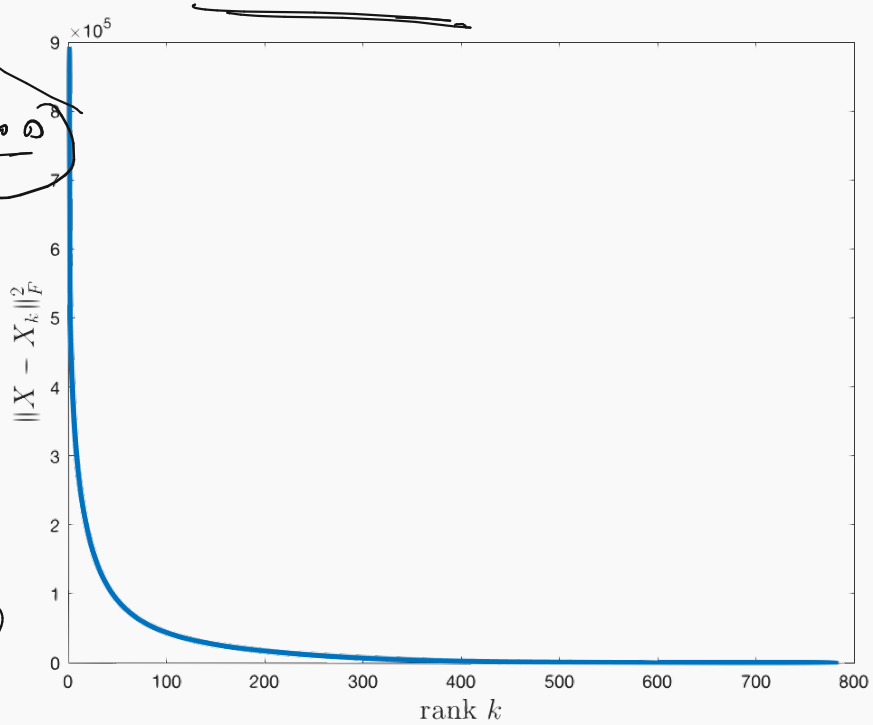
rank 6 approx. rank 7 approx. rank 8 approx. rank 9 approx. rank 50 approx.

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

$$\|\mathbf{X} - \mathbf{X}_k\|_F^2 = \sum_{i=k+1}^{d} \sigma_i^2.$$







saleprice $\approx$ 1 · list price

Which of these data sets would you expect to have a good low-rank approximation? Why?

.05 · list price

1. House data:

$$\mathbf{x} = [\#\text{ bedrooms}, \#\text{ bathrooms}, \text{list price}, \text{sale price}, \text{property tax}]$$

2. Student data:

$$\mathbf{x} = [\text{gender}, \text{year}, \text{age}, \text{GPA}, \text{engineering major}]$$

list price $\approx c + a(\#\text{ bedrooms}) + b(\#\text{ bathrooms})$

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.

$$\text{yard size} \approx \text{lot size} - [\text{illegible}] \cdot \text{square footage}.$$

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

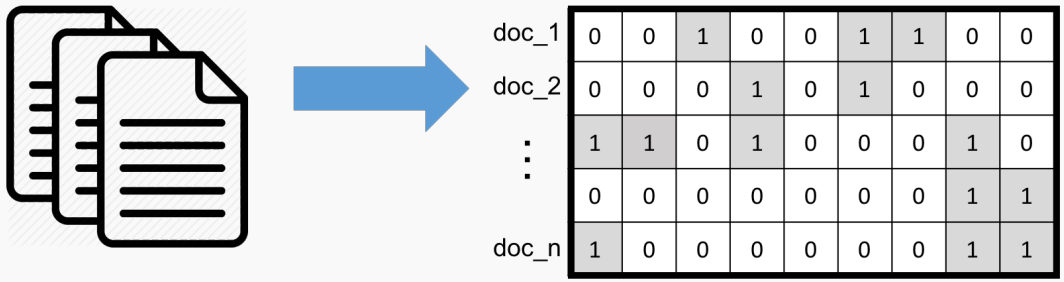
LOW-RANK INTUITION

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

1. Genetic data:

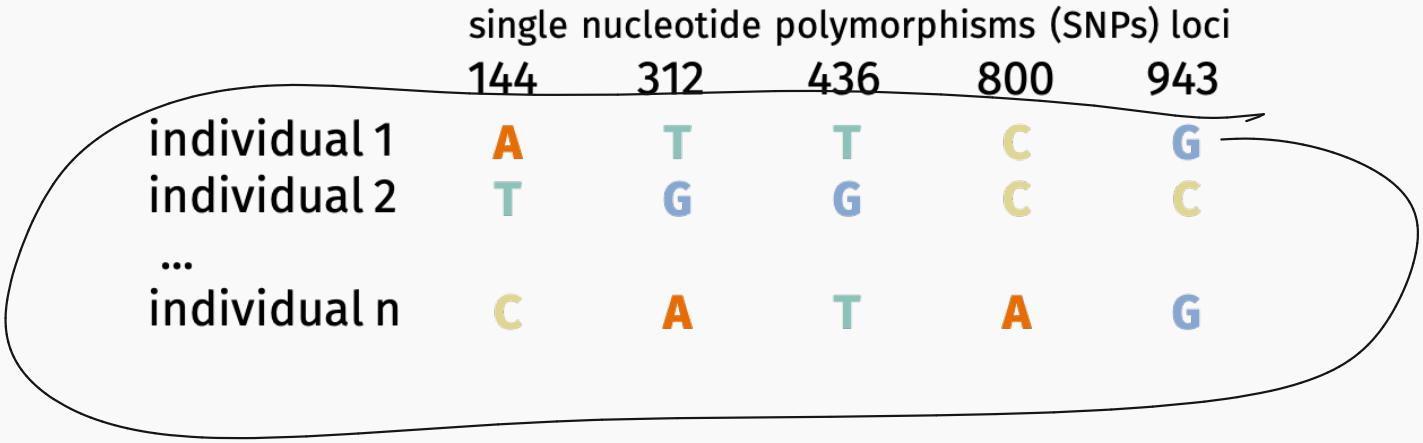
	single nucleotide polymorphisms (SNPs) loci				
	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

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

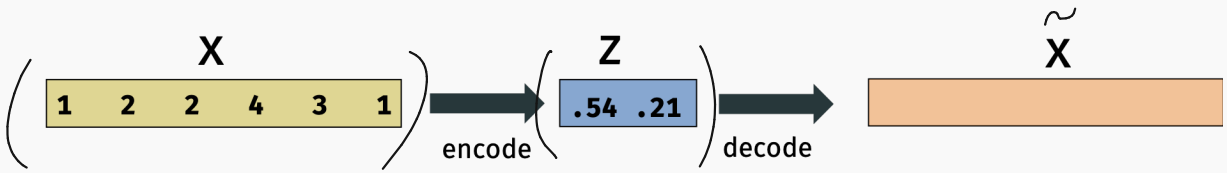


EXAMPLES OF LOW-RANK STRUCTURE

SNPs matrices tend to be very low-rank.

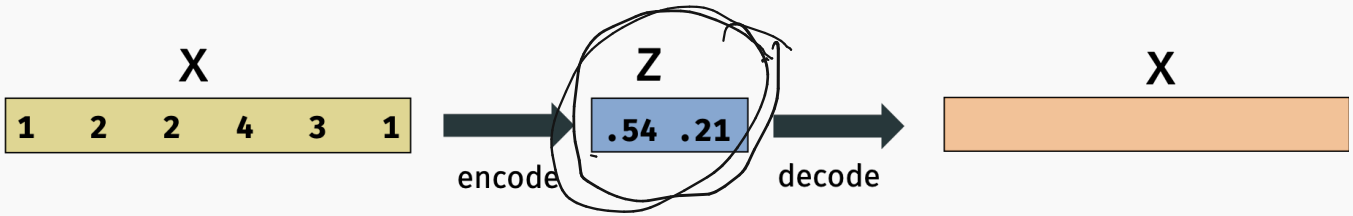


Most of the information in **x** is explained by just a few **latent variable**.



EXAMPLES OF LOW-RANK STRUCTURE

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



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

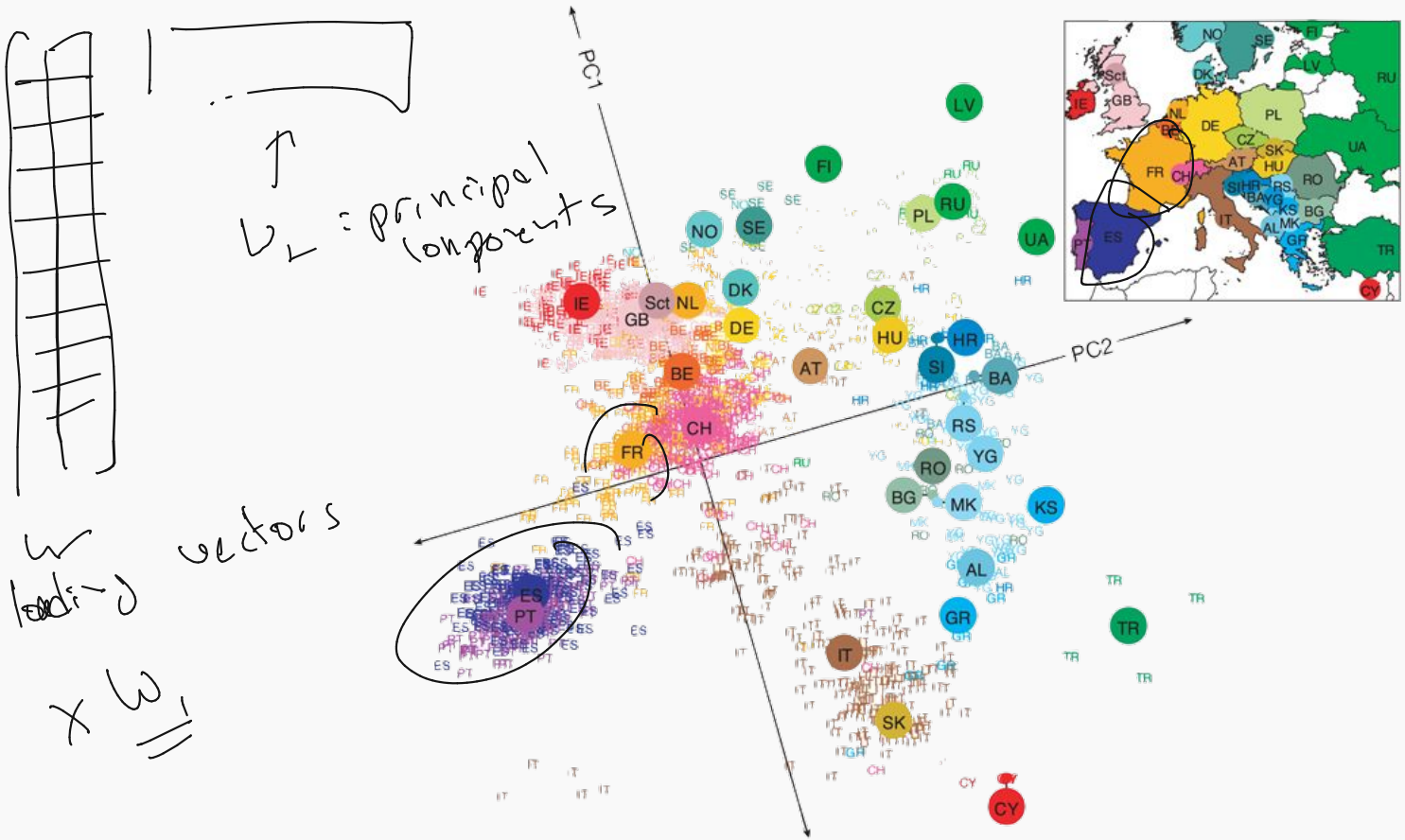
$z[1] \approx$ relative north-south position of birth place

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

Individuals born in similar places tend to have similar genes.

PCA FOR DATA VISUALIZATION

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



Genetic data can be nicely visualized using PCA! Plot each data example $\mathbf{x}$ using two loading variables in $\mathbf{z}$.

SIMILARITY PRESERVATION

Important note 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.

Recall that $\tilde{\mathbf{x}}_i = \mathbf{V}_k \mathbf{z}_i$. Because $\mathbf{V}_k$ is orthogonal, we have:

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

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

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

SIMILARITY PRESERVATION

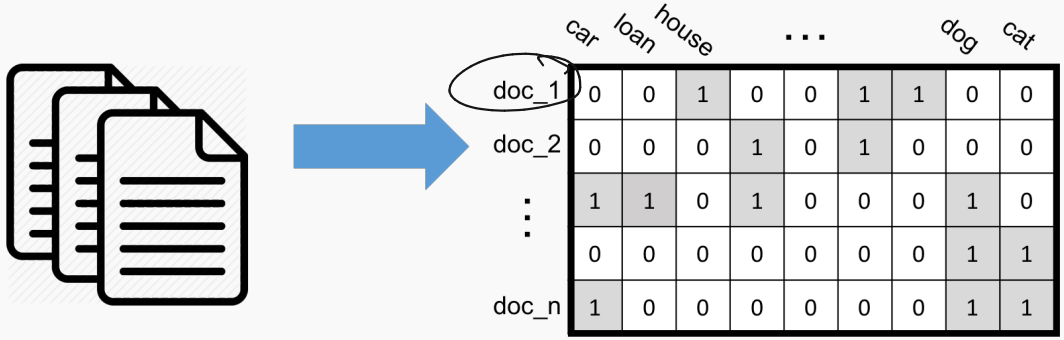
$\|x_i - x_j\|_2^2 \approx \|\tilde{x}_i - \tilde{x}_j\|_2^2$

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

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

TERM DOCUMENT MATRIX

Word-document matrices tend to be low rank.

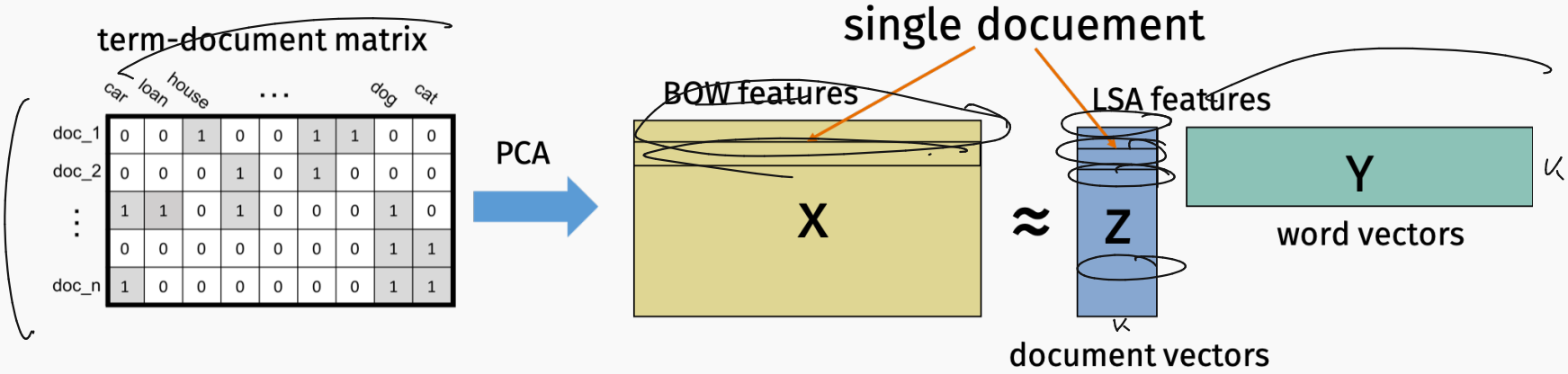


Documents tend to fall into a relatively small number of different categories, which use similar sets of words:

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

Intuition that this data should have co-linear **rows**.

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).

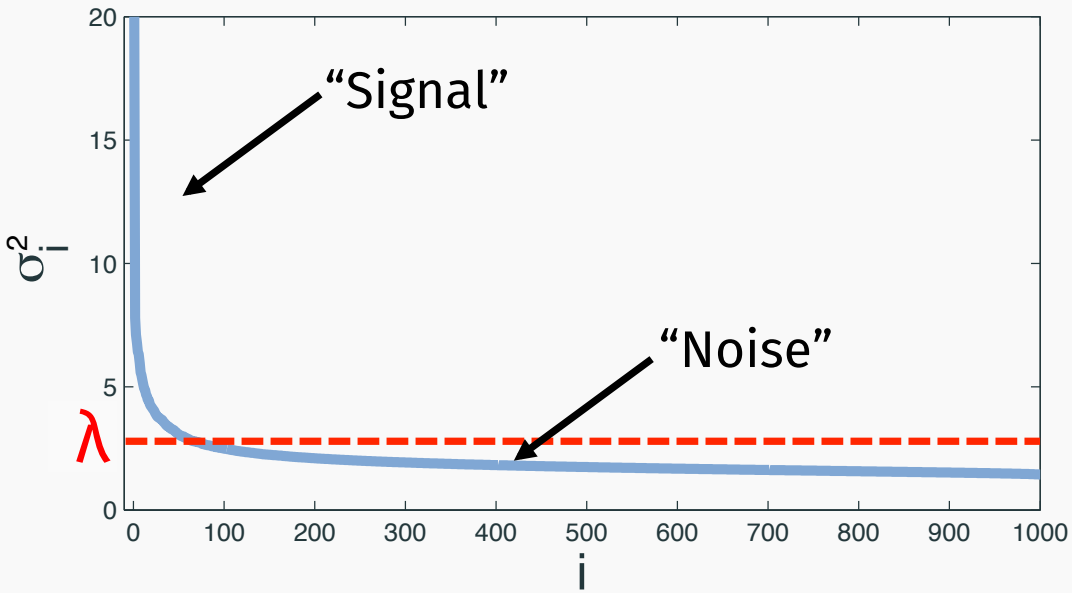


Each column of **Z** corresponds to a latent "category" or "topic".

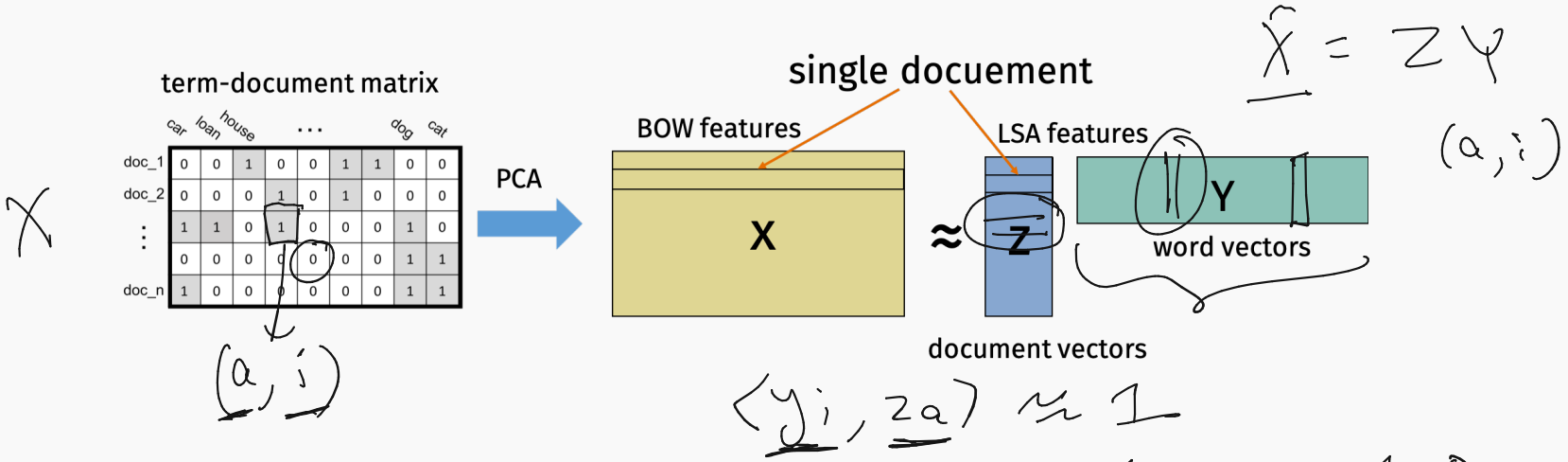
Similar documents have similar LSA document vectors. I.e. $z_i \cdot z_j$ is large. Provide a more compact "finger print" for documents than the long bag-of-words vectors. Useful for e.g search engines.

LSA vectors often provide a more meaningful similarity metric than bag-of-words vectors. Capture high-level categorical information and eliminate document specific quirks.

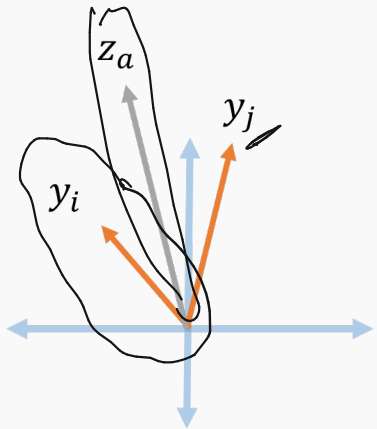
Spectrum of data matrix **X**.



WORD EMBEDDINGS

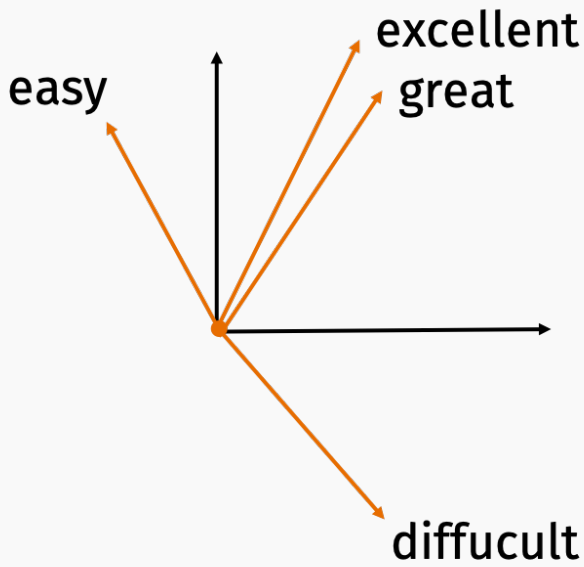


- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$ when doc_i contains $word_a$.
- If doc_i and doc_j both contain $word_a$, $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$.



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.



Word embeddings

Extremely useful "side-effect" of LSA.

WORD EMBEDDINGS: MOTIVATING PROBLEM

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".

BAG-OF-WORDS FEATURES

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.*

[, , , , , , ,]

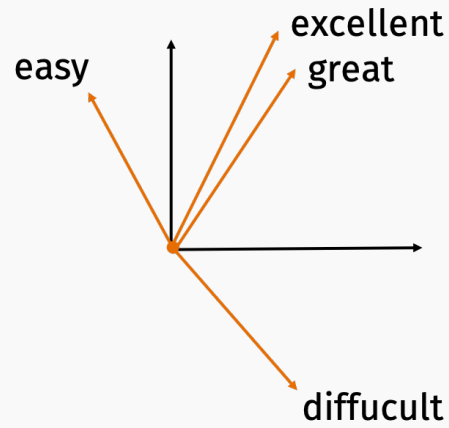
Review 3: *Not entirely sure this was worth* $20. *Mom couldn't figure out how to use it and it's fairly difficult to turn for someone with arthritis.*

[, , , , , , ,]

SEMANTIC EMBEDDINGS

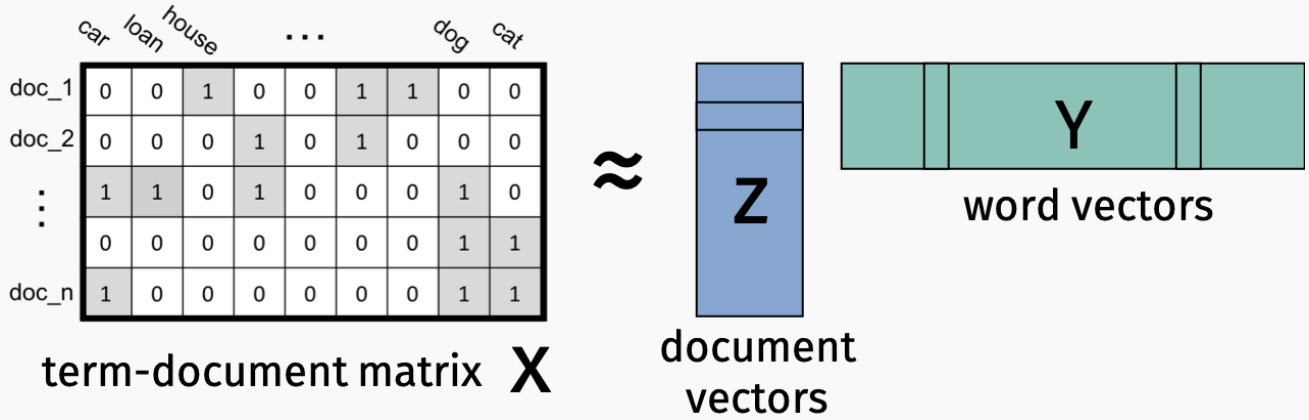
Bag-of-words approach typically only works for large data sets.

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).

Another view on word embeddings from LSA:

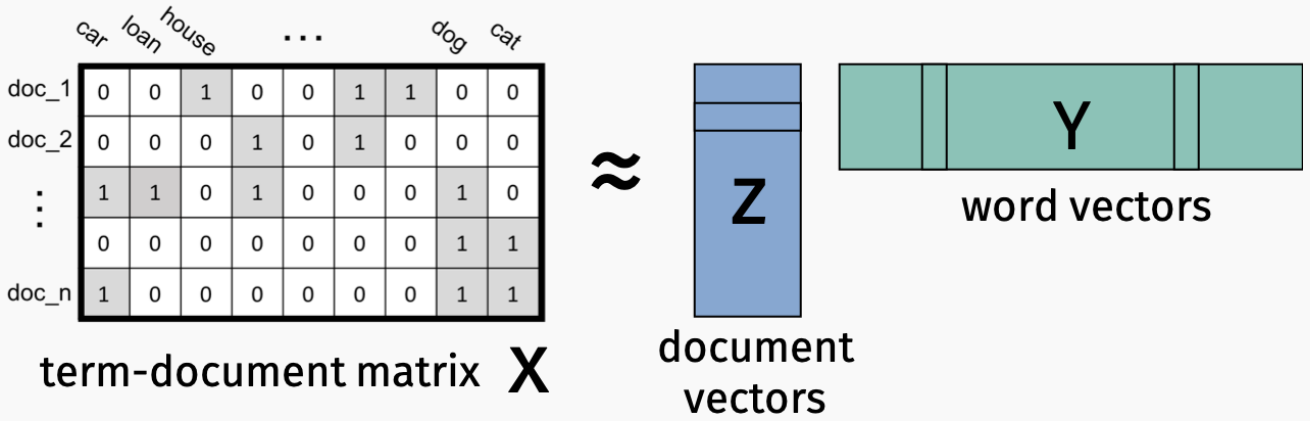


Choose Z to have orthogonal columns. E.g. $Z = U_k$ and $Y = \Sigma_k V_k^T$.

- $X \approx ZY$
- $X^T X \approx Y^T Z^T Z Y = Y^T Y$
- 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?

WORD EMBEDDINGS

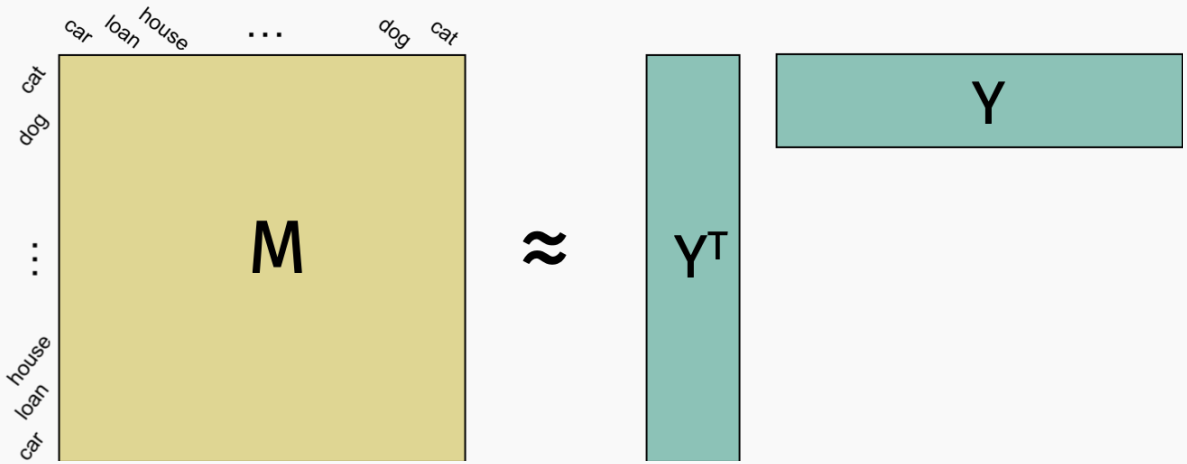


What does the i, j entry of $\mathbf{X}^T\mathbf{X}$ reprent?

$\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ 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 symmetric similarity matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ with $\mathbf{M}_{i,j} = k(word_i, word_j)$.
3. Find symmetric low rank factorization $\mathbf{M} \approx \mathbf{Y}^T\mathbf{Y}$ 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 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.

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

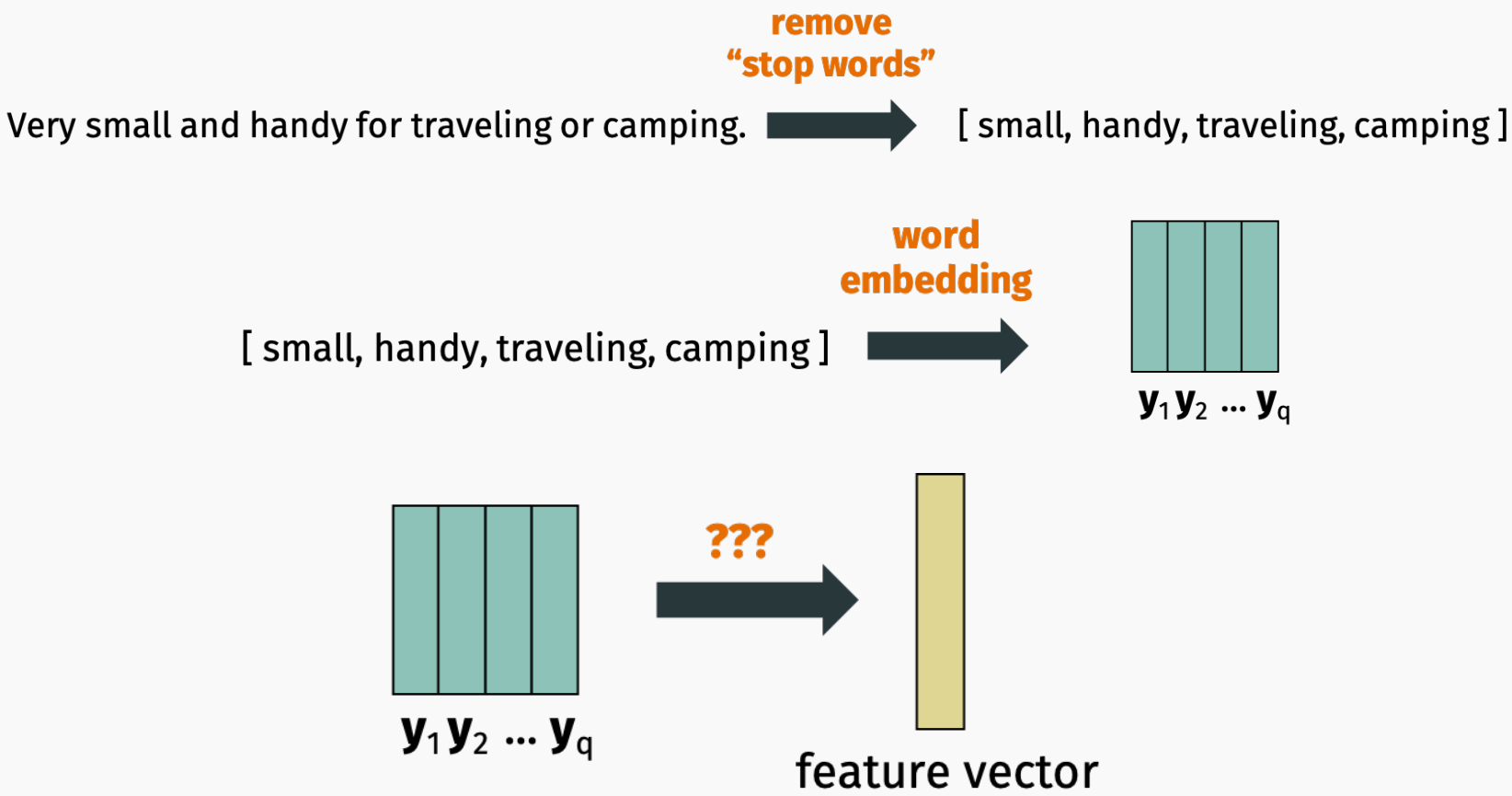
- Based on same principal as LSA.
- `word2vec` was originally presented as a shallow neural network model, but it can be viewed as a matrix factorization method (Levy, Goldberg 2014).
- For `word2vec`, similarity metric is the "point-wise mutual information": $\log \frac{p(i,j)}{p(i)p(j)}$.

EASIEST WAY TO USE WORD EMBEDDINGS

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

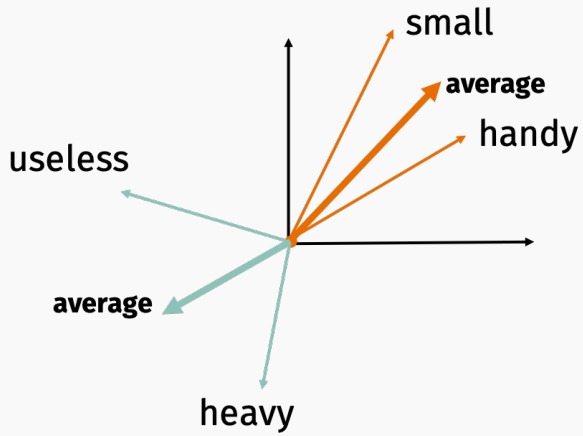
- Original gloVe website: `https://nlp.stanford.edu/projects/glove/`.
- Compilation of many sources: `https://github.com/3Top/word2vec-api`

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

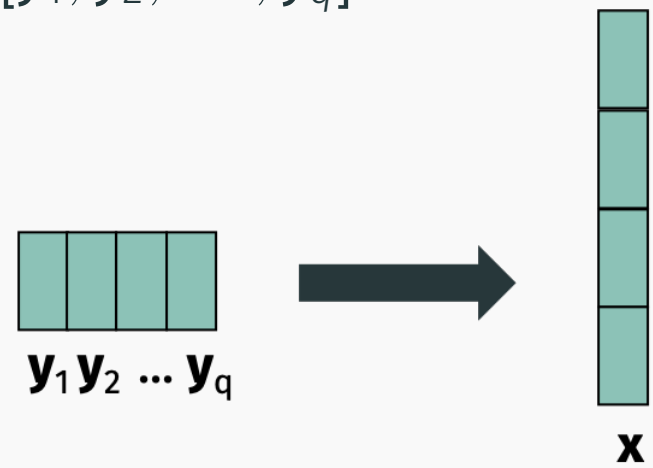


A few simple options:

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



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



Better option than concatenation: 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: