COMP 432 Machine Learning

Dimensionality Reduction & Autoencoders

Computer Science & Software Engineering
Concordia University, Fall 2021

"Curse of dimensionality"

Richard Bellman

High dimensional space is *vast* and *hard to fill up*, even with lots of data!

- When dimensionality of data $\mathbf{x} \in \mathbb{R}^D$ or features $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^M$ is large, problems can quickly arise:
  1. Our intuitions about distance, orthogonality, volume are all based on experience in 2D or 3D.
     In high dimensions, **our intuitions are often wrong**.
  2. Many spatial data structures **do not scale well** with dimension, for example *k*-d trees (nearest neighbour).
  3. Can be **harder to avoid over-fitting**, especially for models that have per-dimension parameters.

In high dimensions, any two random vectors $\mathbf{u}, \mathbf{v}$ are essentially orthogonal.
In high dimensions, nearly all the volume of a ball is near its surface.
In high dimensions, nearly all the probability density of a Gaussian is away from the origin.

"In a 30-dimensional grocery store, anchovies can be next to fish *and* next to pizza toppings." – Geoff Hinton

Ways to reduce dimension

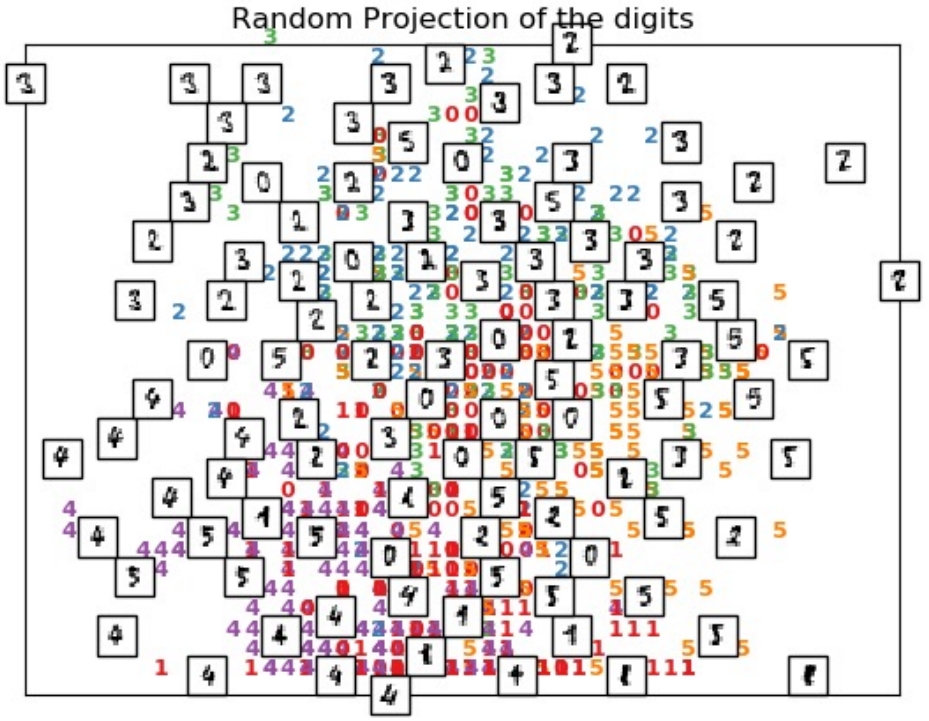
- Perform feature selection, *i.e.* only the top *K* most informative features to be used for a task.
- Engineer new domain-specific features, and use them to replace some of the original raw features. (Or rely on feature selection to choose.)
- Linear transformation to a subspace
  - Random projection, PCA, ICA, ...
- Non-linear transformation to a manifold
  - Kernel PCA, Autoencoders, ...

Dimensionality Reduction

- **Random linear projections** have applications in signal processing, sparse coding, but tend to make classification harder

784-dimensional space

map



2-dimensional space

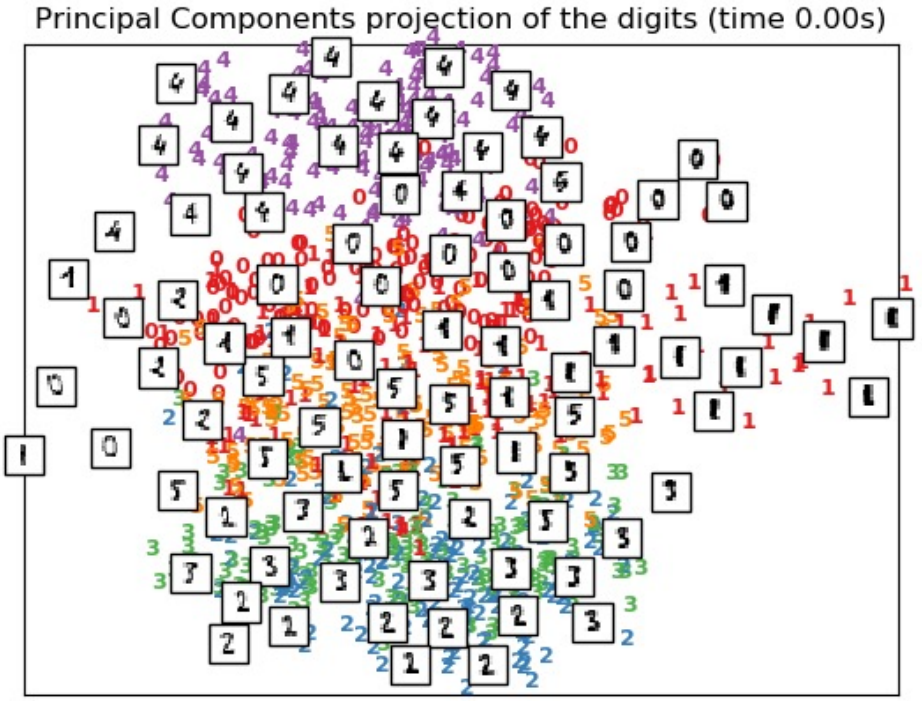
Image: https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html

Dimensionality Reduction

- **Principle component analysis (PCA)** finds linear projection that "preserves the most information" about original data, and discards minor variation

784-dimensional space

map



2-dimensional space

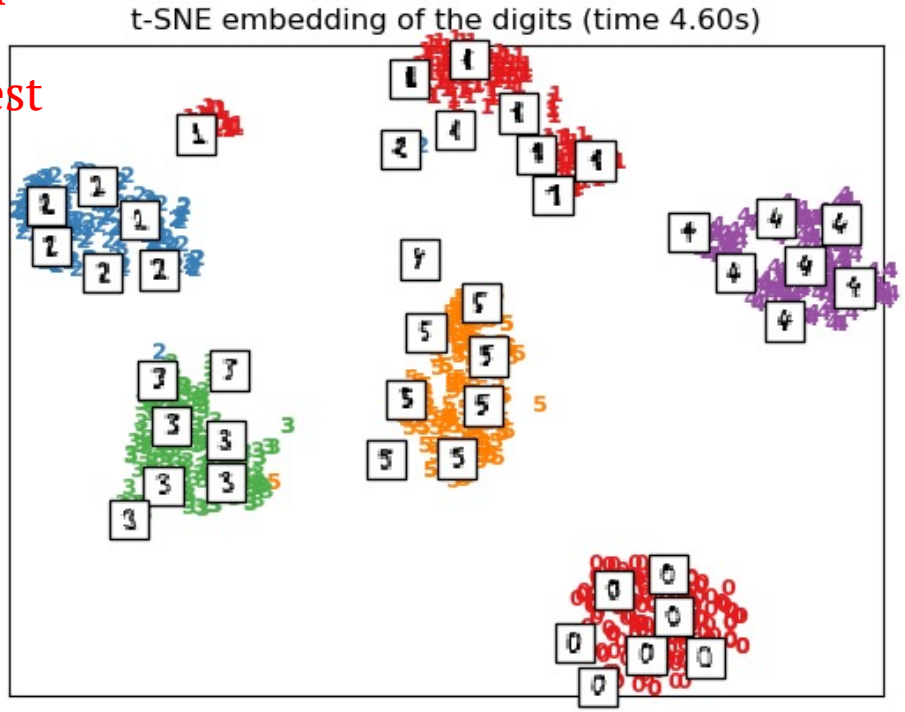
Image: https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html

Dimensionality Reduction

- Stochastic neighbour embedding (SNE, t-SNE) useful for visualizing clusters or topologies.

Note: SNE does *not* learn an explicit map from data space to embedding space, so you can't ask the embedding of a test point. Useless for classification!

784-dimensional space

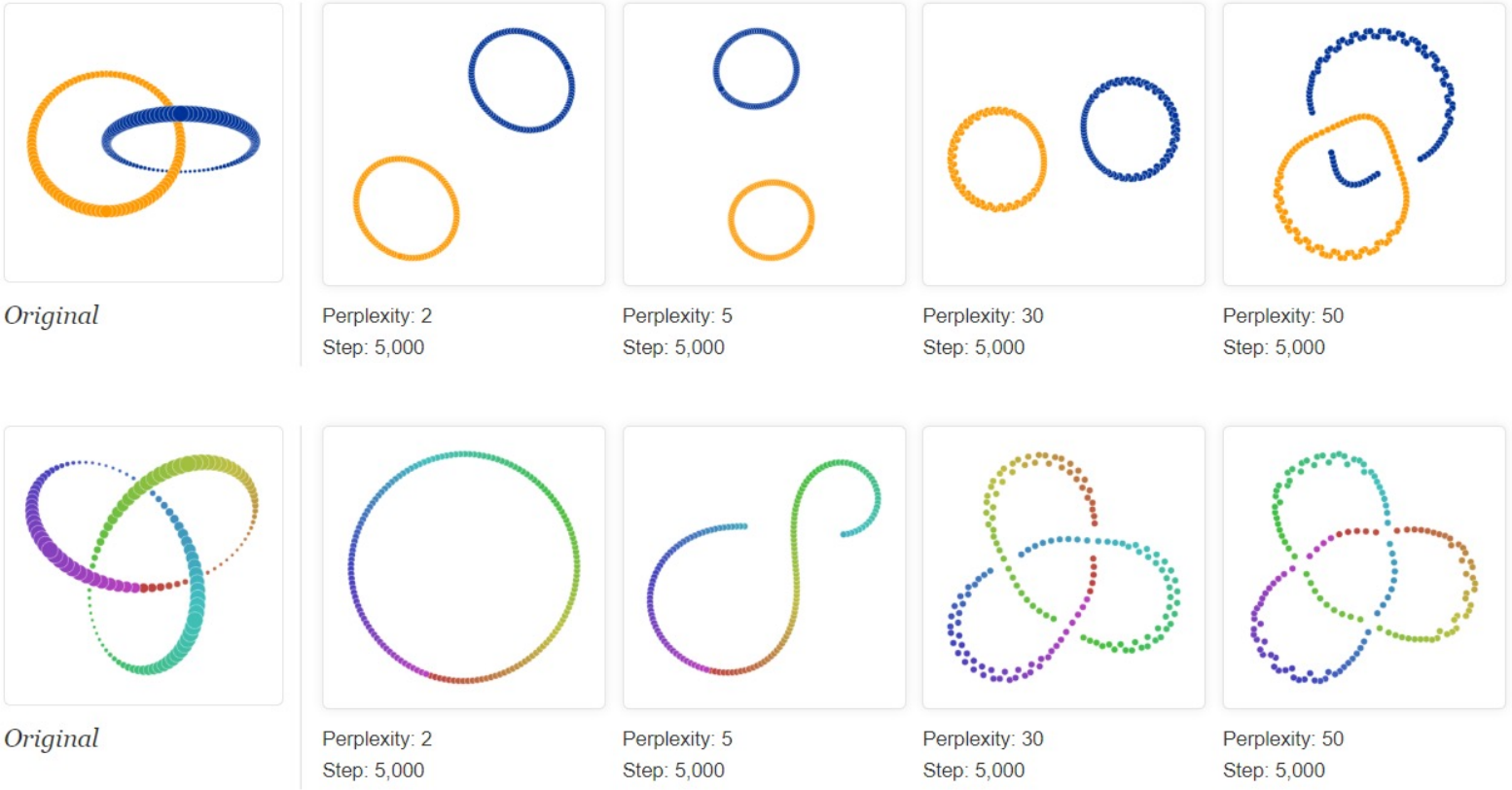


2-dimensional space

Image: https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html

Stochastic Neighbour Embedding

- Check out https://distill.pub/2016/misread-tsne/ for **awesome** t-SNE demos:



sklearn.manifold.TSNE

class `sklearn.manifold.` **`TSNE`** (*n_components=2, perplexity=30.0, early_exaggeration=12.0, learning_rate=200.0, n_iter=1000, n_iter_without_progress=300, min_grad_norm=1e-07, metric='euclidean', init='random', verbose=0, random_state=None, method='barnes_hut', angle=0.5*)

[source]

t-distributed Stochastic Neighbor Embedding.

t-SNE [1] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples. For more tips see Laurens van der Maaten's FAQ [2].

Principle Component Analysis (PCA)

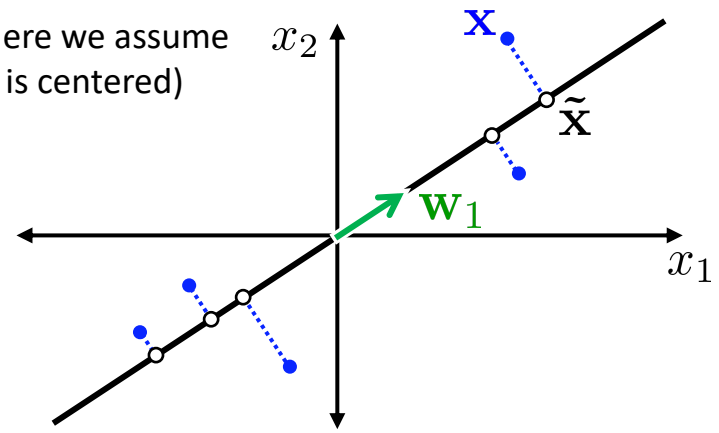
hyperparameter

- **Intuition:** project D-dimensional data to an M-dimensional subspace, in a way that approximates the original data ("preserves most information").
- **Main idea:** given data $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with $\mathbf{x}_i \in \mathbb{R}^D$, find points $\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ with $\mathbf{z}_i \in \mathbb{R}^M$ and an orthonormal basis $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \cdots & \mathbf{w}_M \end{bmatrix} \in \mathbb{R}^{D \times M}$ such that the reconstruction error is minimized:

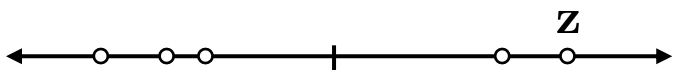
$$\frac{1}{2} \sum_{i=1}^{N} \|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|^2$$

$$\tilde{\mathbf{x}}_i = \mathbf{W}\mathbf{z}_i$$

(Here we assume ***X*** is centered)



(embedding of 1-d subspace in 2-d)



(1-d representations of data)

sklearn.decomposition.PCA

class sklearn.decomposition. **PCA** (*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*) [source]

Principal component analysis (PCA)

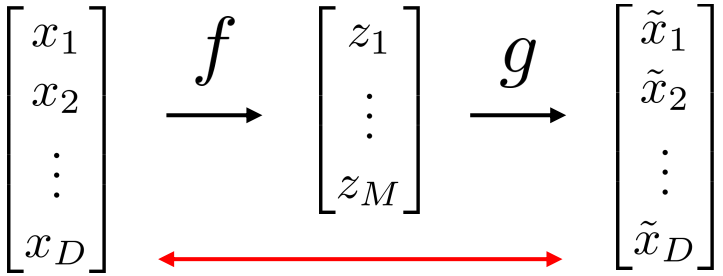
Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

Methods

fit (self, X[, y])	Fit the model with X.
fit_transform (self, X[, y])	Fit the model with X and apply the dimensionality reduction on X.
get_covariance (self)	Compute data covariance with the generative model.
get_params (self[, deep])	Get parameters for this estimator.
get_precision (self)	Compute data precision matrix with the generative model.
inverse_transform (self, X)	Transform data back to its original space.

Autoencoders (AEs)

- With PCA, once we have orthonormal $\mathbf{W}$ we can "encode" the data by $\mathbf{z}_i = \mathbf{W}^T \mathbf{x}_i$
- This means PCA searches for a $\mathbf{W}$ that provides a good (and linear) "self encoding" $\tilde{\mathbf{x}}_i = \mathbf{W}\mathbf{W}^T \mathbf{x}_i$
- **Autoencoders** generalize this to arbitrary encoder and decoder functions:



1. encode: $\mathbf{z}_i = f(\mathbf{x}_i, \mathbf{w})$
2. decode: $\tilde{\mathbf{x}}_i = g(\mathbf{z}_i, \boldsymbol{\omega})$
3. loss:

$$\ell(\mathbf{w}, \boldsymbol{\omega}) = \frac{1}{2} \sum_{i=1}^{N} \|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|^2$$

Autoencoders (AEs)

- Unsupervised “representation learning” technique
- Learns a non-linear encoder and decoder function
  - Early autoencoders were stochastic neural networks where the encoder and decoder shared parameters
  - Encoder and decoder can be any neural network, e.g.

$$\mathbf{z} = \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \quad \tilde{\mathbf{x}} = \tanh(\mathbf{W}_2\mathbf{z} + \mathbf{b}_2)$$

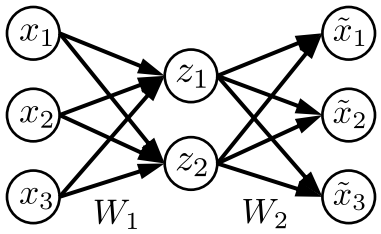
  - Choice of *regularization* has major influence on **z**
- Orthonormality not usually enforced (unlike PCA)
- Reconstruction loss still implicitly prefers encodings that “preserve information”
- Can be used to decrease or *increase* dimensionality

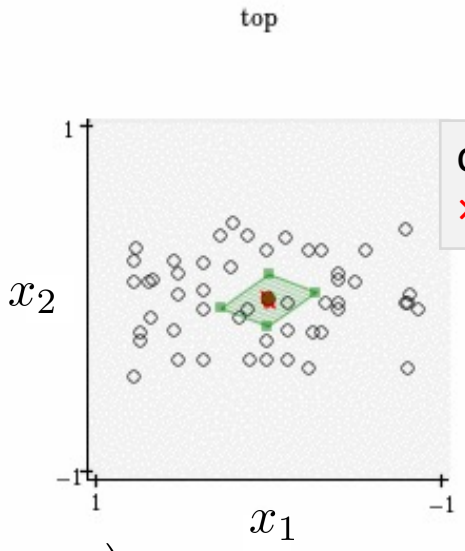
Training a 3-2-3 *tanh* autoencoder w/ SGD

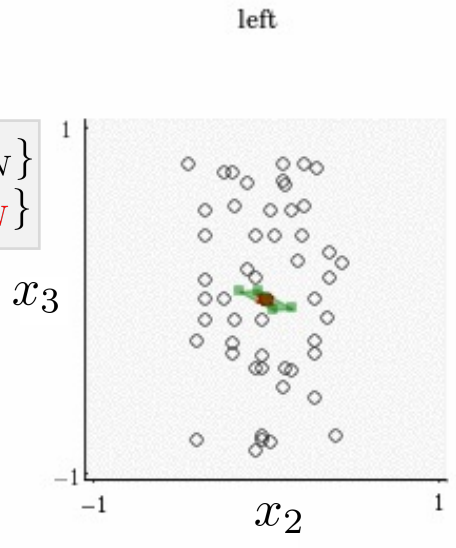
$$\mathbf{z} = f(\mathbf{x}, \mathbf{w})$$
$$\tilde{\mathbf{x}} = g(\mathbf{z}, \boldsymbol{\omega})$$
$$\ell = \tfrac{1}{2} \left\| \mathbf{x} - \tilde{\mathbf{x}} \right\|^2$$

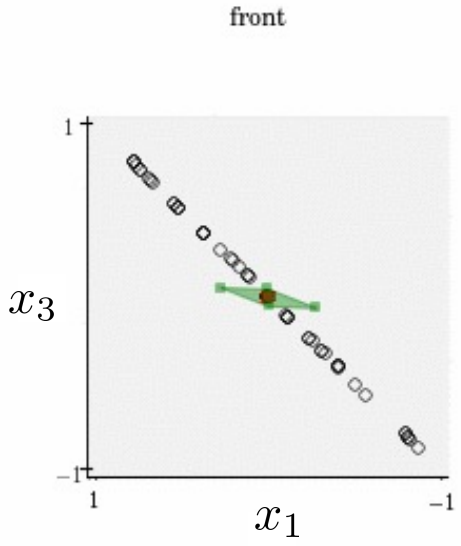
$$f(\mathbf{x}, \mathbf{w}) = \tanh\left(\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} w_7 \\ w_8 \end{bmatrix} \right)$$

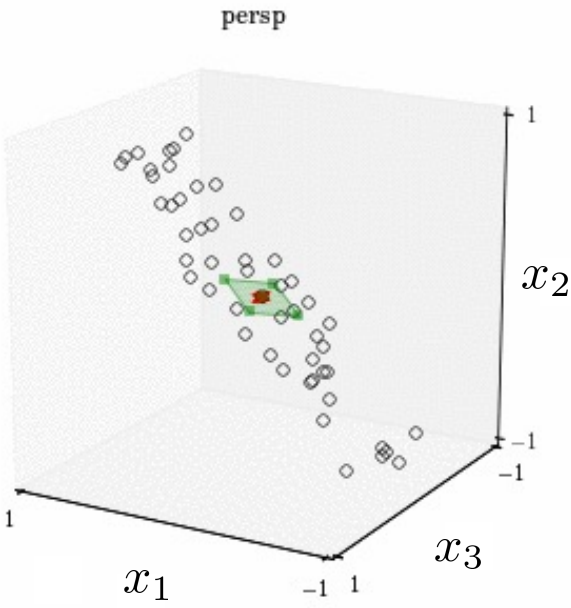
$$g(\mathbf{z}, \boldsymbol{\omega}) = \begin{bmatrix} \omega_1 & \omega_4 \\ \omega_2 & \omega_5 \\ \omega_3 & \omega_6 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} + \begin{bmatrix} \omega_7 \\ \omega_8 \\ \omega_9 \end{bmatrix}$$









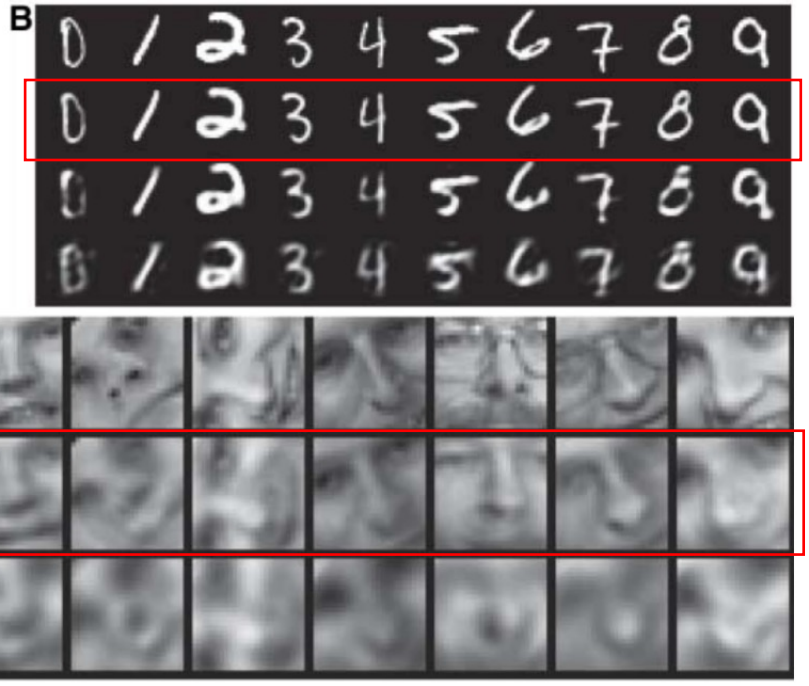


Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton* and R. R. Salakhutdinov

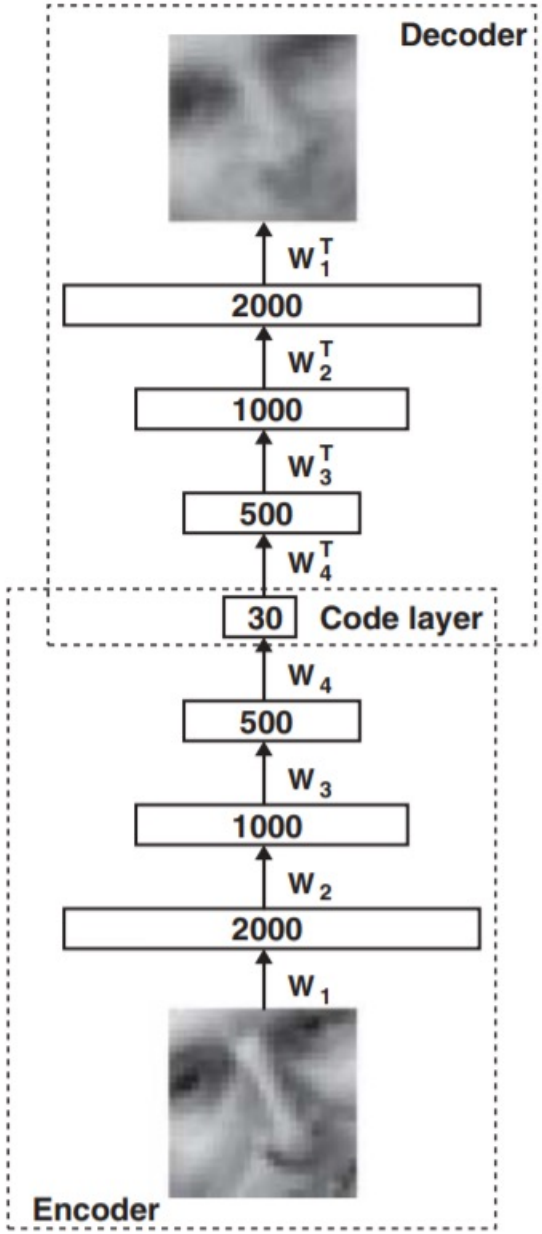
>10,000 citations

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.



(B) Top to bottom: A random test image from each class; reconstructions by the 30-dimensional autoencoder; reconstructions by 30-dimensional logistic PCA and standard PCA. The average squared errors for the last three rows are 3.00, 8.01, and 13.87.

(C) Top to bottom: Random samples from the test data set; reconstructions by the 30-dimensional autoencoder; reconstructions by 30-dimensional PCA. The average squared errors are 126 and 135.



https://www.cs.toronto.edu/~hinton/science.pdf

Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton* and R. R. Salakhutdinov

PCA representation

Auto-encoder (AE) representation

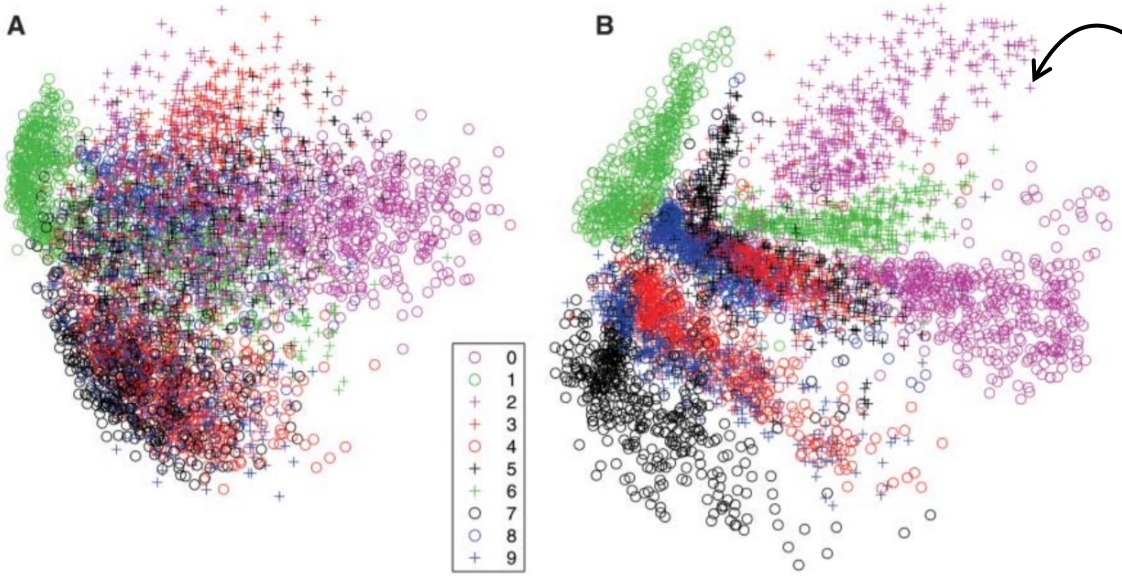
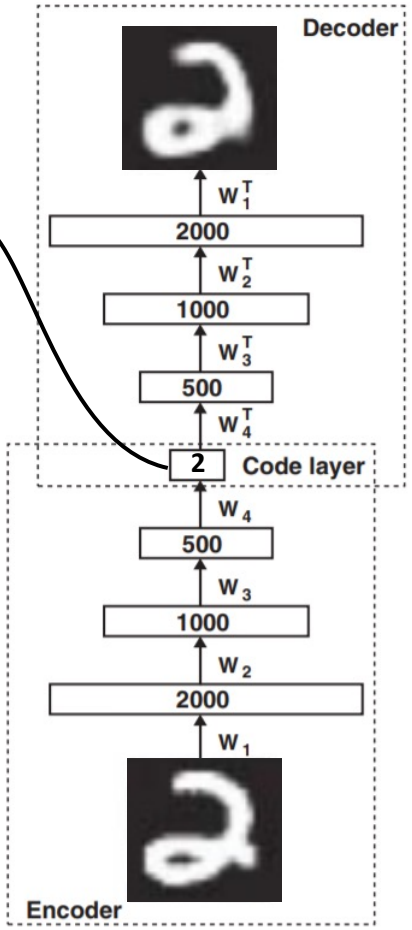


Fig. 3. (**A**) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (**B**) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (*8*).



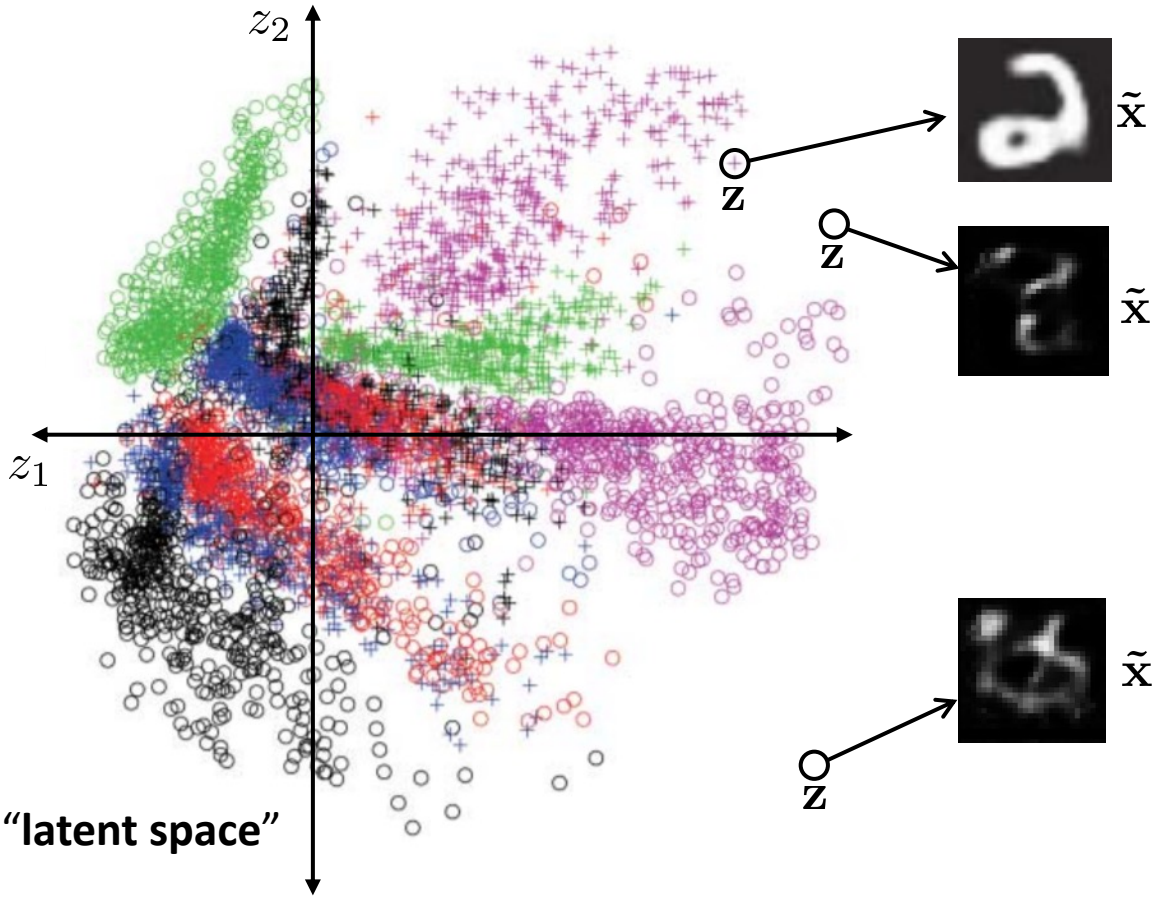
With linear encoder/decoder, PCA representation is worse at reconstruction and worse for classification

With more model capacity, reconstruction loss *incidentally* finds structure that would also be useful for classification!

https://www.cs.toronto.edu/~hinton/science.pdf

Limitation of basic auto-encoders

- After training, most of $\mathbf{z}$-space decodes to nonsense



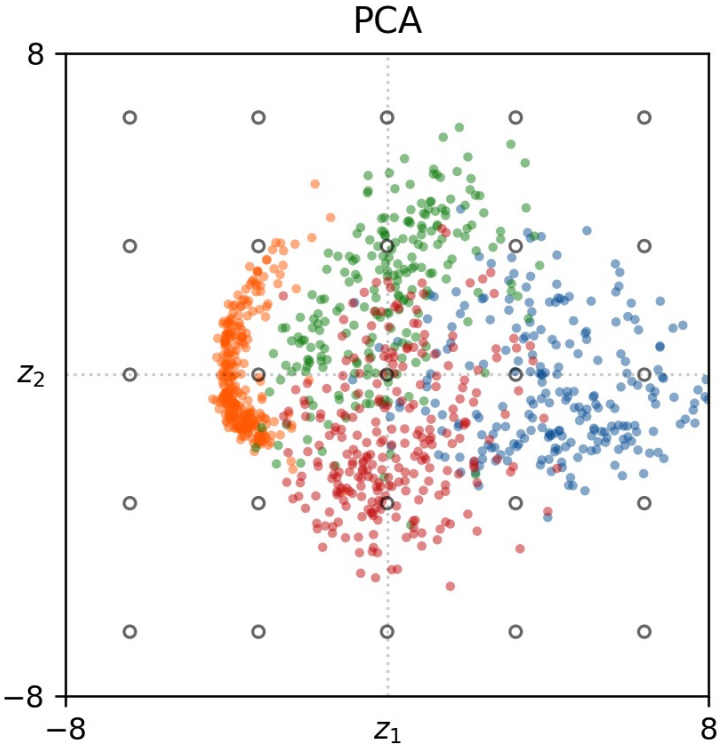
Q: How might you build a *generative* model from a representation like this?

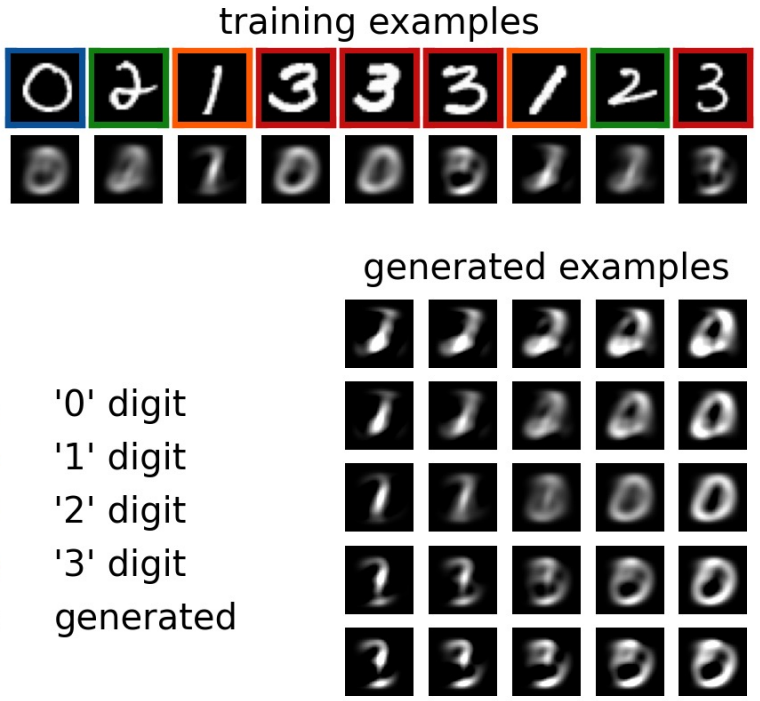
How could we sample a 'latent code' that could be decoded to a 'realistic' data item?

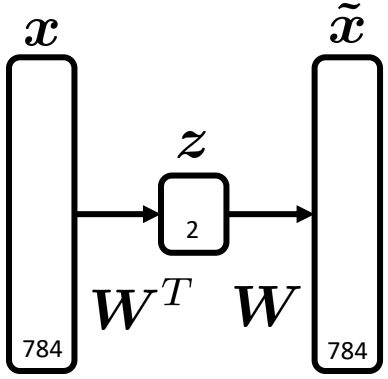
$$\mathbf{z} \sim p$$

$$\tilde{\mathbf{x}} = g(\mathbf{z}, \boldsymbol{\omega})$$

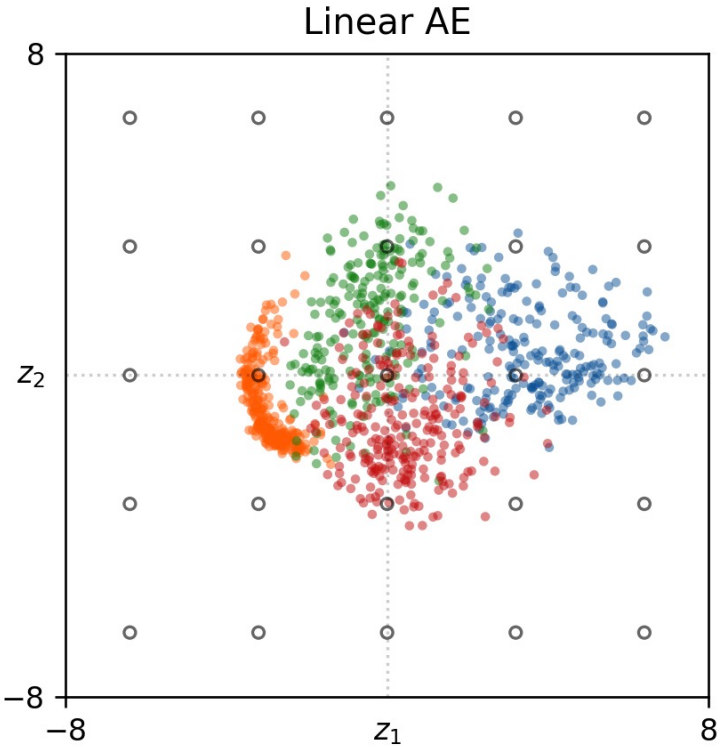
Example: PCA

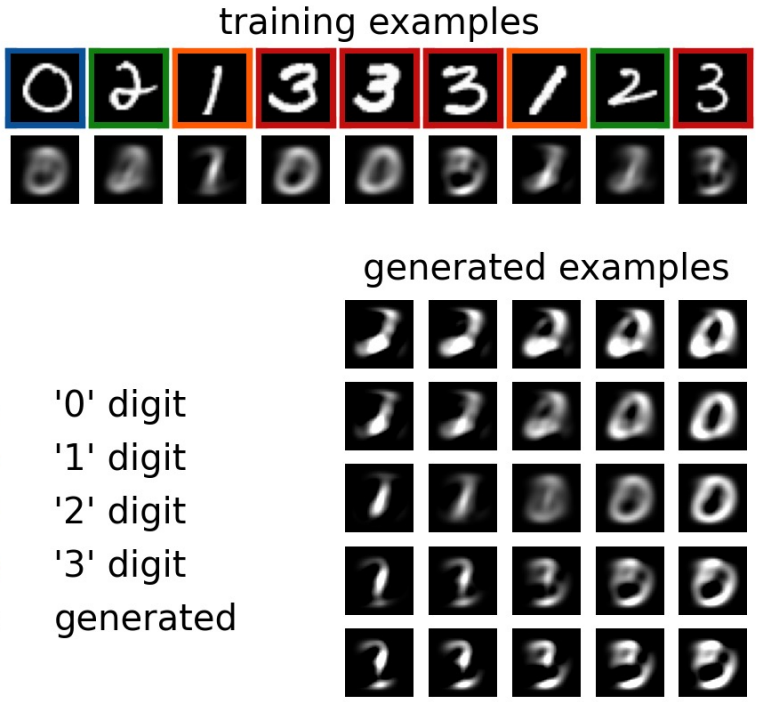

PCA
8
z_2
−8
−8
z_1
8

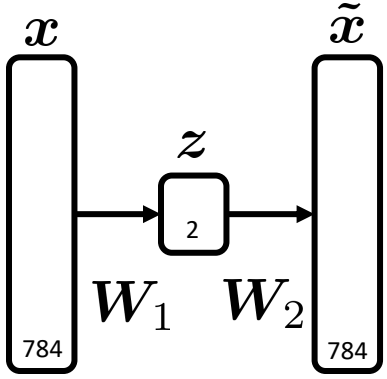

training examples
generated examples
'0' digit
'1' digit
'2' digit
'3' digit
generated


$\boldsymbol{x}$
$\tilde{\boldsymbol{x}}$
$\boldsymbol{z}$
2
$\boldsymbol{W}^T$
$\boldsymbol{W}$
784
784

Example: Linear Autoencoder

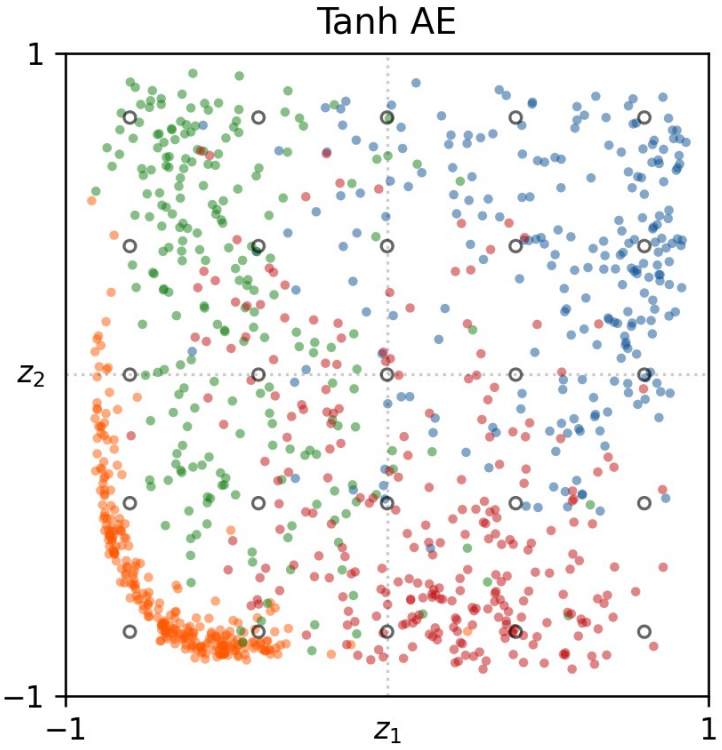


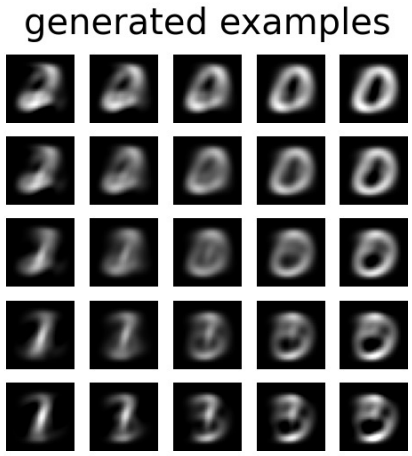




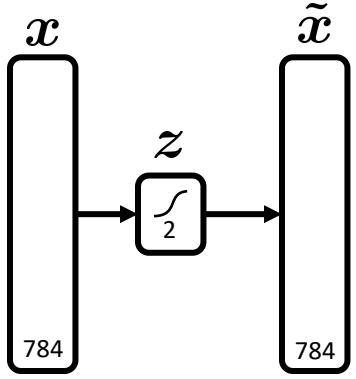
(this slide is a video)

Example: Tanh Autoencoder



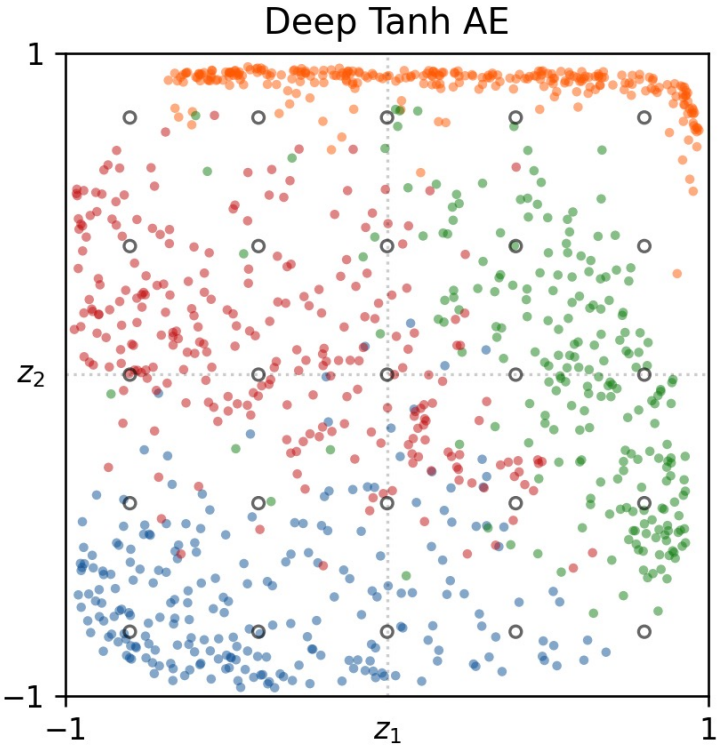


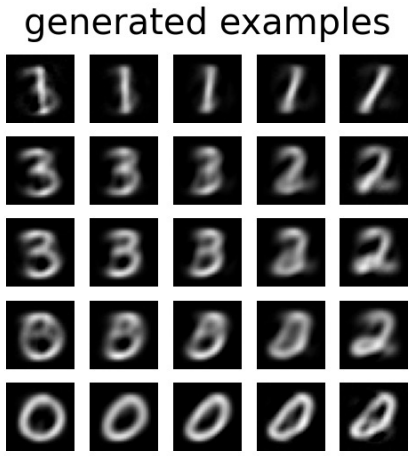
'0' digit
'1' digit
'2' digit
'3' digit
generated

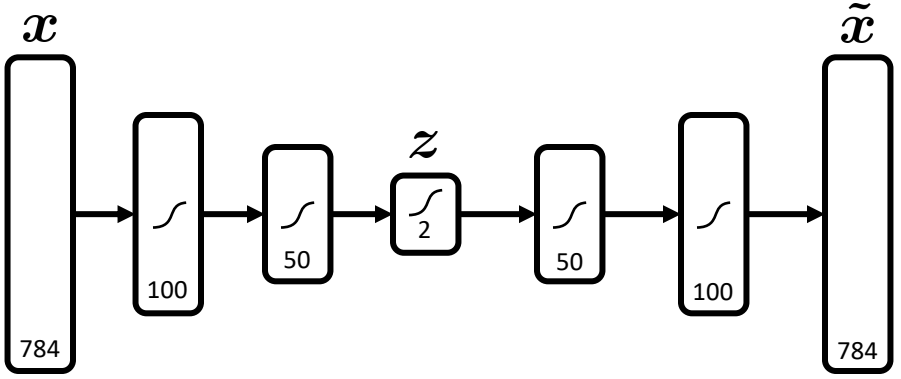


(this slide is a video)

Example: Deep Tanh Autoencoder

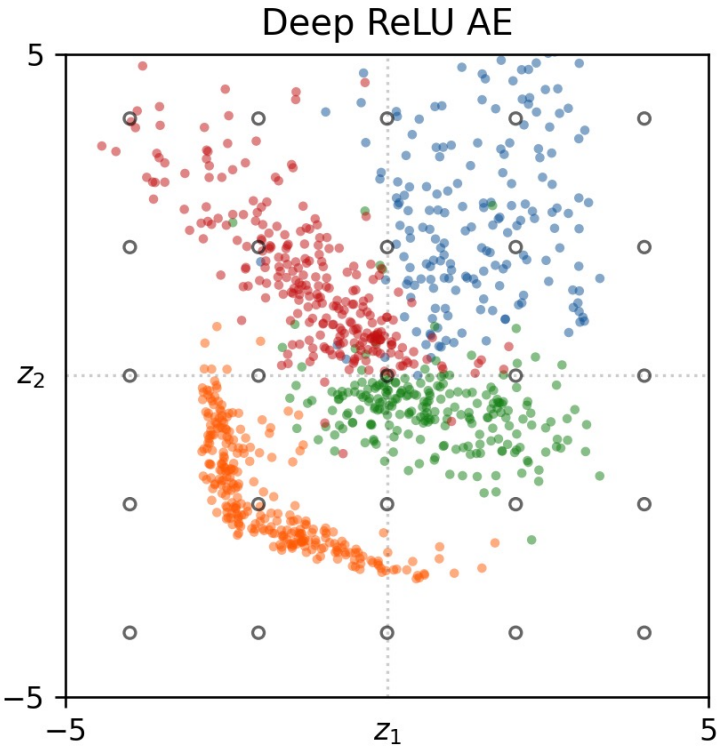


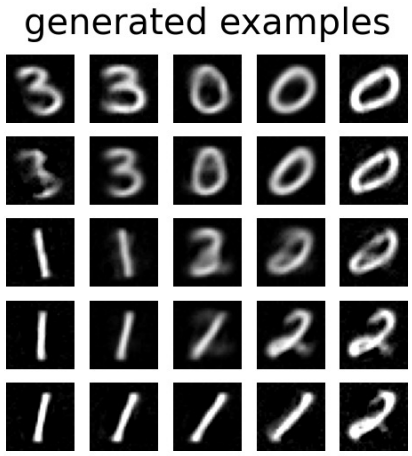


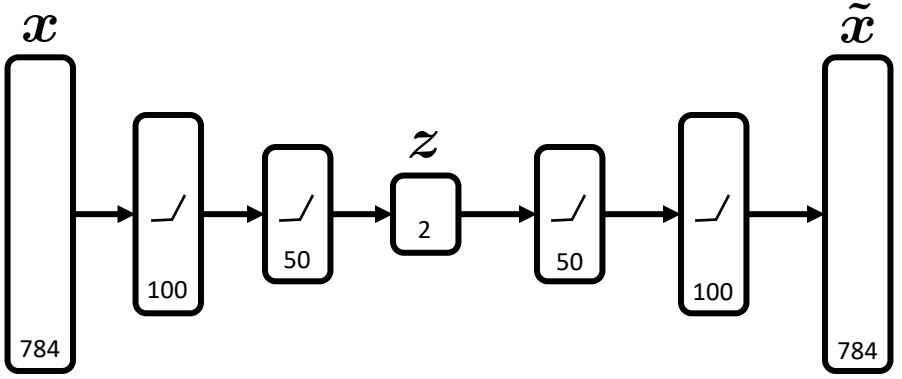


(this slide is a video)

Example: Deep ReLU Autoencoder







(this slide is a video)

Autoencoder types

- **Weight decay:** penalizing $\|\mathbf{w}\|^2$ essentially smooths the embedding while also biasing it
- **Sparse autoencoder:** $\mathbf{z}$ very high dimensional, but encourage $p(z_j = 0 \mid \mathbf{x})$ to be large for most $\mathbf{x}$

  https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf
- **Denoising autoencoder:** encourage "corrupted" data $\mathbf{x}_i + \boldsymbol{\epsilon}$ to have same encoding $\mathbf{z}_i$ as uncorrupted $\mathbf{x}_i$
  - Gives consistent encodings to points $\mathbf{x}$ "around" each $\mathbf{x}_i$
- **Variational autoencoder:** encoder $f(\mathbf{x}, \mathbf{w})$ outputs a *distribution* over $\mathbf{z}$, so each $\mathbf{x}$ has a "soft" encoding
  - Also tries to "pack" the code distributions tightly