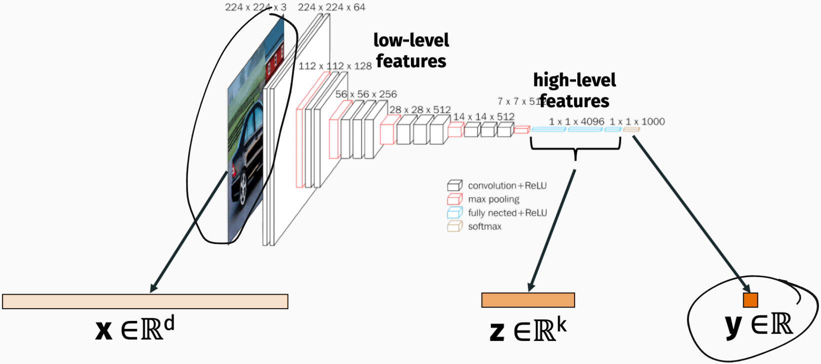
CS-UY 4563: Lecture 21
Auto-encoders, Principal Component Analysis

NYU Tandon School of Engineering, Prof. Christopher Musco

COURSE LOGISTICS

- Next weeks should be focused on project work! Final report due **5/11**.
- I am still working through proposals. If you feel blocked/need my input to move forward on project, please email or come to office hours.
- Each group will give a **5 minute presentation** in class on **5/6** or **5/11**. Link for signing up for a slot is on the course webpage.
- Details on expectations for presentation will be released soon.

Machine learning algorithms like neural networks learn high level features.

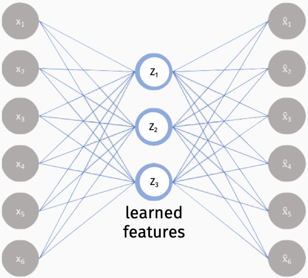


$\mathbf{x} \in \mathbb{R}^d$ $\mathbf{z} \in \mathbb{R}^k$ $\mathbf{y} \in \mathbb{R}$

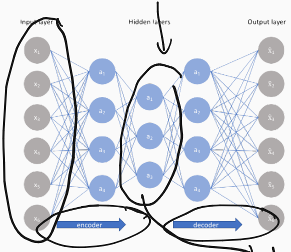
These features are useful for other tasks that the network was not trained specifically to solve.

AUTOENCODER

Idea behind autoencoders: If you have limited labeled data, make the inputs the targets. Learn to reconstruct input data and extract high-level features along the way.

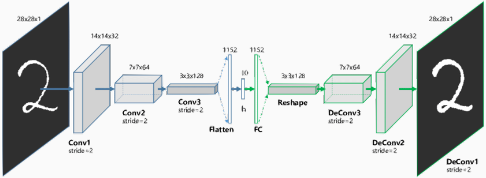


bottleneck
latent variables



d

k for $k \ll d$



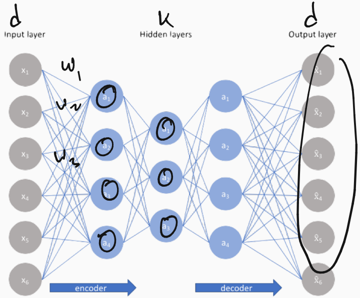
AUTOENCODER

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

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

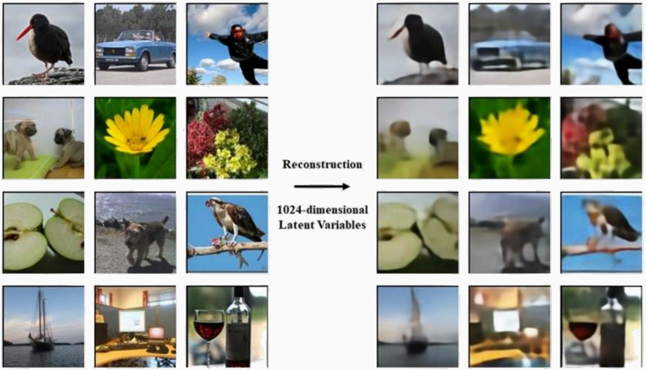
$d(e(\vec{x}))$

$$f(\vec{x}) = \cancel{e(d(\vec{x}))}$$



The number of learned features k is typically $\ll d$.

Example image reconstructions from autoencoder:



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

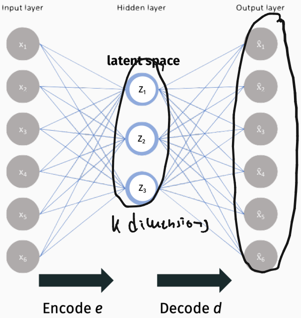
Input parameters: $d = 49152$.

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

Autoencoders also have many other applications besides feature extraction.

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

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



$f(\vec{x}) \approx \vec{x}$

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

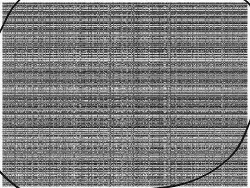
The best lossy compression algorithms are tailor made for specific types of data:

Lossless: No error. Given input $\vec{x}$; $d(e(\vec{x})) = \vec{x}$.

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

Lossy: Error. Given input $\vec{x}$, $d(e(\vec{x})) \approx \vec{x}$

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



$\vec{x} =$

$1\,0\,0\,1\,1\;\underbrace{0\,0\,0\,0\,0\,0}_{(5,0)}\;0\,1\,1\,0\,0\,0\,1\;\underbrace{0\,0\,0\,0\,0\,0}_{(6,0)}$

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

trained autoencoder

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

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-designed" algorithms like JPEG.

AUTOENCODERS FOR DATA RESTORATION

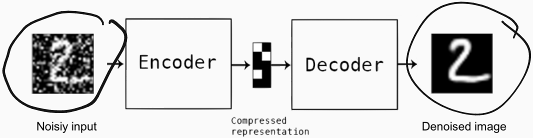


Image denoising

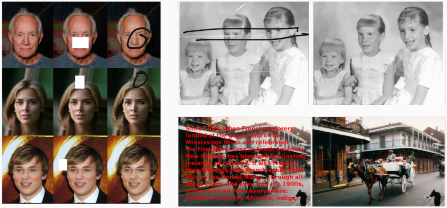
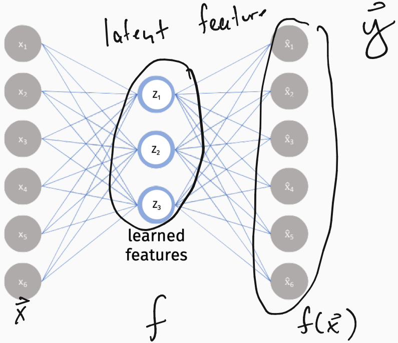


Image inpainting

Train autoencoder on uncorrupted data. Pass corrupted data $\vec{x}$ through autoencoder and return $f(\vec{x})$ as repaired result.[1]

[1]Works much better if trained on corrupted data. More on this later.

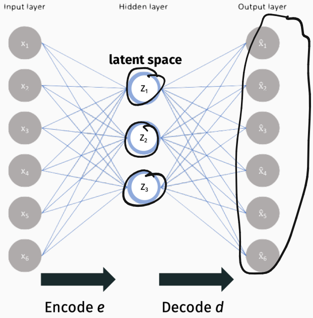
Why does this work?



Definitions:

- Let $\mathcal{A}$ be our original data space. E.g. $\mathcal{A} = \mathbb{R}^d$ for some dimension d.
- Let $\mathcal{S}$ be the set of all data examples which could be the output of our autoencoder f. We have that $\mathcal{S} \subset \mathcal{A}$. Formally, $\mathcal{S} = \{\vec{y} \in \mathbb{R}^d : \vec{y} = f(\vec{x}) \text{ for some } \vec{x} \in \mathbb{R}^d\}$.

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



Consider $128 \times 128 \times 3$ images with pixels values in $0, 1 \ldots, 255$. How many unique images are there in $\mathcal{A}$?

$$|A| = 256^{(128 \cdot 128 \cdot 3)} = O(1)^{d}$$

Suppose $\vec{z}$ holds k values between in $0, .1, .2, \ldots, 1$. Roughly how many unique images are there in $\mathcal{S}$?

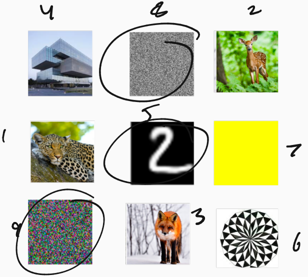
$$|S| \leq 11^{k} = O(1)^{k} \qquad |S| << |A|$$

AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

$|S| << |A|$

So, any autoencoder can only represent a tiny fraction of all possible images. This is a good thing.

Training data



Which images are likely in S?

1 ⟶ 9

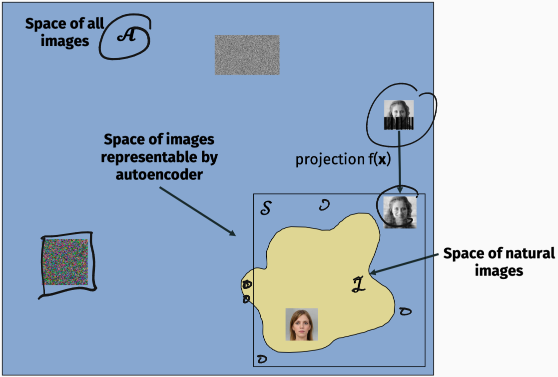
more likely in S

less likely in S

training images $\subseteq$ S

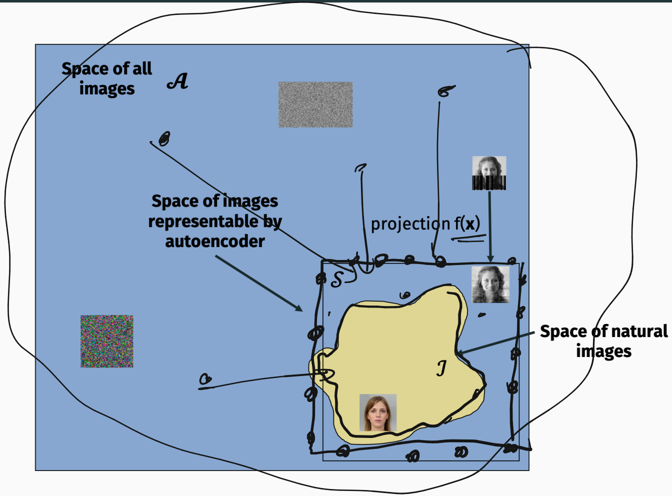
AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS

$$\mathcal{S} = \{\vec{y} \in \mathbb{R}^d : \vec{y} = f(\vec{x}) \text{ for some } \vec{x} \in \mathbb{R}^d\}$$



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

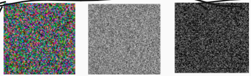
AUTOENCODERS LEARN COMPRESSED REPRESENTATIONS



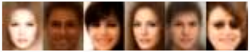
$f(\vec{x})$ projects an image $\vec{x}$ closer to the space of natural images.

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

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

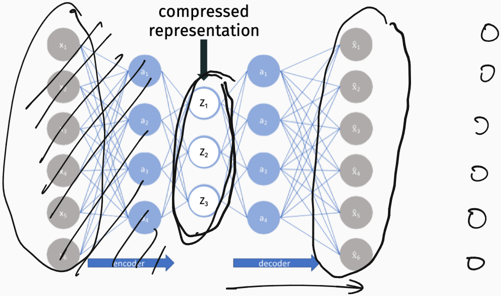


- **Option 2**: Draws $\vec{x}$ randomly image from $\mathcal{S}$.



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

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



Randomly select code $\vec{z}$, then set $\vec{x} = e(\vec{z})$.[2]

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

AUTOENCODERS FOR DATA GENERATION

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

GANs

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

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

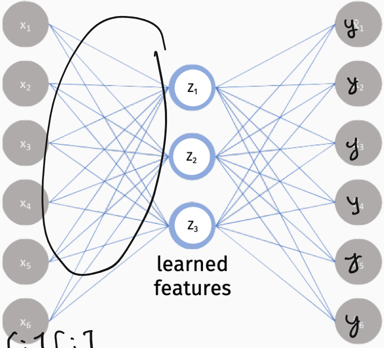
Very important outside machine learning as well.

PRINCIPAL COMPONENT ANALYSIS

Consider the simplest possible autoencoder:

$d = 6$

$k = 3$



$W_1[i][j]$

$=$ weight from x_i to z_j

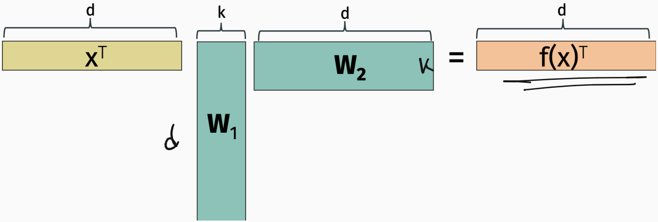
$W_2[i][j]$

$=$ weight from z_i to y_j

$$z_j = \sum_{i=1}^{d} x_i \cdot W_1[i][j]$$

- 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 $\mathbf{W}_2 \in \mathbb{R}^{k \times d}$.

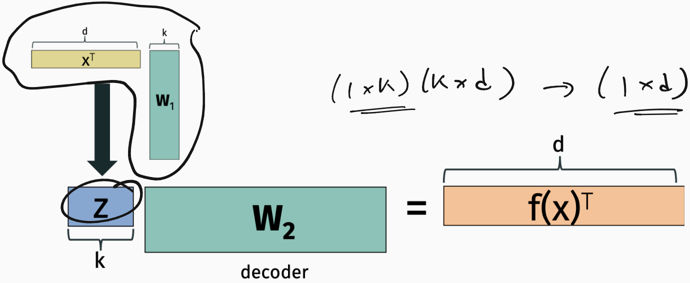
Given input $\vec{x} \in \mathbb{R}^d$, what is $f(\vec{x})$ expressed in linear algebraic terms?



(1 x d) (d x k) (k x d) = (1 x d)

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

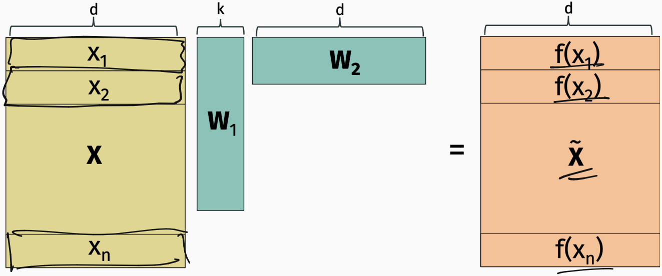
$(1 \times d)(d \times k) \to (1 \times k)$



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

Encoder: $e(\vec{x}) = \vec{x}^T W_1$.   Decoder: $d(\vec{z}) = \vec{z} W_2$

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



Goal of training autoencoder: Learn weights (i.e. learn matrices $\mathbf{W}_1\mathbf{W}_2$) so that $\tilde{\mathbf{X}}$ is as close to $\mathbf{X}$ as possible.

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

$\|x_1 - f(x_1)\|_2^2$

$$L(\mathbf{X}, \tilde{\mathbf{X}}) = \sum_{i=1}^{n} \|\vec{x}_i - f(\vec{x}_i)\|_2^2$$
$$= \sum_{i=1}^{n}\sum_{j=1}^{d} (\vec{x}_i[j] - f(\vec{x}_i)[j])^2 \quad = \sum_{i=1}^{n}\sum_{j=1}^{d} |X_{ij} - \hat{X}_{ij}|^2$$
$$= \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2$$

Recall that for a matrix $\mathbf{M}$, $\|\mathbf{M}\|_F^2$ is called the Frobenius norm. $\|\mathbf{M}\|_F^2 = \sum_{i,j} \mathbf{M}_{i,j}^2$.

Question: How should we find $\mathbf{W}_1, \mathbf{W}_2$ to minimize $\|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 = \|\mathbf{X} - \mathbf{X}\mathbf{W}_1\mathbf{W}_2\|_F^2$?

LOW-RANK APPROXIMATION

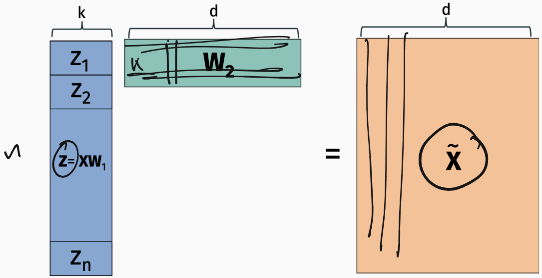
Recall:

- 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**.

k rank

$(n \times d)(d \times k)$

$(n \times k)$

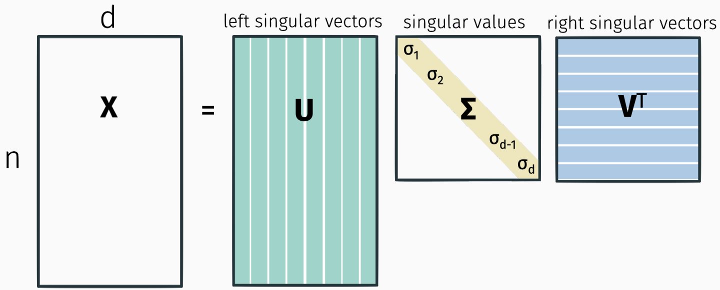


$\tilde{X}$ is a low-rank matrix since it has rank k for $k \ll d$.

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.

In general, $\mathbf{X}$ will have rank d.

Any matrix **X** can be written:

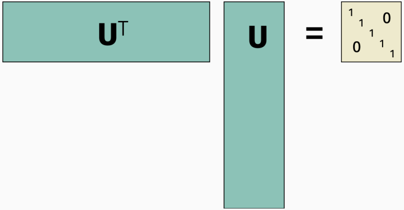


Where $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_d \geq 0$. I.e. **U** and **V** are orthogonal matrices.

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

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

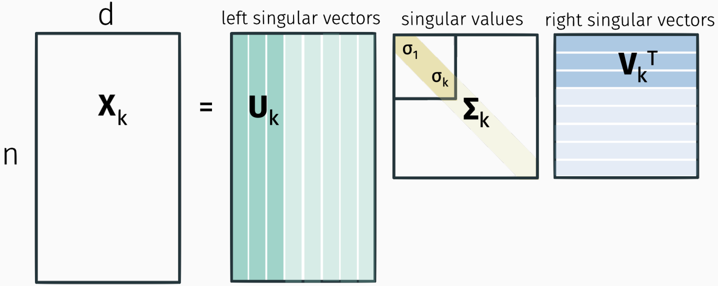
Let $\mathbf{u}_1, \ldots, \mathbf{u}_n \in \mathbb{R}^n$ denote the columns of $\mathbf{U}$. I.e. the top left singular vectors of $\mathbf{X}$.



$\|u_i\|_2^2 =$

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

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

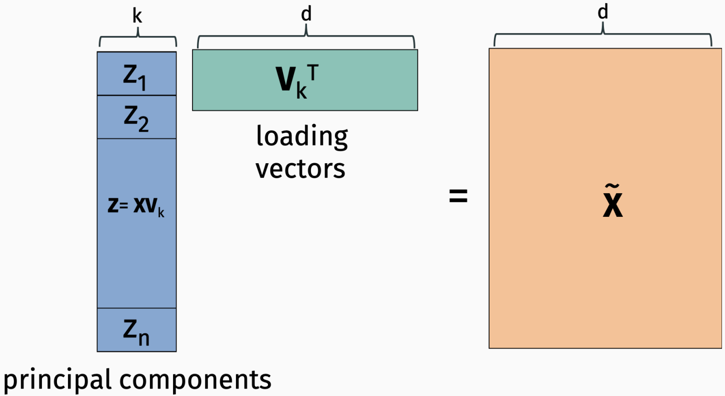


Eckart–Young–Mirsky Theorem: For any $k \leq d$, $\mathbf{X}_k = \mathbf{U}_k \boldsymbol{\Sigma}_k \mathbf{V}_k^T$ is the optimal k rank approximation to $\mathbf{X}$:

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

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

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



To be continued...