Machine Learning 2

DS 4420 - Spring 2020

Dimensionality reduction 2

Byron C Wallace

Today

- A bit of wrap up on PCA
- **Then**: Non-linear dimensionality reduction! (SNE/t-SNE)

In Sum: Principal Component Analysis

Data

$$\mathbf{X} = \begin{pmatrix} | & & | \\ \mathbf{x}_1 & \cdots\cdots & \mathbf{x}_n \\ | & & | \end{pmatrix} \in \mathbb{R}^{d \times n}$$

Eigenvectors of Covariance

$$\mathbf{C} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{x}_j \mathbf{x}_j^\top = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$$

$$\mathbf{C}\mathbf{u}_j = \lambda_j \mathbf{u}_j$$

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_d \end{pmatrix}$$

Idea: Take top-*k* eigenvectors to maximize variance

Why?

$$\mathbf{C} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{x}_j \mathbf{x}_j^\top = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$$

$$\mathbf{C}\mathbf{u}_j = \lambda_j \mathbf{u}_j$$

Idea: Take top-*k* eigenvectors to maximize variance

Last time, we saw that we can derive this by *maximizing the variance in the compressed space*

Why?

$$\mathbf{C} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{x}_j \mathbf{x}_j^\top = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$$

$$\mathbf{C}\mathbf{u}_j = \lambda_j \mathbf{u}_j$$

Idea: Take top-*k* eigenvectors to maximize variance

Last time, we saw that we can derive this by *maximizing the variance in the compressed space*

Can also motivate by explicitly minimizing *reconstruction error*

Minimizing *reconstruction error*

Getting the eigenvalues, two ways

- Direct eigenvalue decomposition of the covariance matrix

$$\boldsymbol{S} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^\top$$

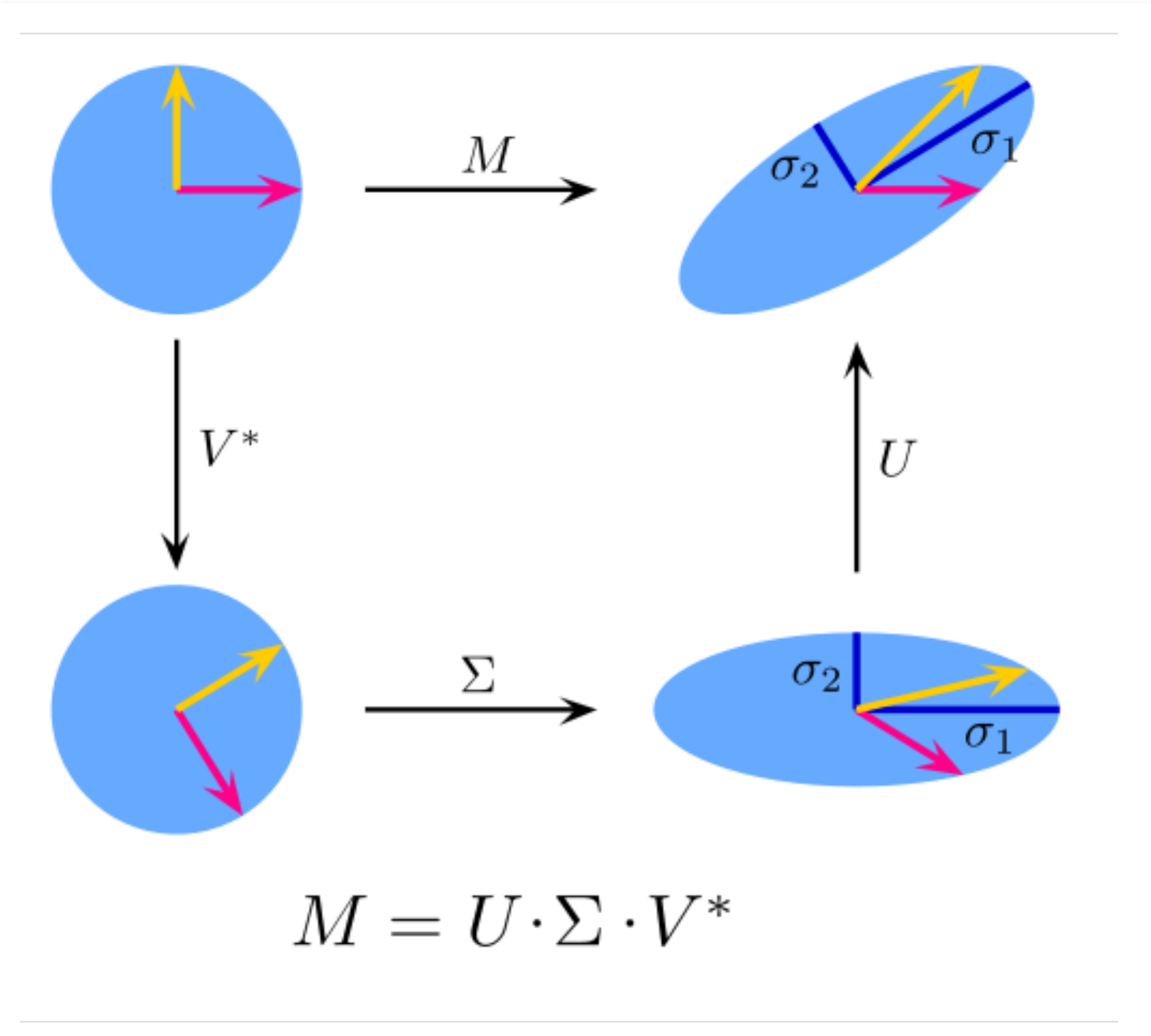
Getting the eigenvalues, two ways

- Direct eigenvalue decomposition of the covariance matrix

$$\boldsymbol{S} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^\top$$

- Singular Value Decomposition (SVD)

Singular Value Decomposition



Idea: Decompose the **d x n** matrix ***X*** into

1. A n x n basis *V* (unitary matrix)
2. A d x n matrix ***Σ*** (diagonal projection)
3. A d x d basis ***U*** (unitary matrix)

$$\mathbf{X} = \mathbf{U}_{d\times d}\Sigma_{d\times n}\mathbf{V}^{\top}_{n\times n}$$

1. Rotation

$$V^T \vec{x} = \sum_{i=1}^{n} \langle \vec{v}_i, \vec{x} \rangle \, \vec{e}_i$$

2. Scaling

$$SV^T \vec{x} = \sum_{i=1}^{n} s_i \, \langle \vec{v}_i, \vec{x} \rangle \, \vec{e}_i$$

3. Rotation

$$USV^T \vec{x} = \sum_{i=1}^{n} s_i \, \langle \vec{v}_i, \vec{x} \rangle \, \vec{u}_i$$

SVD for PCA

$$\underbrace{\boldsymbol{X}}_{D\times N} = \underbrace{\boldsymbol{U}}_{D\times D}\underbrace{\boldsymbol{\Sigma}}_{D\times N}\underbrace{\boldsymbol{V}^\top}_{N\times N}$$

$$\boldsymbol{S} = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^\top = \frac{1}{N}\boldsymbol{U}\boldsymbol{\Sigma}\underbrace{\boldsymbol{V}^\top\boldsymbol{V}}_{=\boldsymbol{I}_N}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top = \frac{1}{N}\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top$$

SVD for PCA

$$\underbrace{\boldsymbol{X}}_{D\times N} = \underbrace{\boldsymbol{U}}_{D\times D}\underbrace{\boldsymbol{\Sigma}}_{D\times N}\underbrace{\boldsymbol{V}^\top}_{N\times N}$$

$$\boldsymbol{S} = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^\top = \frac{1}{N}\boldsymbol{U}\boldsymbol{\Sigma}\underbrace{\boldsymbol{V}^\top\boldsymbol{V}}_{=\boldsymbol{I}_N}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top = \frac{1}{N}\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top$$

It turns out the columns of $\boldsymbol{U}$ are the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$

Computing PCA

Method 1: eigendecomposition

$\mathbf{U}$ are eigenvectors of covariance matrix $C = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$

Computing C already takes $O(nd^2)$ time (very expensive)

Method 2: singular value decomposition (SVD)

Find $\mathbf{X} = \mathbf{U}_{d\times d}\Sigma_{d\times n}\mathbf{V}_{n\times n}^\top$

where $\mathbf{U}^\top\mathbf{U} = I_{d\times d}$, $\mathbf{V}^\top\mathbf{V} = I_{n\times n}$, Σ is diagonal

Computing top k singular vectors takes only $O(ndk)$

Computing PCA

Method 1: eigendecomposition

$\mathbf{U}$ are eigenvectors of covariance matrix $C = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$

Computing C already takes $O(nd^2)$ time (very expensive)

Method 2: singular value decomposition (SVD)

Find $\mathbf{X} = \mathbf{U}_{d\times d}\Sigma_{d\times n}\mathbf{V}_{n\times n}^\top$

where $\mathbf{U}^\top\mathbf{U} = I_{d\times d}$, $\mathbf{V}^\top\mathbf{V} = I_{n\times n}$, Σ is diagonal

Computing top k singular vectors takes only $O(ndk)$

Relationship between eigendecomposition and SVD:

Left singular vectors are principal components ($C = \mathbf{U}\Sigma^2\mathbf{U}^\top$)

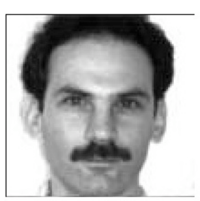
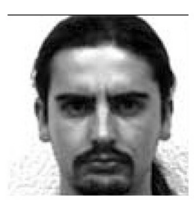
Eigen-faces [Turk & Pentland 1991]

- $d =$ number of pixels
- Each $\mathbf{x}_i \in \mathbb{R}^d$ is a face image
- $\mathbf{x}_{ji} =$ intensity of the j-th pixel in image i

Eigen-faces [Turk & Pentland 1991]

- $d =$ number of pixels
- Each $\mathbf{x}_i \in \mathbb{R}^d$ is a face image
- $\mathbf{x}_{ji} =$ intensity of the j-th pixel in image i

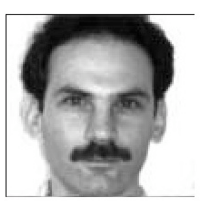
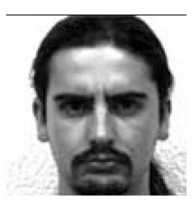
$$\mathbf{X}_{d\times n} \quad \approxeq \quad \mathbf{U}_{d\times k} \quad \mathbf{Z}_{k\times n}$$

( $\dots$ ) $\approxeq$ () $\begin{pmatrix} | & & | \\ \mathbf{z}_1 & \dots & \mathbf{z}_n \\ | & & | \end{pmatrix}$

Eigen-faces [Turk & Pentland 1991]

- d = number of pixels
- Each $\mathbf{x}_i \in \mathbb{R}^d$ is a face image
- $\mathbf{x}_{ji}$ = intensity of the j-th pixel in image i

$$\mathbf{X}_{d\times n} \quad \approxeq \quad \mathbf{U}_{d\times k} \quad \mathbf{Z}_{k\times n}$$

( $\ldots$ ) $\approxeq$ () $\begin{pmatrix} | & & | \\ \mathbf{z}_1 & \ldots & \mathbf{z}_n \\ | & & | \end{pmatrix}$

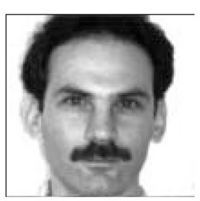
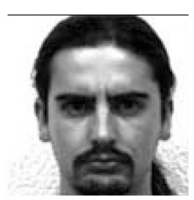
Idea: $\mathbf{z}_i$ more "meaningful" representation of i-th face than $\mathbf{x}_i$

Can use $\mathbf{z}_i$ for nearest-neighbor classification

Eigen-faces [Turk & Pentland 1991]

- d = number of pixels
- Each $\mathbf{x}_i \in \mathbb{R}^d$ is a face image
- $\mathbf{x}_{ji}$ = intensity of the j-th pixel in image i

$$\mathbf{X}_{d\times n} \approxeq \mathbf{U}_{d\times k} \quad \mathbf{Z}_{k\times n}$$

( $\dots$ ) $\approxeq$ () $\begin{pmatrix} | & & | \\ \mathbf{z}_1 & \dots & \mathbf{z}_n \\ | & & | \end{pmatrix}$

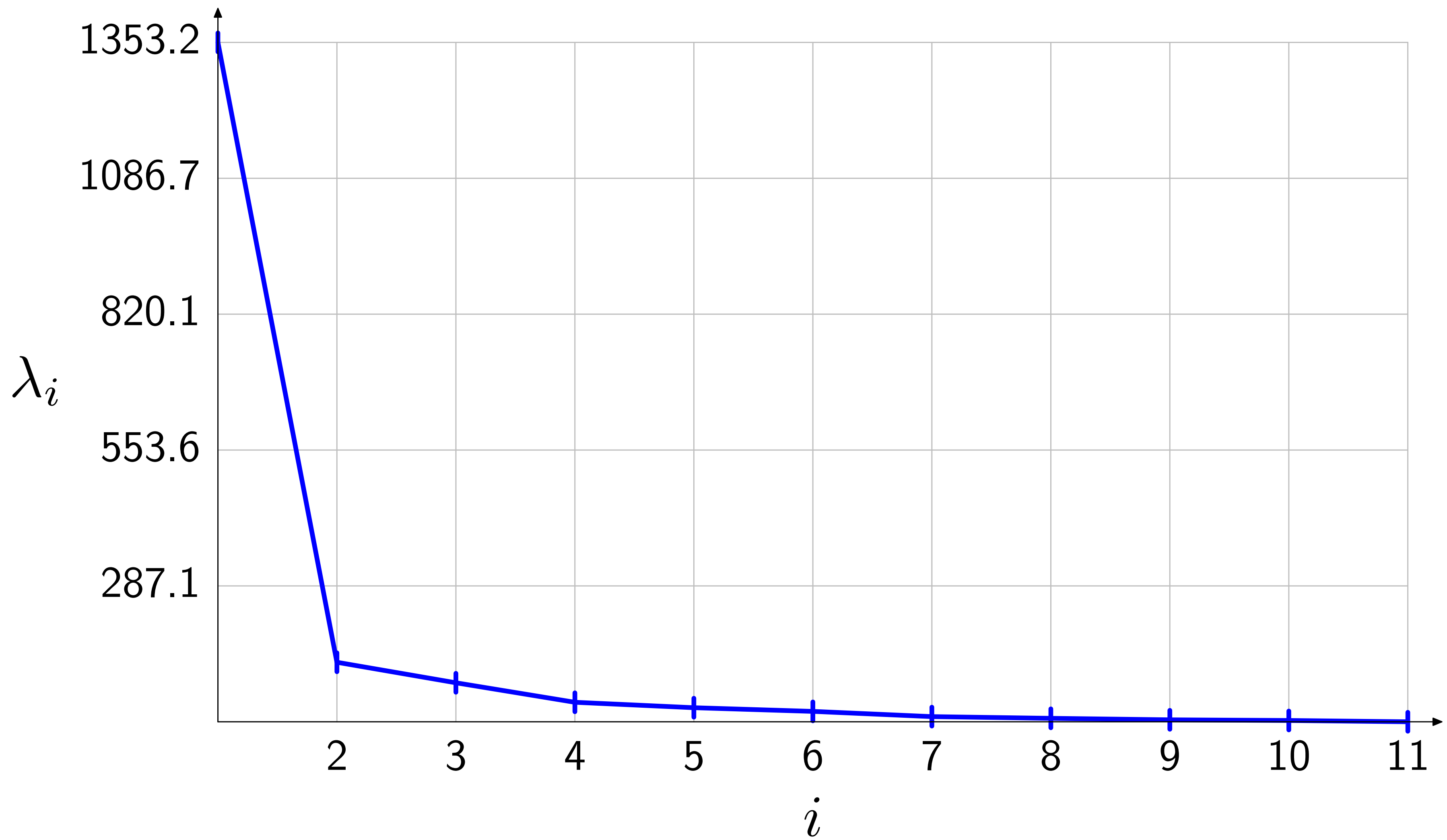
Idea: $\mathbf{z}_i$ more "meaningful" representation of i-th face than $\mathbf{x}_i$

Can use $\mathbf{z}_i$ for nearest-neighbor classification

Much faster: $O(dk + nk)$ time instead of $O(dn)$ when $n, d \gg k$

Aside: How many components?

- Magnitude of eigenvalues indicate fraction of variance captured.
- Eigenvalues on a face image dataset:



- Eigenvalues typically drop off sharply, so don't need that many.
- Of course variance isn't everything...

Wrapping up PCA

- PCA is a linear model for dimensionality reduction which finds a mapping to a lower dimensional space that maximizes variance

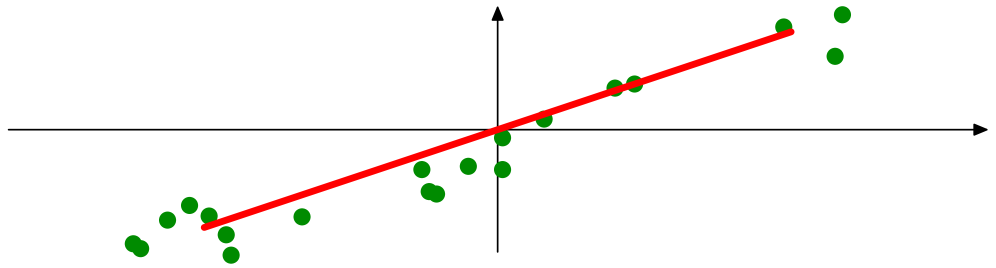
Wrapping up PCA

- PCA is a linear model for dimensionality reduction which finds a mapping to a lower dimensional space that maximizes variance

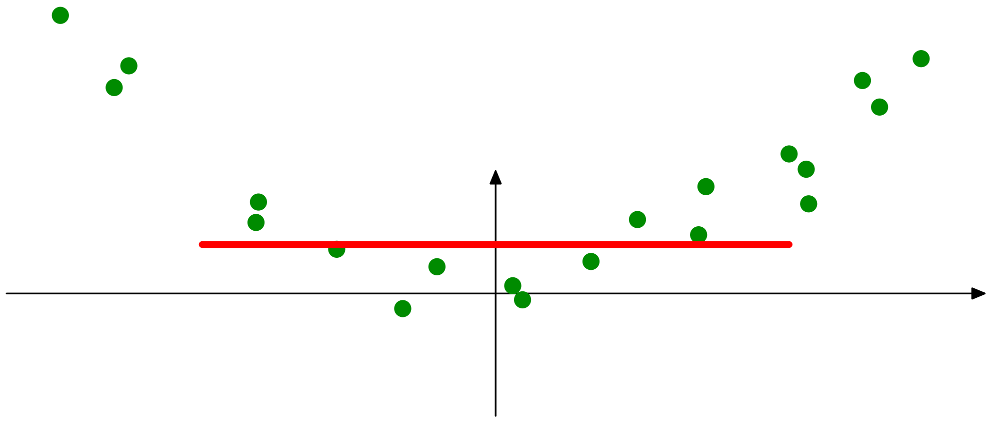
- We saw that this is equivalent to performing an eigendecomposition on the covariance matrix of $\boldsymbol{X}$

Non-linear dimensionality reduction

Limitations of Linearity

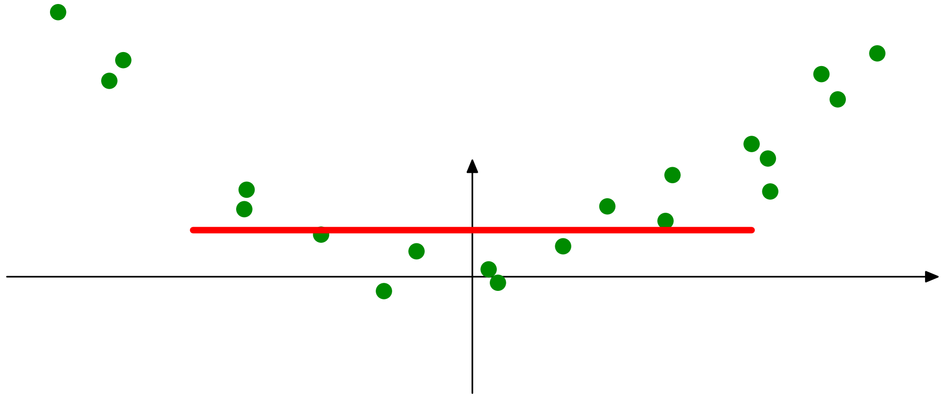


PCA is effective

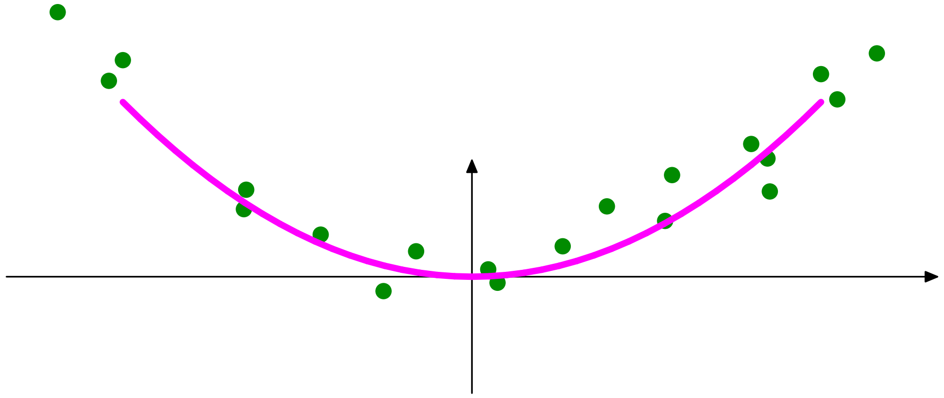


PCA is ineffective

Nonlinear PCA

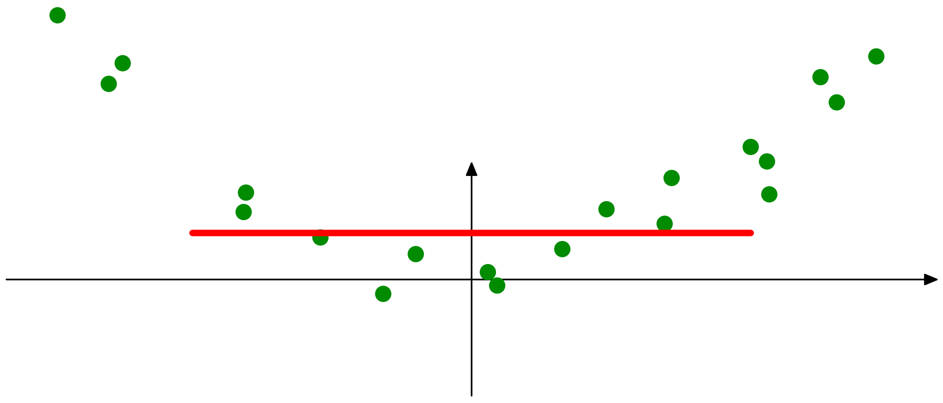


Broken solution

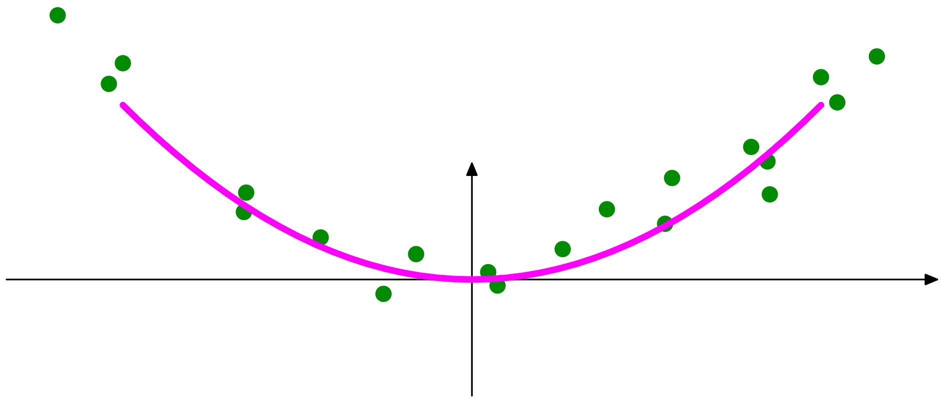


Desired solution

Nonlinear PCA



Broken solution

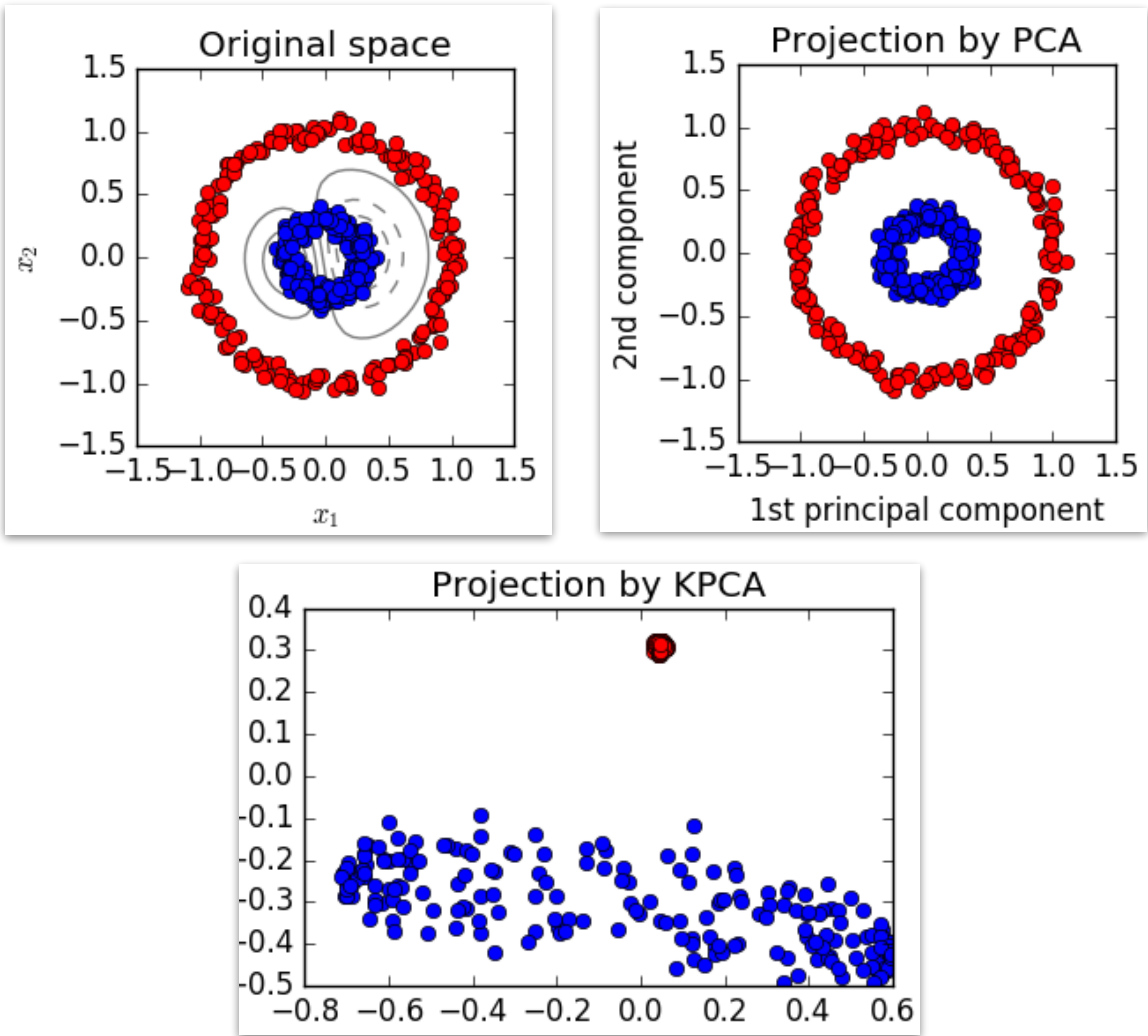


Desired solution

Idea: Use kernels

$$\boxed{\begin{array}{c}\text{Linear dimensionality reduction in } \phi(\mathbf{x}) \text{ space} \\ \Updownarrow \\ \text{Nonlinear dimensionality reduction in } \mathbf{x} \text{ space}\end{array}}$$

Kernel PCA



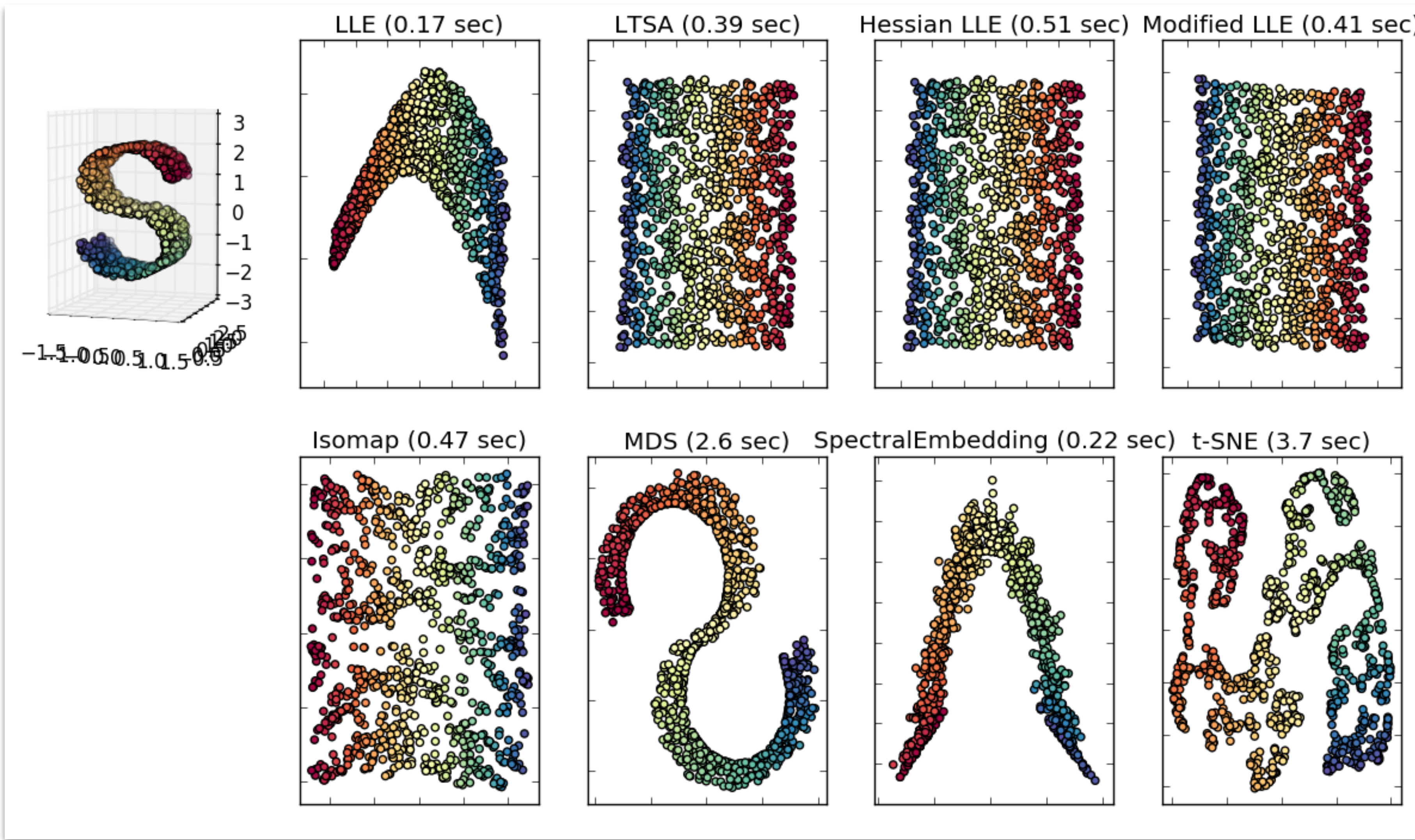
Original space
x_1
x_2
Projection by PCA
1st principal component
2nd component
Projection by KPCA

Alternatively: *t-SNE*!

Stochastic Neighbor Embeddings

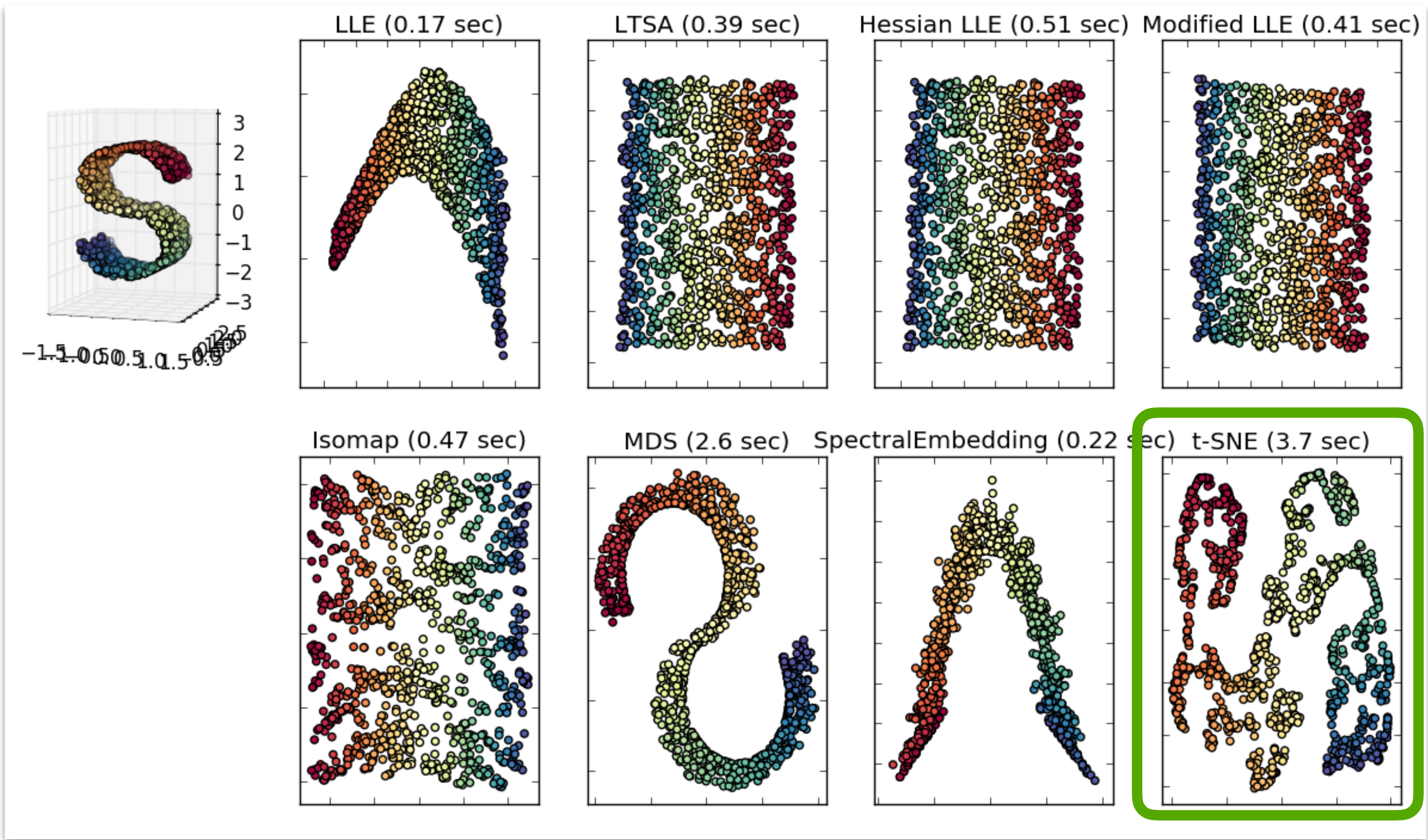
Borrowing from:
Laurens van der Maaten
(Delft -> Facebook AI)

Manifold learning



Idea: Perform a *non-linear* dimensionality reduction
in a manner that preserves proximity (but not distances)

Manifold learning



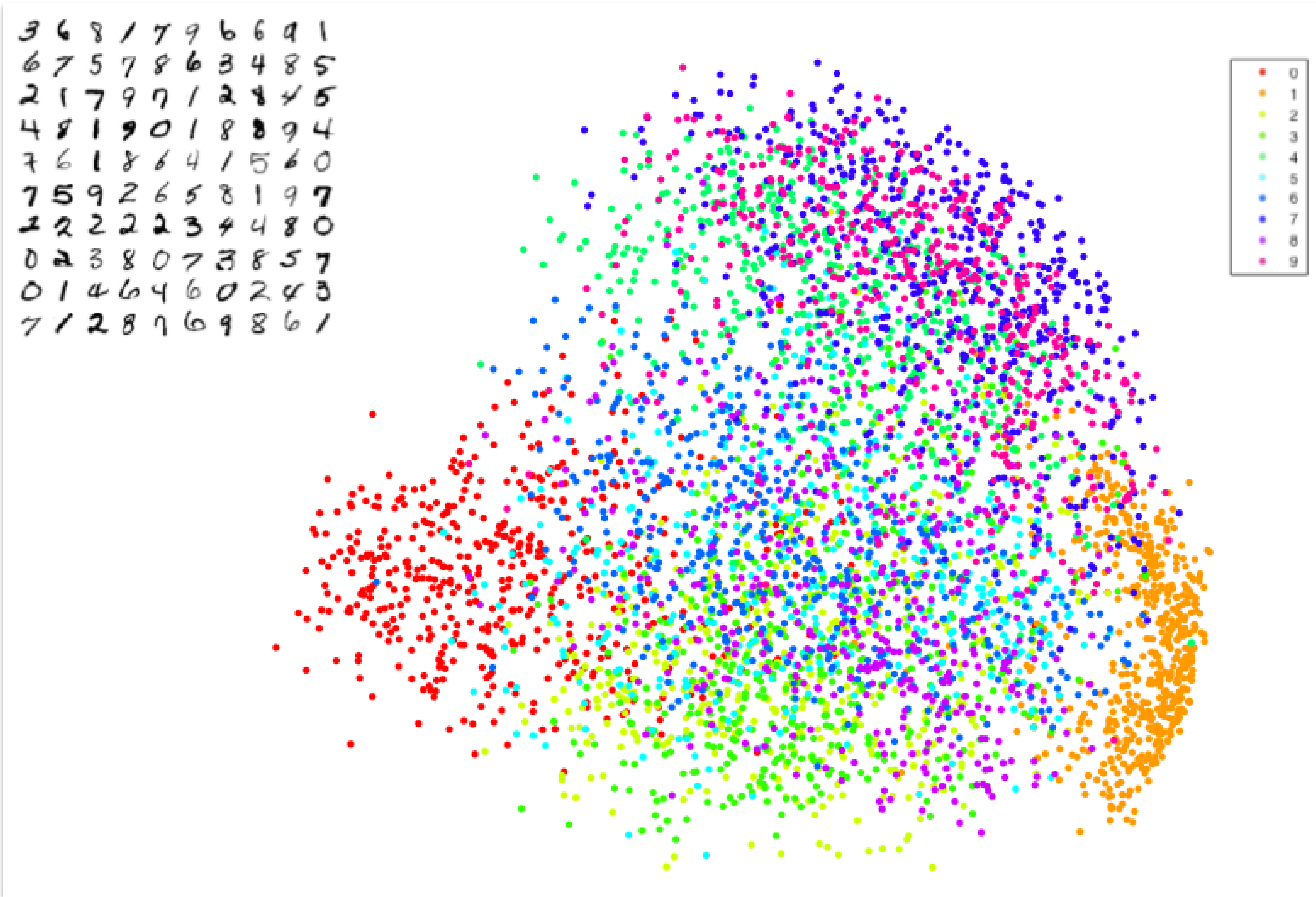
Visualizing data using **t-SNE**

L Maaten, G Hinton - Journal of machine learning research, 2008 - jmlr.org

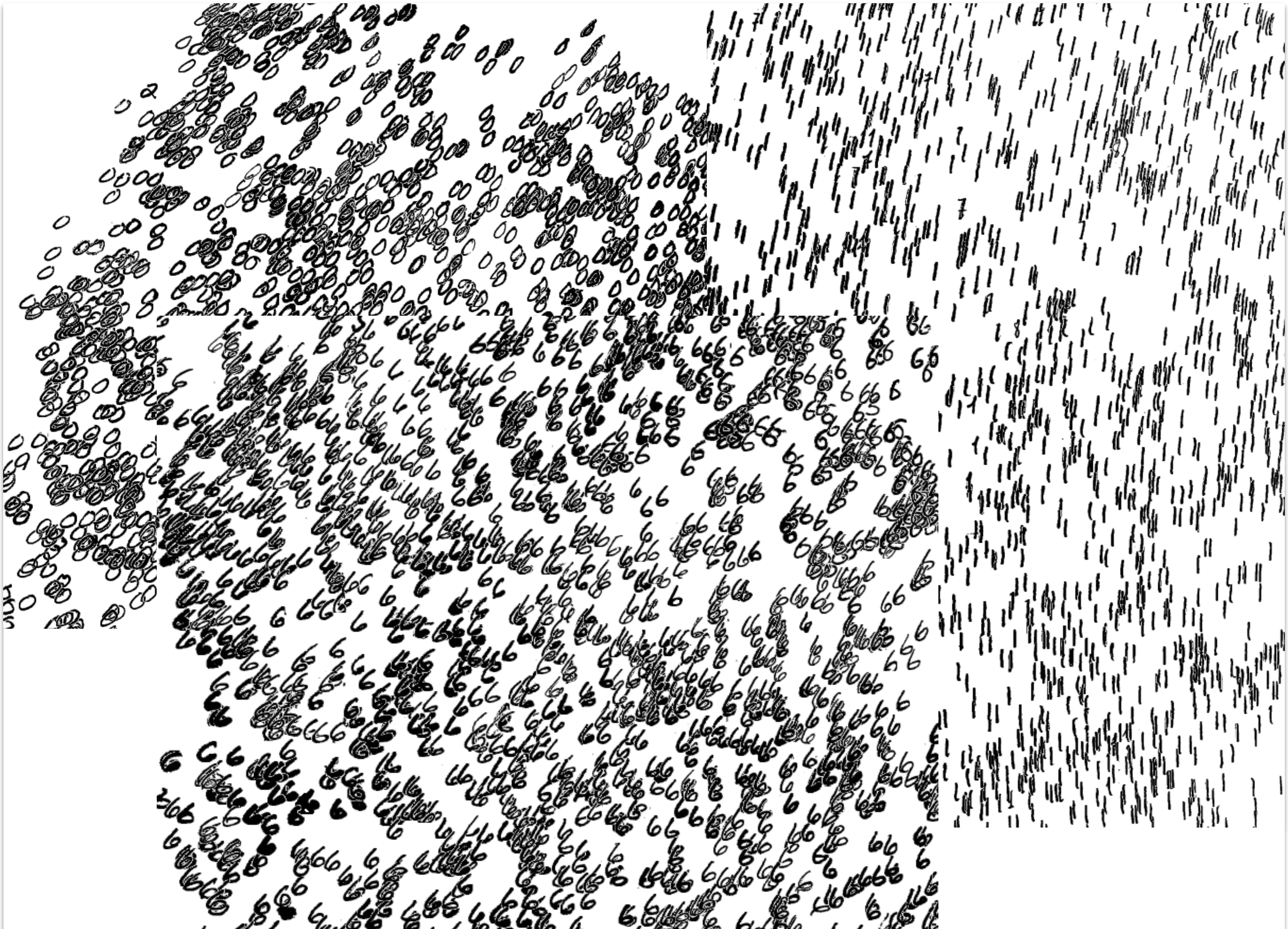
We present a new technique called" **t-SNE**" that visualizes high-dimensional data by giving each datapoint a location in a two or three-dimensional map. The technique is a variation of Stochastic Neighbor Embedding (Hinton and Roweis, 2002) that is much easier to optimize …

Cited by 11621 Related articles All 39 versions Import into BibTeX

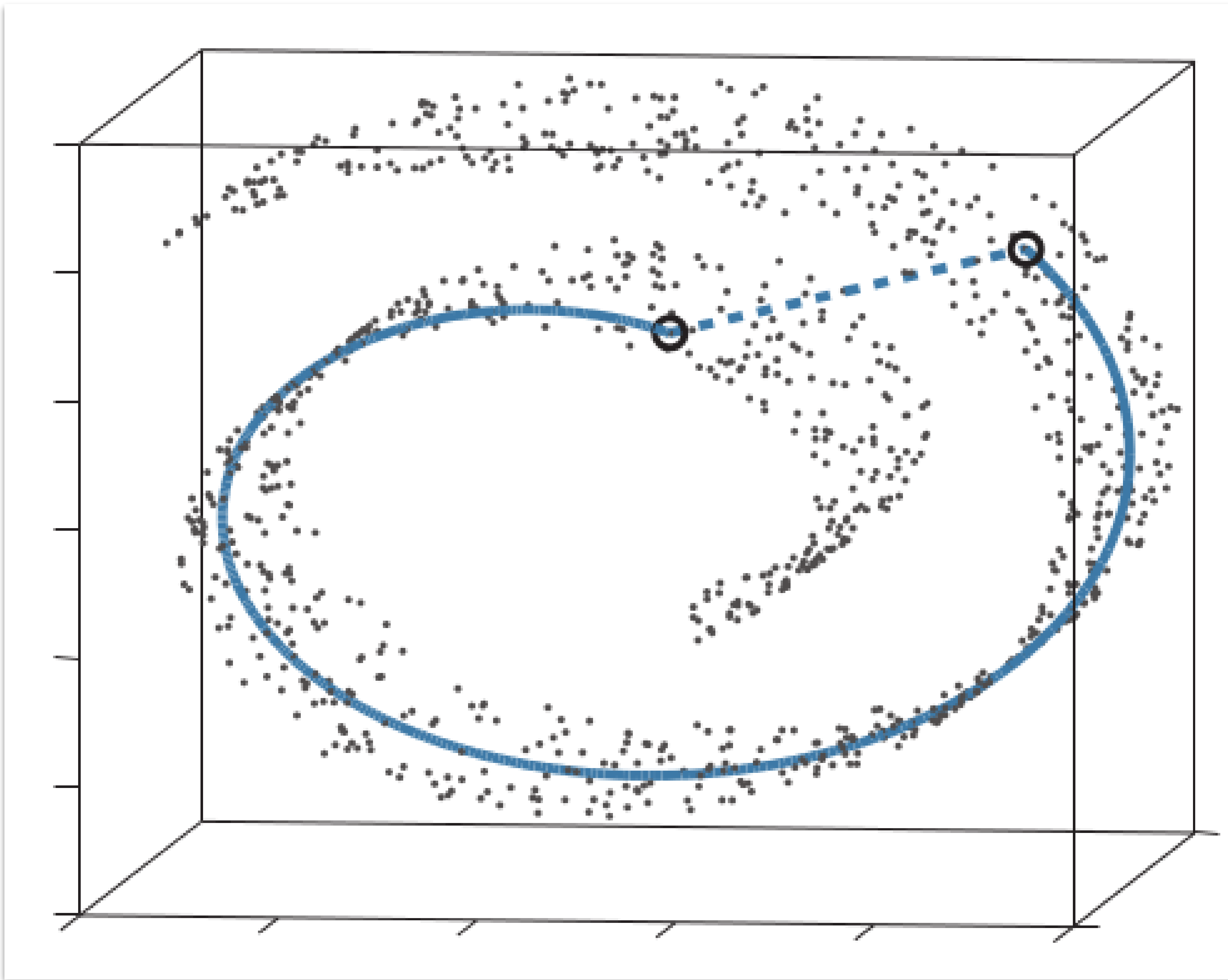
PCA on MNIST digits



t-SNE on MNIST Digits

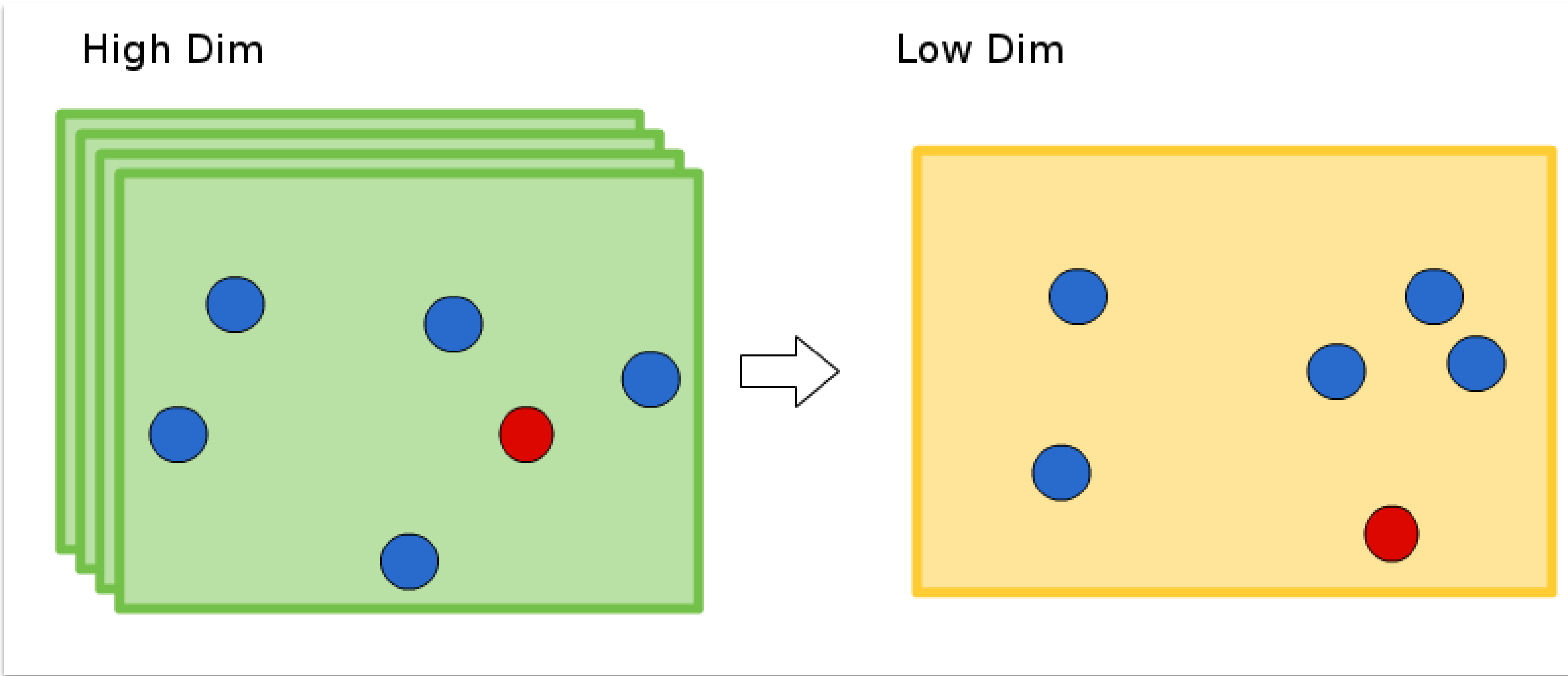


Swiss roll



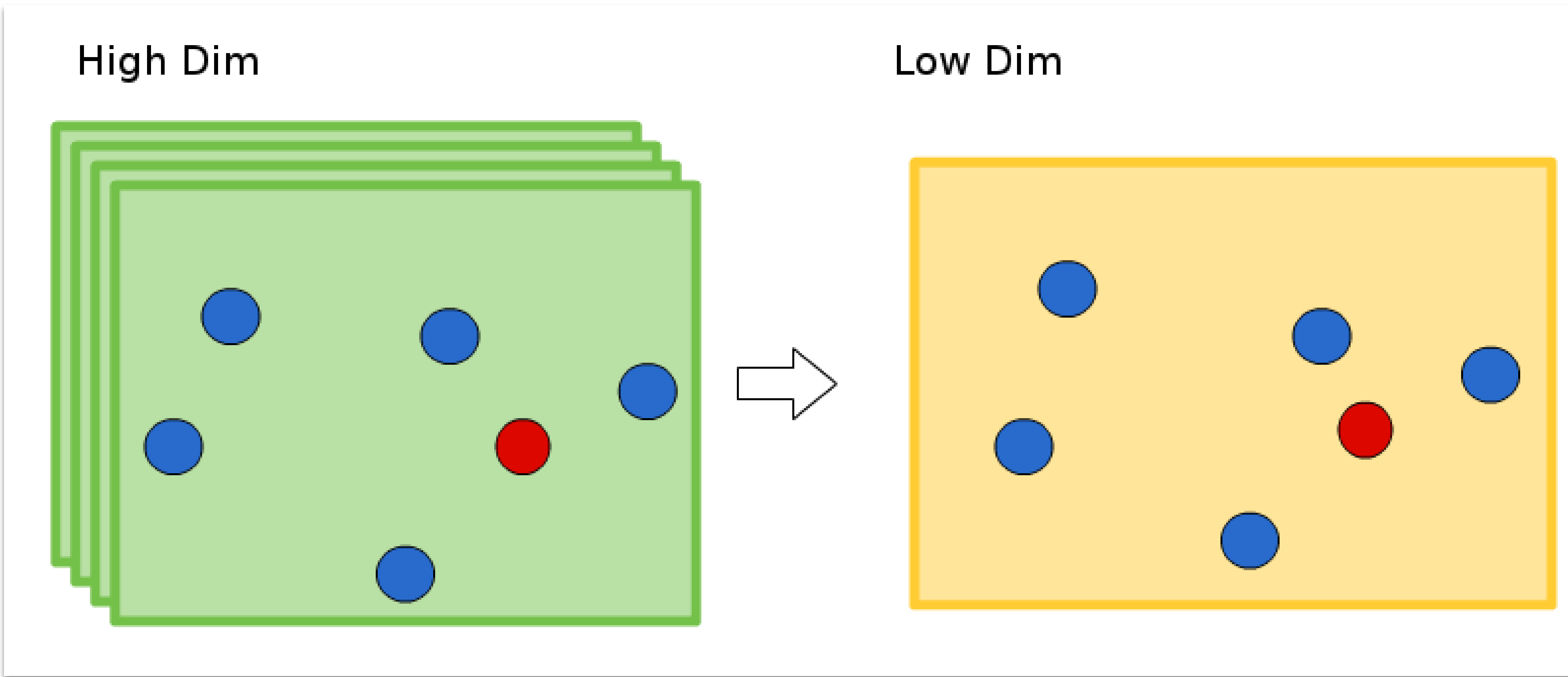
Euclidean distance is not always
a good notion of proximity

Non-linear projection



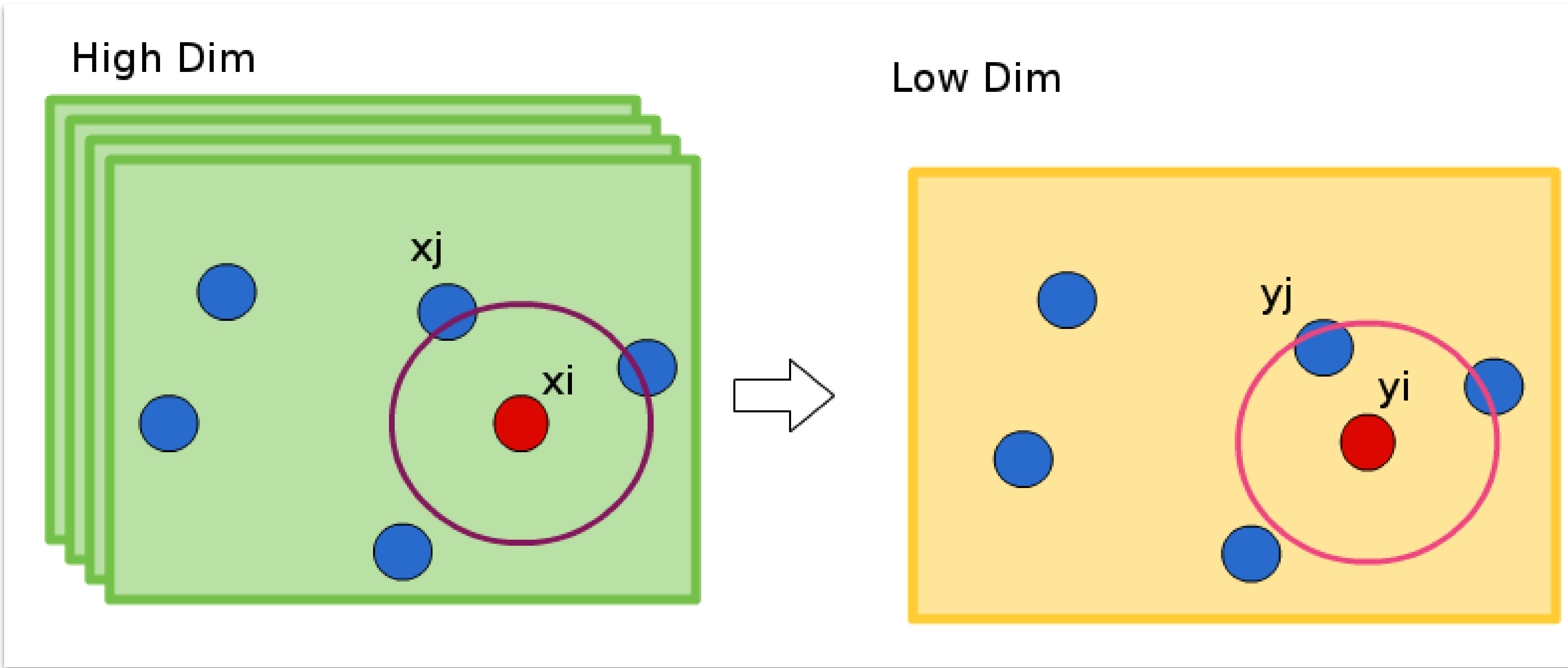
Bad projection: relative position to neighbors changes

Non-linear projection



Intuition: Want to preserve ***local*** neighborhood

Stochastic Neighbor Embedding



Original space

The *map*

SNE to t-SNE
(*on board*)

t-SNE: SNE with a t-Distribution

Similarity in High Dimension

$$p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq l} exp(-||x_l - x_k||^2/2\sigma^2)}$$

Similarity in Low Dimension

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$$

Gradient

$$\frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j)$$

Algorithm 1: Simple version of t-Distributed Stochastic Neighbor Embedding.

Data: data set $\mathcal{X} = \{x_1, x_2, ..., x_n\}$,
cost function parameters: perplexity $Perp$,
optimization parameters: number of iterations T, learning rate η, momentum $\alpha(t)$.
Result: low-dimensional data representation $\mathcal{Y}^{(T)} = \{y_1, y_2, ..., y_n\}$.
begin
compute pairwise affinities $p_{j|i}$ with perplexity $Perp$ (using Equation 1)
set $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$
sample initial solution $\mathcal{Y}^{(0)} = \{y_1, y_2, ..., y_n\}$ from $\mathcal{N}(0, 10^{-4}I)$
for *t=1* **to** T **do**
compute low-dimensional affinities q_{ij} (using Equation 4)
compute gradient $\frac{\delta C}{\delta \mathcal{Y}}$ (using Equation 5)
set $\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta \mathcal{Y}} + \alpha(t) \left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)} \right)$
end
end

Algorithm 1: Simple version of t-Distributed Stochastic Neighbor Embedding.

Data: data set $\mathcal{X} = \{x_1, x_2, ..., x_n\}$,
cost function parameters: perplexity $Perp$,
optimization parameters: number of iterations T, learning rate η, momentum $\alpha(t)$.
Result: low-dimensional data representation $\mathcal{Y}^{(T)} = \{y_1, y_2, ..., y_n\}$.
begin
  compute pairwise affinities $p_{j|i}$ with perplexity $Perp$ (using Equation 1)
  set $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$
  sample initial solution $\mathcal{Y}^{(0)} = \{y_1, y_2, ..., y_n\}$ from $\mathcal{N}(0, 10^{-4}I)$
  for *t=1* **to** T **do**
    compute low-dimensional affinities q_{ij} (using Equation 4)
    compute gradient $\frac{\delta C}{\delta \mathcal{Y}}$ (using Equation 5)
    set $\boxed{\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta \mathcal{Y}} + \alpha(t) \left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)} \right)}$
  end
end

$$\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta \mathcal{Y}} + \alpha(t) \left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)} \right)$$

$$\boxed{\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\delta C}{\delta \mathcal{Y}}} + \alpha(t)\left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)}\right)$$

Regular gradient descent

“momentum”

$$\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta\frac{\delta C}{\delta \mathcal{Y}} + \boxed{\alpha(t)\left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)}\right)}$$

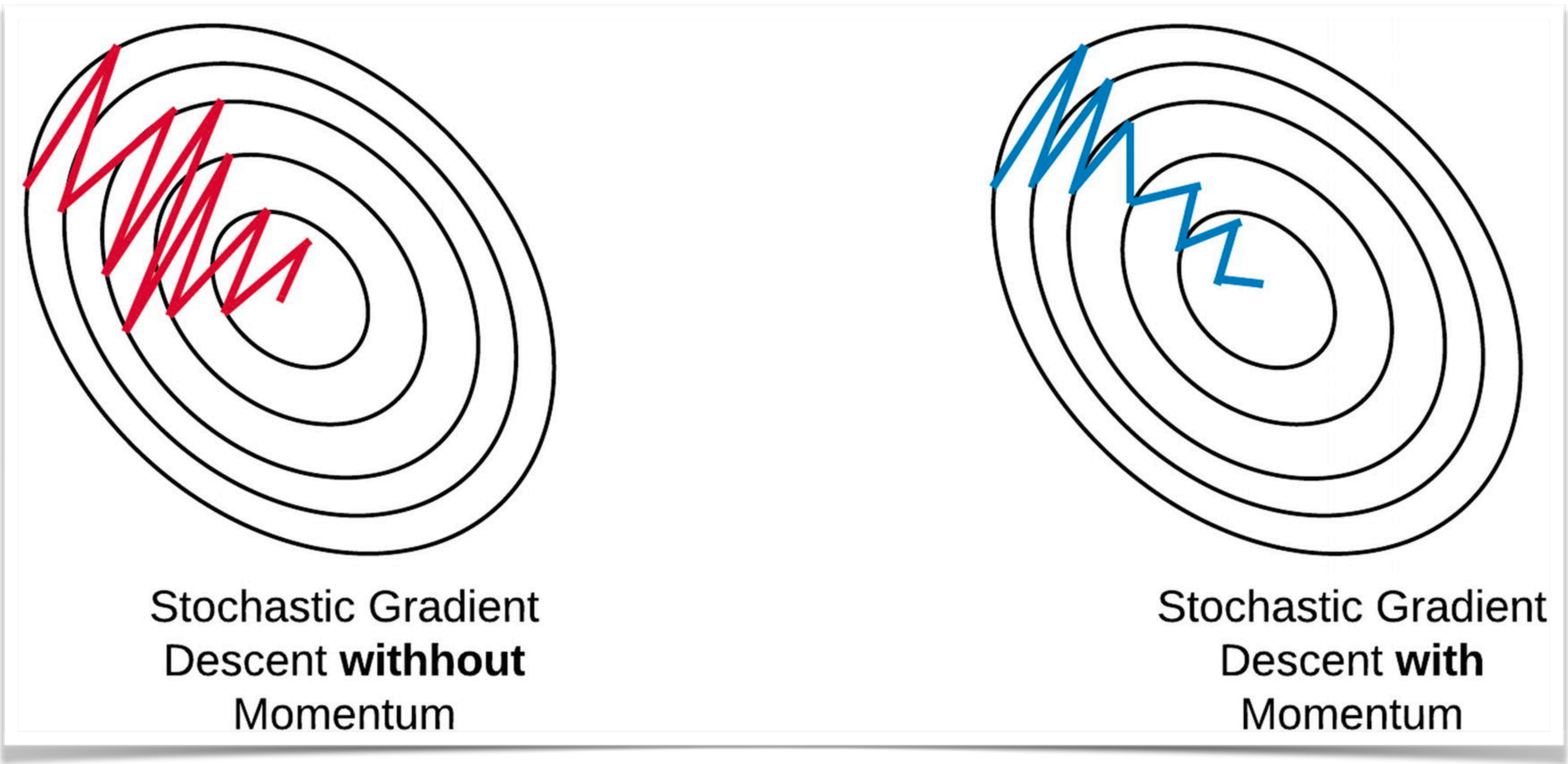
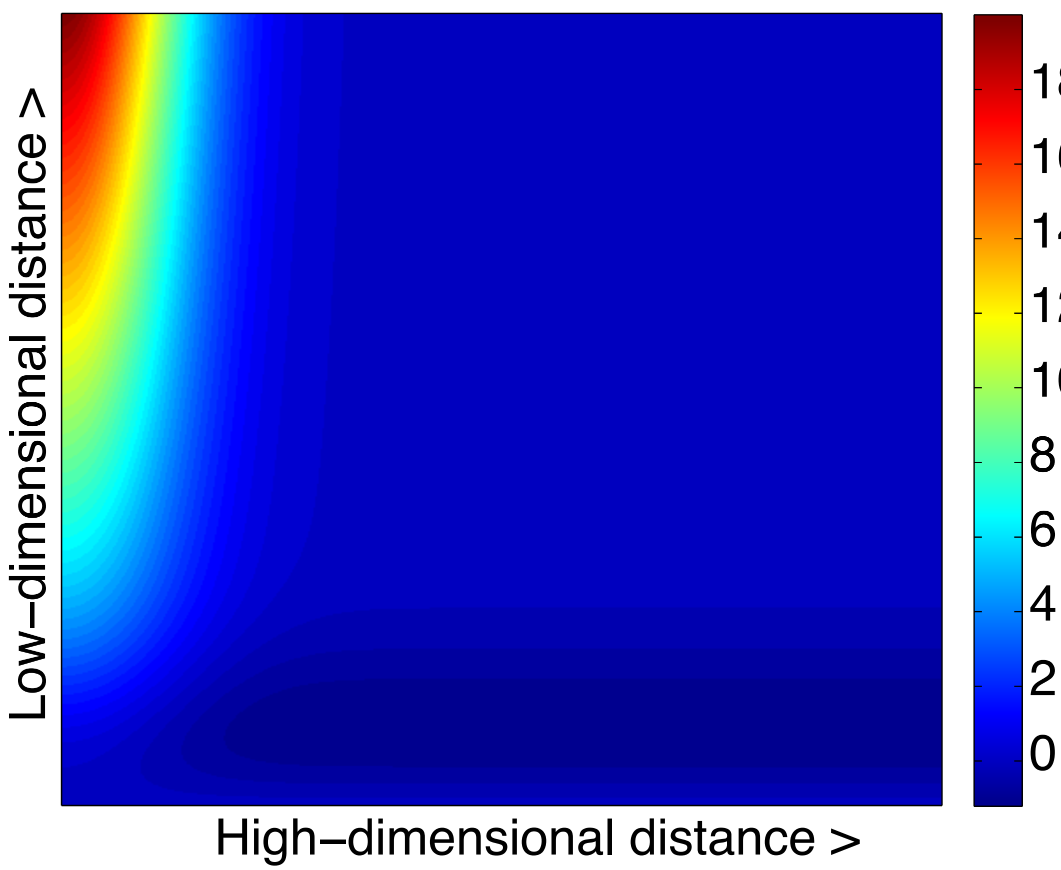


Figure credit: Bisong, 2019

What's magical about *t*?

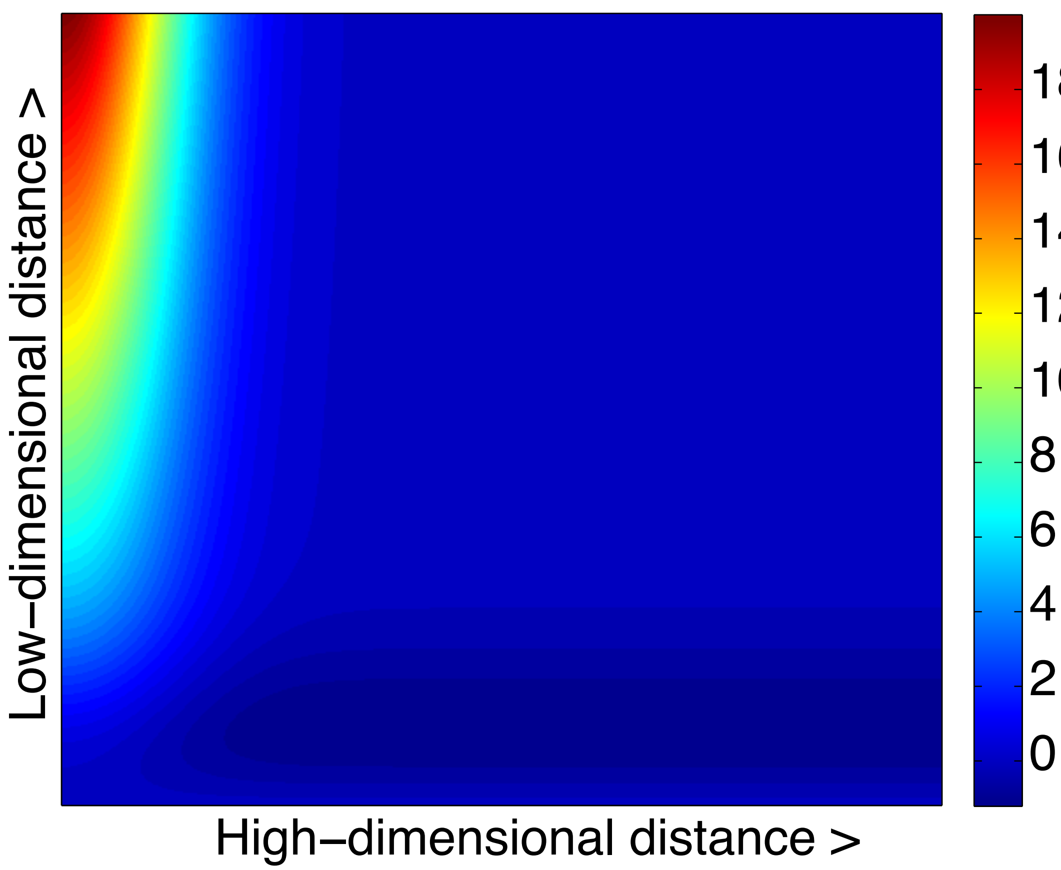
Basically, the gradient has nice properties



(a) Gradient of SNE.

What's magical about *t*?

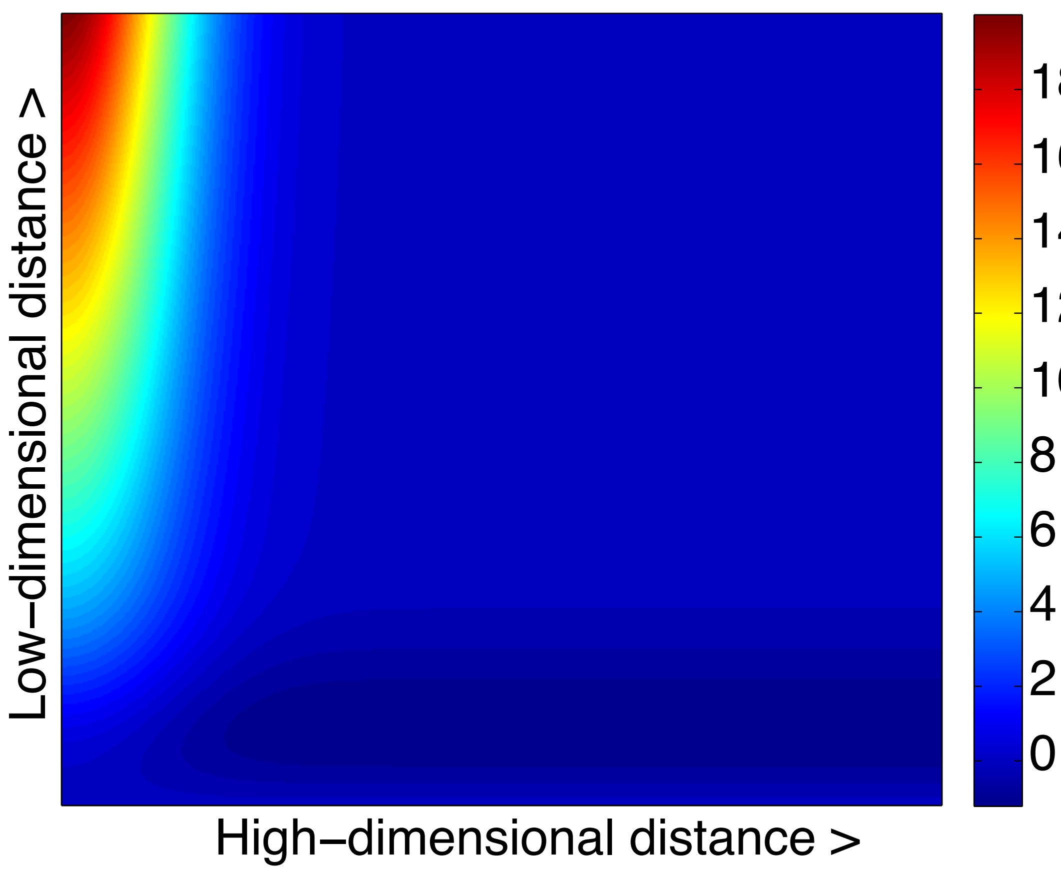
Basically, the gradient has nice properties



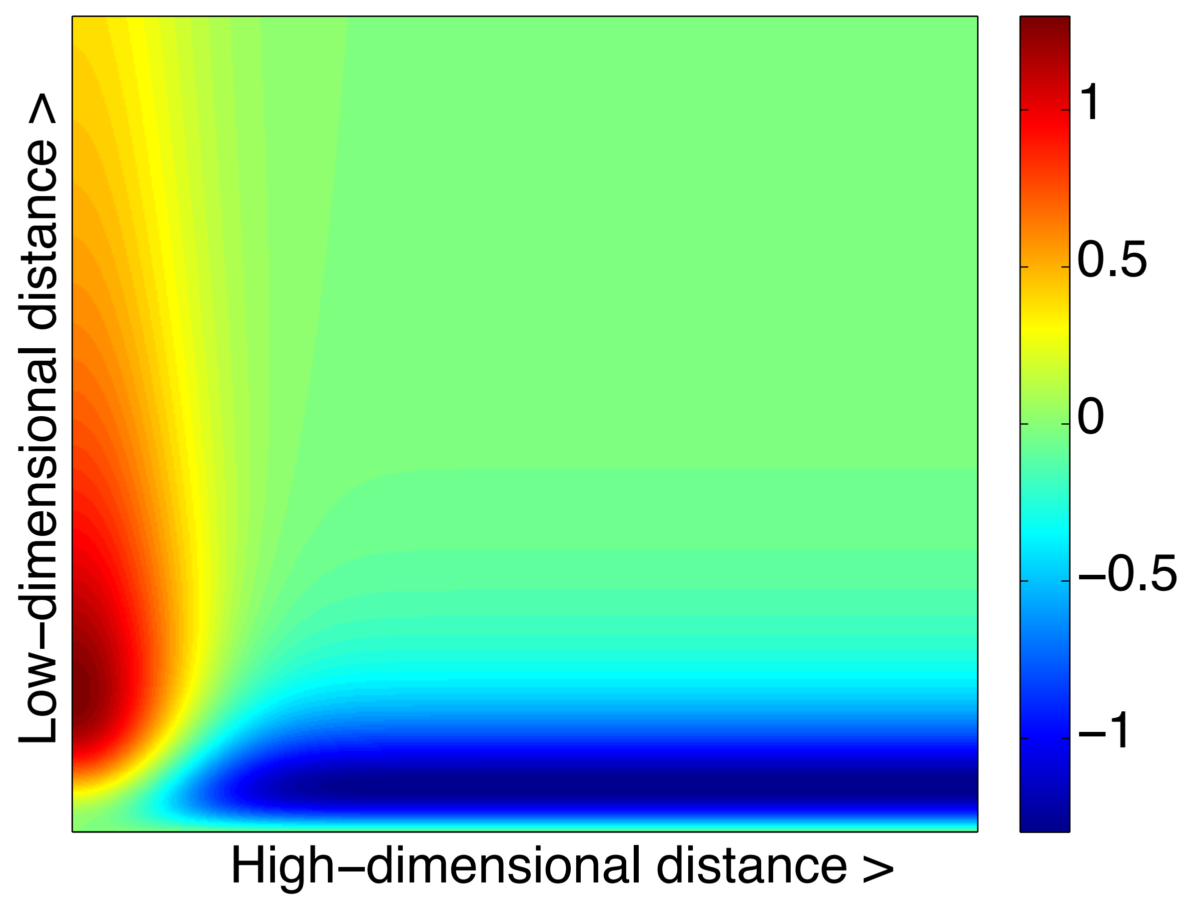
(a) Gradient of SNE.

Positive gradient —> "attraction" between points

What's magical about *t*?



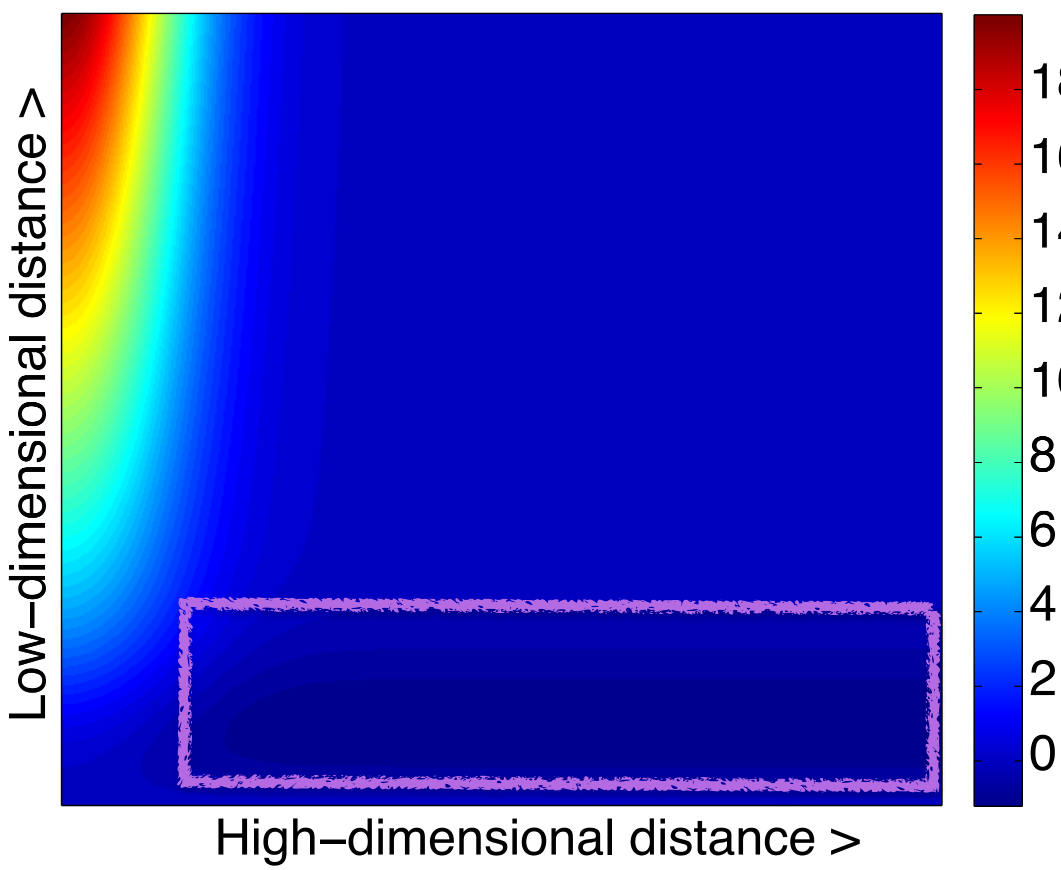
(a) Gradient of SNE.



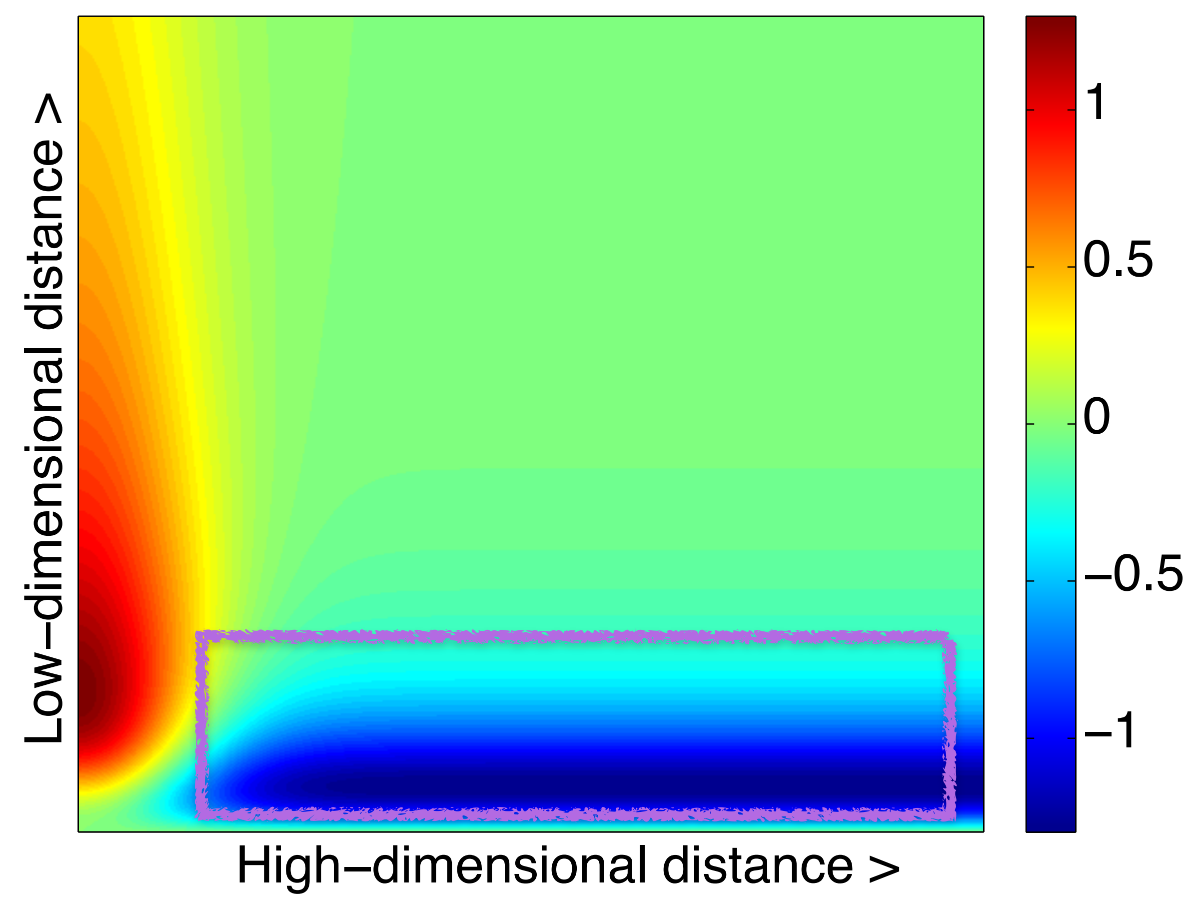
(c) Gradient of t-SNE.

Positive gradient —> "attraction" between points

What's magical about *t*?



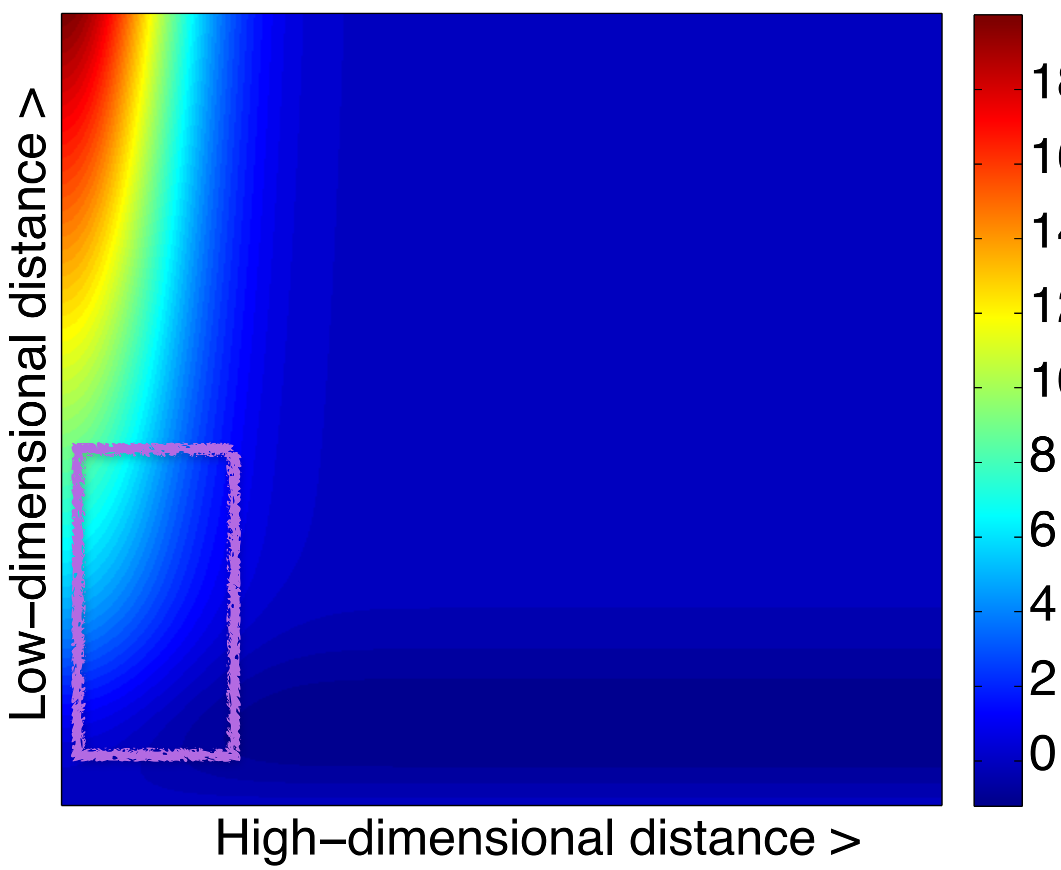
(a) Gradient of SNE.



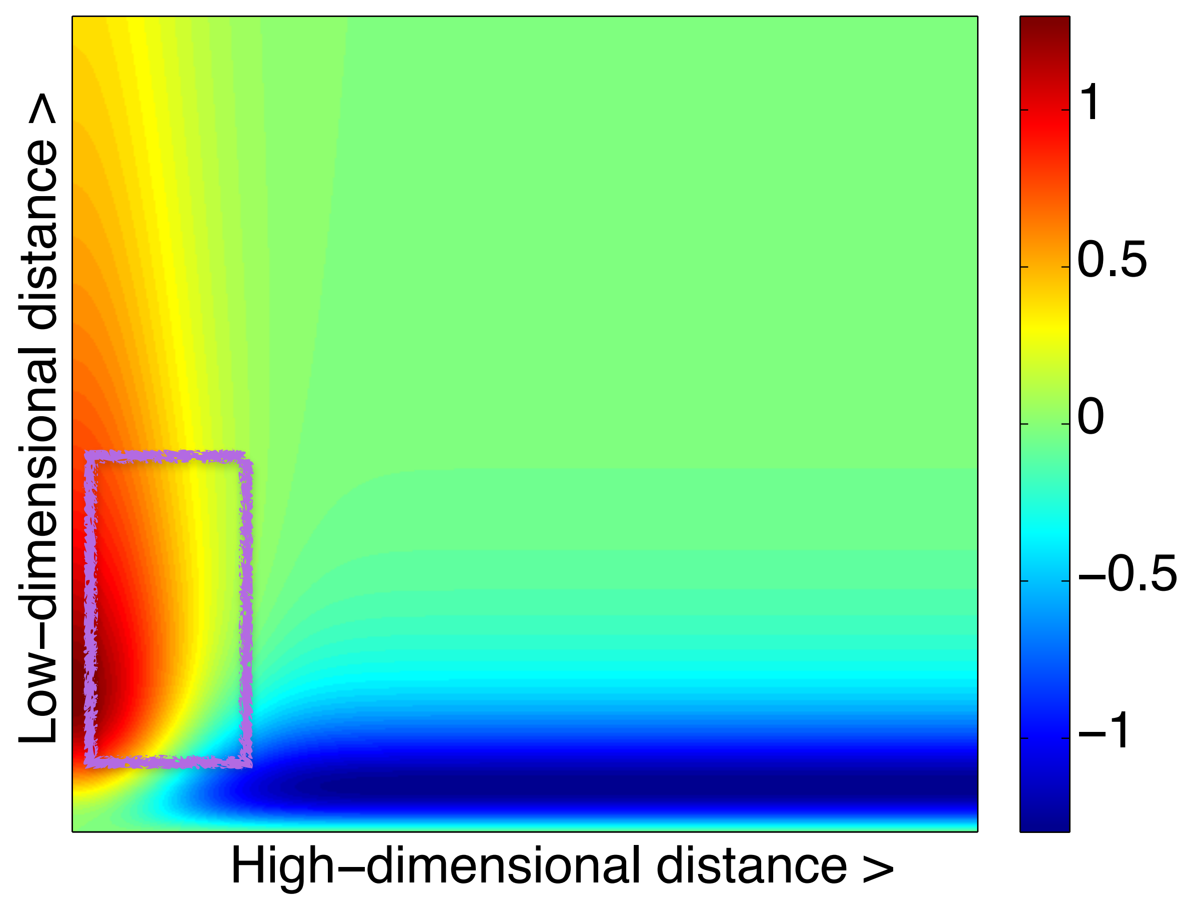
(c) Gradient of t-SNE.

t-SNE *repels* points in low dim space that are different in the high dim space

What's magical about *t*?



(a) Gradient of SNE.



(c) Gradient of t-SNE.

Also strongly attracts points nearby in high dim space

Let’s see some code

Another perspective:
Auto-encoders

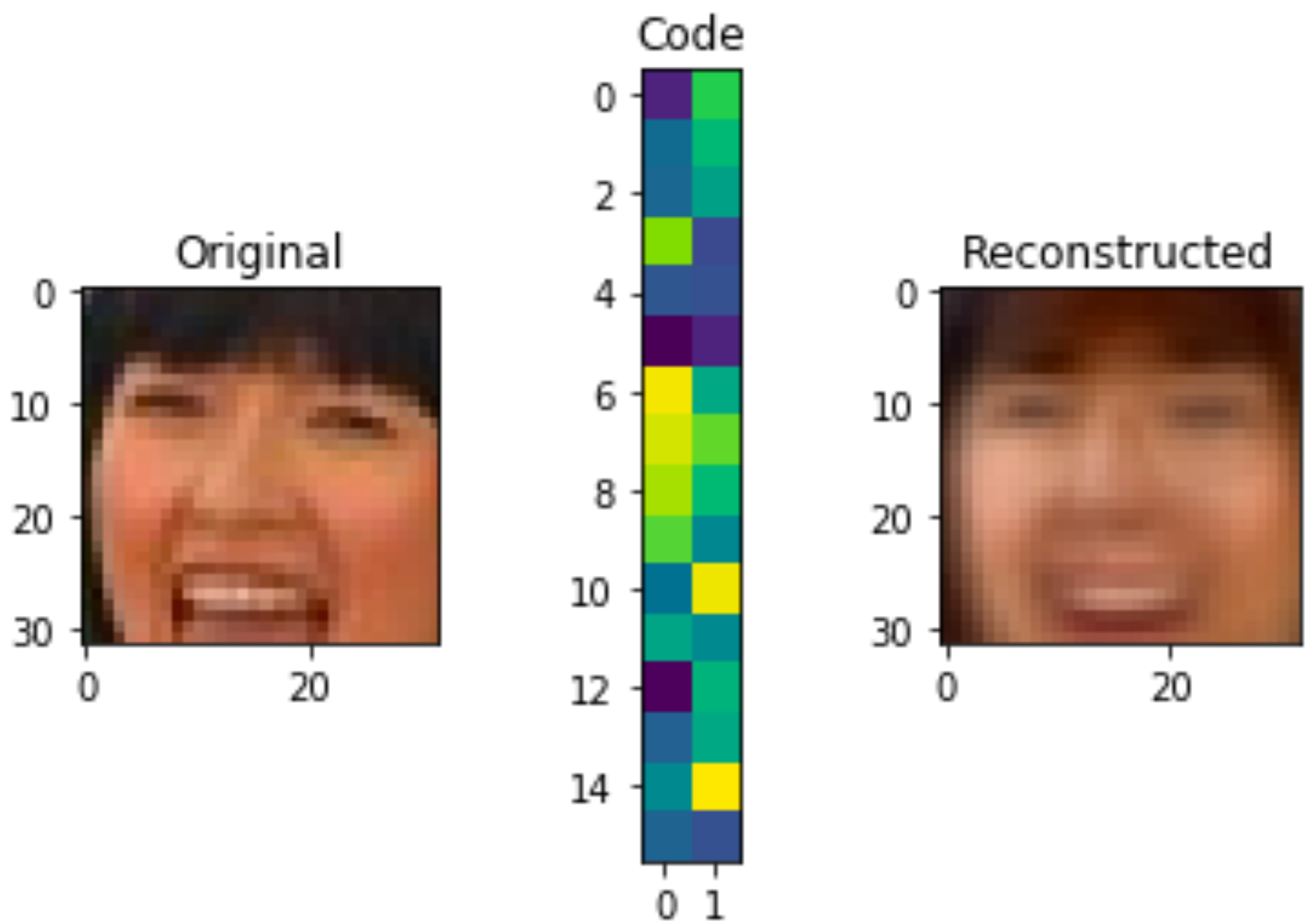


Figure credit: https://stackabuse.com/autoencoders-for-image-reconstruction-in-python-and-keras/