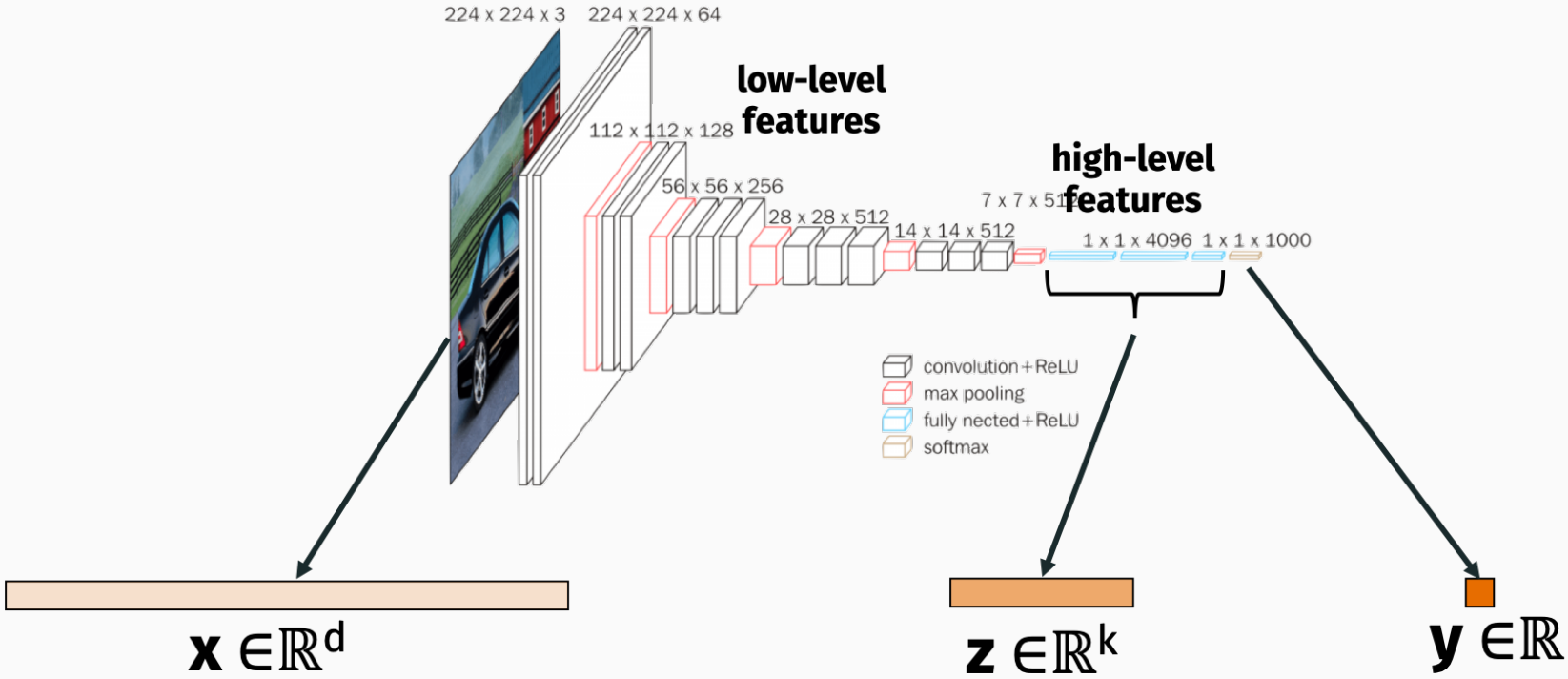
CS-GY 6923: Lecture 13
Principal Component Analysis, Semantic Embeddings

NYU Tandon School of Engineering, Prof. Christopher Musco

Empirical observation: Features learned when training models like deep neural nets are often useful for problems beyond what the model was trained on.



TRANSFER LEARNING

Very useful in domains like computer vision where we have huge labeled datasets to train deep models on. Approach:

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 for a new problem (e.g., quidditch ball detection), typically using a simpler machine learning algorithm that requires less data (nearest neighbor, logistic regression, etc.).

UNSUPERVISED FEATURE LEARNING

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.

First of many simple but clever ideas: 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.

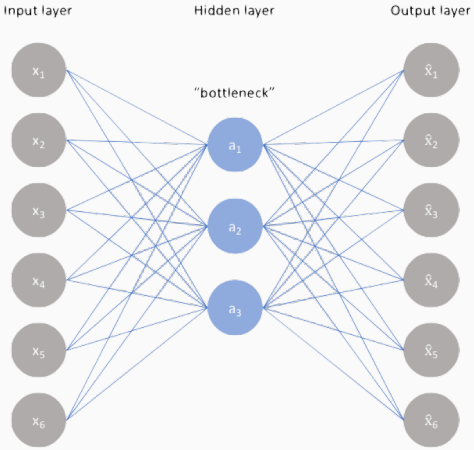
- Let $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$ be our model.
- Let $L_{\boldsymbol{\theta}}$ be a loss function. E.g. squared loss: $L_{\boldsymbol{\theta}}(\mathbf{x}) = \|\mathbf{x} - f_{\boldsymbol{\theta}}(\mathbf{x})\|_2^2$.
- Train model: $\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}}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

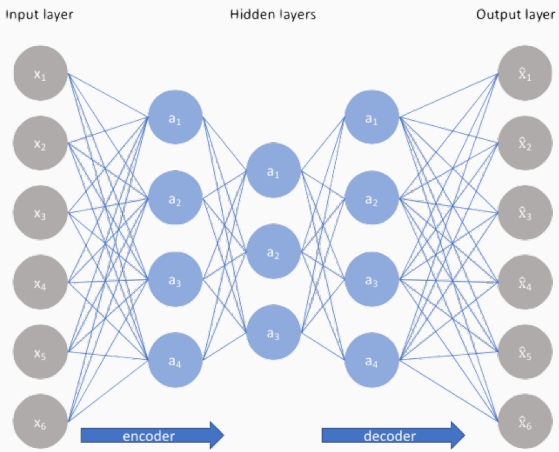
If $f_{\boldsymbol{\theta}}$ is a model that incorporates feature learning, then these features can be used for supervised tasks.

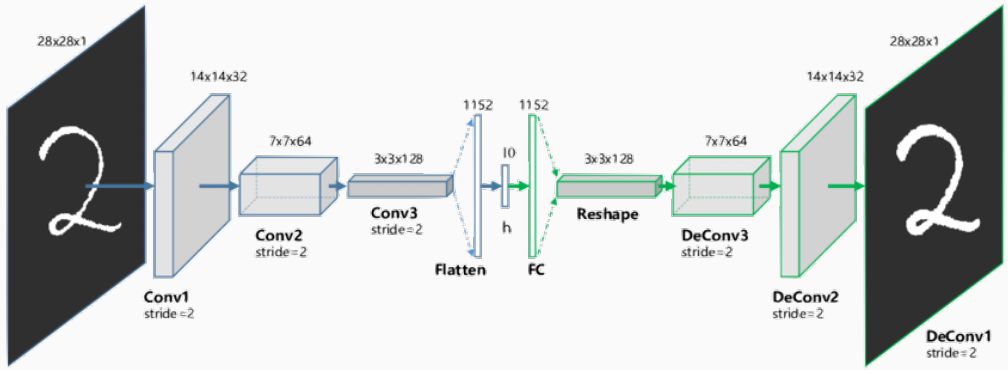
$f_{\boldsymbol{\theta}}$ is called an autoencoder. It maps input space to input space (e.g. images to images, french to french, PDE solutions to PDE solutions).

AUTOENCODER

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.





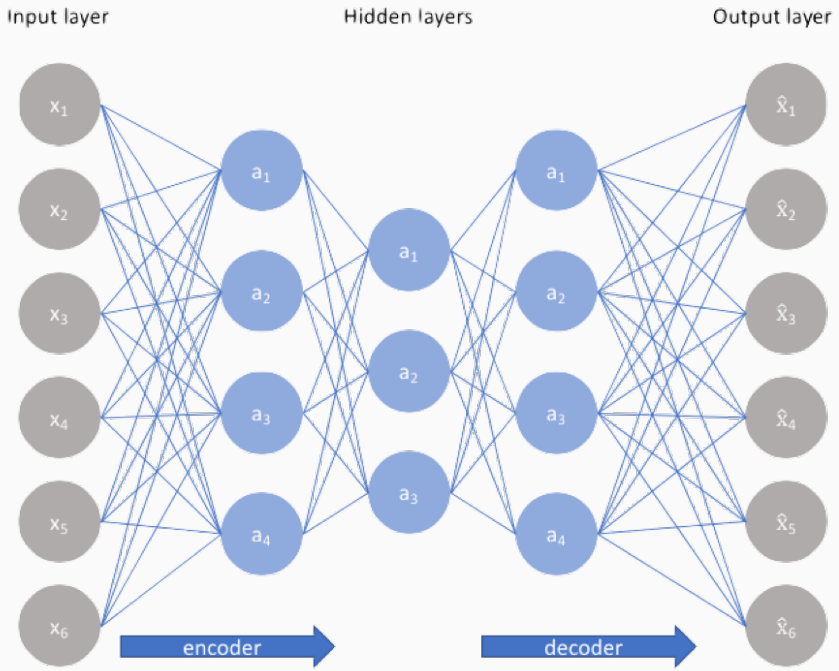


AUTOENCODER

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

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

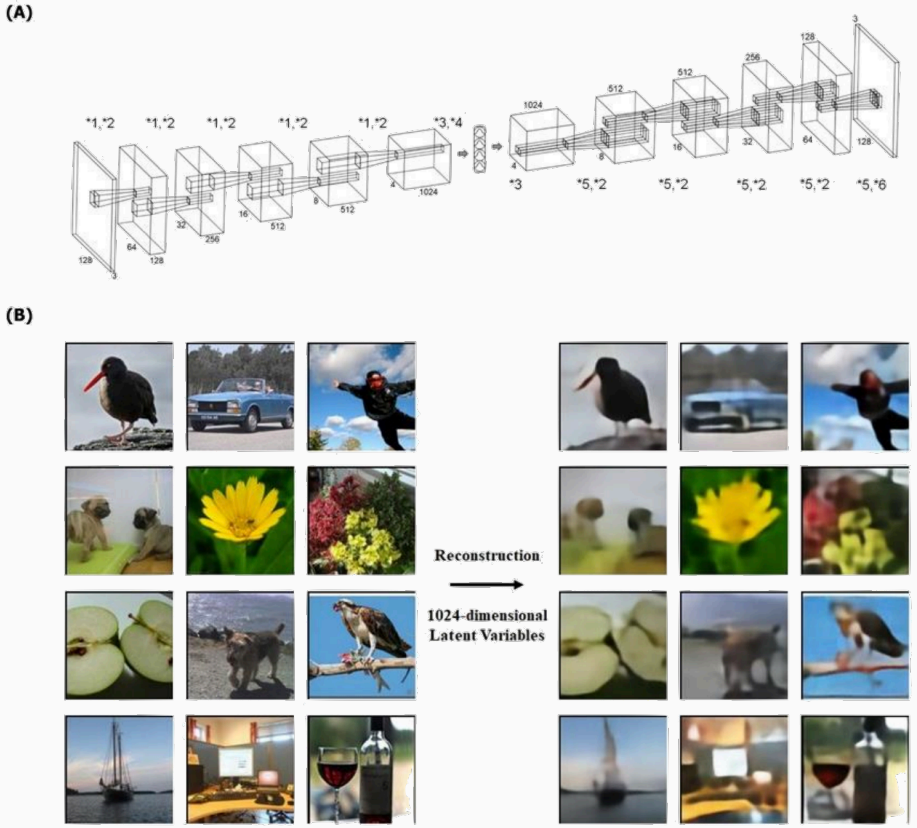
$$f(\mathbf{x}) =$$



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

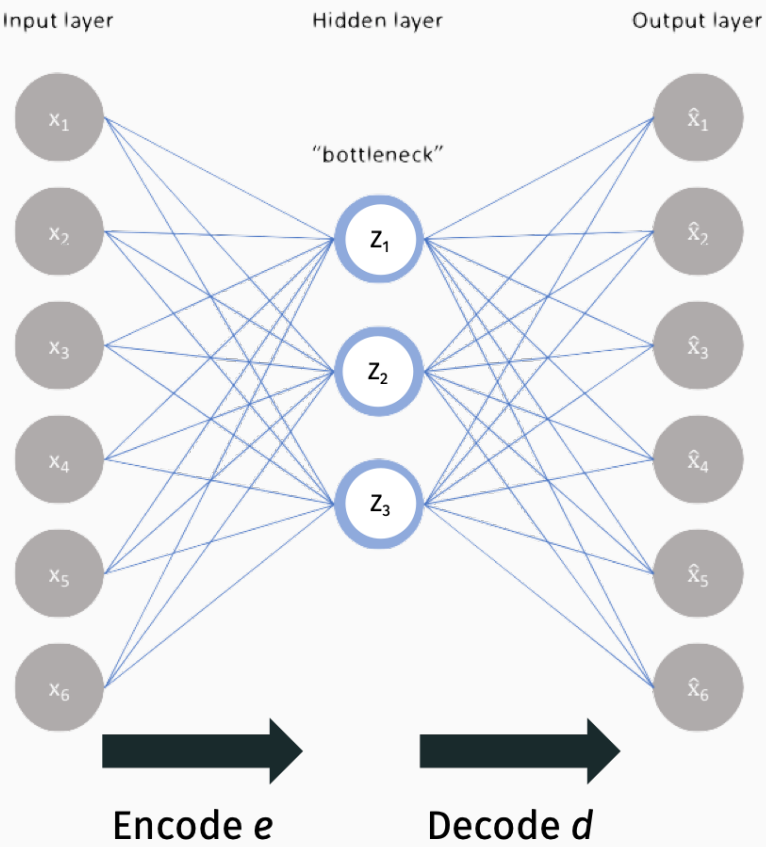
The best autoencoders do not work as well as supervised methods for feature extraction, but they require no labeled data.

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

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

AUTOENCODERS FOR DATA COMPRESSION

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



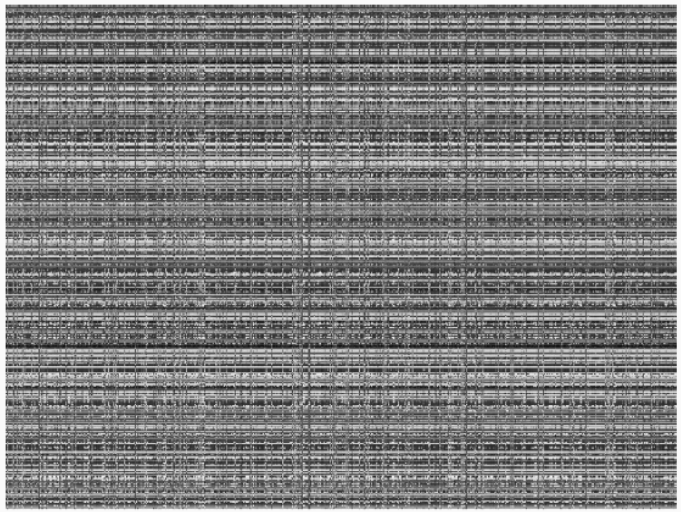
Given input $\mathbf{x}$, we can completely recover $f(\mathbf{x})$ from $\mathbf{z} = e(\mathbf{x})$. $\mathbf{z}$ typically has many fewer dimensions than $\mathbf{x}$ and for a typical

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.

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.

AUTOENCODERS FOR IMAGE CORRECTION

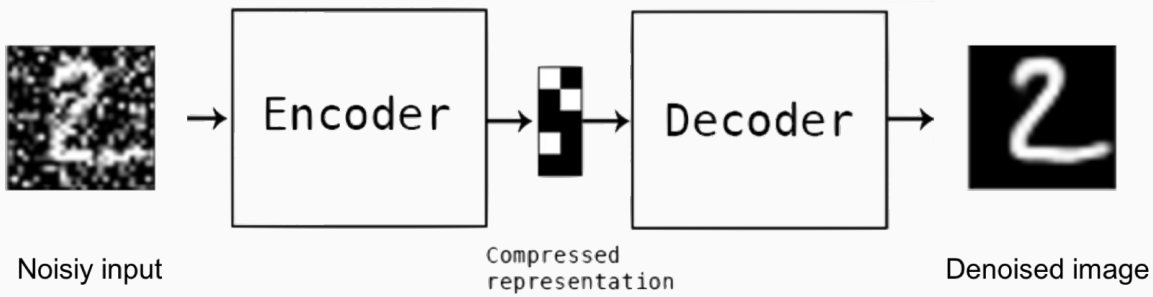


Image denoising

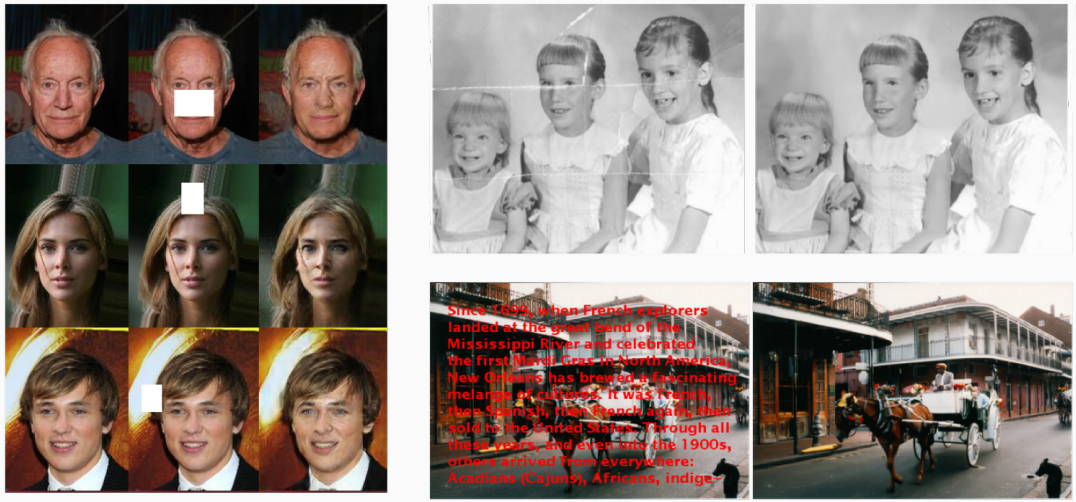
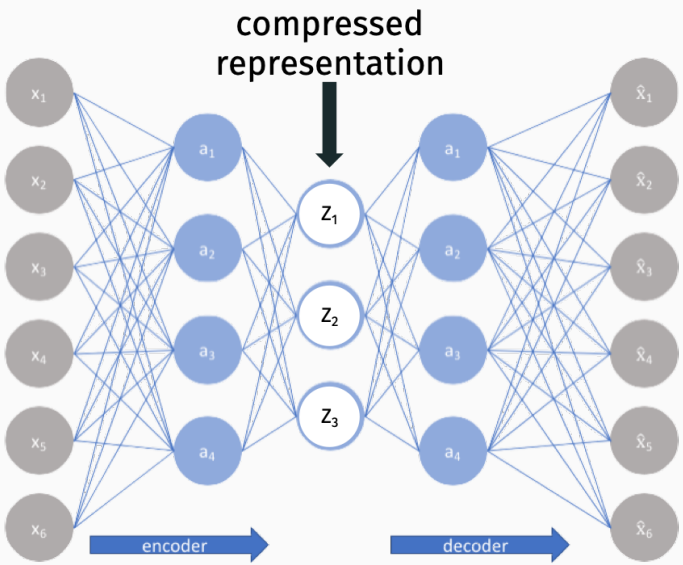


Image inpainting

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

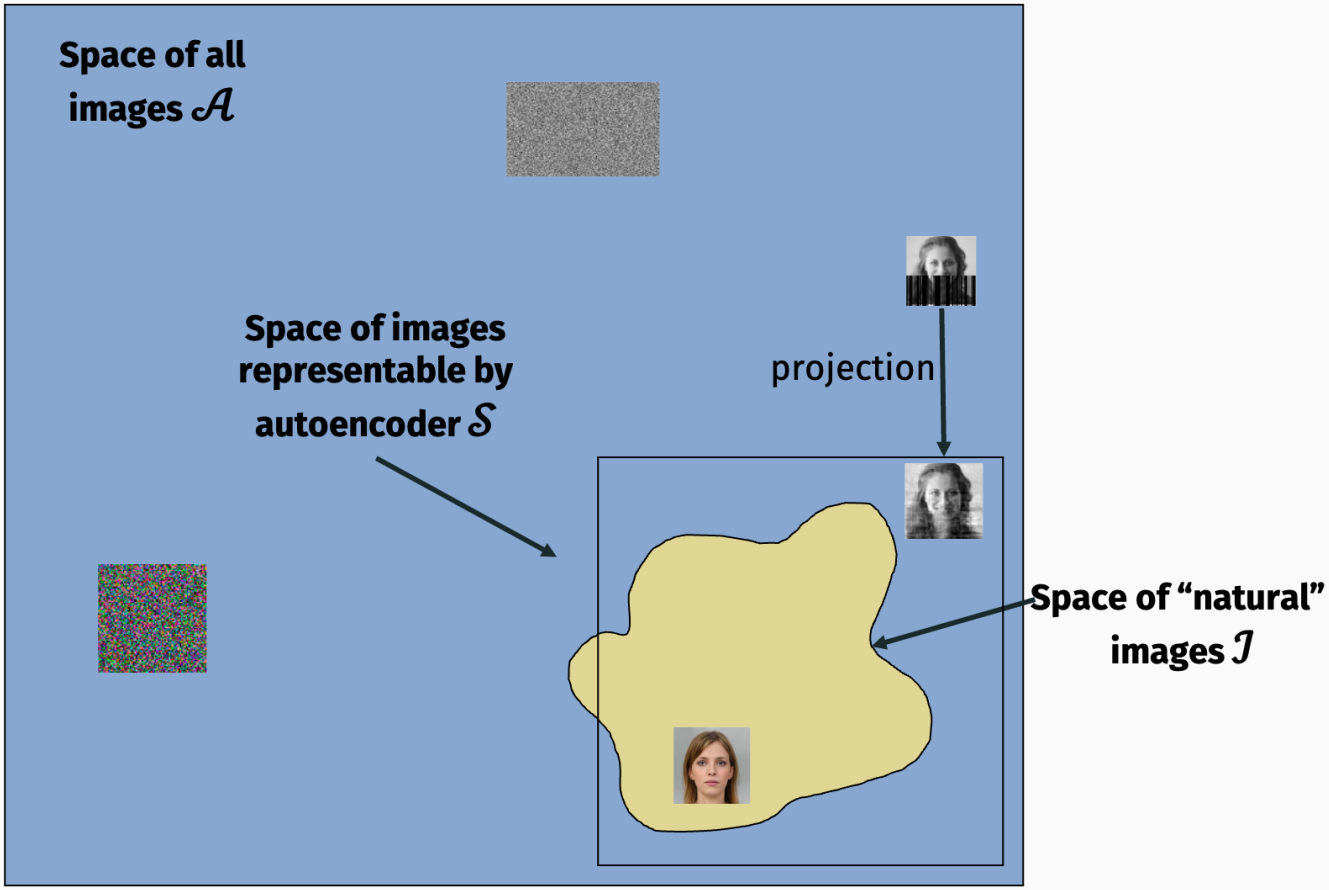
Why does this work?



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

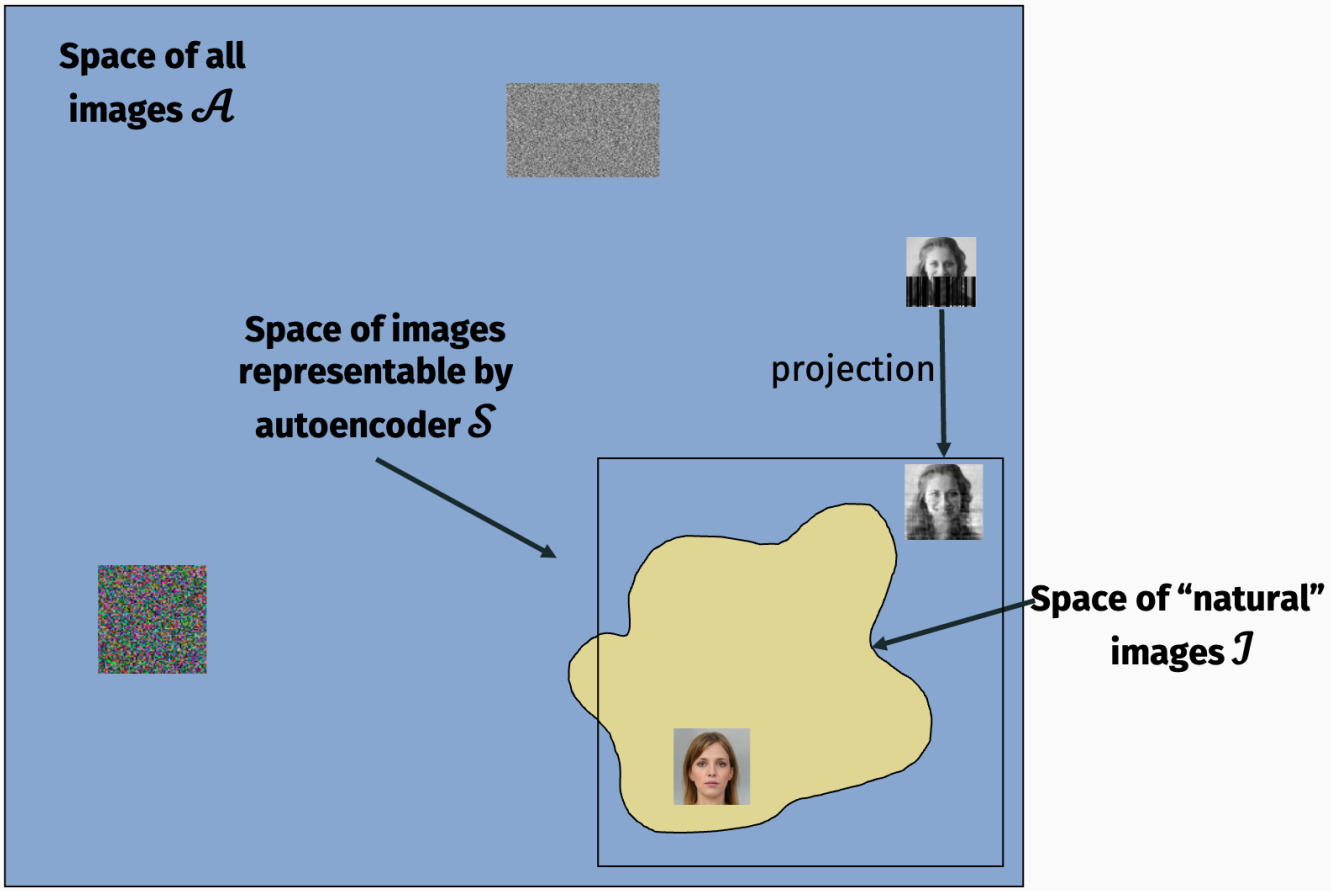
If $\mathbf{z}$ holds k, 8 bit values, how many unique images $\mathbf{w}$ can be output by the autoencoder function f?

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



An accurate autoencoder with a small bottleneck must have a representation space $\mathcal{S}$ that closely approximates $\mathcal{I}$. Both will be much smaller than $\mathcal{A}$.

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

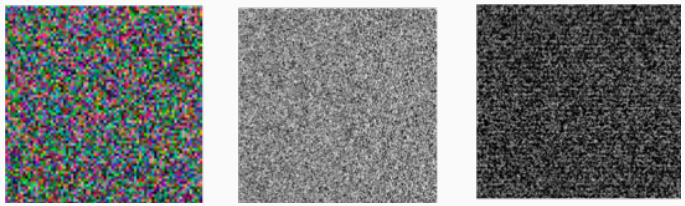


$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

AUTOENCODERS FOR DATA GENERATION

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

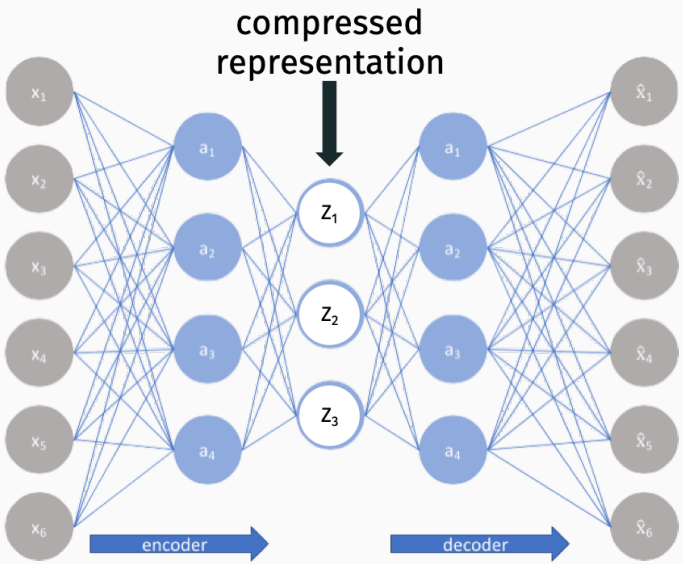
- **Option 1**: Draw each pixel value in $\mathbf{x}$ uniformly at random. Draws a random image from $\mathcal{A}$.



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

PRINCIPAL COMPONENT ANALYSIS

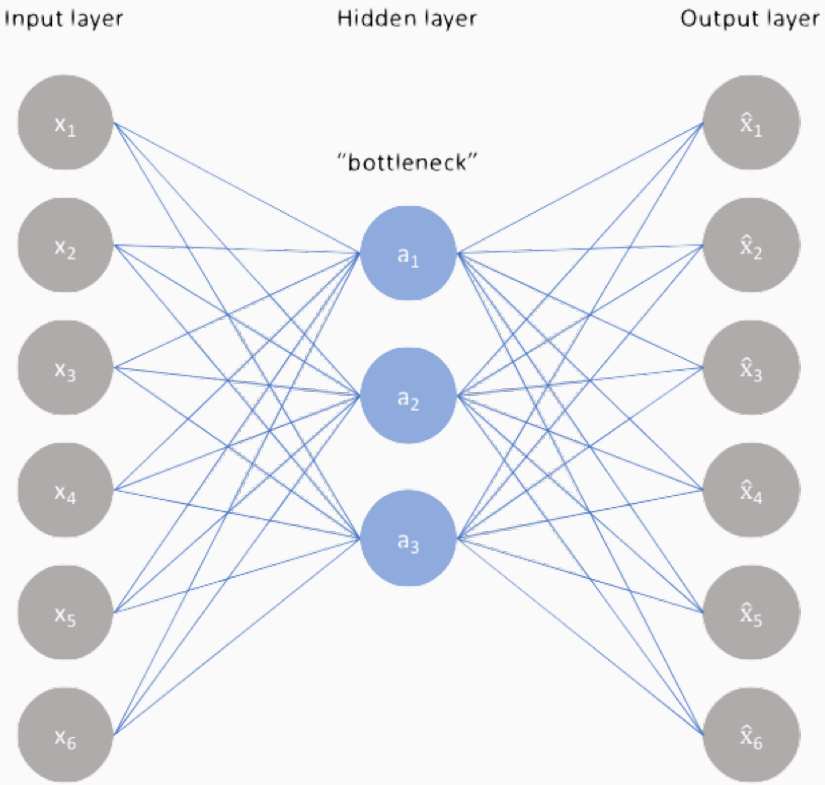
PRINCIPAL COMPONENT ANALYSIS

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 denoising, dimensionality reduction, etc.

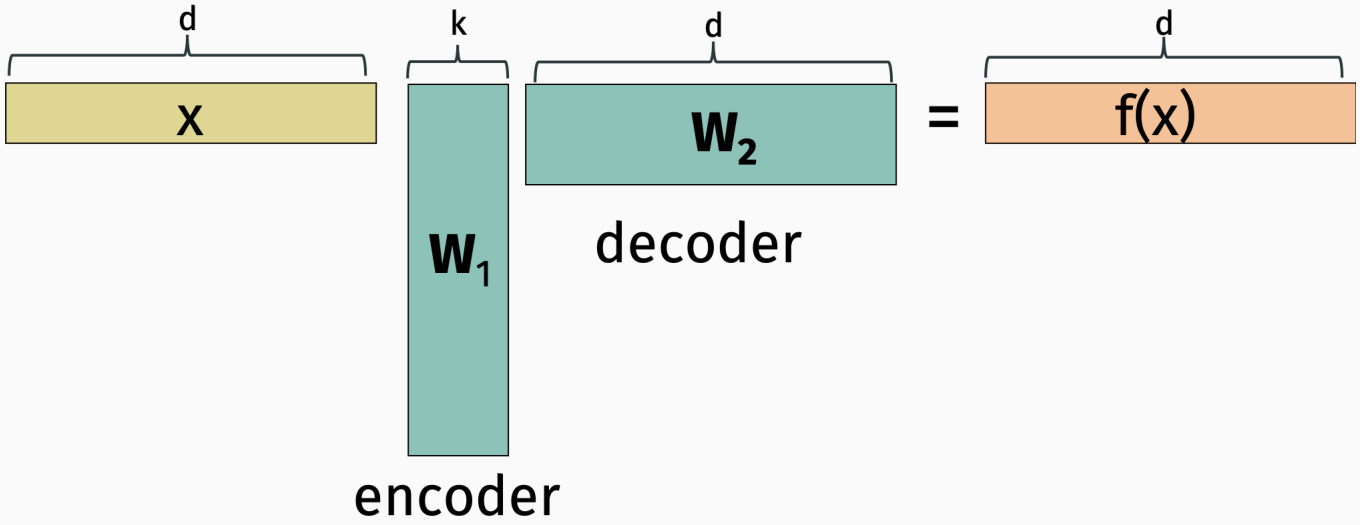
Very important outside machine learning as well.

Consider the simplest possible autoencoder:



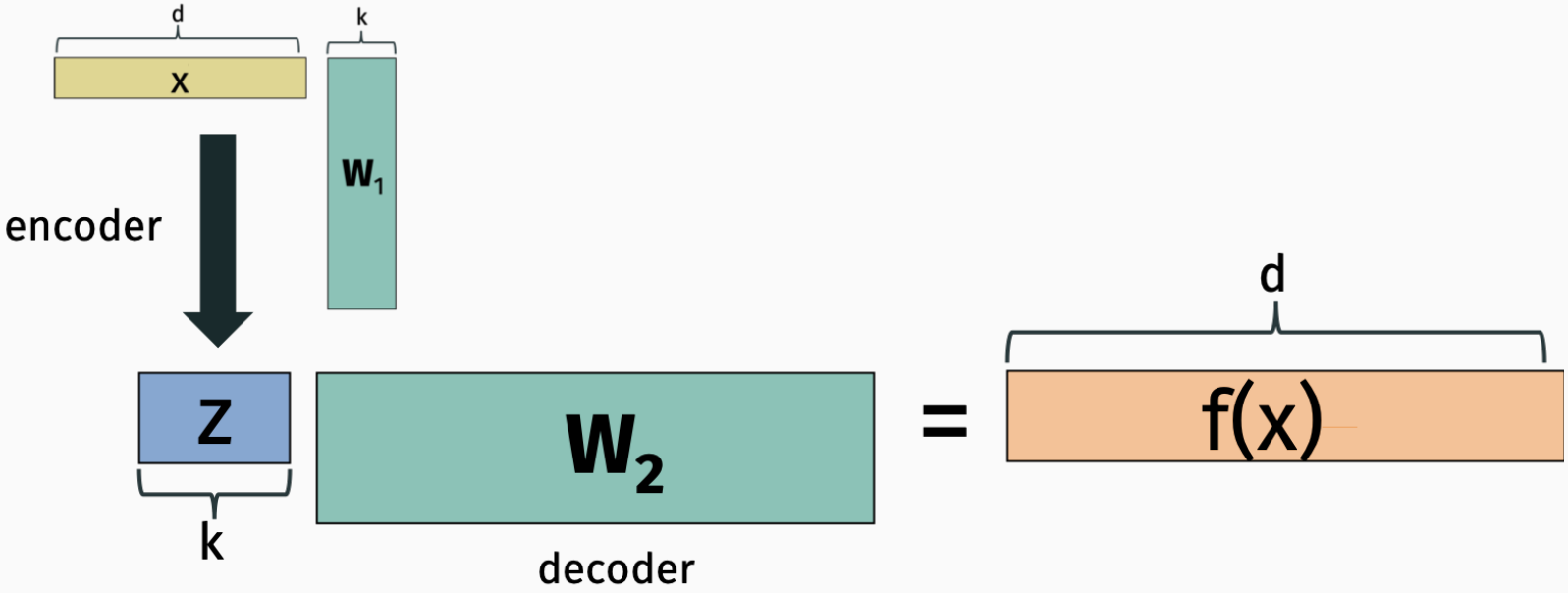
- One hidden layer. No non-linearity. No biases.
- Latent space of dimension k.
- Weight matrices are $\mathbf{W}_1 \in \mathbb{R}^{d \times k}$ and $\mathbf{W}_2 \in \mathbb{R}^{k \times d}$.

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



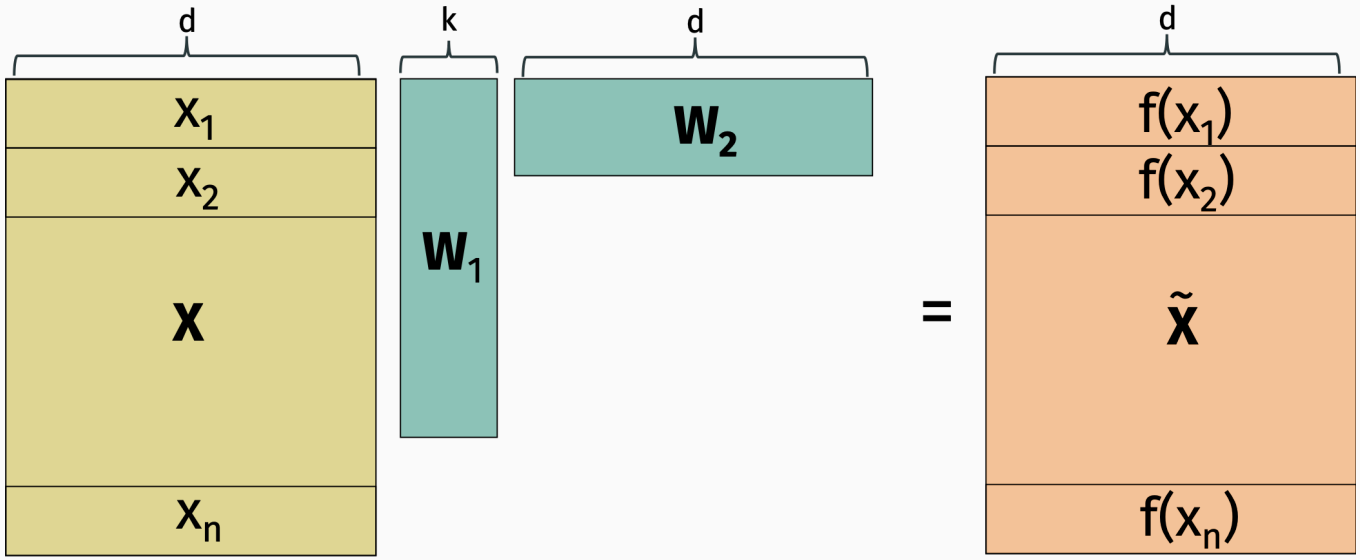
$$f(\mathbf{x})^T = \mathbf{x}^T \mathbf{W}_1 \mathbf{W}_2$$

PRINCIPAL COMPONENT ANALYSIS



Encoder: $z = e(x) = x^T W_1$. Decoder: $d(z) = zW_2$

Given training data set $\mathbf{x}_1, \ldots, \mathbf{x}_n$, let $\mathbf{X}$ denote our data matrix. Let $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$.



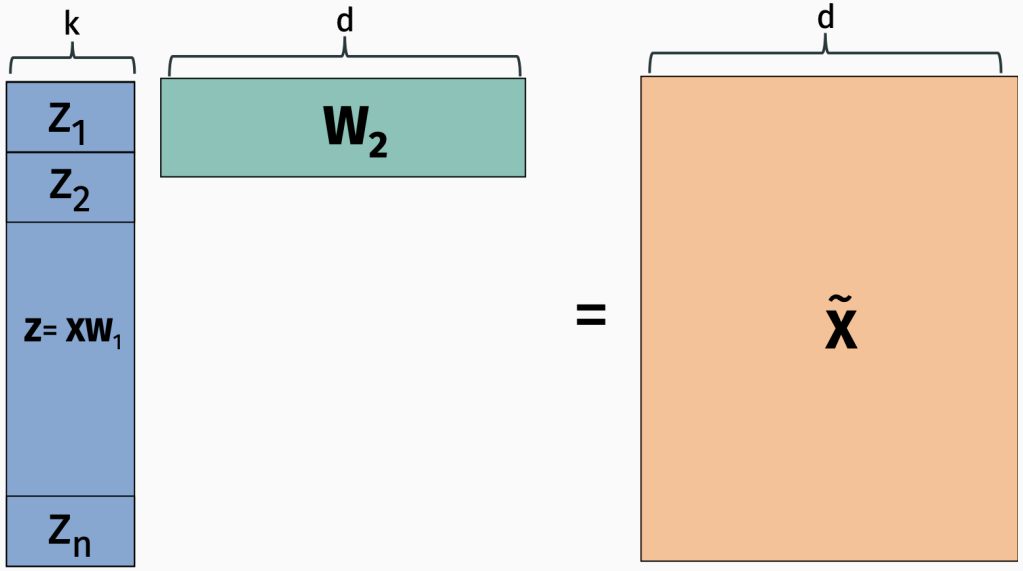
Natural squared autoencoder loss: Minimize $L(\mathbf{X}, \tilde{\mathbf{X}})$ where:

$$\begin{aligned} L(\mathbf{X}, \tilde{\mathbf{X}}) &= \sum_{i=1}^{n} \|\mathbf{x}_i - f(\mathbf{x}_i)\|_2^2 \\ &= \sum_{i=1}^{n} \sum_{j=1}^{d} (\mathbf{x}_i[j] - f(\mathbf{x}_i)[j])^2 \\ &= \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 \end{aligned}$$

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 $k \ll d$.

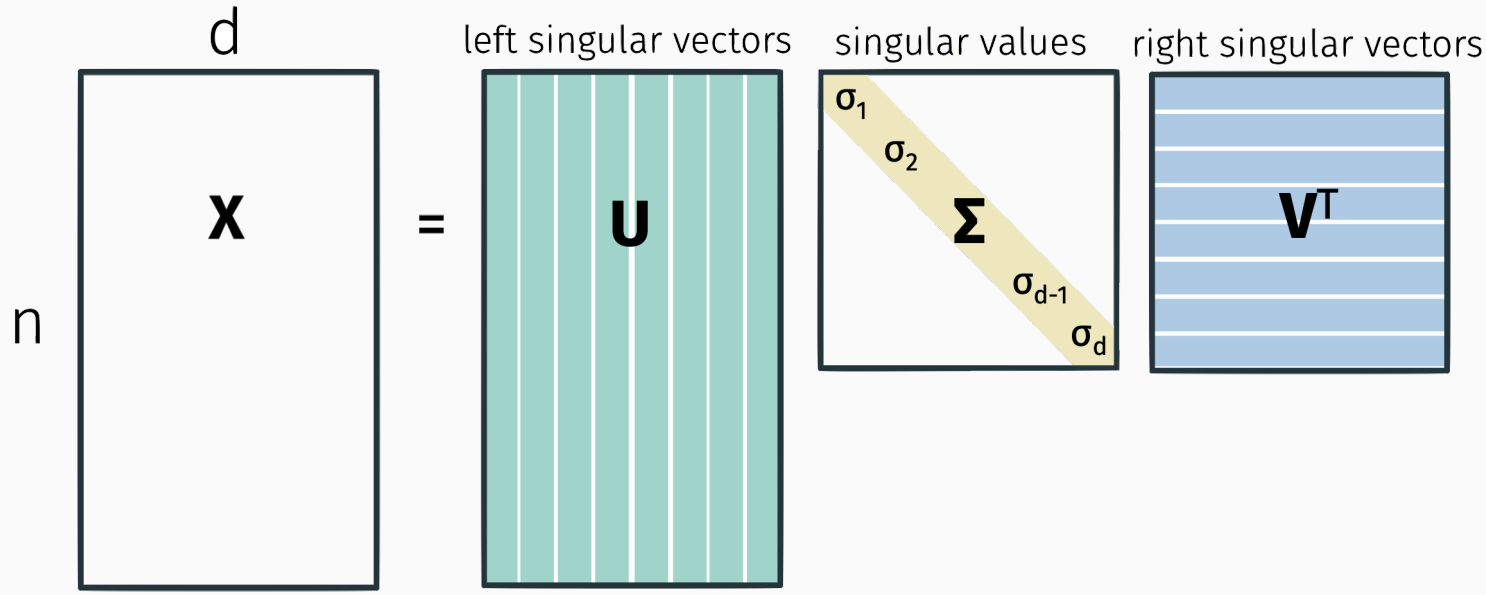
LOW-RANK APPROXIMATION

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:

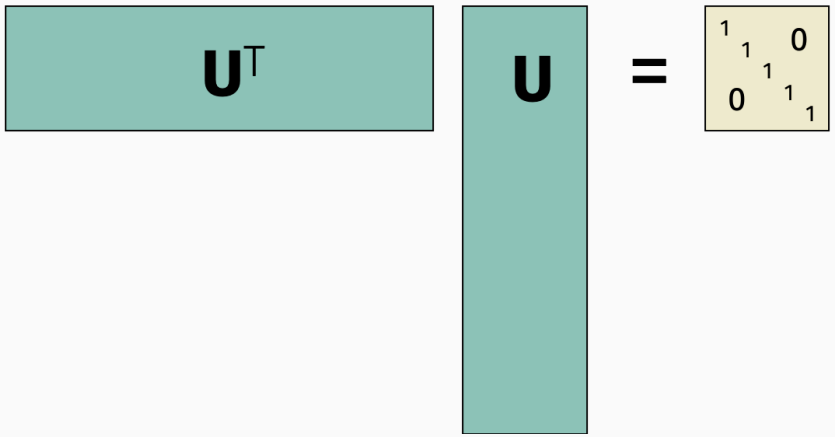


Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices.

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

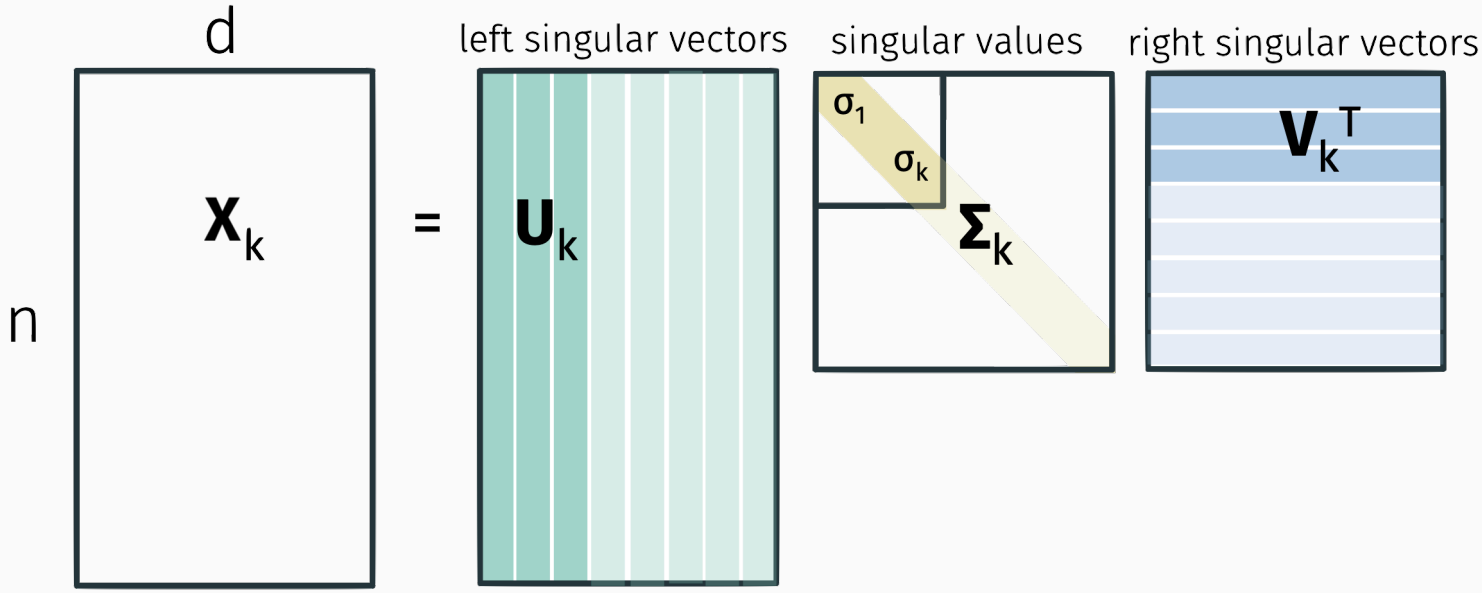
Can be computed in $O(nd^2)$ time (faster with approximation algos).

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 $\mathbf{X}$. Recall that orthogonality means that:



$$\|\mathbf{u}_i\|_2^2 = \qquad \mathbf{u}_i^T \mathbf{u}_j =$$

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



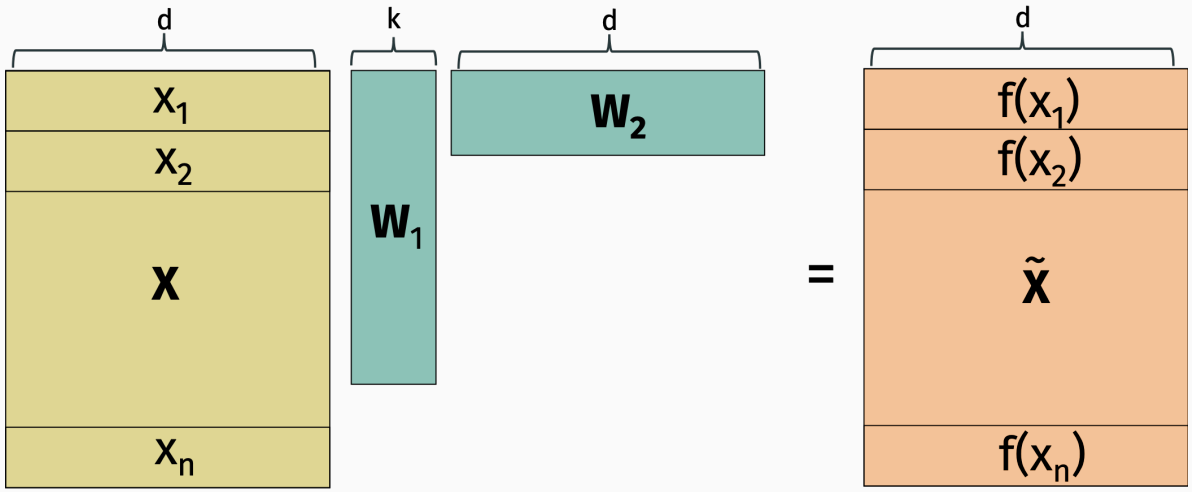
Eckart–Young–Mirsky Theorem: For any $k \leq d$, $\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

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

OPTIMAL LOW-RANK APPROXIMATION

That's great, but not quite in the form we wanted. Optimal rank k approximation is $\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T$. We want an approximation of the form:

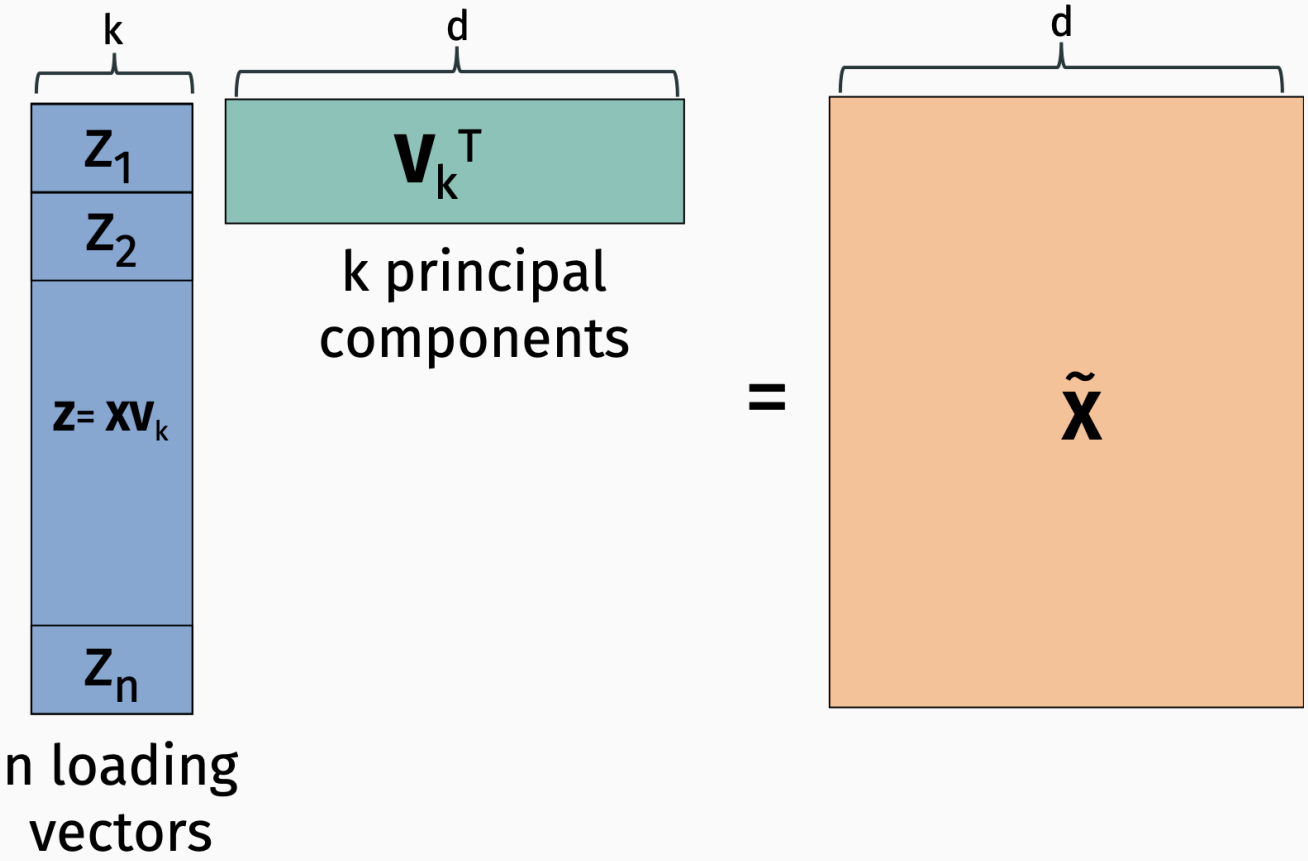
$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$$



Optimal low-rank approximation

Claim: $X_k = U_k \mathbf{\Sigma}_k V_k^T = X V_k V_k^T$. I.e., can choose $W_1 = V_k$, $W_2 = V_k^T$.

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_k, S_k, V_k `= 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.

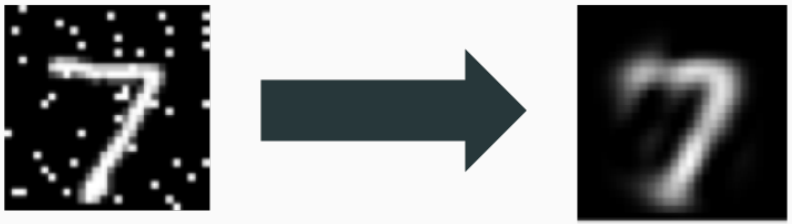
- $\mathbf{U}$'s columns (the left singular vectors) are the orthonormal eigenvectors of $\mathbf{X}\mathbf{X}^T$.
- $\mathbf{V}$'s columns (the right singular vectors) are the orthonormal eigenvectors of $\mathbf{X}^T\mathbf{X}$.
- $\sigma_i^2 = \lambda_i(\mathbf{X}\mathbf{X}^T) = \lambda_i(\mathbf{X}^T\mathbf{X})$

Exercise: Verify this directly. This means you can use any 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.

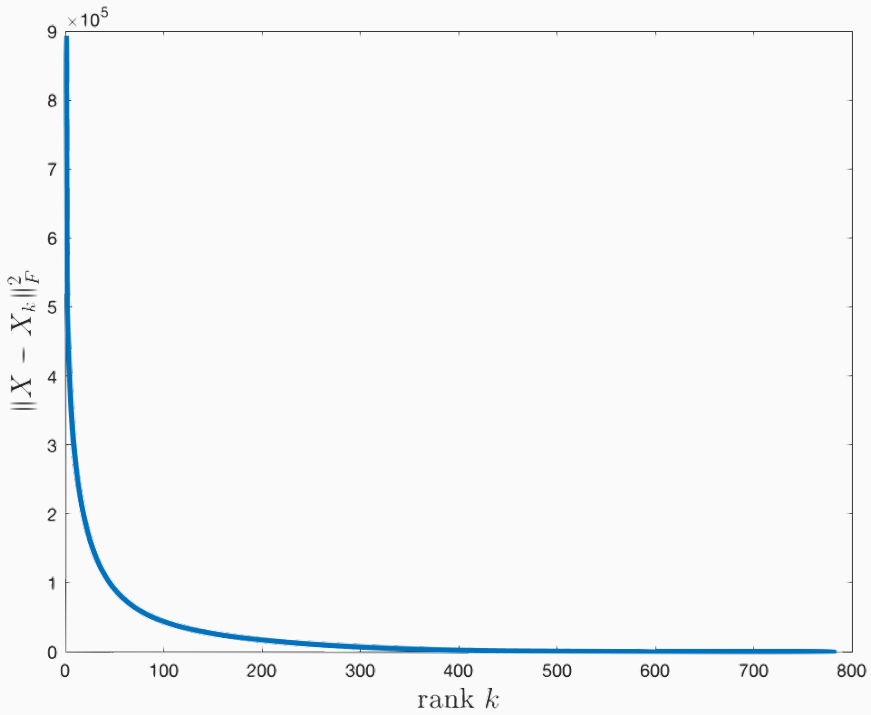
original data

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 X's singular values. The singular values are often called the **spectrum** of X.

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



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.

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

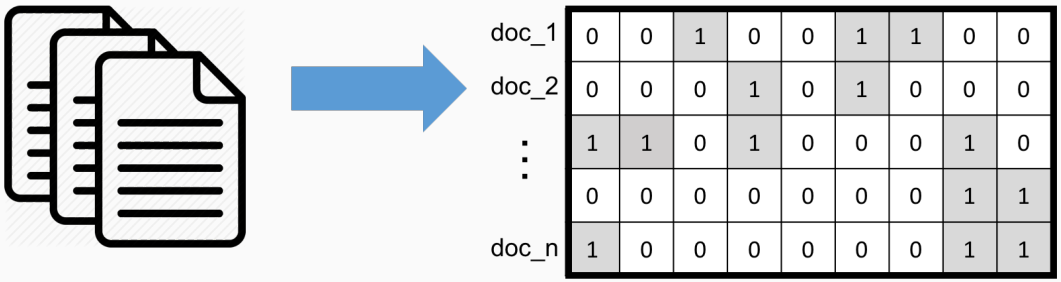
$$\begin{aligned}\text{cumulative GPA} \approx &\frac{1}{4} \cdot \text{year 1 GPA} + \frac{1}{4} \cdot \text{year 2 GPA} \\ &+ \frac{1}{4} \cdot \text{year 3 GPA} + \frac{1}{4} \cdot \text{year 4 GPA}.\end{aligned}$$

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

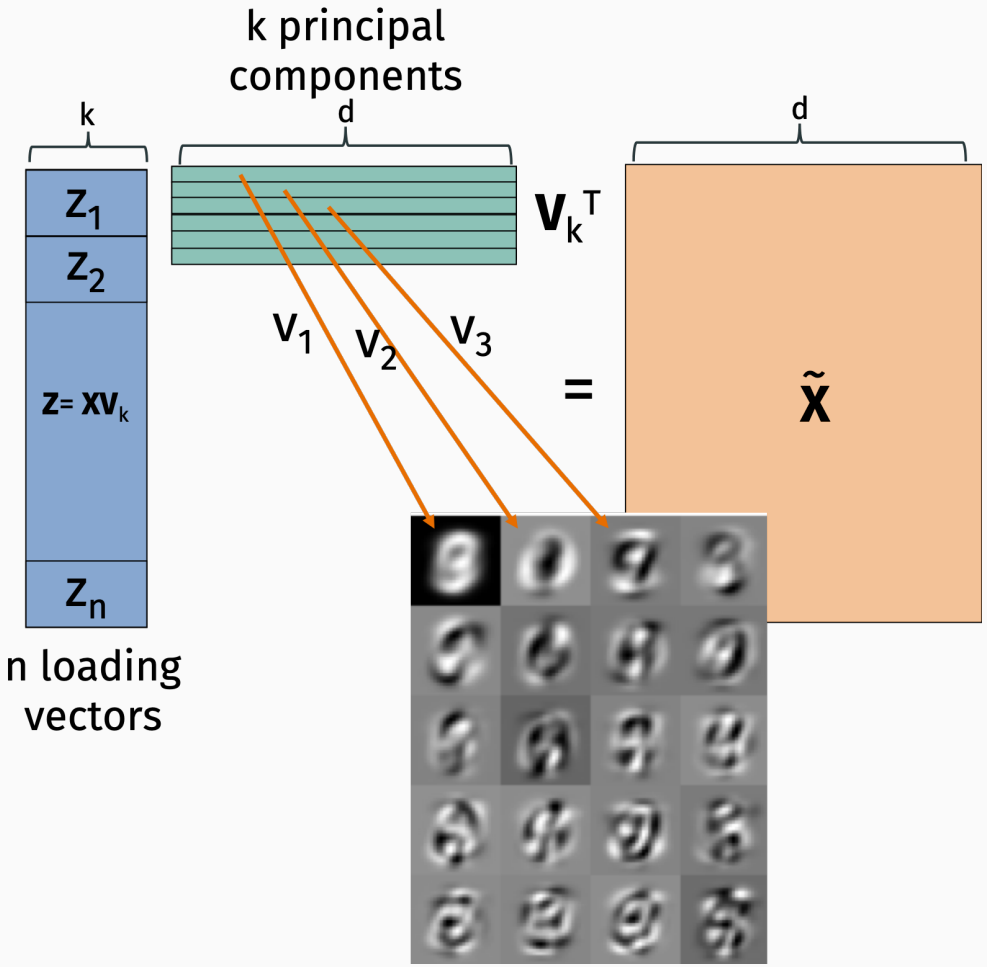
2. "Term-document" matrix with bag-of-words data:



PRINCIPAL COMPONENTS

What do principal components and loading vectors look like?

MNIST **principal components**:

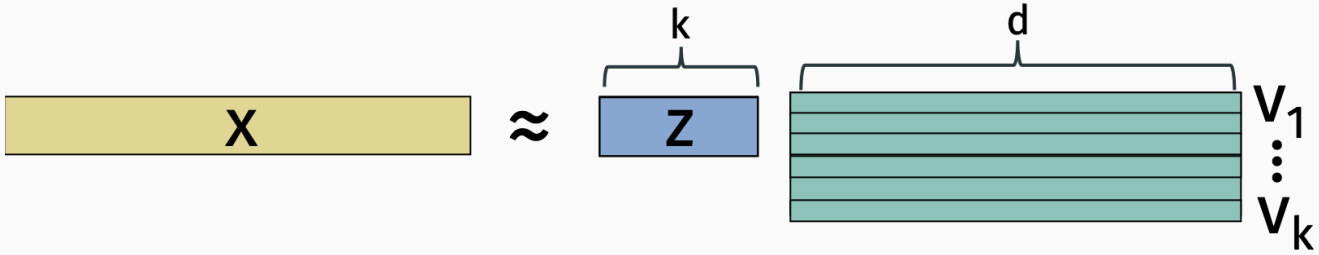


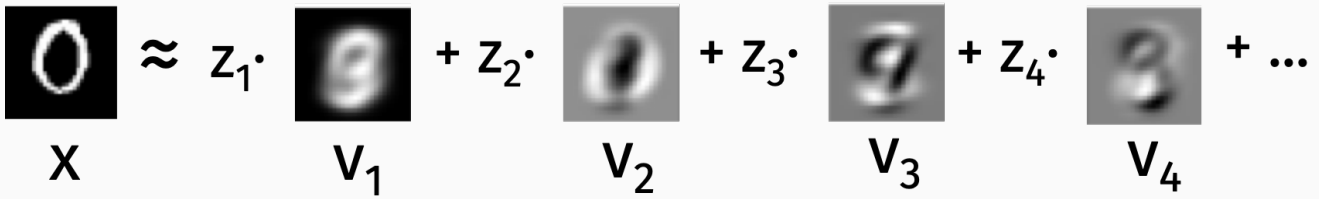
Often principal components are difficult to interpret.

LOADING VECTORS

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 $\mathbf{x}$.

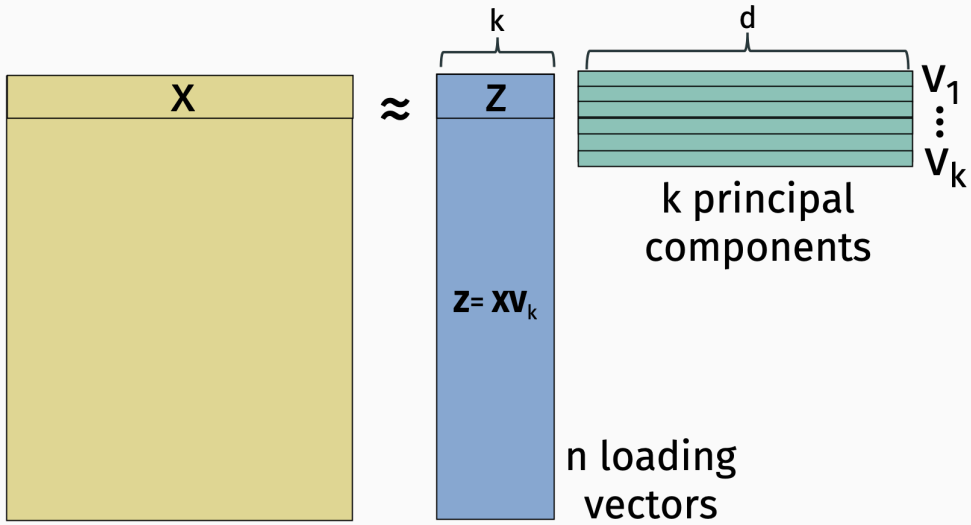




Provide a short "finger print" for any image $\mathbf{x}$ which can be used to reconstruct that image.

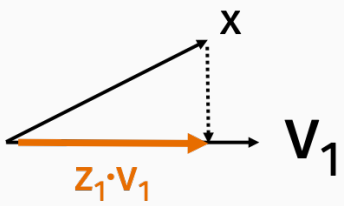
LOADING VECTORS: SIMILARITY VIEW

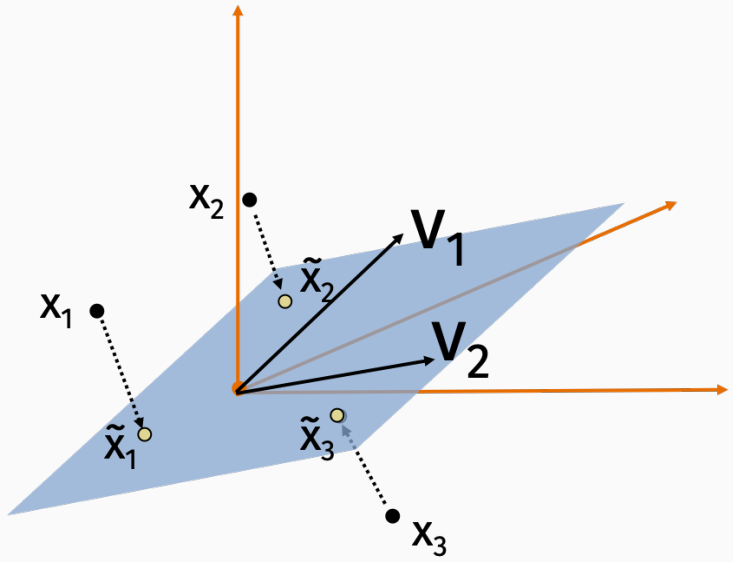
For any $\mathbf{x}$ with loading vector $\mathbf{z}$, the i^{th} entry z_i is the inner product similarity between $\mathbf{x}$ and the i^{th} principal component, $\mathbf{v}_i$.



LOADING VECTORS: PROJECTION VIEW

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



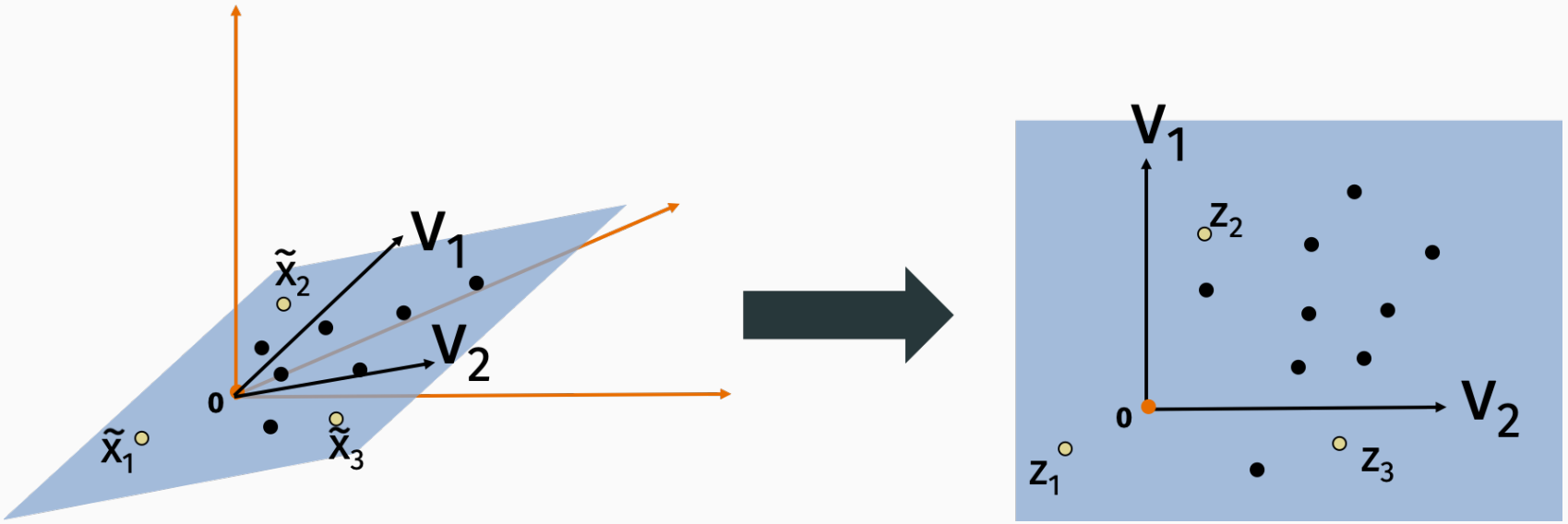


Since $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are orthonormal, this operation is a projection onto first k principal components.

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:



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:

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

Conclusion: If our data had a good low rank approximation, i.e. $\|\tilde{\mathbf{x}}_i\|_2^2 \approx \|\mathbf{x}_i\|_2^2, \langle \tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j \rangle \approx \langle \mathbf{x}_i, \mathbf{x}_j \rangle$, and $\|\tilde{\mathbf{x}}_i - \tilde{\mathbf{x}}_j\|_2^2 \approx \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$, we expect that:

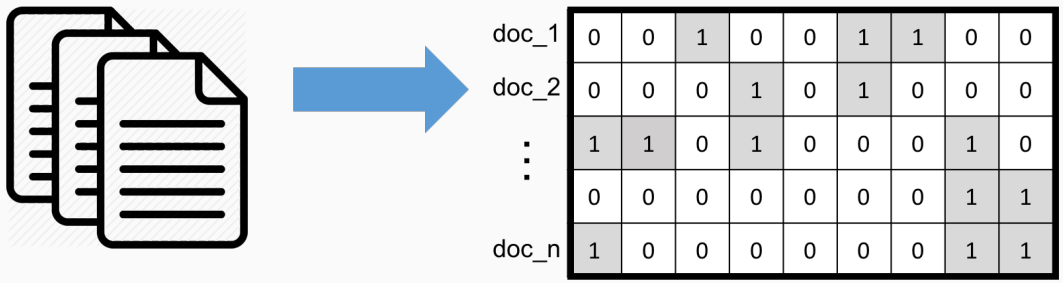
$$\|\mathbf{x}_i\|_2^2 \approx \|\mathbf{z}_i\|_2^2$$
$$\langle \mathbf{x}_i, \mathbf{x}_j \rangle \approx \langle \mathbf{z}_i, \mathbf{z}_j \rangle$$
$$\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \approx \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

Useful in obtaining short "finger prints" for complex data.

Note: this is not true of most autoencoders, but unique to PCA. Typically compressions themselves cannot be directly used to approximate distance, similarity,. etc.

TERM DOCUMENT MATRIX

Word-document matrices tend to be low rank.

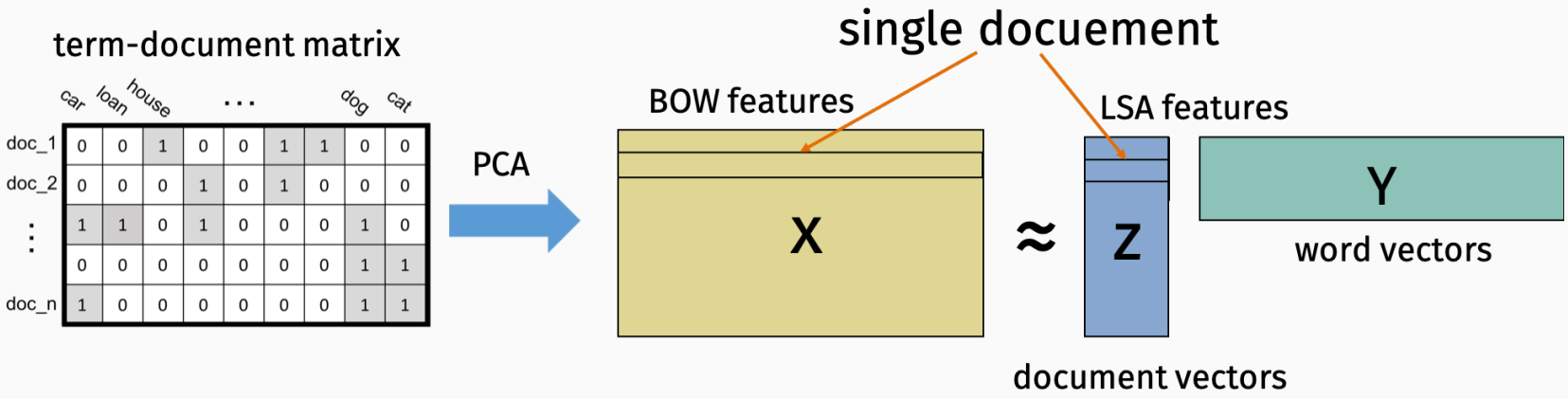


Documents tend to fall into a relatively small number of different categories, which use similar sets of words:

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

LATENT SEMANTIC ANALYSIS

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).



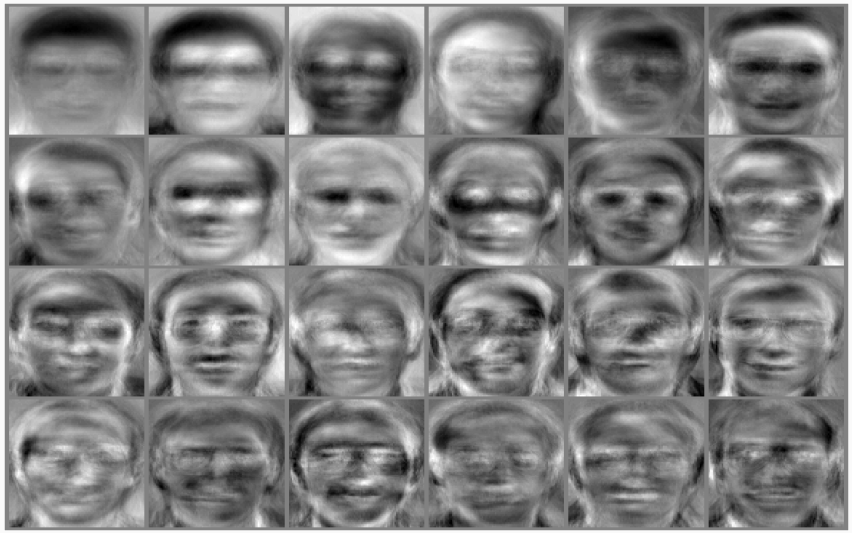
Each column of **Z** corresponds to a latent "category" or "topic". Corresponding row in **Y** corresponds to the "frequency" with which different words appear in documents on that topic.

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.

EIGENFACES

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

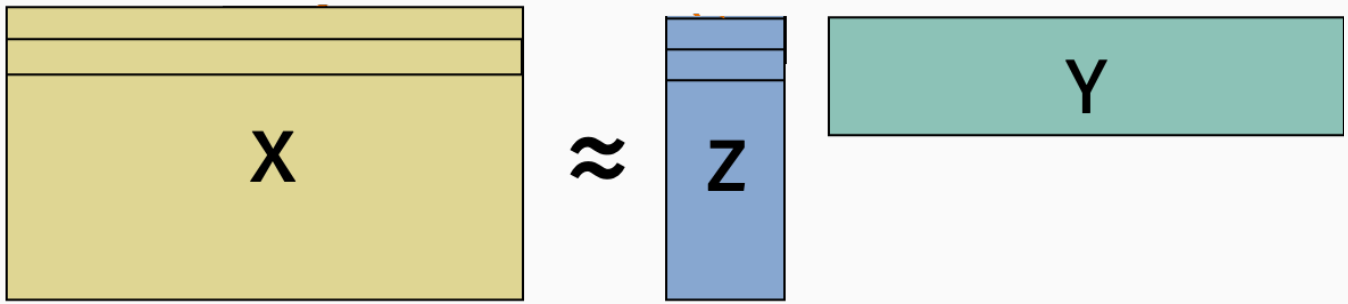


Each image above is one of the principal components of a dataset containing images of faces.

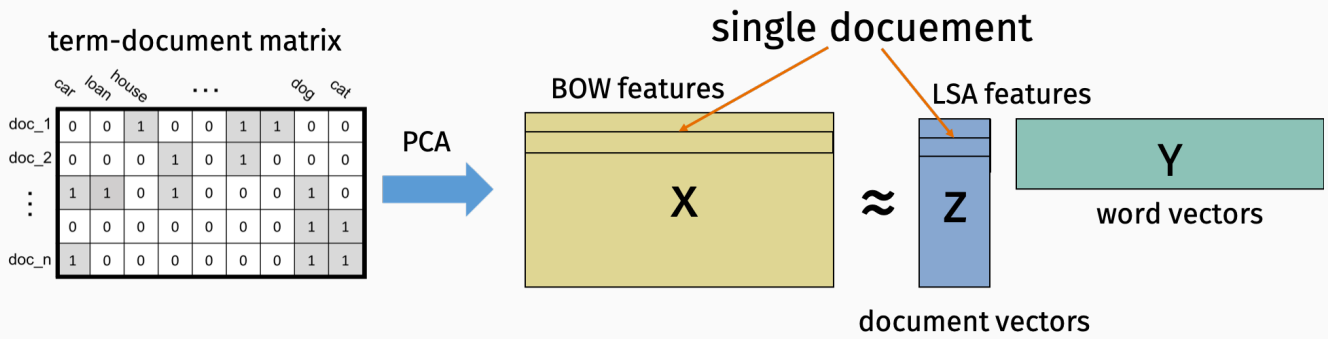
SEMANTIC EMBEDDINGS

Document embeddings are clearly useful. What about the word embeddings? It turns out these are super useful as well!

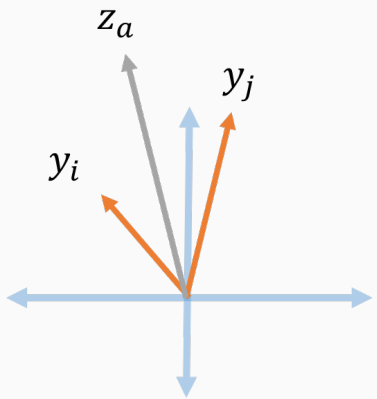
Reminder: The i, j entry of $\tilde{\mathbf{X}}$ equals $\langle \mathbf{z}_i, \mathbf{y}_j \rangle$.



WORD EMBEDDINGS

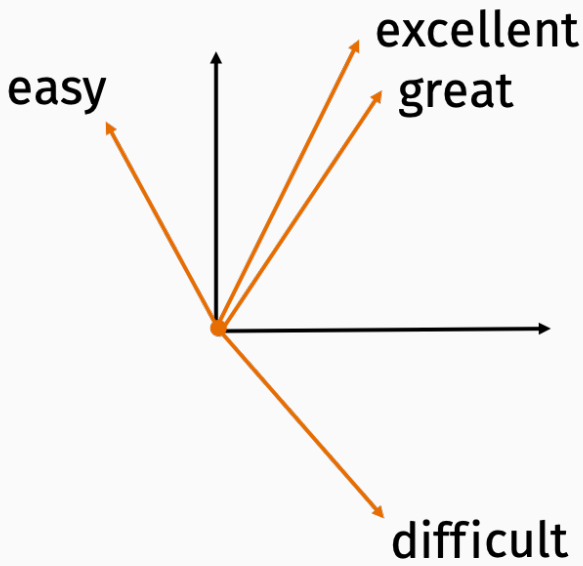


- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$ when doc_a contains $word_i$.
- If $word_i$ and $word_j$ both appear in doc_a, then $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$, so we expect $\langle \mathbf{y}_j, \mathbf{y}_j \rangle$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



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

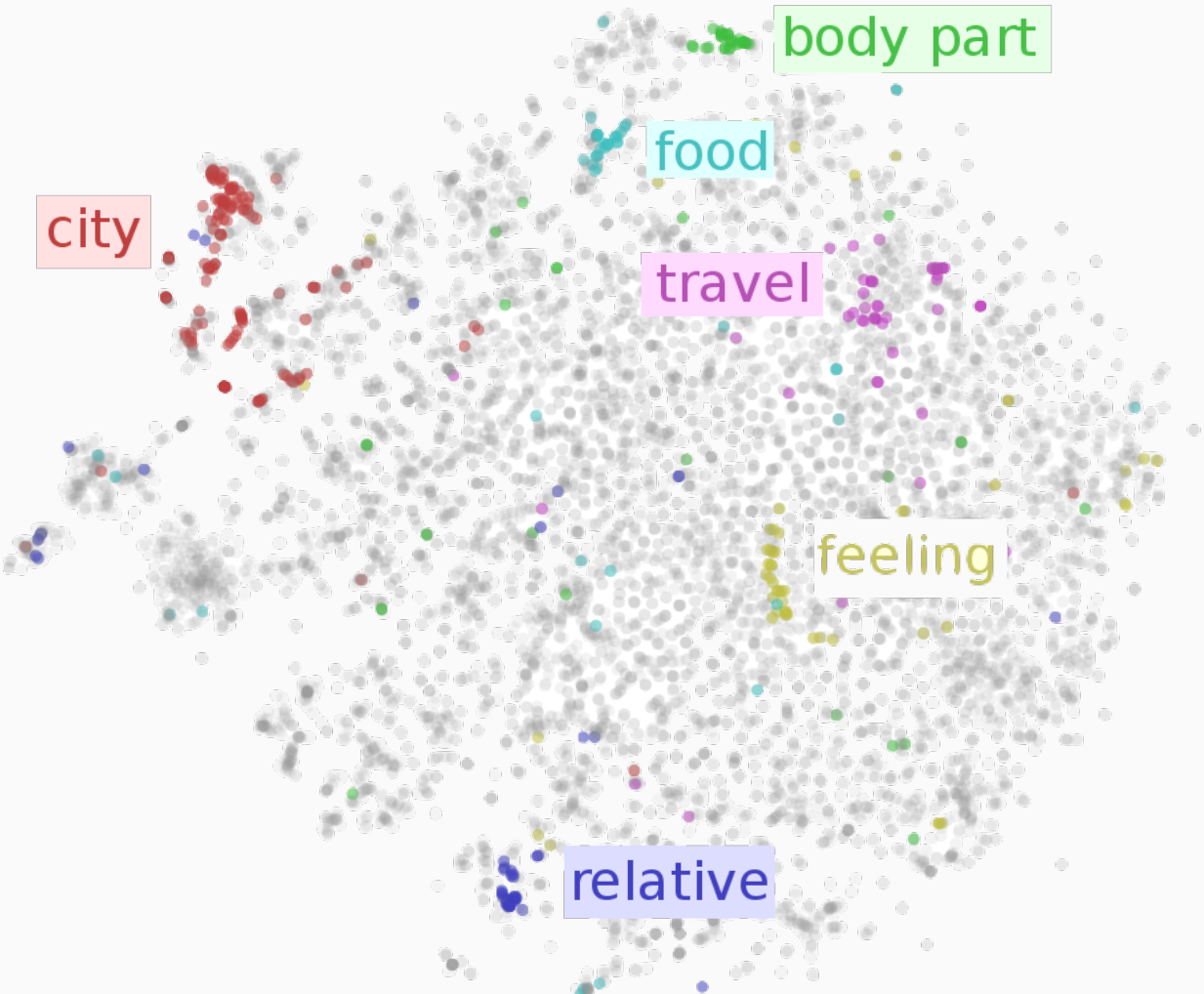
Result: Map words to numerical vectors in a semantically meaningful way. Similar words map to similar vectors. Dissimilar words to dissimilar vectors.



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.

For similar words, $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ should be large. I.e. $\mathbf{y}_i$ and $\mathbf{y}_j$ point in the same direction.



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.*

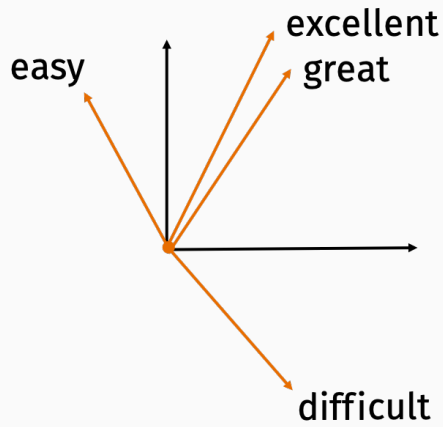
[, , , , , , ,]

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.*

[, , , , , , ,]

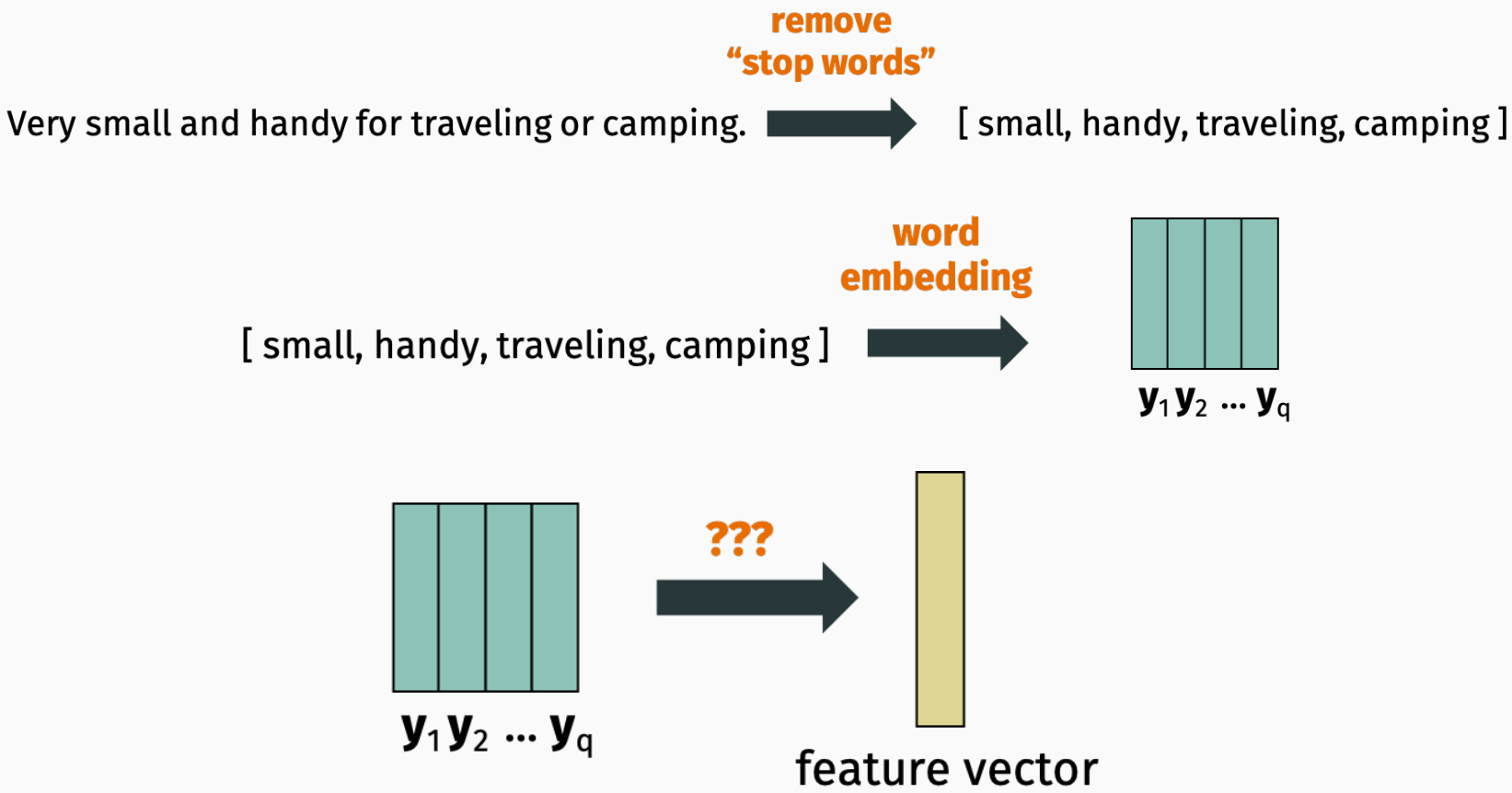
Bag-of-words approach typically only works for large data sets.

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



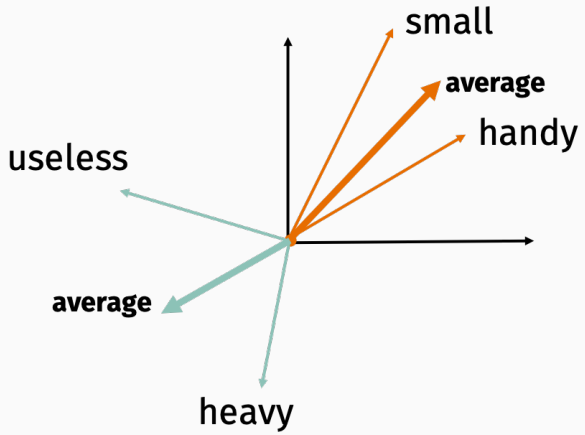
This can be addressed by first mapping words to semantically meaningful vectors. That mapping can be trained using a much large corpus of text than the data set you are working with (e.g. Wikipedia, Twitter, news data sets).

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

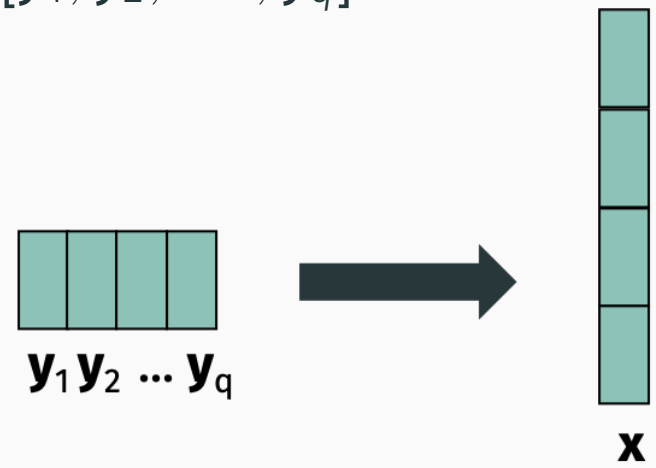


A few simple options:

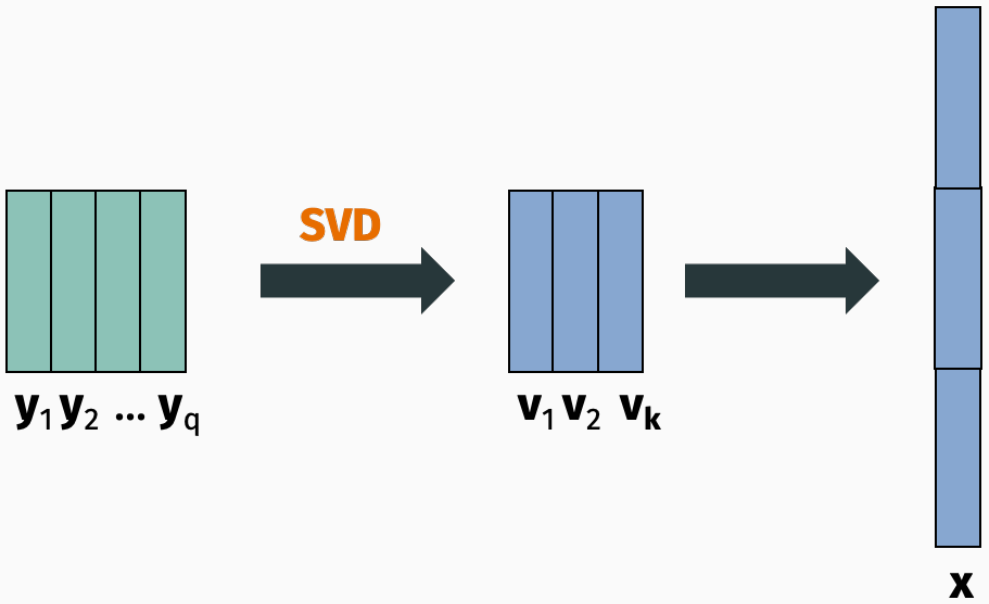
Feature vector $\mathbf{x} = \frac{1}{q} \sum_{i=1}^{q} \mathbf{y}_q$.



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

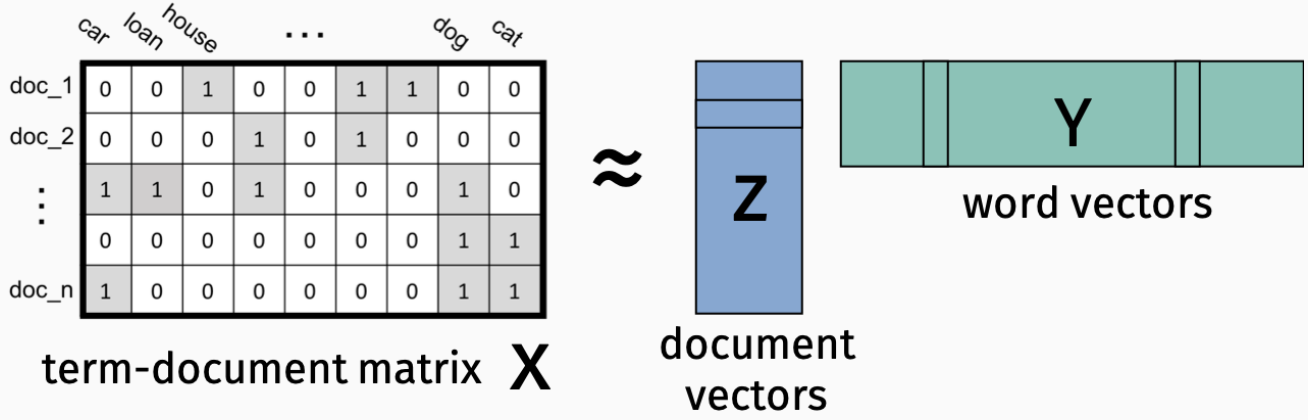


To avoid issues with inconsistent sentence length, word ordering, etc., can concatenate a fixed number of top principal components of the matrix of word vectors:



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

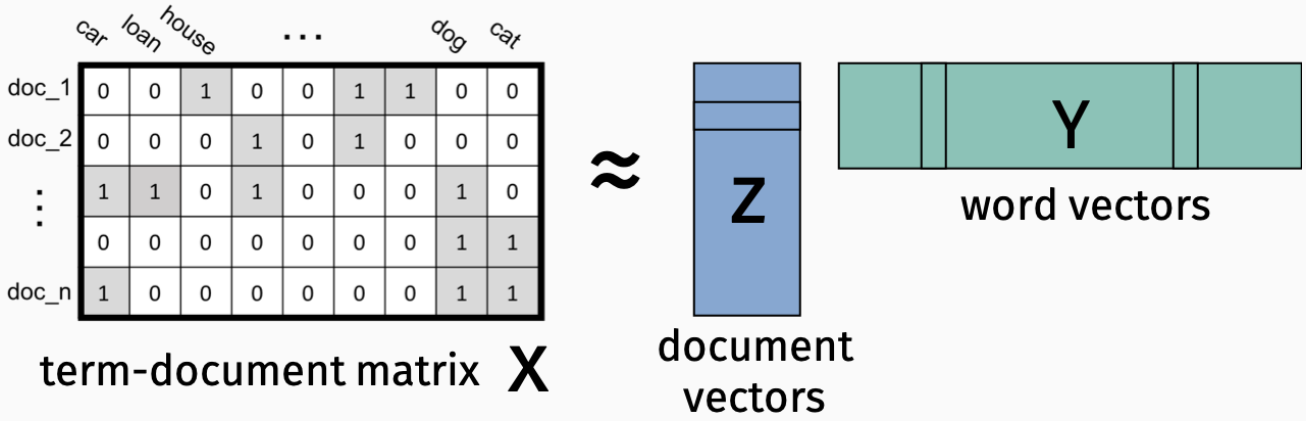
Another view on word embeddings from LSA:



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

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

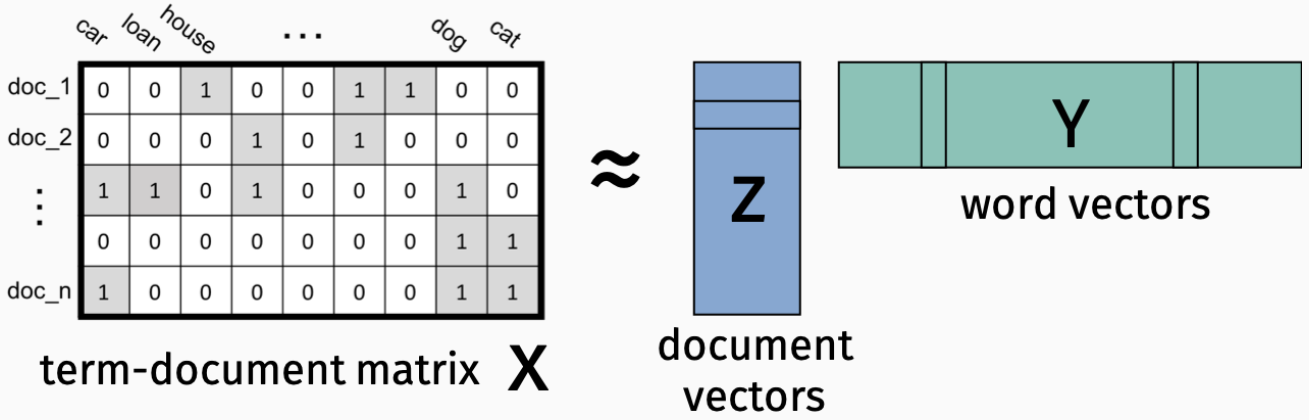
Another view on word embeddings from LSA:



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

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

WORD EMBEDDINGS



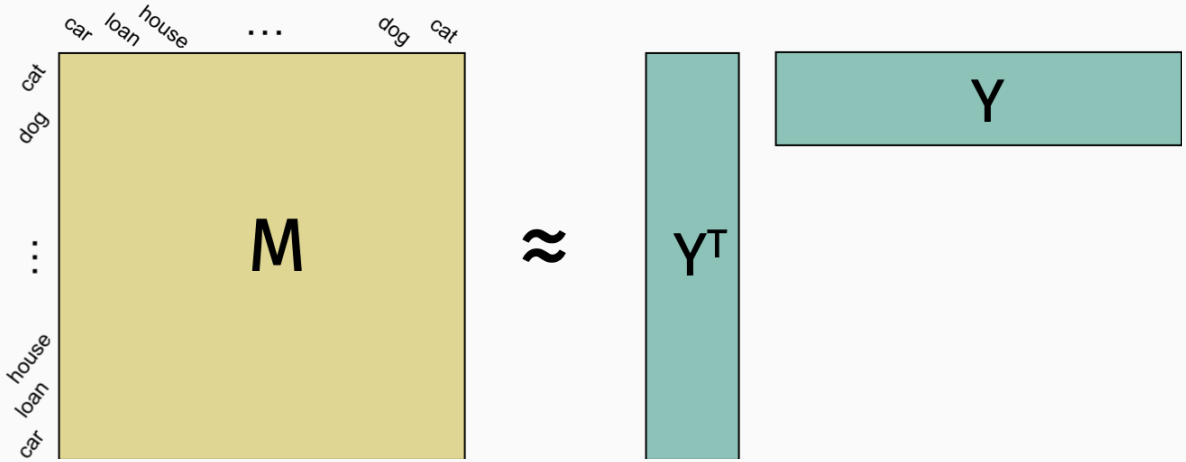
What does the i, j entry of $X^T X$ reprent?

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

We expect that $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ will be larger for more similar words.



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.

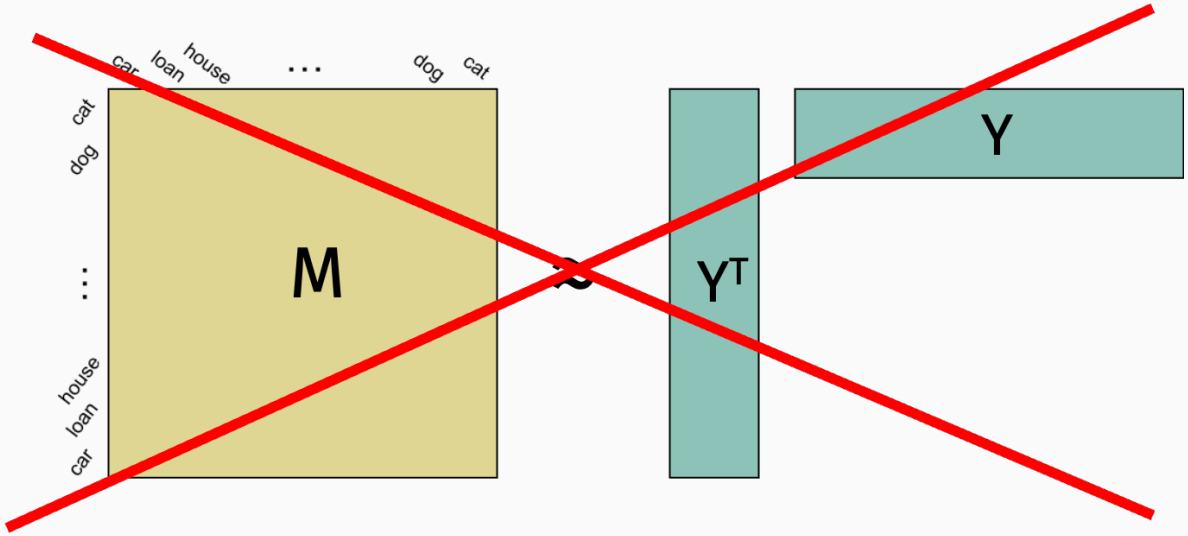
The girl walks to her dog to the park.
It can take a long time to park your car in NYC.
The dog park is always crowded on Saturdays.

	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": $\log \frac{p(i,j)}{p(i)p(j)}$.

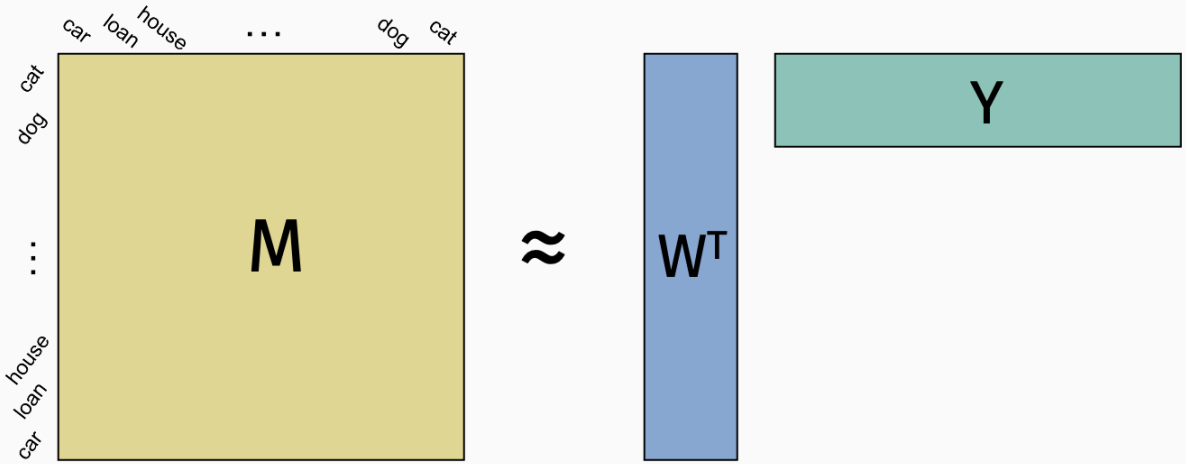
CAVEAT ABOUT FACTORIZATION



SVD will not return a symmetric factorization in general. In fact, if $\mathbf{M}$ is not positive semidefinite[2] then the optimal low-rank approximation does not have this form.

[2] 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 $[\mathbf{w}_i, \mathbf{y}_i]$

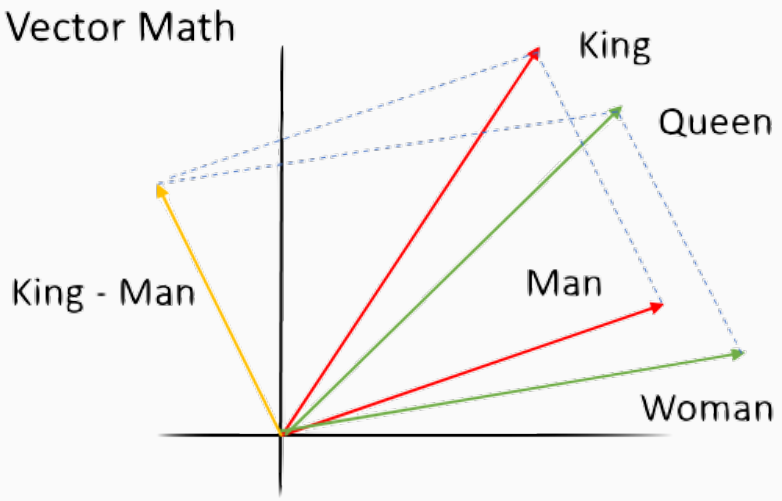
EASIEST WAY TO USE WORD EMBEDDINGS

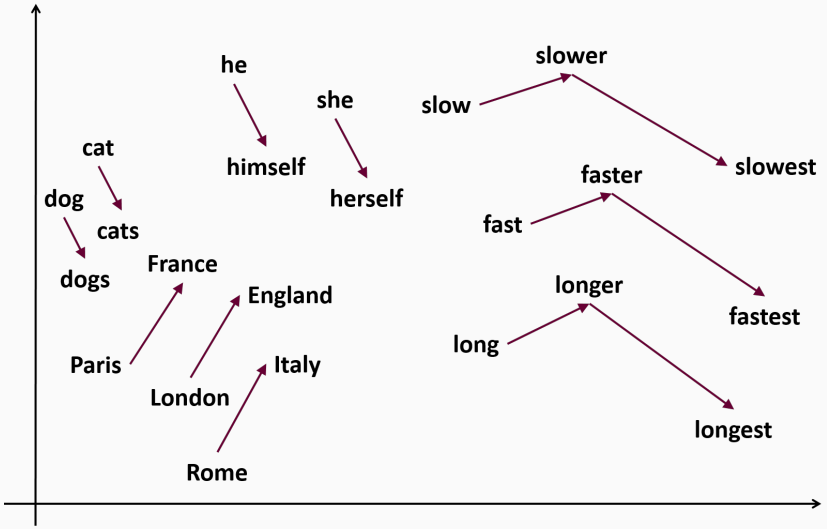
Lots of pre-trained word vectors are available online:

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

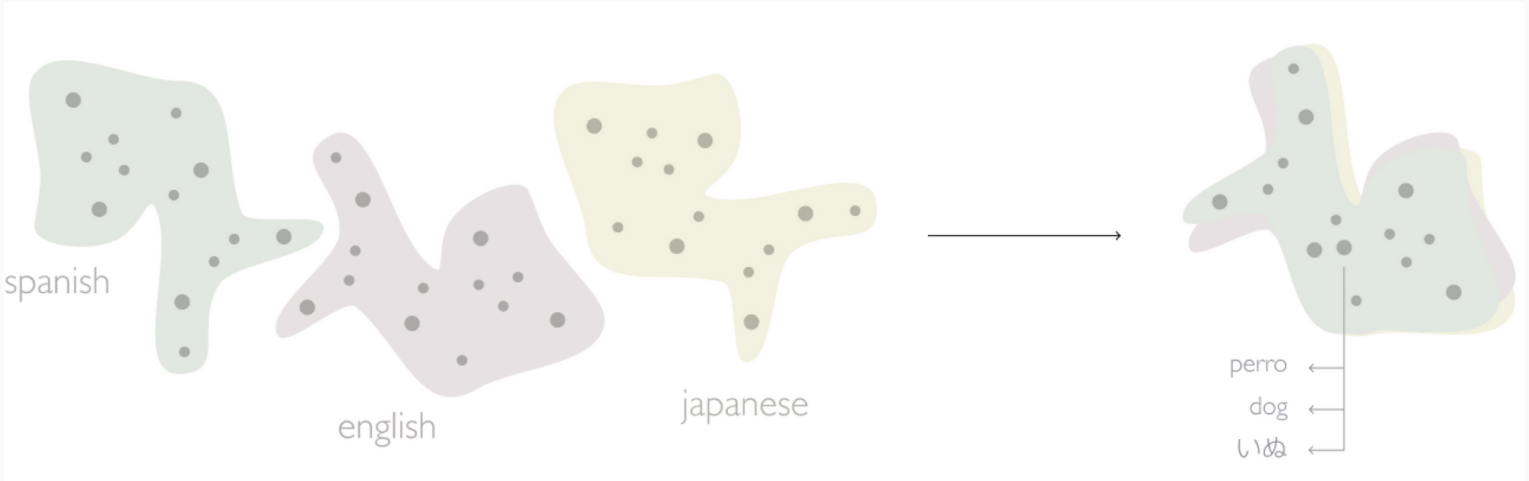
WORD EMBEDDINGS MATH

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





FORWARD LOOKING APPLICATION: UNSUPERVISED TRANSLATION



- Train word-embeddings for languages separately. Obtain lowish dimensional point clouds of words.
- Perform rotation/alignment to match up these point clouds.
- Equivalent words should land on top of each other.

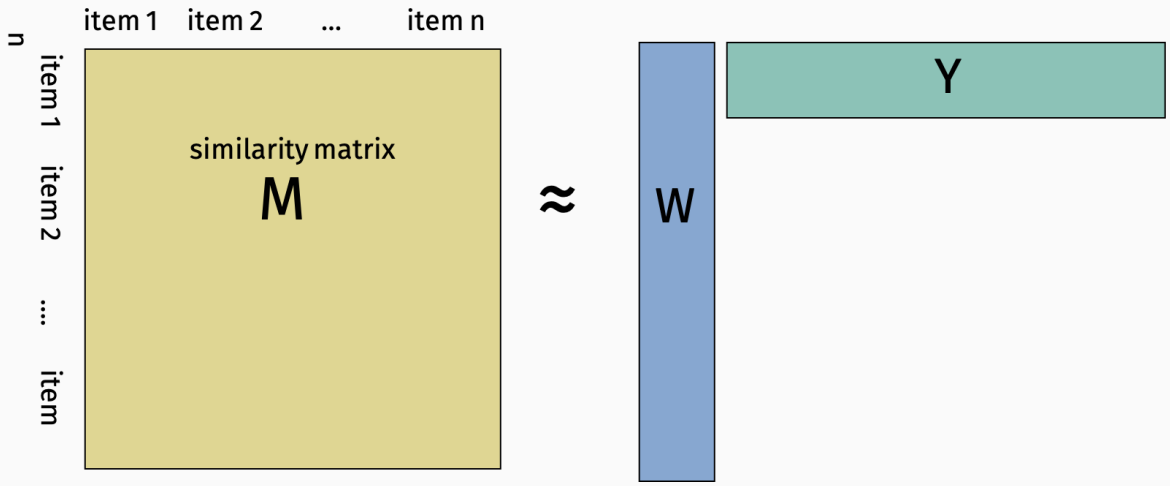
No needs for labeled training data like translated pairs of sentences!

FORWARD LOOKING APPLICATION: UNSUPERVISED TRANSLATION

Why not monkey or whale language?

Earth Species Project (www.earthspecies.org), CETI Project (www.projectceti.org)

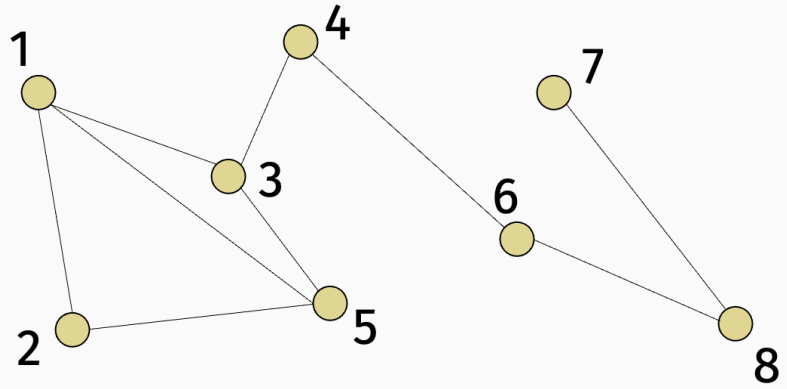
The same approach used for word embeddings can be used to obtain meaningful numerical features for any other data where there is a natural notion of similarity.



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.

NODE EMBEDDINGS

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

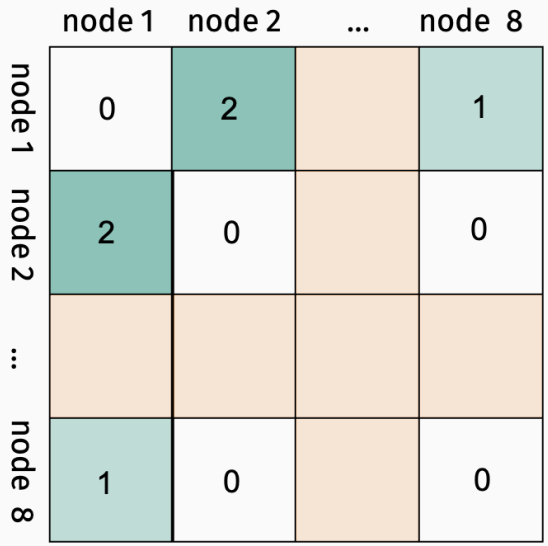


1, 3, 4, 4, 5, 2, 1, 2, 5
6, 8, 6, 4, 3, 1, 5, 3, 4
7, 8, 6, 8, 7, 8, 6, 8, 6
⋮
4, 6, 8, 6, 4, 3, 1, 2, 5

NODE EMBEDDINGS

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

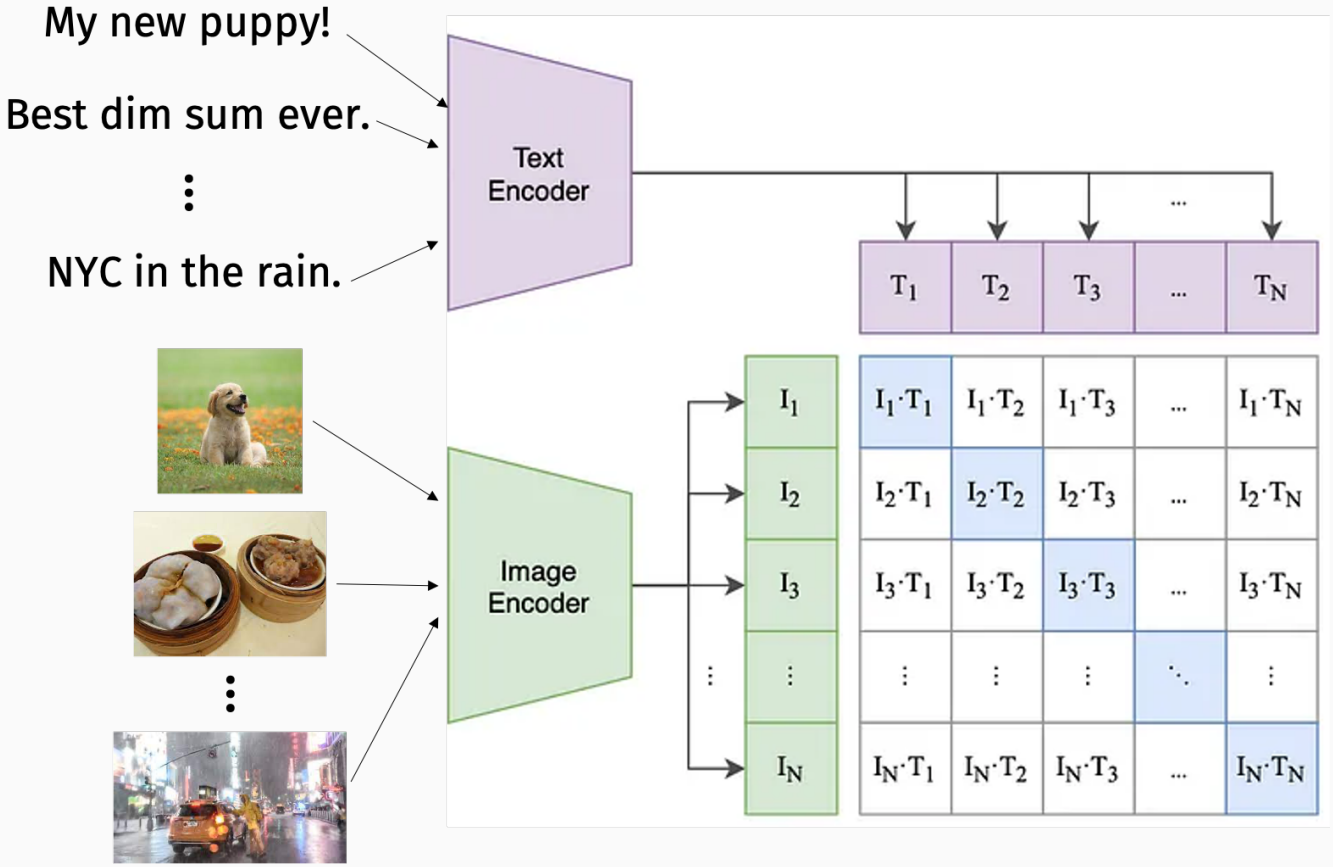
1, 3, 4, 4, 5, 2, 1, 2, 5
6, 8, 6, 4, 3, 1, 5, 3, 4
7, 8, 6, 8, 7, 8, 6, 8, 6
⋮
4, 6, 8, 6, 4, 3, 1, 2, 5



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

BIMODAL EMBEDDINGS

We can also create embeddings that represent different types of data. OpenAI's clip architecture:



Goal: Train embedding architectures so that $\langle \mathbf{T}_i, \mathbf{I}_j \rangle$ are similar if image and sentence are similar.

CLIP TRAINING

What do we use as ground truth similarities during training? Sample a batch of sentence/image pairs and just use identity matrix.



My new puppy!	1	0	0
Best dim sum ever.	0	1	0
NYC in the rain.	0	0	1

This is called contrastive learning. Train unmatched text/image pairs to have nearly orthogonal embedding vectors.

CLIP FOR ZERO-SHOT LEARNING

Learning Transferable Visual Models From Natural Language Supervision

Alec Radford [*1] Jong Wook Kim [*1] Chris Hallacy [1] Aditya Ramesh [1] Gabriel Goh [1] Sandhini Agarwal [1]
Girish Sastry [1] Amanda Askell [1] Pamela Mishkin [1] Jack Clark [1] Gretchen Krueger [1] Ilya Sutskever [1]

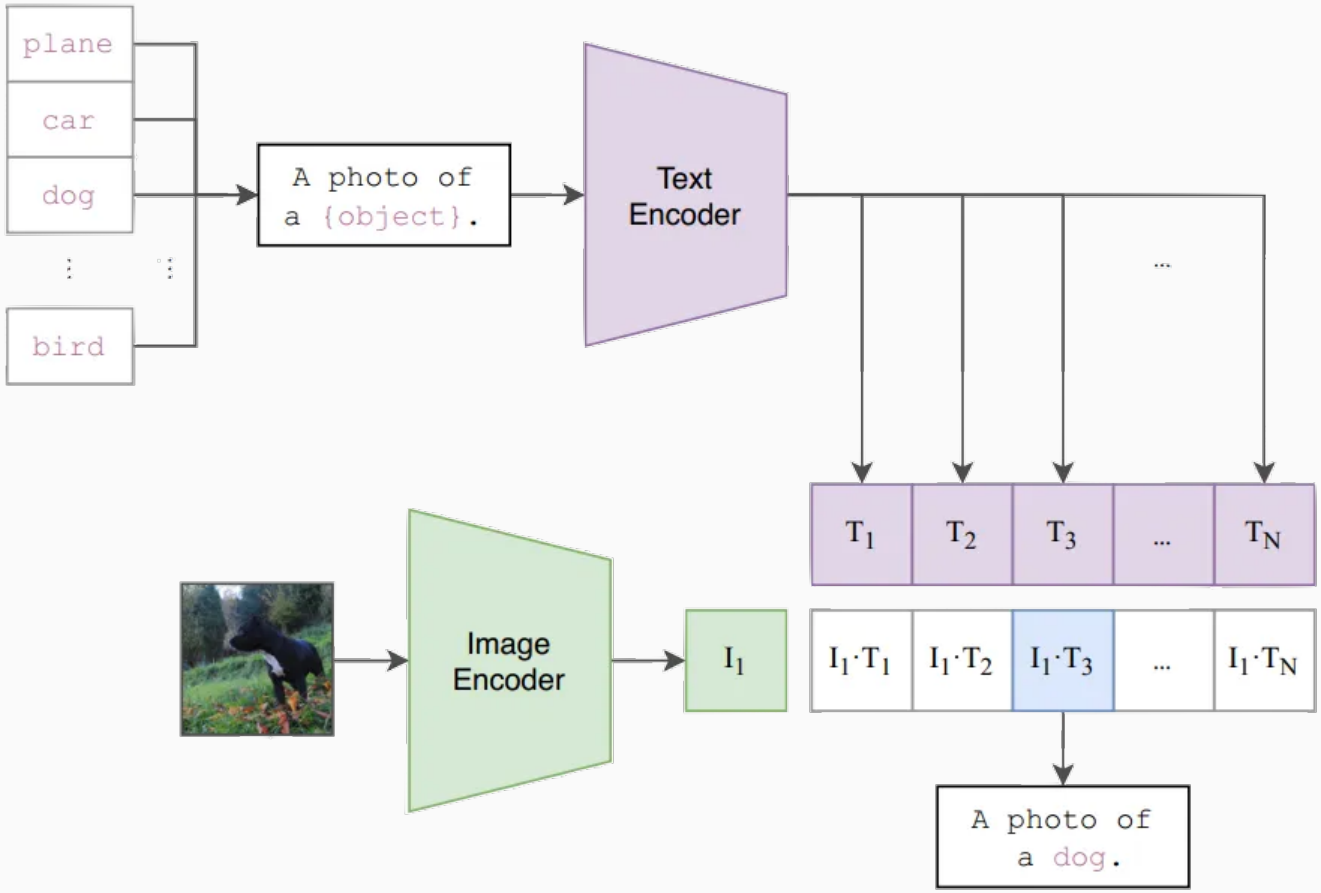
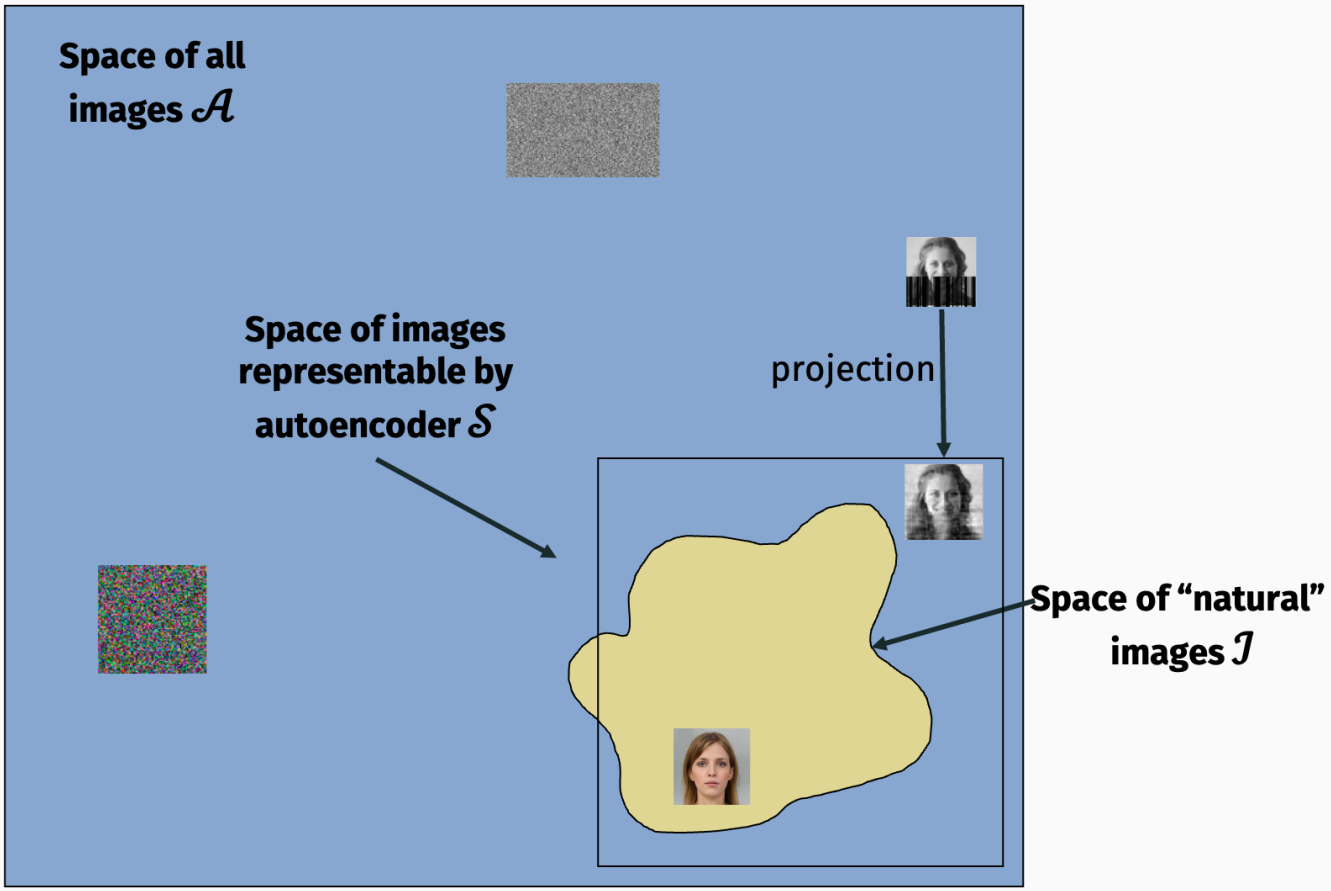


IMAGE SYNTHESIS

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

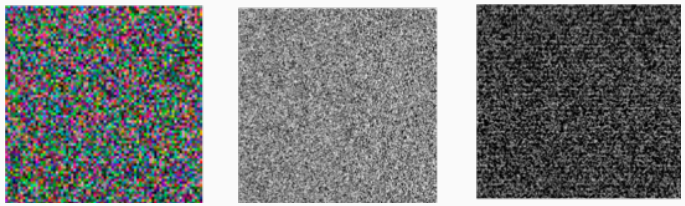


$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.

AUTOENCODERS FOR DATA GENERATION

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

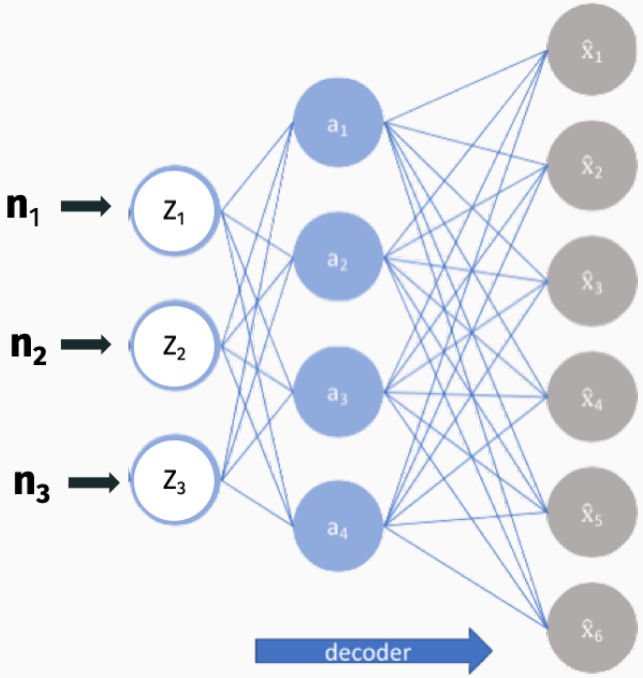
- **Option 1**: Draw each pixel value in $\mathbf{x}$ uniformly at random. Draws a random image from $\mathcal{A}$.

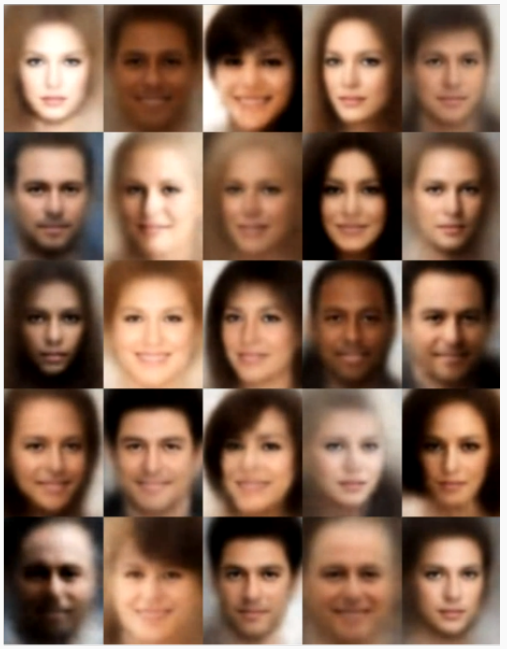


- **Option 2**: Draw $\mathbf{x}$ randomly from $\mathcal{S}$, the space of images representable by the autoencoder.

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

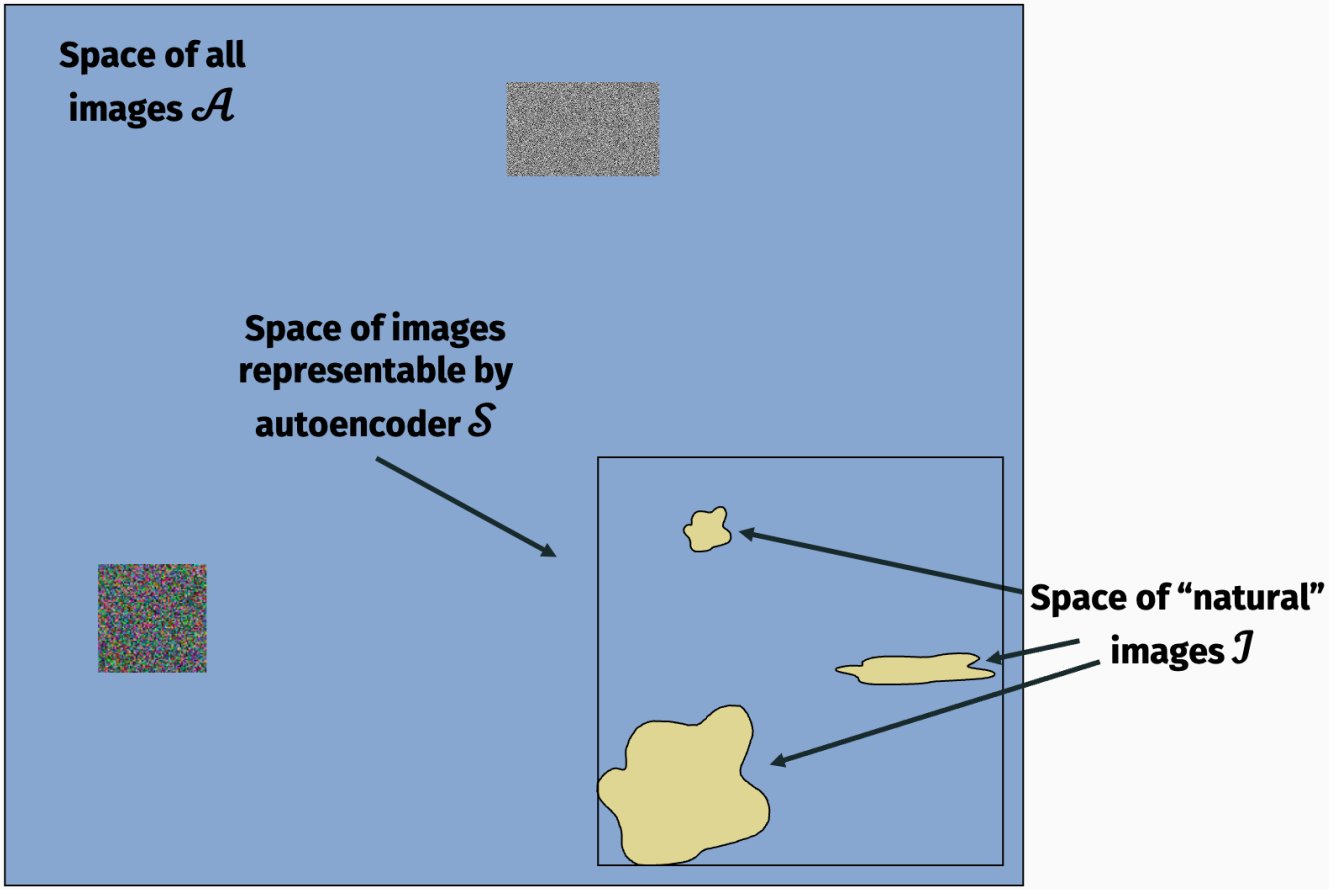
Autoencoder approach to generative ML: Feed random inputs into decode to produce random realistic outputs.





Main issue: most random inputs words will “miss” and produce garbage results.

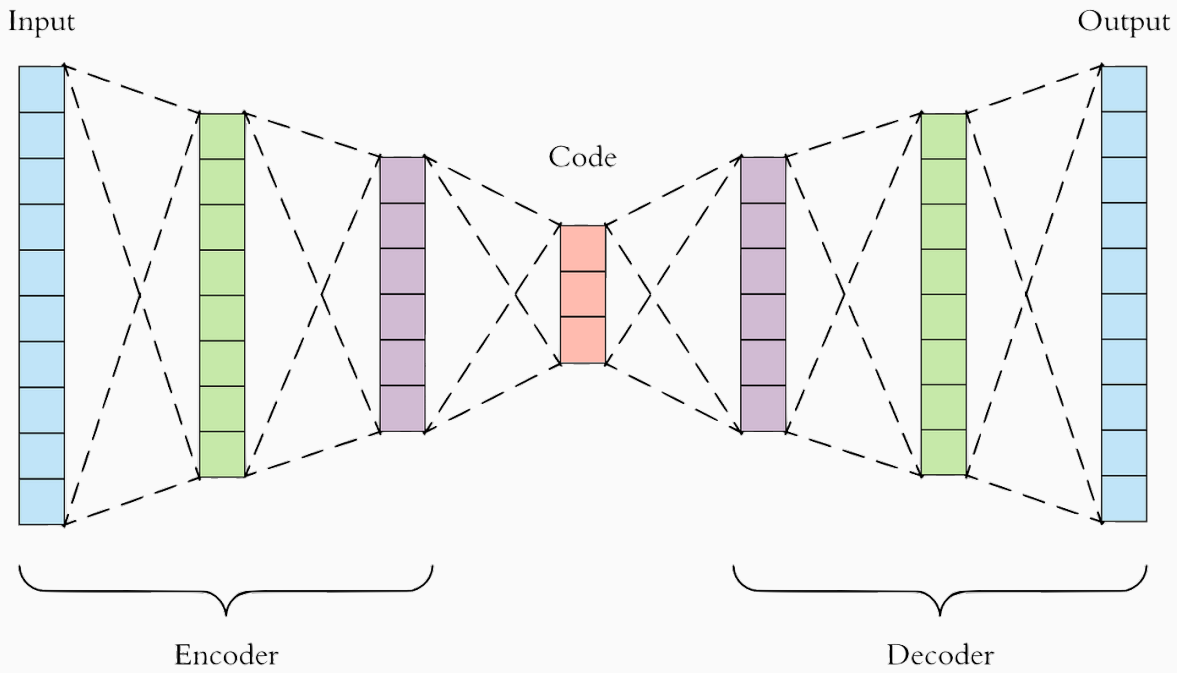
AUTOENCODERS FOR DATA GENERATION



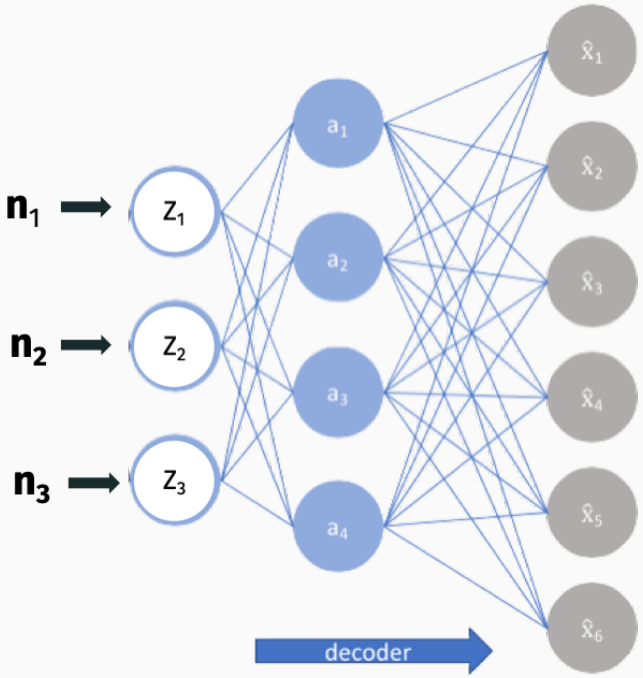
Variational auto-encoders attempt to resolve this issue.

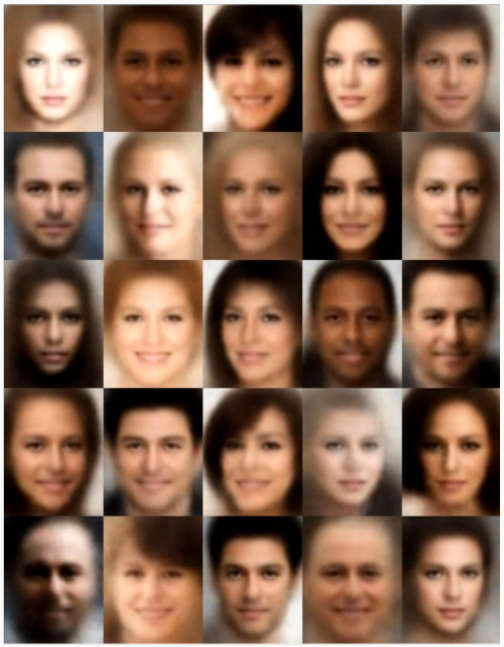
Variational auto-encoders attempt to resolve this issue. Basic ideas:

- Add noise during training.
- Add penalty term so that distribution of code vectors generated looks like mean 0, variance 1 Gaussian.



Variation AE's give very good results, but tends to produce images with immediately recognizable flaws (e.g. soft edges, high-frequency artifacts).



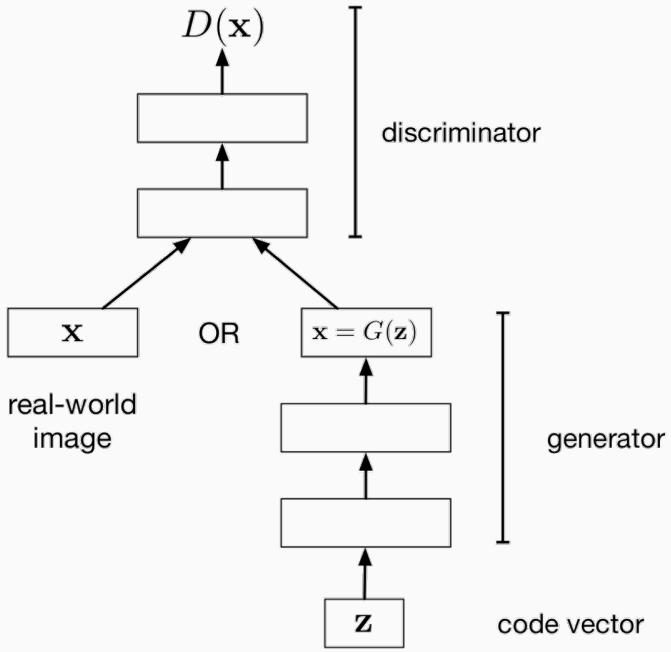


Lots of efforts to hand-design regularizers that penalize images that don't look realisitic to the human eye.

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

$$\min_{\theta} L(\theta) + P(\theta)$$

GENERATIVE ADVERSARIAL NETWORKS (GANS)

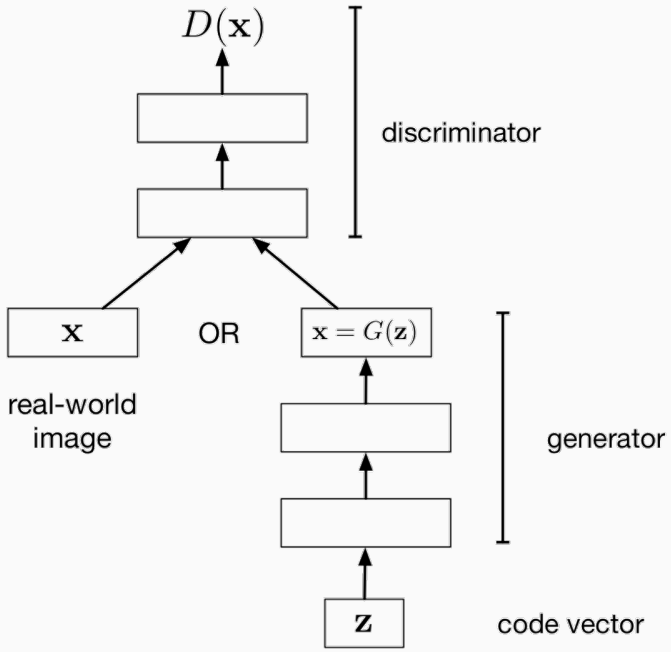


Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be real images and let $\mathbf{z}_1, \ldots, \mathbf{z}_m$ be random code vectors. The goal of the discriminator is to output a number between $[0, 1]$ which is close to 0 if the image is fake, close to 1 if it's real.

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

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

GENERATIVE ADVERSARIAL NETWORKS (GANS)



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

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

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

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

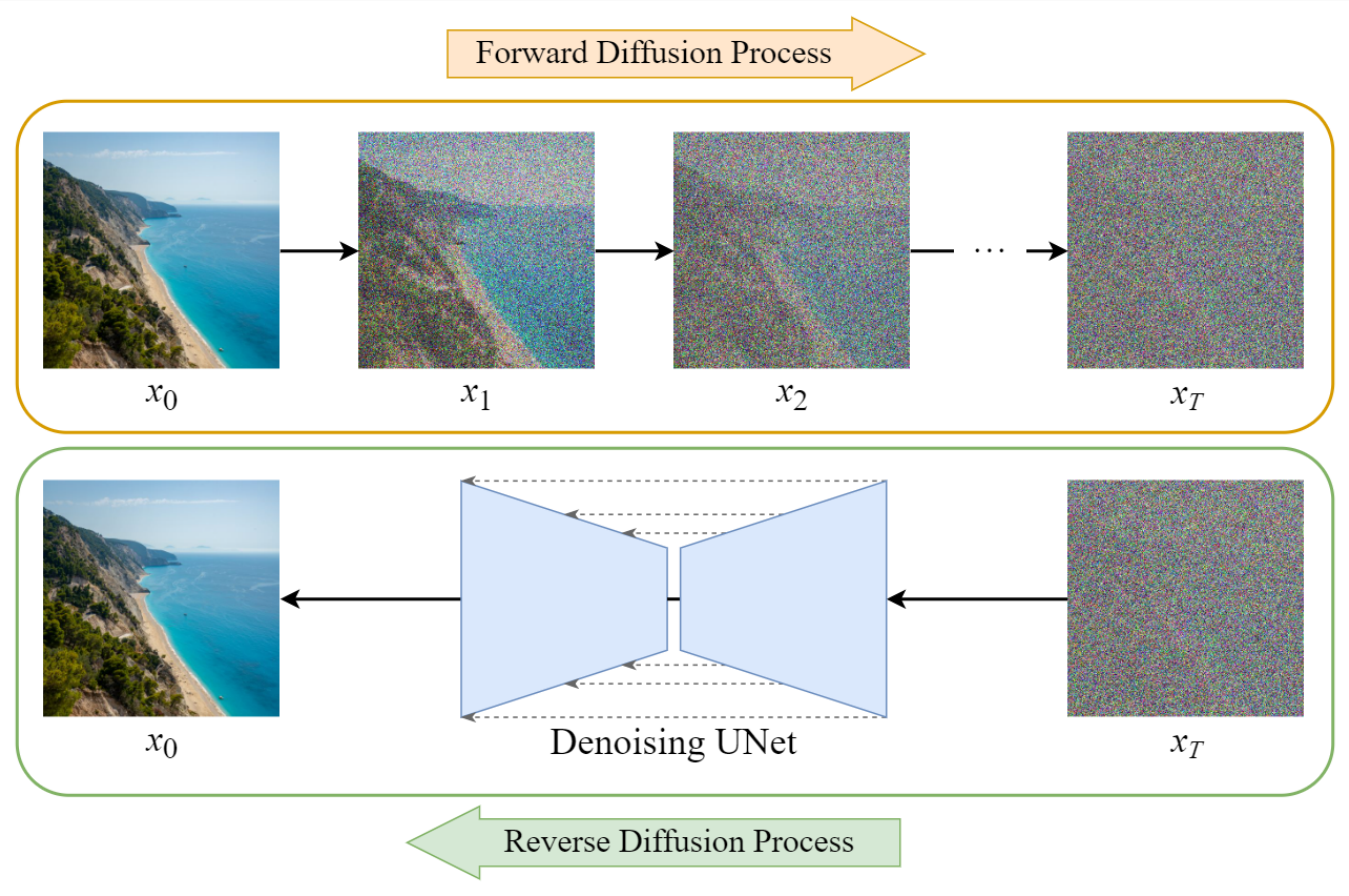
This is called a minimax optimization problem. Really tricky to solve in practice.

- **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., slightly different loss functions) can help.

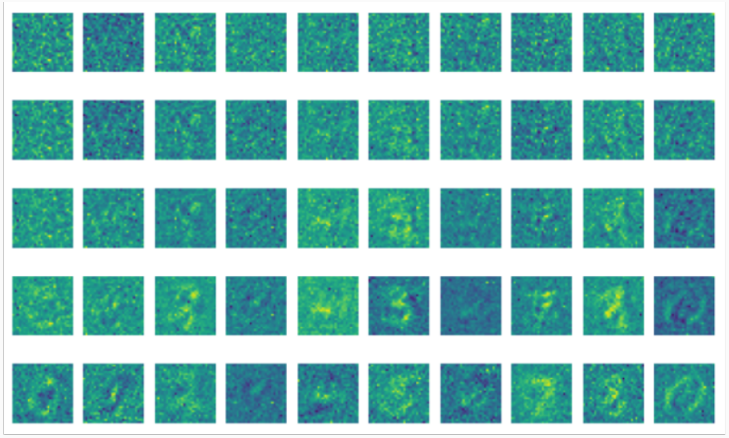
State of the art until a few years ago.

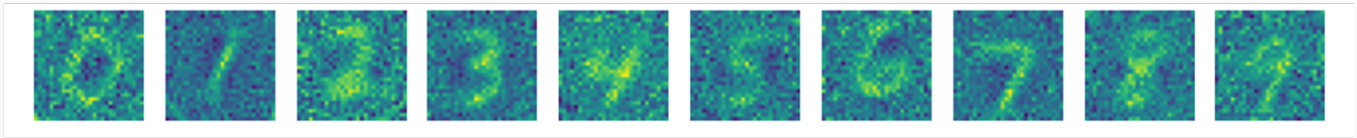
Auto-encoder/GAN approach: Input noise, map directly to image.

Diffusion: Slowly move from noise to image.



We will post a demo for generating digits by training on MNIST.





Text to image synthetsis: Dall-E, Imagen, Stable Diffusion

”A chair that looks like an avocado”