

# CS-GY 6923: Lecture 13

## Semantic Embeddings, Beyond Autoencoders

---

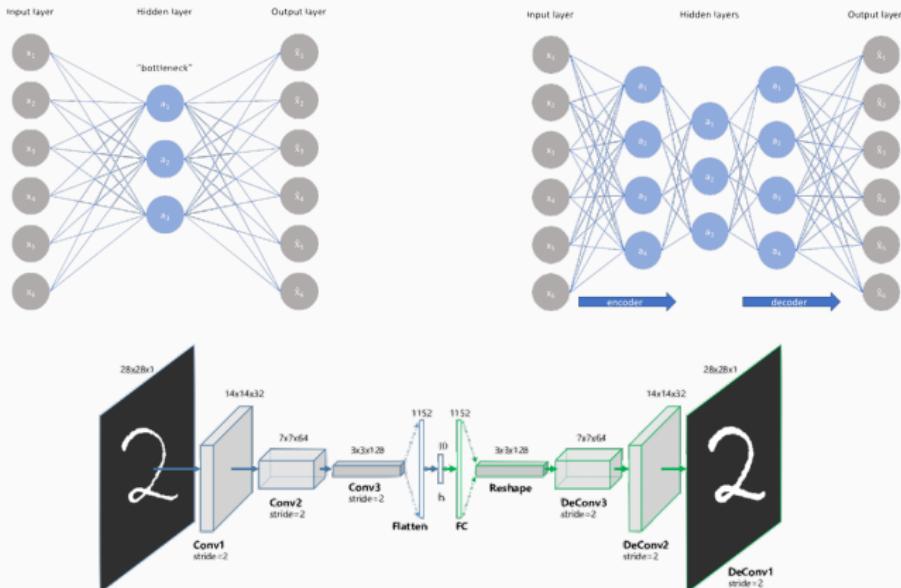
NYU Tandon School of Engineering, Prof. Christopher Musco

## AUTOENCODER

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

# AUTOENCODER

**Important property of autoencoders:** no matter what architecture is used, 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.



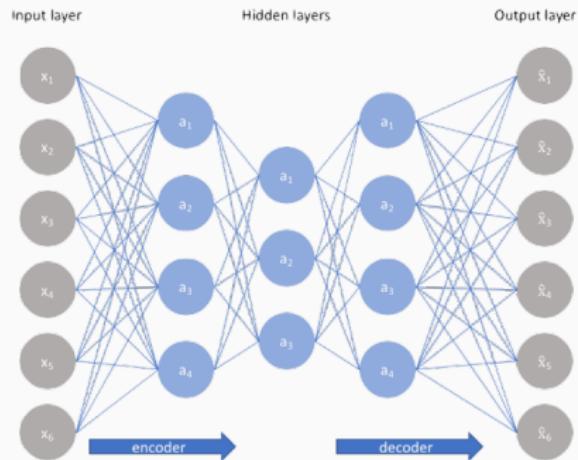
# AUTOENCODER

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

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

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

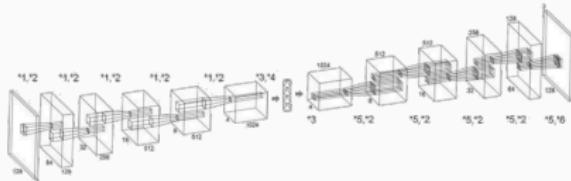
$$f(x) = d(e(x))$$



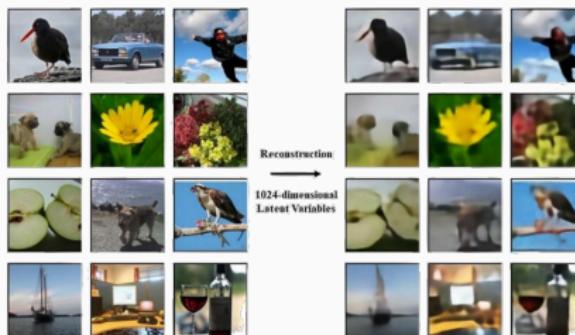
# AUTOENCODER RECONSTRUCTION

Example image reconstructions from autoencoder:

(A)



(B)



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

Input parameters:  $d = 49152$ .

Bottleneck “latent” parameters:  $k = 1024$ .

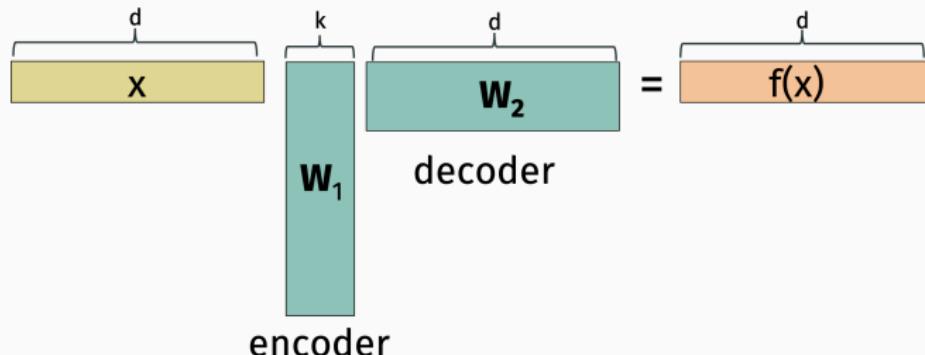
# AUTOENCODERS FOR FEATURE EXTRACTION

Lots of applications:

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

# PRINCIPAL COMPONENT ANALYSIS

Simple linear autoencoder:

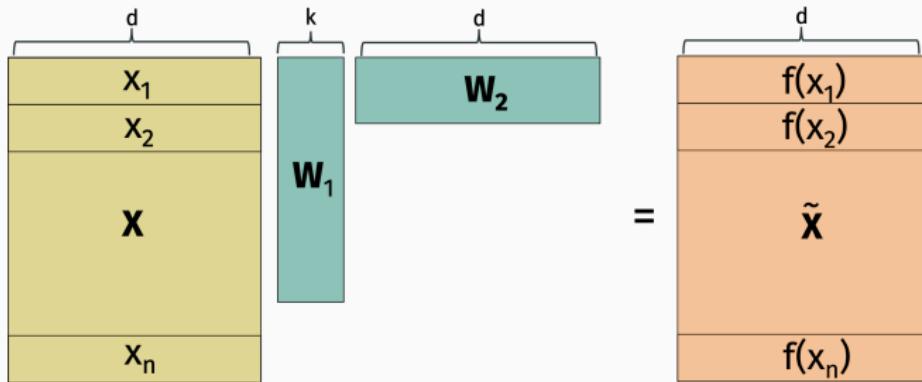


$$f(x)^T = x^T W_1 W_2$$

## PRINCIPAL COMPONENT ANALYSIS

Given training data set  $x_1, \dots, x_n$ , let  $X$  denote our data matrix.

Let  $\tilde{X} = XW_1W_2$  denote the autoencoded data. Want to minimize  $\|X - \tilde{X}\|_F^2$ .



$\tilde{X}$  is a low-rank (rank  $k$ ) matrix. The optimal choice of  $W_1$  and  $W_2$  can be found using algorithms for optimal low-rank approximation.

# SINGULAR VALUE DECOMPOSITION

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

$$\begin{matrix} & d \\ X & = \end{matrix} \begin{matrix} \text{left singular vectors} \\ U \end{matrix} \begin{matrix} \text{singular values} \\ \Sigma \end{matrix} \begin{matrix} \text{right singular vectors} \\ V^T \end{matrix}$$

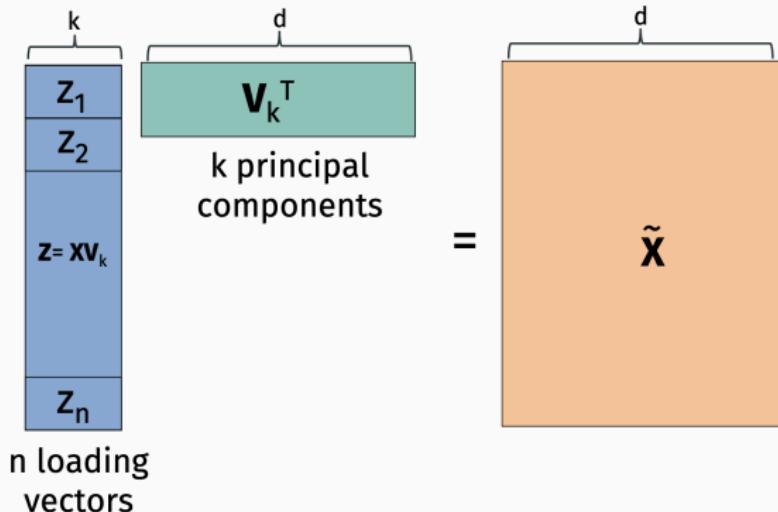
The diagram illustrates the Singular Value Decomposition (SVD) of a matrix  $X$ . Matrix  $X$  is shown as a rectangle with dimensions  $n$  (height) by  $d$  (width). It is equated to the product of three matrices:  $U$ ,  $\Sigma$ , and  $V^T$ . Matrix  $U$  is labeled "left singular vectors" and contains vertical teal bars. Matrix  $\Sigma$  is labeled "singular values" and is represented as a diagonal matrix with entries  $\sigma_1, \sigma_2, \dots, \sigma_{d-1}, \sigma_d$ . Matrix  $V^T$  is labeled "right singular vectors" and contains horizontal blue bars.

Where  $U^T U = I$ ,  $V^T V = I$ , and  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_d \geq 0$ . i.e.  $U$  and  $V$  are orthogonal matrices.

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

## SINGULAR VALUE DECOMPOSITION

We obtain an optimal autoencoder by setting  $W_1 = V_k$ ,  $W_2 = V_k^T$ .  
 $f(x) = xV_kV_k^T$ .



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

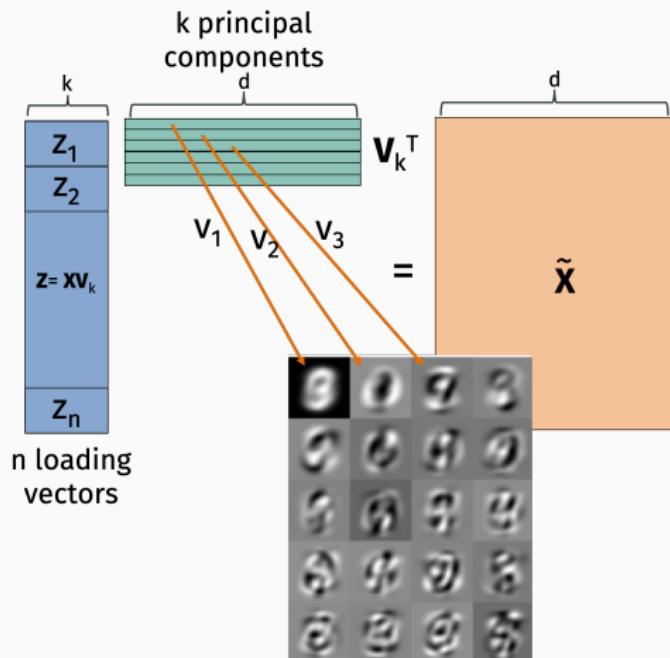
## PRINCIPAL COMPONENTS

---

What do principal components and loading vectors look like?

# PRINCIPAL COMPONENTS

MNIST principal components:

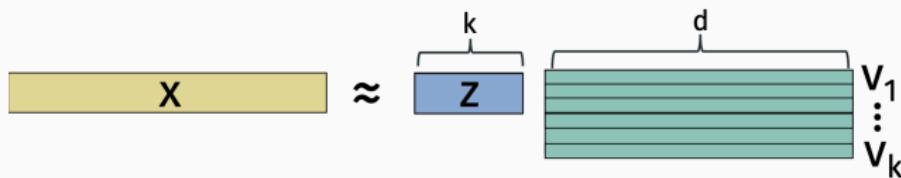


Often principal components are difficult to interpret.

## 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  $\mathbf{v}_1, \dots, \mathbf{v}_k$  to approximately reconstruct  $\mathbf{x}$ .

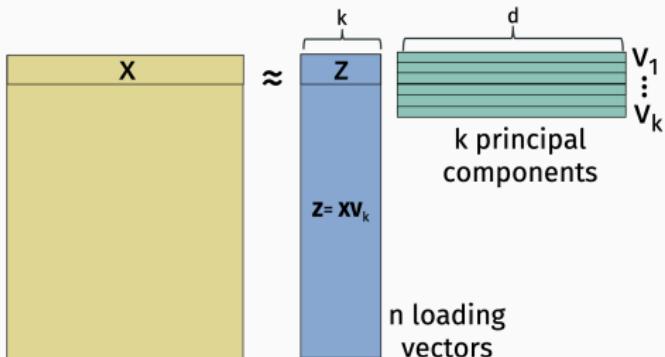


$$\mathbf{x} \approx z_1 \cdot \mathbf{v}_1 + z_2 \cdot \mathbf{v}_2 + z_3 \cdot \mathbf{v}_3 + z_4 \cdot \mathbf{v}_4 + \dots$$

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

## LOADING VECTORS: SIMILARITY VIEW

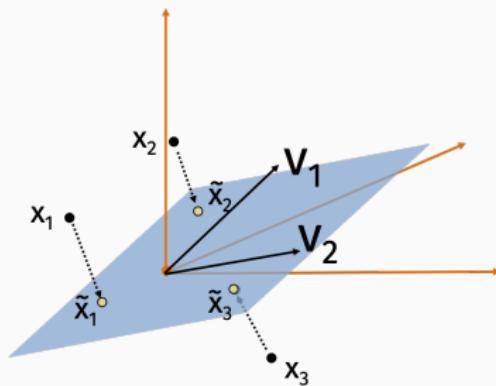
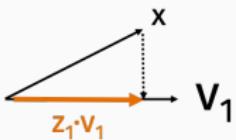
For any  $x$  with loading vector  $z$ ,  $z_i$  is the inner product similarity between  $x$  and the  $i^{\text{th}}$  principal component  $v_i$ .



$$z_1 = \langle \begin{matrix} 0 \\ x \end{matrix}, \begin{matrix} 0 \\ v_1 \end{matrix} \rangle \quad z_2 = \langle \begin{matrix} 0 \\ x \end{matrix}, \begin{matrix} 0 \\ v_2 \end{matrix} \rangle \quad z_3 = \langle \begin{matrix} 0 \\ x \end{matrix}, \begin{matrix} 0 \\ v_3 \end{matrix} \rangle \dots$$

## LOADING VECTORS: PROJECTION VIEW

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

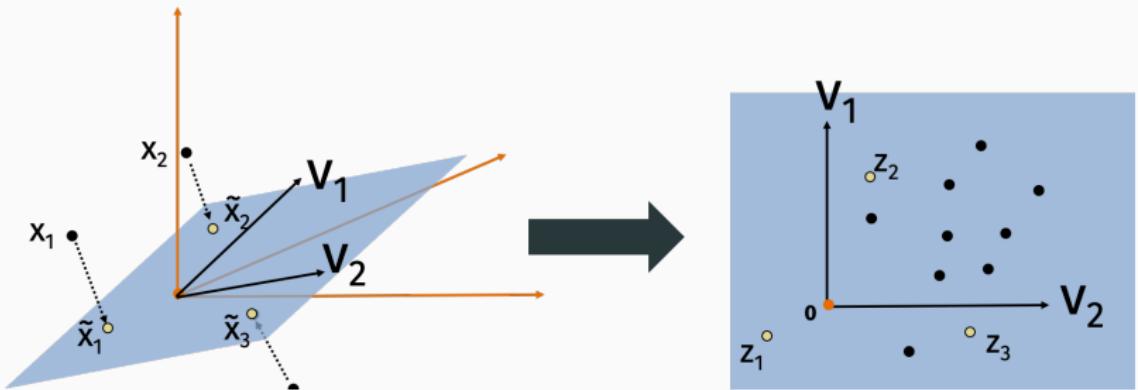


Since  $\mathbf{v}_1, \dots, \mathbf{v}_k$  are orthonormal, this operation is a **projection** onto first  $k$  principal components.

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

## LOADING VECTORS: PROJECTION VIEW

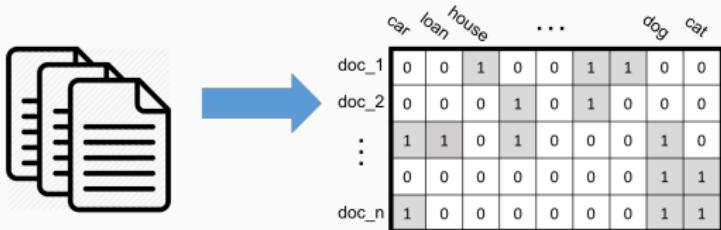
For an example  $x_i$ , the loading vector  $z_i$  contains the coordinates in the projection space:



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

## TERM DOCUMENT MATRIX

Word-document matrices tend to be low rank.

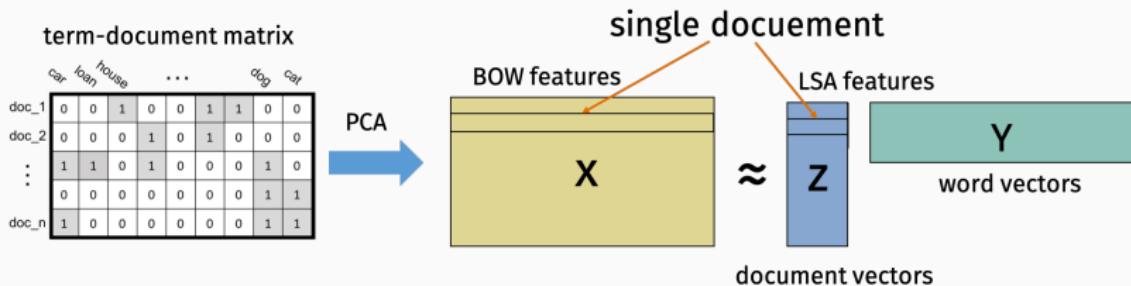


Documents tend to fall into a relatively small number of different categories, which use similar sets of words:

- **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.

## LATENT SEMANTIC ANALYSIS

Similar documents have similar LSA document vectors. 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 more effective 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.

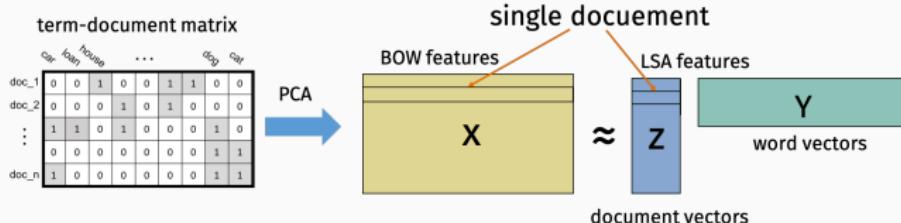
## EIGENFACES

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

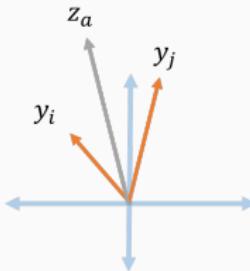


Each image above is one of the principal components of a dataset containing images of faces.

# WORD EMBEDDINGS



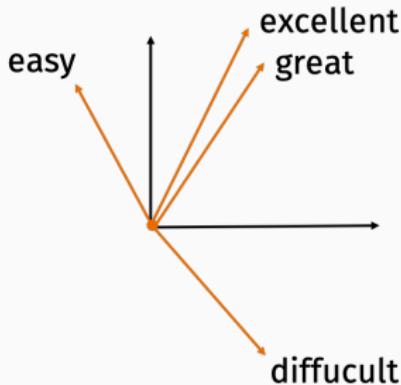
- $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx 1$  when  $doc_a$  contains word<sub>i</sub>.
- If word<sub>i</sub> and word<sub>j</sub> both appear in  $doc_a$ , then  $\langle \mathbf{y}_i, \mathbf{z}_a \rangle \approx \langle \mathbf{y}_j, \mathbf{z}_a \rangle \approx 1$ , so we expect  $\langle \mathbf{y}_i, \mathbf{y}_j \rangle$  to be large.



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

## SEMANTIC EMBEDDINGS

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



Extremely useful “side-effect” of LSA.

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

## WORD EMBEDDINGS

---

Word embeddings are considered a type of semantic embedding.

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

## WORD EMBEDDINGS: MOTIVATING PROBLEM

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

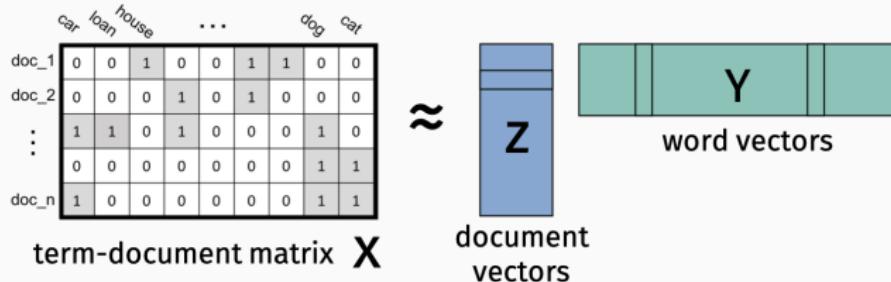
**Review 2:** *So far this thing is great. Well designed, compact,  
and easy to use. I'll never use another can opener.*

**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”.**

## WORD EMBEDDINGS

Another view on word embeddings from LSA:

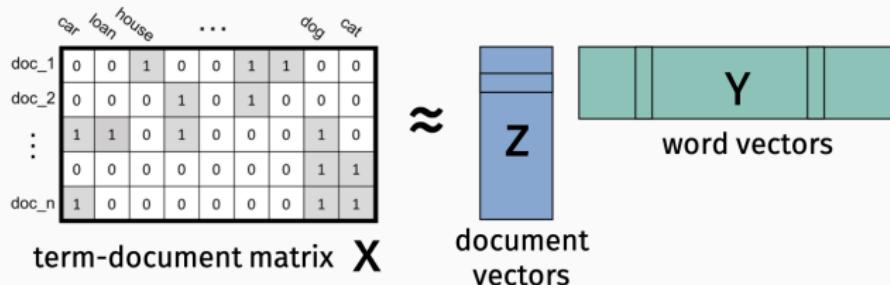


We chose  $Z$  to equal  $XV_k = U_k \Sigma_k$  and  $Y = V_k^T$ .

Could have just as easily set  $Z = U_k$  and  $Y = \Sigma_k V_k^T$ , so  $Z$  has orthonormal columns.

# WORD EMBEDDINGS

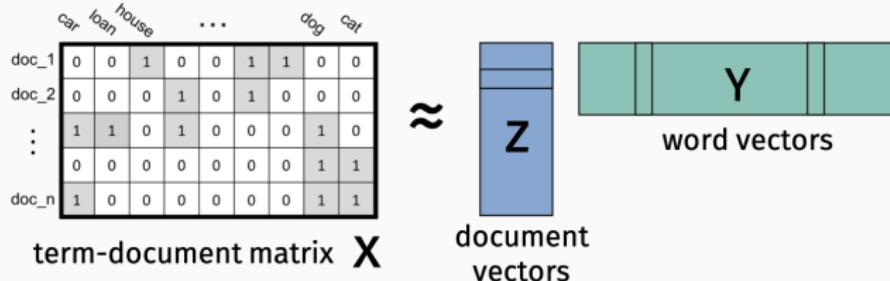
Another view on word embeddings from LSA:



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

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

# WORD EMBEDDINGS



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

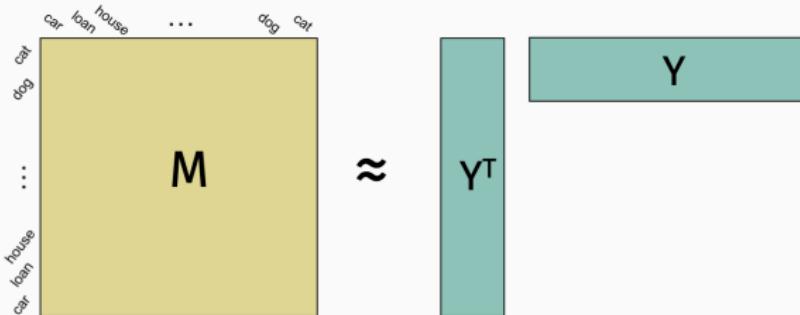
## WORD EMBEDDINGS

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

## 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.  
 $k(\text{word}_i, \text{word}_j) = \frac{p(i,j)}{p(i)p(j)}$  where  $p(i, j)$  is the frequency both  $i, j$  appeared together, and  $p(i), p(j)$  is the frequency either one appeared.

# MODERN WORD EMBEDDINGS

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.

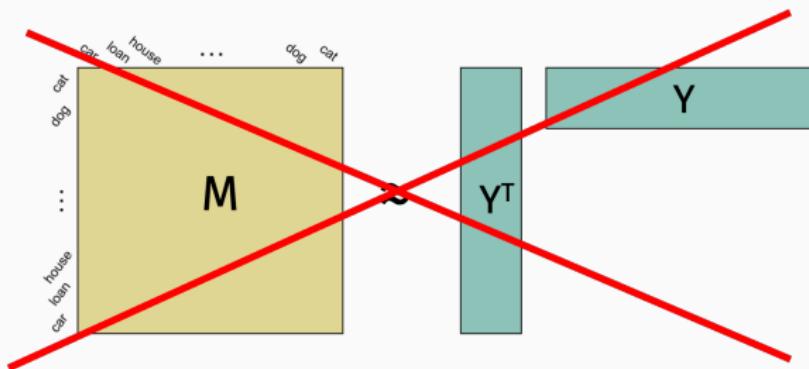
The girl walks to her dog to the park.  
It can take a long time to park your car in NYC.  
The dog park is always crowded on Saturdays.

	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

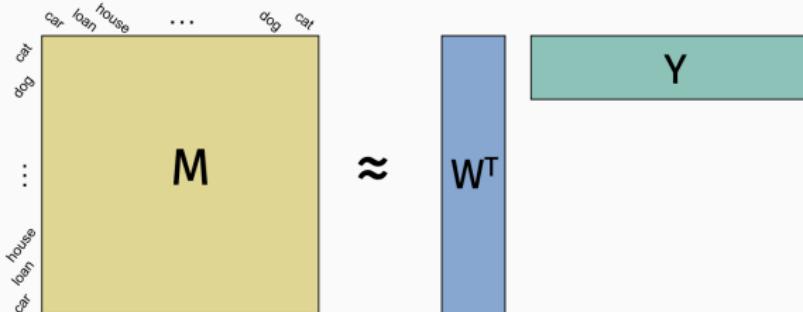


SVD will not return a symmetric factorization in general. In fact, if  $M$  is not positive semidefinite<sup>1</sup> then the optimal low-rank approximation does not have this form.

---

<sup>1</sup>I.e.,  $\kappa$  is not a positive semidefinite kernel.

## CAVEAT ABOUT FACTORIZATION



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

## EASIEST WAY TO USE WORD EMBEDDINGS

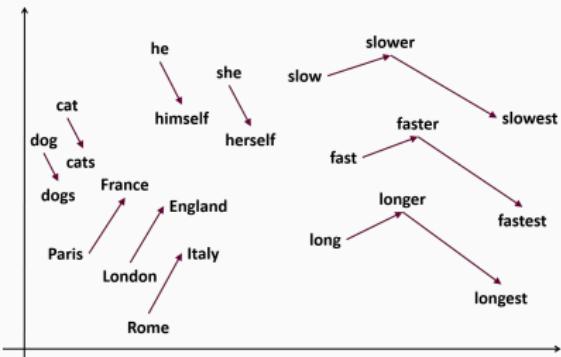
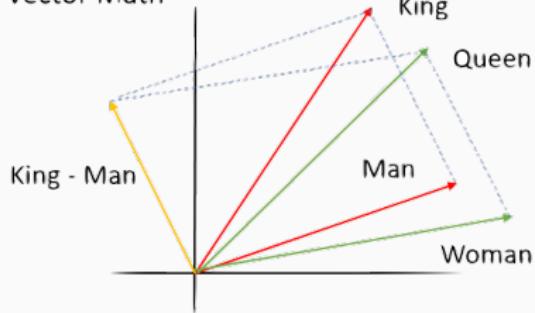
If you want to use word embeddings for your project, the easiest approach is to download pre-trained word vectors:

- Original glove website:  
[https://nlp.stanford.edu/projects/glove/.](https://nlp.stanford.edu/projects/glove/)
- Compilation of many sources:  
<https://github.com/3Top/word2vec-api>

# WORD EMBEDDINGS MATH

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

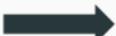
Vector Math



## USING WORD EMBEDDINGS

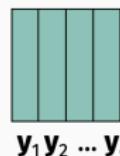
How to go from word embeddings to features for a whole sentence or chunk of text?

remove  
“stop words”

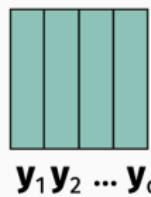
Very small and handy for traveling or camping.  [ small, handy, traveling, camping ]

word  
embedding

[ small, handy, traveling, camping ]



$y_1 y_2 \dots y_q$

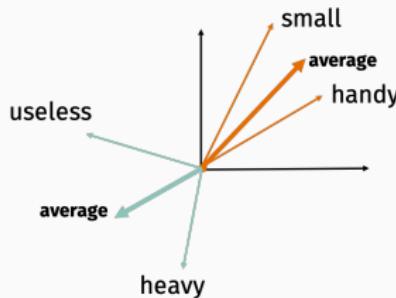


feature vector

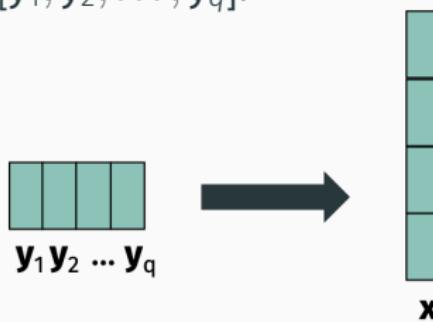
# USING WORD EMBEDDINGS

A few simple options:

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

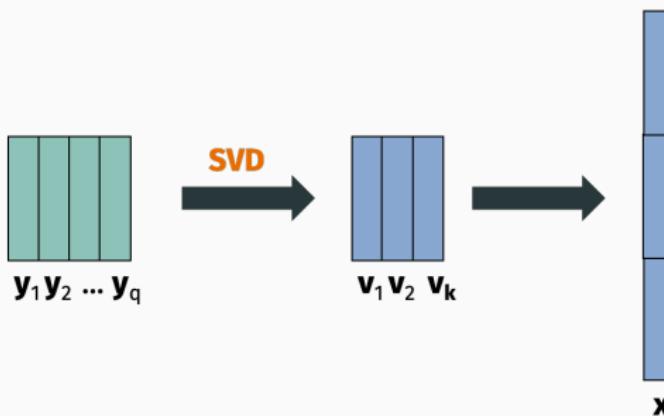


$$\text{Feature vector } \mathbf{x} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_q].$$



## USING WORD EMBEDDINGS

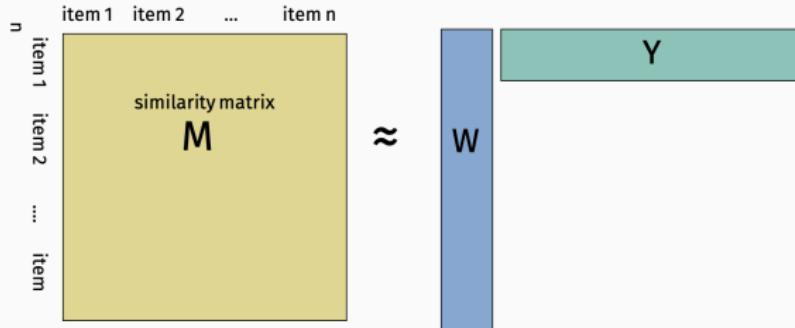
To avoid issues with inconsistent sentence length, word ordering, etc., can concatenate a fixed number of top principal components of the matrix of word vectors:



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

## 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.

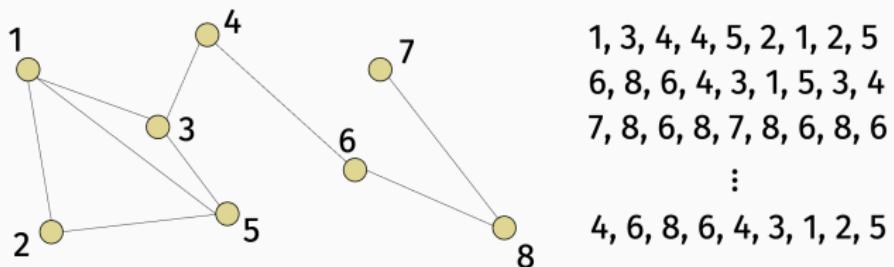


For example, the items could be nodes in a social network graph. Maybe we want to predict an individual's age, level of interest in a particular topic, political leaning, etc.

## NODE EMBEDDINGS



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



## NODE EMBEDDINGS

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

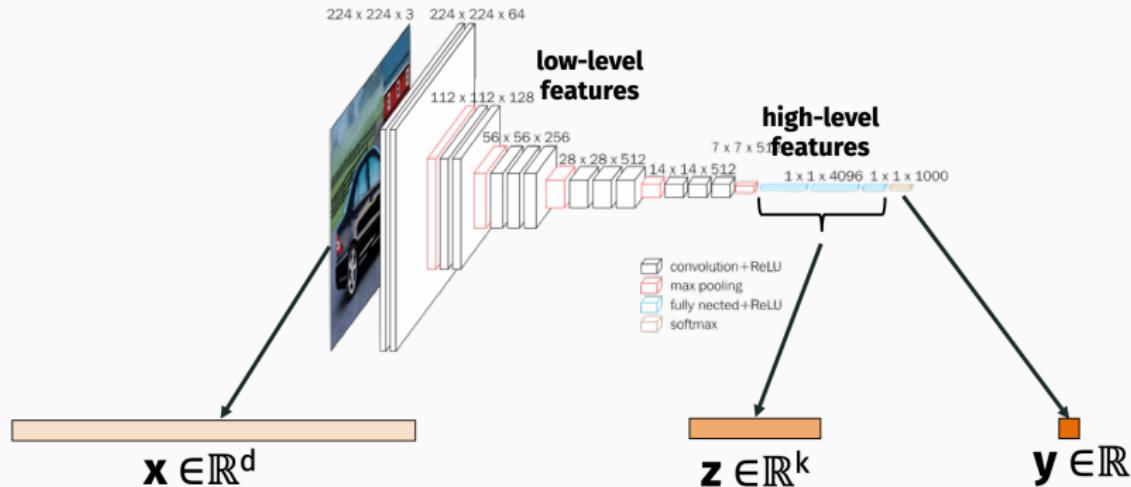
1, 3, 4, 4, 5, 2, 1, 2, 5  
6, 8, 6, 4, 3, 1, 5, 3, 4  
7, 8, 6, 8, 7, 8, 6, 8, 6  
⋮  
4, 6, 8, 6, 4, 3, 1, 2, 5

	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).

## BEYOND AUTOENCODERS

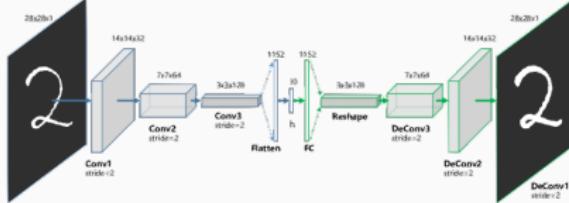
While they have many applications, one of the original goals of autoencoders was to learn “high level” features when we did not have substantial training data.



Typical supervised approach to transfer learning.

# BEYOND AUTOENCODERS

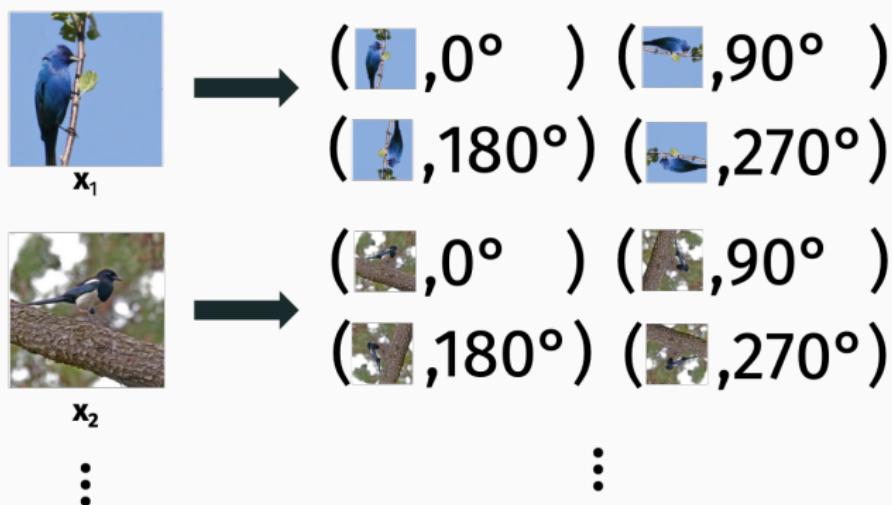
**Growing realization:** The types of features needed to precisely reconstruct an image (e.g. with small  $\ell_2$  error) don't exactly match up with the features required to understand an image.



## SELF-SUPERVISED LEARNING

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

## Example: Rotation Learning.



4 class learning problem. Train a supervised neural network

# SELF-SUPERVISED LEARNING

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

Input
You are given two strings $s$ and $t$ , both consisting of lowercase English letters. You are going to type the string $s$ character by character, from the first character to the last one.
When typing a character, instead of pressing the button corresponding to it, you can press the 'Backspace' button. It deletes the last character you have typed among those that aren't deleted yet (or does nothing if there are no characters in the current string). For example, if $s$ is "abcc" and you press Backspace instead of typing the first and the fourth character, you will get the string "bc" (the first press of Backspace deletes no character, and the second press deletes the character 'c'). Another example, if $s$ is "aacc" and you press Backspace instead of the last two letters, then the resulting text is "a".
Your task is to determine whether you can obtain the string $t$ , if you type the string $s$ and press 'Backspace' instead of typing several (maybe zero) characters of $s$ .
<b>Output</b>
For each test case, print "YES" if you can obtain the string $t$ by typing the string $s$ and replacing some characters with presses of 'Backspace' button, or "NO" if you cannot.
You may print each letter in any case (YES, yes, Yes will all be recognized as positive answer, NO, no and no will be recognized as negative answer).



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

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

# SELF-SUPERVISED LEARNING

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

```
# Python code to reverse a  
string  
# using loop  
  
def reverse(s):  
    str = ""  
    for i in s:  
        str = i + str  
    return str
```



```
# Python code to reverse a  
string  
# using loop  
  
def reverse(s):  
    str = ""  
    for i in s:  
        str = i + str  
    return str
```

$x_1$   
 $y_1 = \text{def}$

```
# Python code to reverse a  
string  
# using loop  
  
def reverse(s):  
    str = ""  
    for i in s:  
        str = i + str  
    return str
```

$x_2$   
 $y_2 = \text{reverse}$

# SELF-SUPERVISED LEARNING

There are a lot of different possibilities!



In-painting task

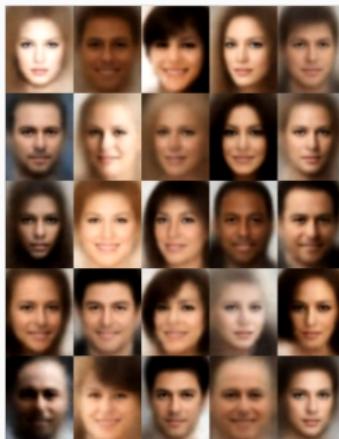
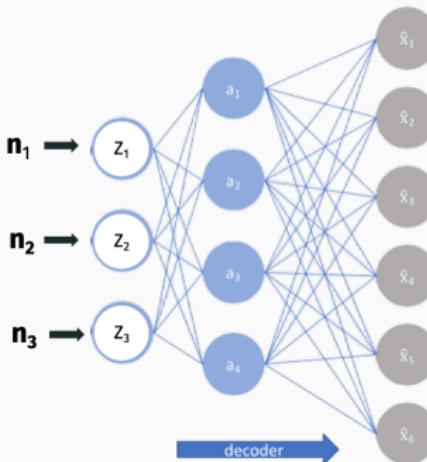
Colorization task

Puzzle task

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

# GENERATIVE ADVERSARIAL NETWORKS

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



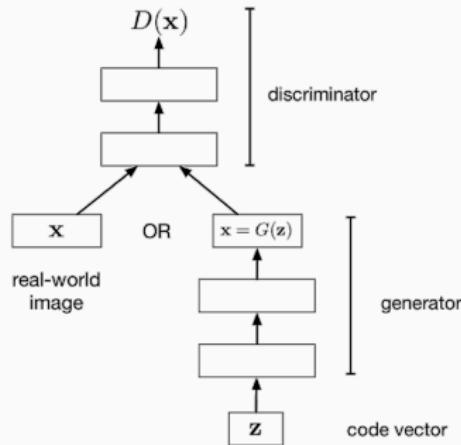
Pretty cool, but tends to produce images with immediately recognizable flaws (e.g. soft edges, high-frequency artifacts).

## GENERATIVE ADVERSARIAL NETWORKS (GANs)

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

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

# GENERATIVE ADVERSARIAL NETWORKS (GANS)



Let  $x_1, \dots, x_n$  be real images and let  $z_1, \dots, z_m$  be random code vectors.

Train weights of discriminator  $D$  to maximize:

$$\sum_{i=1}^n$$