CS-GY 6923: Lecture 13 PCA, Semantic Embeddings, Image Generation
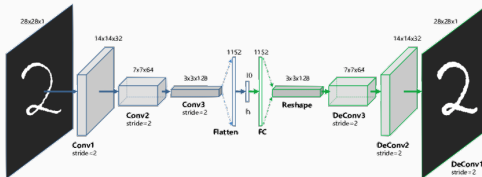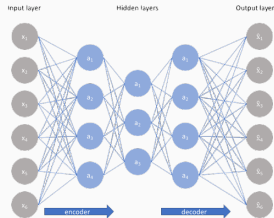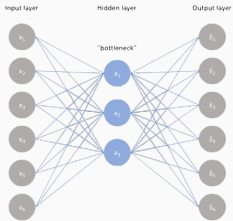
NYU Tandon School of Engineering, Akbar Rafiey
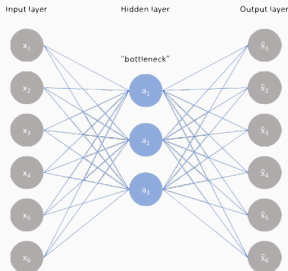
Recap: Goal of autoencoder models is to map input data to a close approximation of the original that takes less space to represent.

Principal Component Analysis

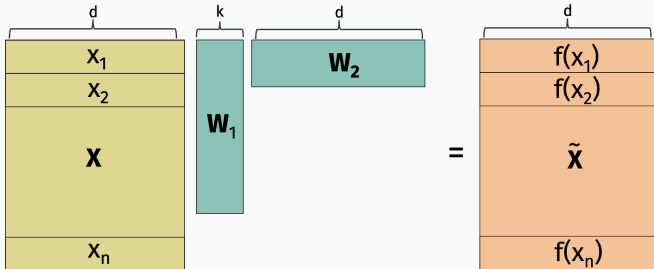PCA is the "linear regression" of autoencoders:

- Simplest possible model. One layer, no non-linearities.
  - $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$ where $\mathbf{X} \in \mathbb{R}^{n \times d}, \mathbf{W}_1 \in \mathbb{R}^{d \times k}, \mathbf{W}_2 \in \mathbb{R}^{k \times d}$.
  - Want to minimize $\min_{\mathbf{W}_1, \mathbf{W}_2} \|\mathbf{X} - \mathbf{X}\mathbf{W}_1\mathbf{W}_2\|_F^2$.

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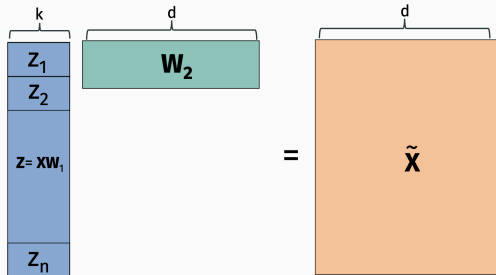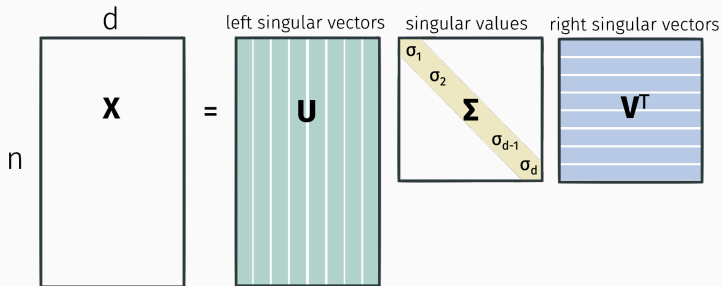Let $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{W}_1\mathbf{W}_2$.

Low-rank approximation

- $\tilde{\mathbf{X}}$ is a **low-rank matrix**. It only has rank (at most) k for $k \ll d$.

- Finding best $\mathbf{W}_1, \mathbf{W}_2$: equivalent to low-rank approximation. Can be efficiently and provable optimized using the SVD.

Singular Value Decomposition

Any matrix $\mathbf{X}$ can be written:

left singular vectors   singular values   right singular vectors

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 <u>orthogonal matrices</u>. Can be computed in $O(nd^2)$ time (faster with approximation algos).
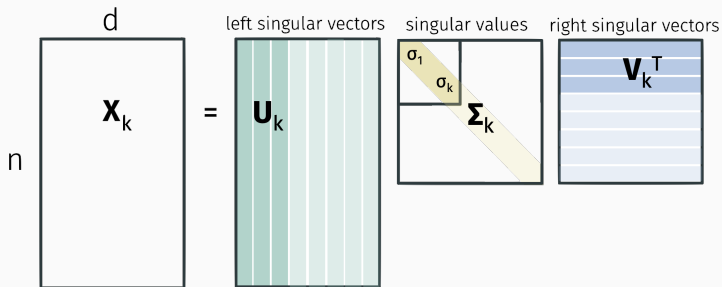
Can be computed in roughly $O(ndk)$ time.

Singular value decomposition

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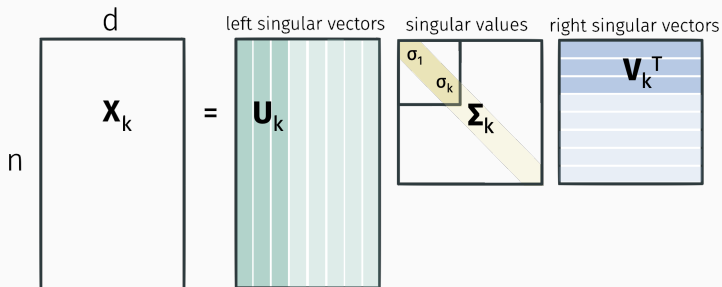
Eckart–Young–Mirsky Theorem: For any $k \leq d$,
$\mathbf{X}_k = \mathbf{U}_k \boldsymbol{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

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

Singular value decomposition

Claim: $\mathbf{X}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^T = \mathbf{X} \mathbf{V}_k \mathbf{V}_k^T$.

So for a model with k hidden variables, we obtain an optimal autoencoder by setting $\mathbf{W}_1 = \mathbf{V}_k$, $\mathbf{W}_2 = \mathbf{V}_k^T$. $f(\mathbf{x}) = \mathbf{x} \mathbf{V}_k \mathbf{V}_k^T$.

Principal Component Analysis
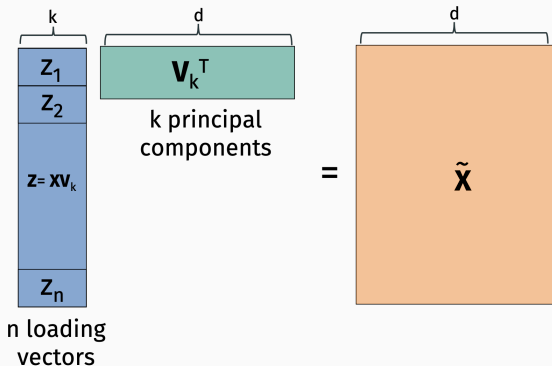
Eckart–Young–Mirsky Theorem: $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{V}_k\mathbf{V}_k^T$ is the optimal low-rank approximation to $\mathbf{X}$. So $\mathbf{W}_1 = \mathbf{V}_k$ and $\mathbf{W}_2 = \mathbf{V}_k^T$ are optimal autoencoder parameters.

Principal Component Analysis (PCA)
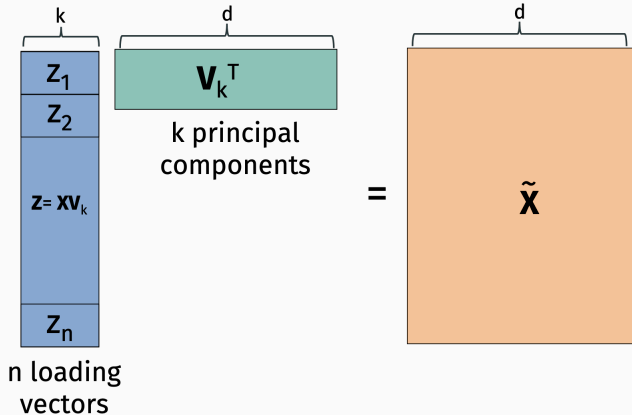
Usually $\mathbf{x}$'s columns (features) are mean centered and normalized to variance 1 before computing principal components.

Singular value decomposition

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 <u>roughly</u> $O(ndk)$ time.

Connection to eigen-decomposition

Recall that for a matrix $\mathbf{M} \in \mathbb{R}^{p \times p}$, $\mathbf{q}$ is an <u>eigenvector</u> of $\mathbf{M}$ if $\lambda \mathbf{q} = \mathbf{M}\mathbf{q}$ for a 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

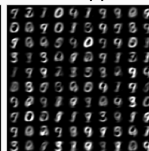| rank 1 approx. | rank 2 approx. | rank 3 approx. | rank 4 approx. | rank 5 approx. |

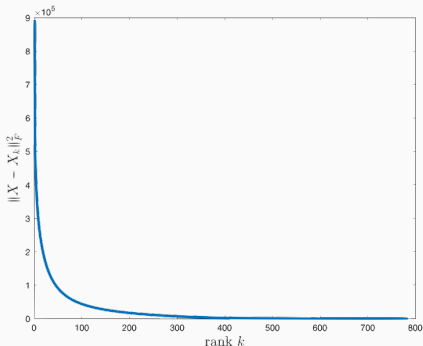| rank 6 approx. | rank 7 approx. | rank 8 approx. | rank 9 approx. | rank 50 approx. |

Low rank approximation

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

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

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.

Column redundancy

Sometimes these relationships are simple, other times more complex. But as long as there exists <u>linear</u> 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:

SNPs matrices tend to be very low-rank.

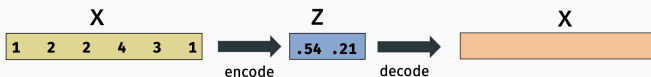| | single nucleotide polymorphisms (SNPs) loci | | | | |
	144	312	436	800	943
individual 1	A	T	T	C	G
individual 2	T	G	G	C	C
...					
individual n	C	A	T	A	G

Most of the information in **x** is explained by just a few **latent variable**.
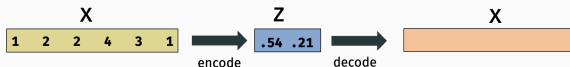
Examples of low-rank structure

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

In data collected from European populations, latent variables capture information about <u>geography</u>.
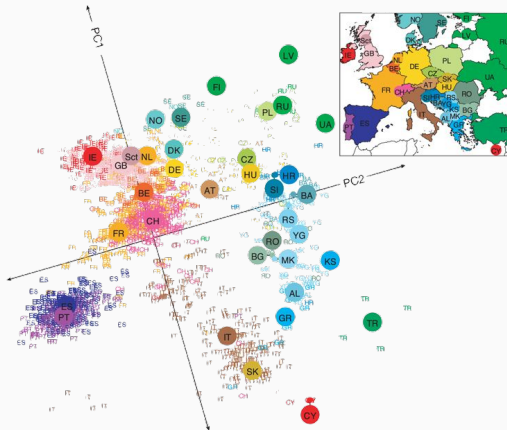
$z[1] \approx$ relative north-south position of birth place

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

Individuals born in similar places tend to have similar genes.

PCA for data visualization

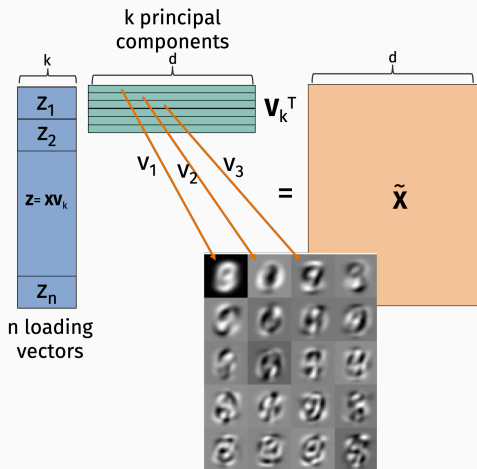"Genes Mirror Geography Within Europe" – Nature, 2008.

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

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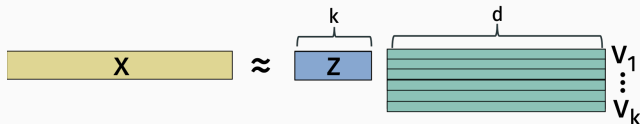
MNIST **principal components**:

Principal components are a small set of vectors that can be recombined to approximate rows in $\tilde{\mathbf{X}}$.

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

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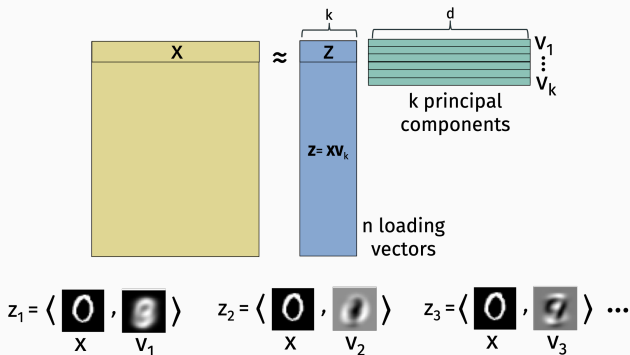

Provide a short "finger print" for any image **x** which can be used to reconstruct that image.

Loading vectors: similarity view

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

$z_1 = \langle \ \boxed{0}\ ,\ \boxed{}\ \rangle$
 $\mathbf{x}$ $\mathbf{v}_1$

$z_2 = \langle \ \boxed{0}\ ,\ \boxed{}\ \rangle$
 $\mathbf{x}$ $\mathbf{v}_2$

$z_3 = \langle \ \boxed{0}\ ,\ \boxed{}\ \rangle$ ...
 $\mathbf{x}$ $\mathbf{v}_3$

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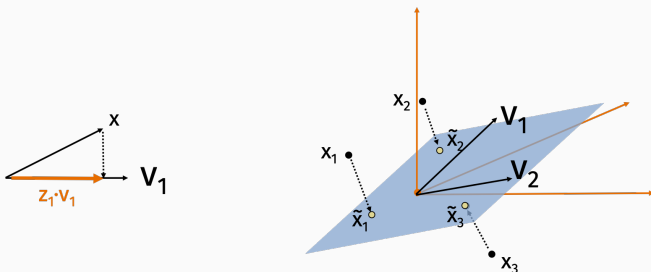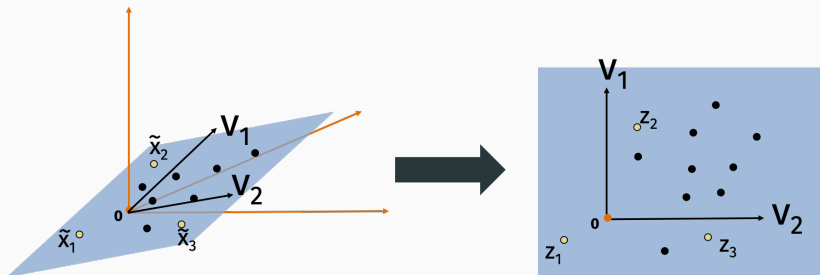

Since $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are orthonormal, this operation is a **projection** onto first k principal components.

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

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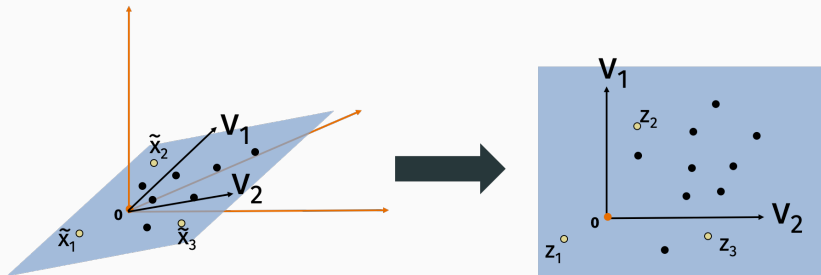
Similarity preservation

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:

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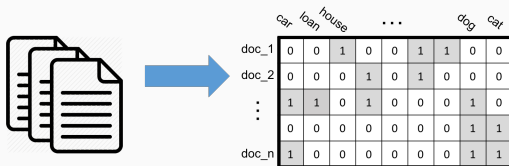
$$\|\mathbf{x}_i\|_2^2 \approx \|\mathbf{z}_i\|_2^2$$
$$\langle \mathbf{x}_i, \mathbf{x}_j \rangle \approx \langle \mathbf{z}_i, \mathbf{z}_j \rangle$$
$$\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \approx \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$$

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*

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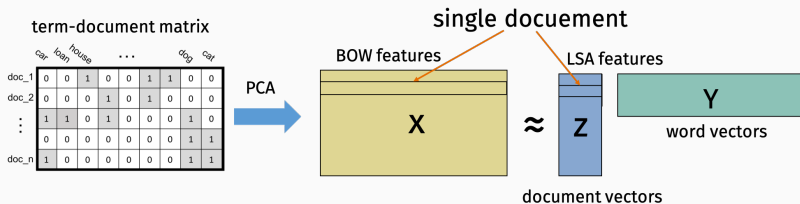
Each column of **z** corresponds to a latent "category" or "topic". Corresponding row in **Y** corresponds to the "frequency" with which different words appear in documents on that topic.

Word-document matrix:
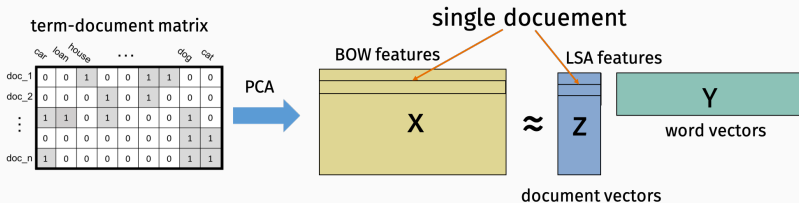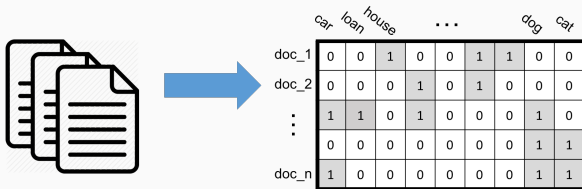
For documents with a lot of shared words, $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ is a large positive number.

Latent semantic analysis

Similar documents have similar <u>LSA document vectors</u>. I.e. $\langle z_i, z_j \rangle$ is large.

- 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 <u>more effective</u> than comparing raw BOW features. Two documents can have $\langle z_i, z_j \rangle$ 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.

Document embeddings

For similar documents, $\langle z_i, z_j \rangle$ should be large. I.e. z_i and z_j point in the same direction.

TSNE Visualization of Book Embeddings

Genre: Crime novel, Science fiction novel, Fantasy novel, Children's novel, Fantasy, Historical novel, Fiction, Non-fiction, Novel, Science fiction
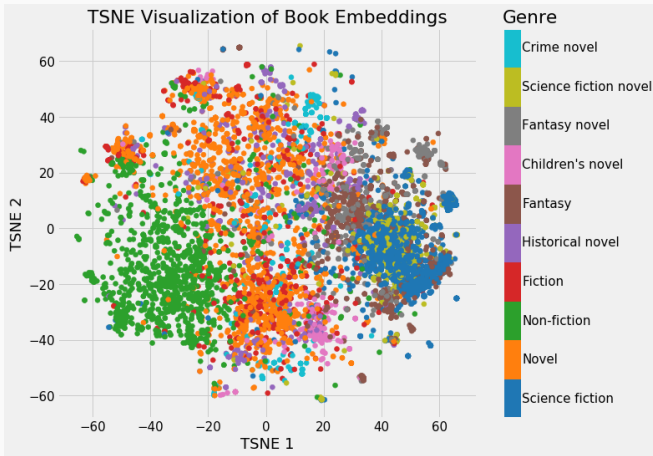
Simple but useful observation: The i, j entry of $\tilde{\mathbf{X}}$ equals $\langle \mathbf{z}_i, \mathbf{y}_j \rangle$.
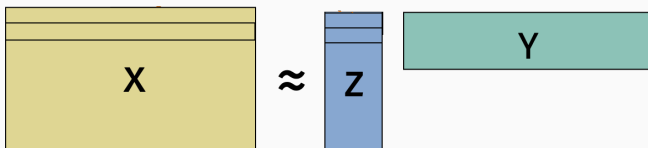
Word embeddings
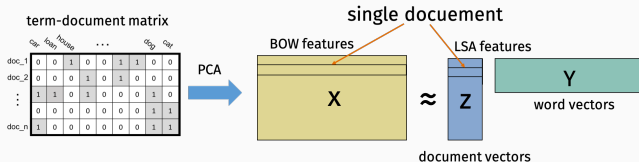
- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$ when doc_a contains $word_i$.
- If $word_i$ and $word_j$ both appear in doc_a, then
  $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$, so we expect $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ to be large.
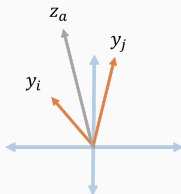
If two words appear in the same document their, word vectors tend to point more in the same direction.

Word embeddings

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

Extremely useful "side-effect" of LSA.

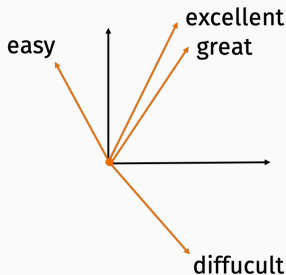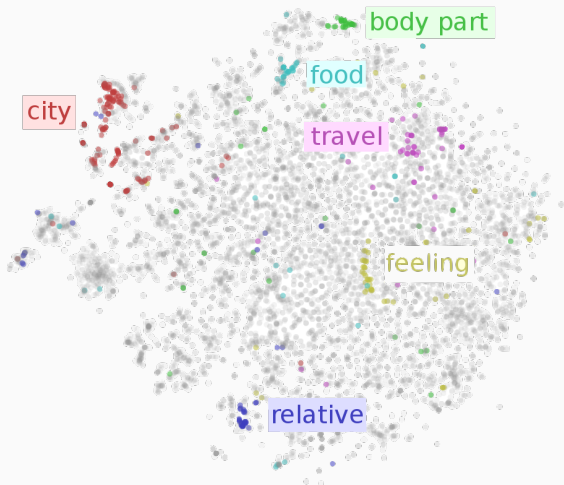Capture e.g. the fact that "great" and "excellent" are near synonyms. Or that "difficult" and "easy" are antonyms.

Word embeddings

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.*

$$[\quad , \quad , \quad , \quad , \quad , \quad , \quad , \quad]$$

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

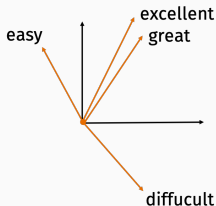$$[\quad , \quad , \quad , \quad , \quad , \quad , \quad , \quad]$$

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.*

$$[\quad , \quad , \quad , \quad , \quad , \quad , \quad , \quad]$$

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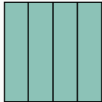
How to go from word embeddings to features for a whole sentence or chunk of text?

remove "stop words"

Very small and handy for traveling or camping. ➡ [small, handy, traveling, camping]

word embedding

[small, handy, traveling, camping] ➡

$\mathbf{y}_1\mathbf{y}_2 \ldots \mathbf{y}_q$

???

$\mathbf{y}_1\mathbf{y}_2 \ldots \mathbf{y}_q$ ➡ feature vector

A few simple options:

Feature vector $\mathbf{x} = \frac{1}{q} \sum_{i=1}^{q} \mathbf{y}_q$.
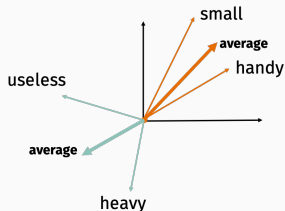
Feature vector $\mathbf{x} = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_q]$.
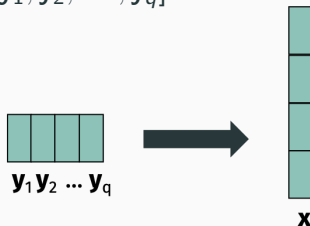
Using word embeddings

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:

$$\mathbf{y}_1 \mathbf{y}_2 \dots \mathbf{y}_q \xrightarrow{\text{SVD}} \mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_k \rightarrow \mathbf{x}$$
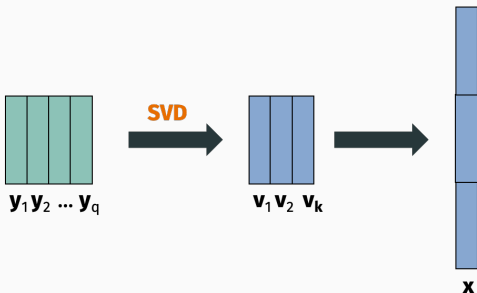
There are much more complicated approaches that account for word position in a sentence. Lots of pretrained libraries available (e.g. Facebook's `InferSent`).

Another view on word embeddings from LSA:

We chose $\mathbf{Z}$ to equal $\mathbf{XV}_k = \mathbf{U}_k\boldsymbol{\Sigma}_k$ 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.

Word embeddings

Another view on word embeddings from LSA:
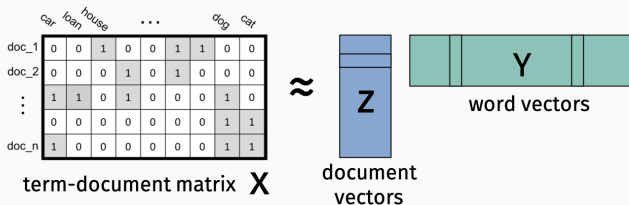
- $\mathbf{X} \approx \mathbf{ZY}$
- $\mathbf{X}^T\mathbf{X} \approx \mathbf{Y}^T\mathbf{Z}^T\mathbf{ZY} = \mathbf{Y}^T\mathbf{Y}$
- So for $word_i$ and $word_j$, $\langle \mathbf{y}_i, \mathbf{y}_j \rangle \approx [\mathbf{X}^T\mathbf{X}]_{i,j}$.

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

term-document matrix **X**

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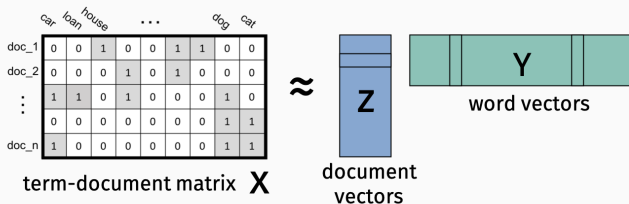
What does the i, j entry of $\mathbf{X}^T\mathbf{X}$ represent?

The number of documents where words i and j were both used.

Word embeddings

$\langle \mathbf{y}_i, \mathbf{y}_j \rangle$ is <u>larger</u> 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.

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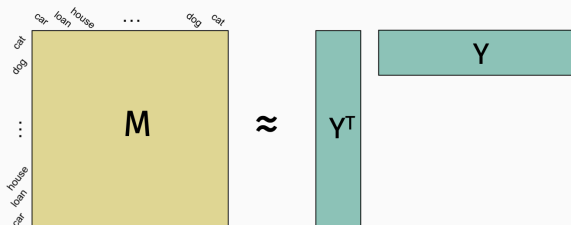
How do current state-of-the-art methods differ from LSA?

- 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

51

Modern word embeddings

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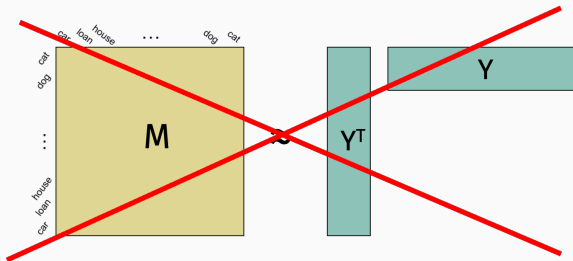
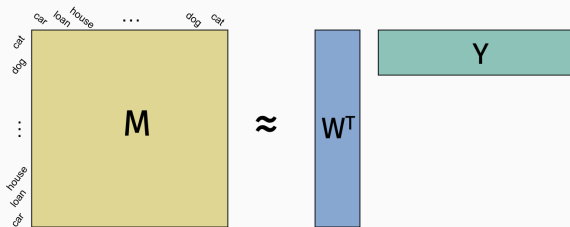SVD will not return a symmetric factorization in general. In fact, if **M** is not positive semidefinite[1] then the optimal low-rank approximation does not have this form.

---

[1] 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:
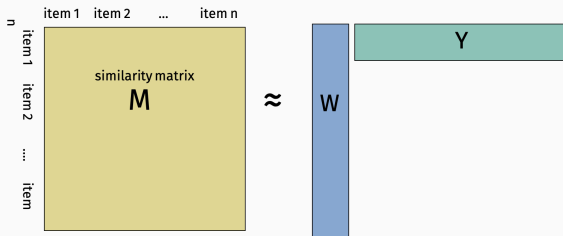
- **Original gloVe website:**
  https://nlp.stanford.edu/projects/glove/.

- **Compilation of many sources:**
  https://github.com/3Top/word2vec-api

Semantic embeddings

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.
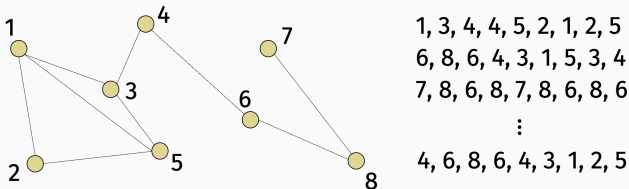

For example, the items could be nodes in a social network graph. Maybe be want to predict an individuals age, level of interest in a particular topic, political leaning, etc.

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

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

	node 1	node 2	...	node 8
node 1	0	2		1
node 2	2	0		0
...				
node 8	1	0		0

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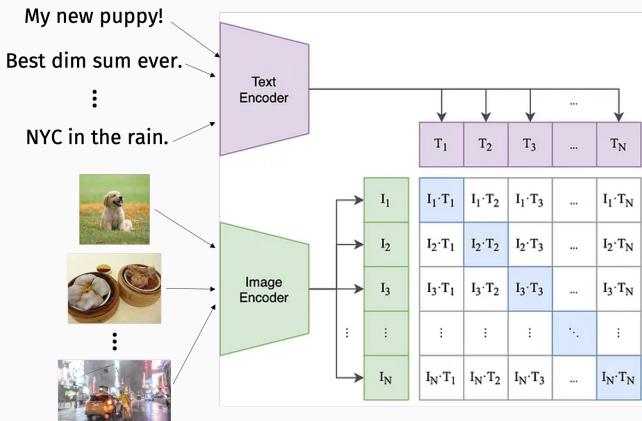
Goal: Train embedding architectures so that $\langle \mathbf{T}_i, \mathbf{I}_j \rangle$ are similar if
image and sentence are similar.

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 <u>contrastive learning</u>. Train unmatched text/image
pairs to have nearly orthogonal embedding vectors.

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]
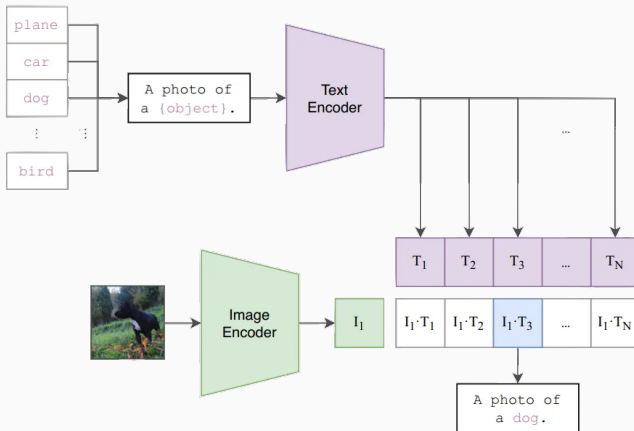
Image Synthesis

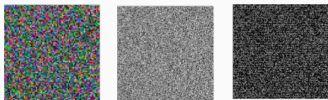## Autoencoders learn compressed representations
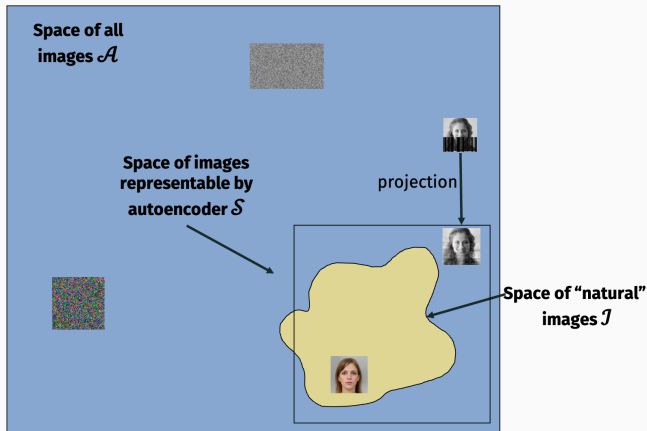
$f(\mathbf{x}) = d(e(\mathbf{x}))$ projects an image $\mathbf{x}$ closer to the space of natural images.
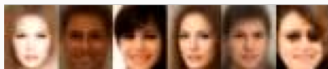
Autoencoders for data generation

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

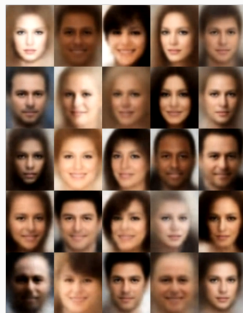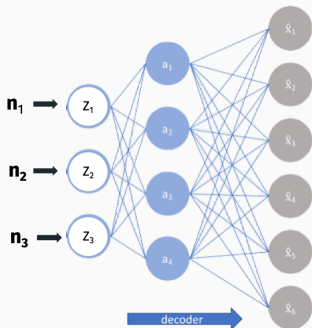- **Option 1**: Draw each pixel value in **x** uniformly at random. Draws a random image from $\mathcal{A}$.

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

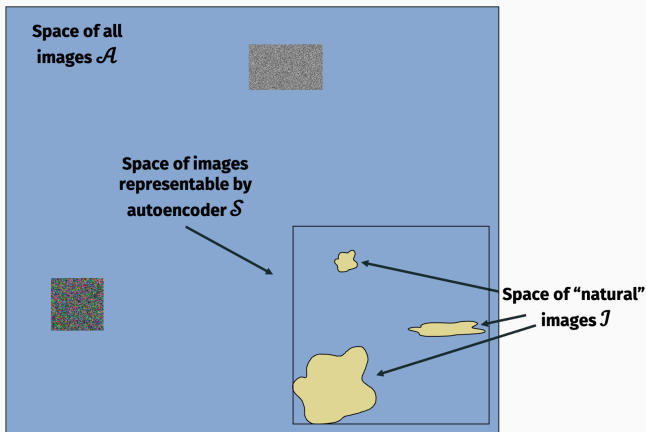How do we randomly select an image from $\mathcal{S}$?

Autoencoders for data generation

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.
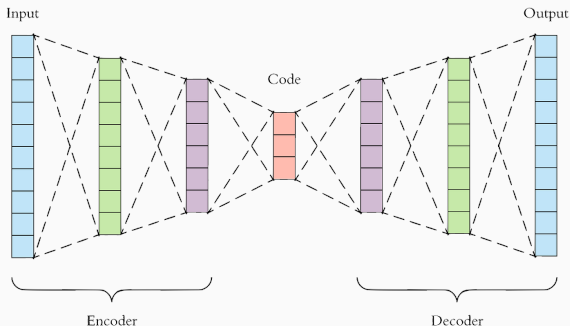
Variational auto-encoders attempt to resolve this issue.

Variational autoencoders

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.

Generative Adversarial Networks

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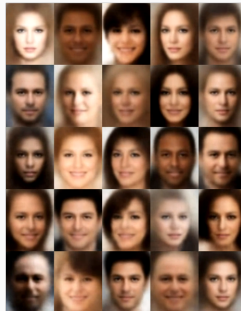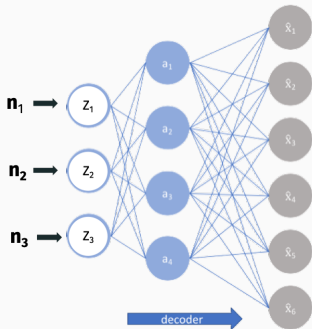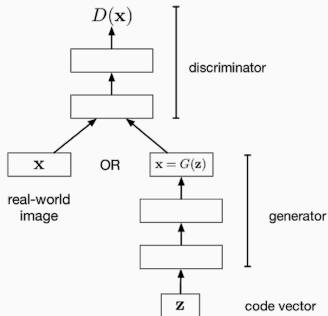
Generative Adversarial Networks (GANs)

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 $x_1, \ldots, x_n$ be real images and let $z_1, \ldots, z_m$ 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_θ to minimize:

$$\min_\theta \sum_{i=1}^{n} -\log\left(D_\theta(x_i)\right) + \sum_{i=1}^{m} -\log\left(1 - D_\theta(G_{\theta'}(z_i))\right)$$

Generative Adversarial Networks (GANs)
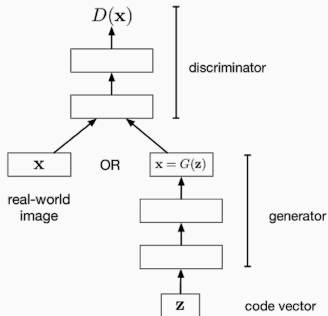
Goal of the generator $G_{\theta'}$ is the opposite. We want to maximize:

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

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

Generative Adversarial Networks (GANs)

θ^*, θ'^* solve $\min\limits_{\theta} \max\limits_{\theta'} \sum\limits_{i=1}^{n} -\log\left(D_{\theta}(\mathbf{x}_i)\right) + \sum\limits_{i=1}^{m} -\log\left(1 - D_{\theta}(G_{\theta'}(\mathbf{z}_i))\right)$

This is called a minimax optimization problem. <u>Really tricky to solve in practice.</u>

- **Repeatedly play:** Fix one of θ^* or θ'^*, train the other to convergence, repeat.

- **Simultaneous gradient descent:** Run a single gradient descent step for each of θ^*, θ'^* and update D and G accordingly. Difficult to balance learning rates.

- Lots of tricks (e.g. slight 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.

Denoising Diffusion Probabilistic Models

Jonathan Ho
UC Berkeley
jonathanho@berkeley.edu

Ajay Jain
UC Berkeley
ajayj@berkeley.edu

Pieter Abbeel
UC Berkeley
pabbeel@cs.berkeley.edu

Abstract

We present high quality image synthesis results using diffusion probabilistic models, a class of latent variable models inspired by considerations from nonequilibrium thermodynamics. Our best results are obtained by training on a weighted variational bound designed according to a novel connection between diffusion probabilistic

How diffusion models work

- Forward Process:
  - Gradually add noise to data until it becomes pure noise.
- Reverse Process:
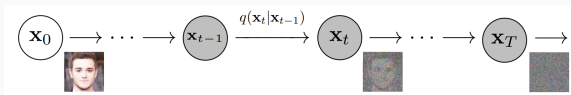 - Train a neural network to remove the noise step by step.
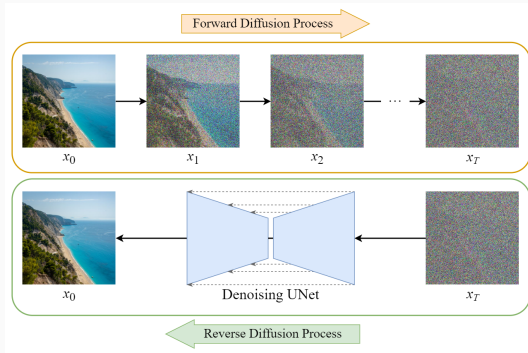
Key Question: How do we predict and reverse noise effectively?

74

Mathematical Formulation (1/2)

Forward Process (Adding Noise):

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$$

- β_t: Noise schedule.
- After T steps, for large enough T, $\mathbf{x}_T$ is pure noise.

Cumulative Noise:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1-\alpha_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I})$$
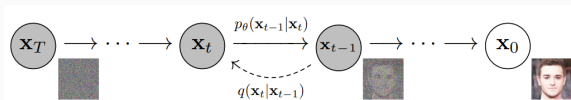
with retention factor $\alpha_t = \prod_{s=1}^{t}(1-\beta_s)$.

Mathematical formulation (2/2)

Reverse Process (Denoising):

$$p_\theta(\mathbf{x}_{t-1}|x_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

- μ_θ: Predicted mean of the clean image.
- Σ_θ: Predicted variance (optional).

Training objective:

$$\mathcal{L}_{\mathsf{simple}} = \mathbb{E}_{\mathbf{x}_0, t, \epsilon} \left[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2 \right]$$

Training process

Data Preparation:

- Use large datasets of images $\mathbf{x}_0$.

Noise Addition:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1 - \alpha_t}\epsilon$$

Model Training:

- Train $\epsilon_\theta(\mathbf{x}_t, t)$ to predict the noise.

Loss Function:

$$\mathcal{L}_{\mathsf{simple}} = \mathbb{E}_{\mathbf{x}_0, t, \epsilon}\left[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2\right]$$

Image generation using diffusion

Sampling Process:

1. Start with pure noise $\mathbf{x}_T$.

2. Iteratively denoise using:

$$\mathbf{x}_{t-1} = \mu_\theta(\mathbf{x}_t, t) + \sqrt{\Sigma_\theta} z, \quad z \sim \mathcal{N}(0, I)$$

3. Final output: a clean image $\mathbf{x}_0$.

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

"A chair that looks like a pineapple"