Nonlinear Dimensionality Reduction: (Deep) Autoencoder

Principal Component Analysis (PCA)

Maximum Variance Subspace

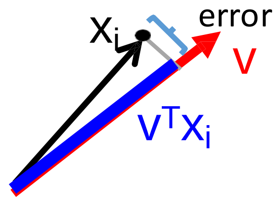
- PCA finds vectors v such that projections on to the vectors capture ***maximum variance in the data***

$$max_{\boldsymbol{v}} \ \frac{1}{n}\sum_{i=1}^{n}(\mathbf{v}^T\mathbf{x}_i)^2 = \mathbf{v}^T\mathbf{X}\mathbf{X}^T\mathbf{v}$$

Minimum Reconstruction Error

- PCA finds vectors v such that projection on to the vectors yields ***minimum MSE reconstruction***

$$min_{\boldsymbol{v}} \frac{1}{n}\sum_{i=1}^{n}\|\mathbf{x}_i - (\mathbf{v}^T\mathbf{x}_i)\mathbf{v}\|^2$$



Principal Component Analysis (PCA)

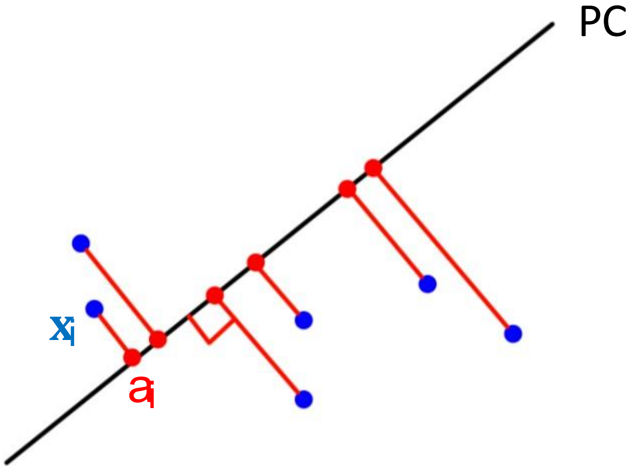
- Input: $\mathbf{x}_1 \cdots, \mathbf{x}_n \in \mathbb{R}^d$)with zero-mean).

- Output: $a_1 \cdots, a_n \in \mathbb{R}^k$ $(k \ll d)$.

- PCA finds principal components: $d \times k$ column orthogonal matrix V ($V^T V = I$)

$$\min_{\mathbf{V}} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{V}\mathbf{V}^T \mathbf{x}_i\|_2^2$$

- This process can be written in two steps



Principal Component Analysis (PCA)

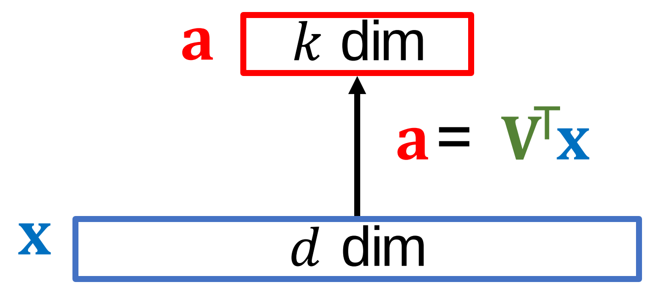
- Input: $\mathbf{x}_1 \cdots, \mathbf{x}_n \in \mathbb{R}^d$)with zero-mean).

- Output: $a_1 \cdots, a_n \in \mathbb{R}^k$ $(k \ll d)$.

- PCA finds principal components: $d \times k$ column orthogonal matrix V ($V^TV=I$)

$$\min_{\mathbf{V}} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{V}\mathbf{V}^T\mathbf{x}_i\|_2^2$$

- This process can be written in two steps
  - Dimensionality reduction ($\mathbf{a} = \mathbf{V}^T\mathbf{x}$)

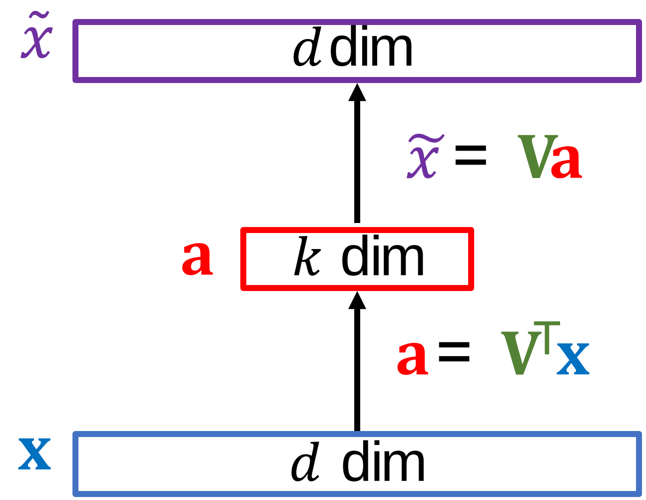


Principal Component Analysis (PCA)

- Input: $\mathbf{x}_1 \cdots, \mathbf{x}_n \in \mathbb{R}^d$)with zero-mean).
- Output: $\mathrm{a}_1 \cdots, \mathrm{a}_n \in \mathbb{R}^k$ ($k \ll d$).
- PCA finds principal components: $d \times k$ column orthogonal matrix V ($V^TV=I$)

$$\tilde{x} \approx VV^T x$$

- This process can be written in two steps
  - Dimensionality reduction ($\mathbf{a} = \mathbf{V}^T\mathbf{x}$)
  - Data reconstruction ($\tilde{x} = \mathbf{V}\mathbf{a}$)

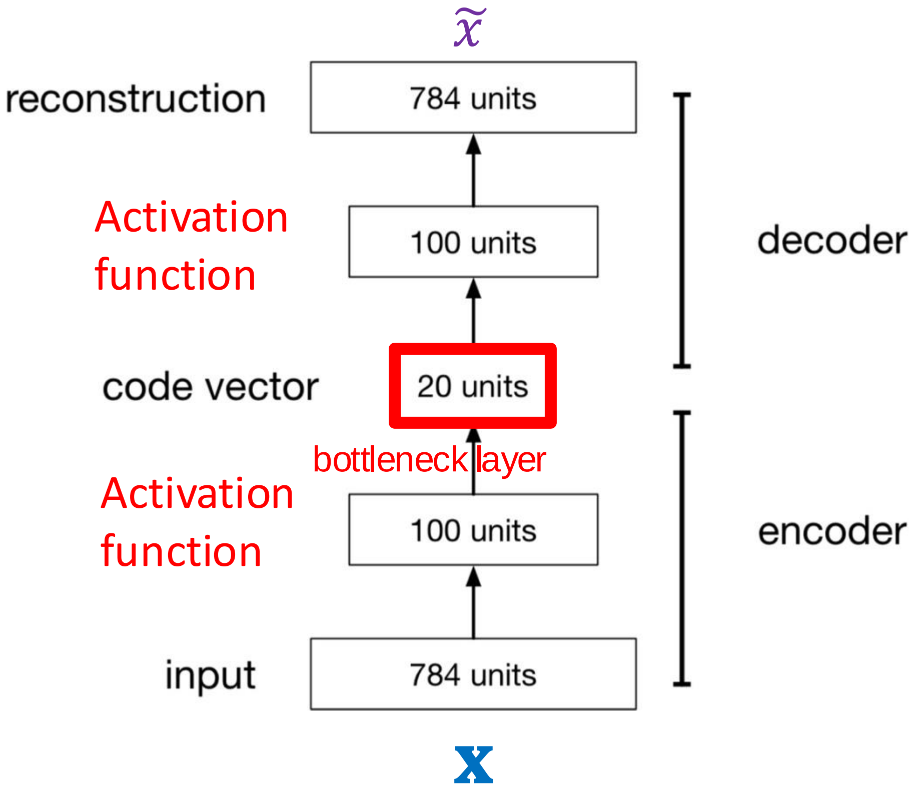


$$\tilde{x} \approx VV^T x$$

Autoencoder

- A ***neural network*** that **encodes** an input into latent codes/features (lower-dim) and then ***decodes*** the latent codes/features to reconstruct the input
- **Encoder** f
  - ***Compress*** input x into a latent-space of smaller dimension: h = f(x)
- **Decoder** g
  - ***Reconstruct*** input $\tilde{x}$ from the latent space h: $\tilde{x}$ = g(h) with $\tilde{x}$ close to x
- **Dimensionality reduction**: use latent (low-dim) codes/features
  - For downstream tasks, e.g., classification, clustering, visualization, etc.

Autoencoder: structure



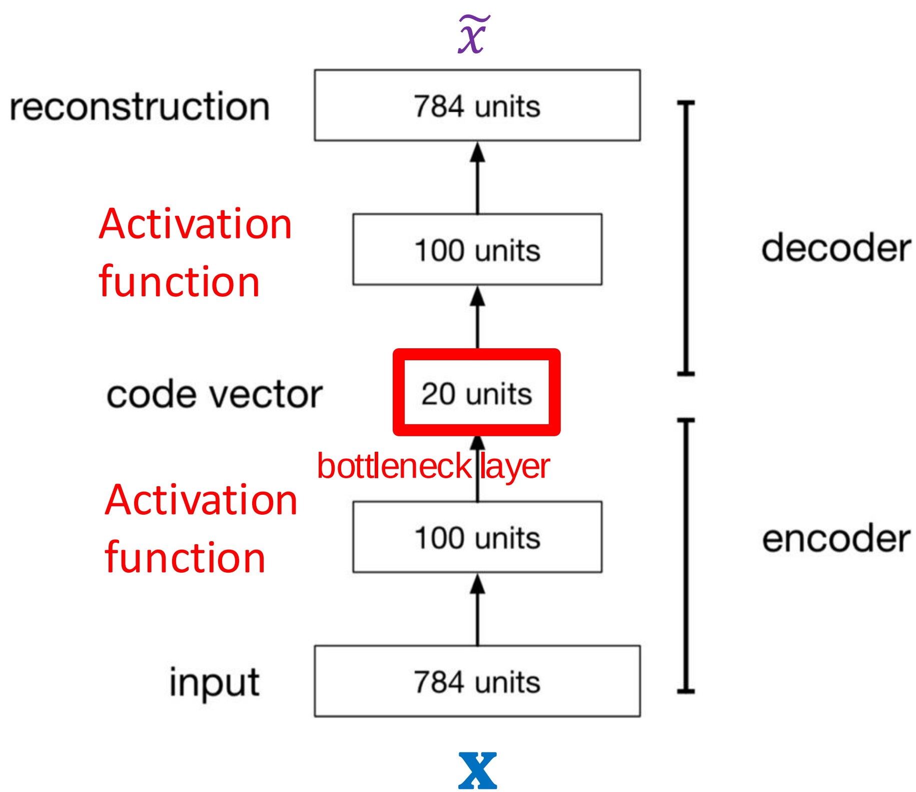
- Takes **x** as an input and outputs its reconstruction $\tilde{x}$
  - $\tilde{x}$ close to **x**
- For dimensionality reduction, we need a bottleneck layer whose dim is smaller than the input's dim
- Loss function

$$min||x - \tilde{x}||_2^2$$

Autoencoder vs PCA

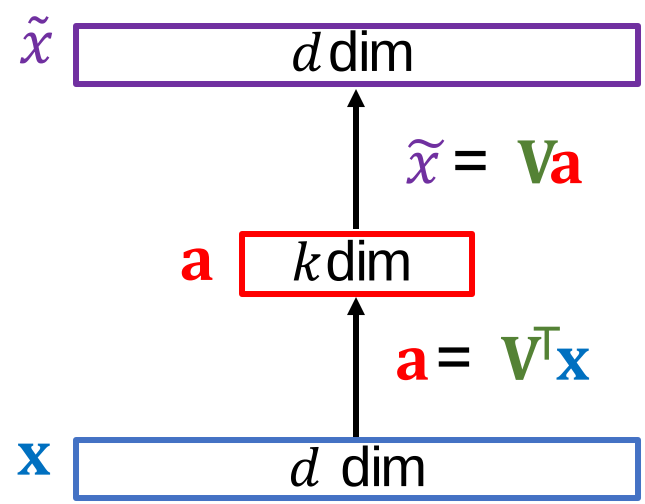
Autoencoder is nonlinear

$\tilde{x} = \text{decoder}\left(\text{encoder}(\mathbf{x})\right)$



PCA is linear

$\tilde{x} = \mathbf{V}\mathbf{V}^{\mathsf{T}}\mathbf{x}$

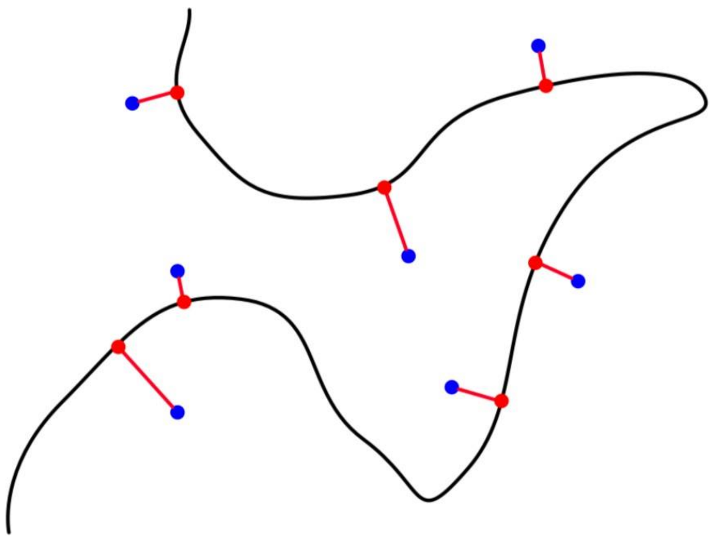


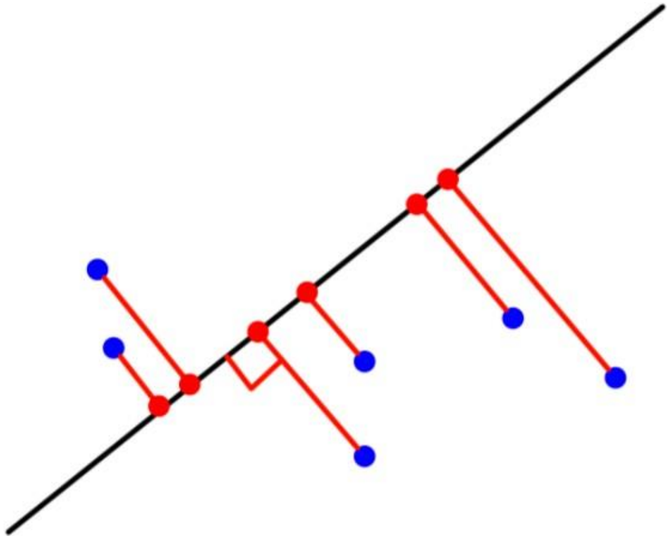
PCA: Autoencoder with linear activation

Autoencoder vs PCA

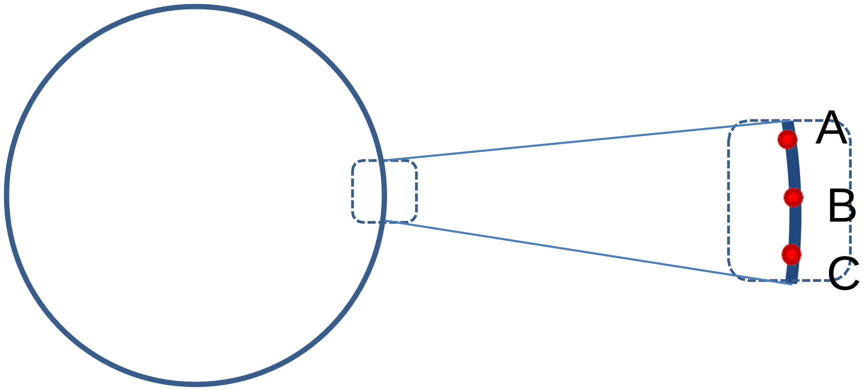
Autoencoder projects data onto ***nonlinear*** manifold.

PCA projects data onto a ***linear*** subspace.





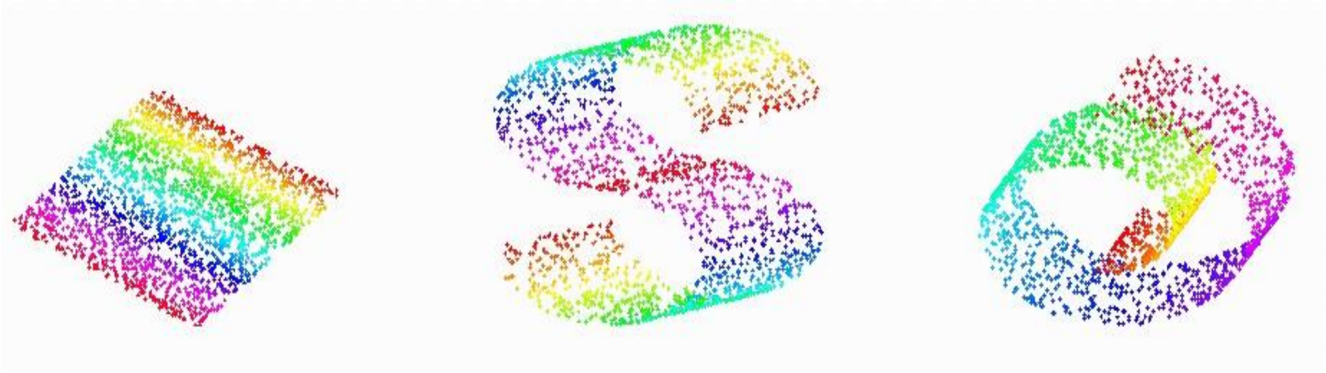
Manifold



$AC \approx AB + BC$

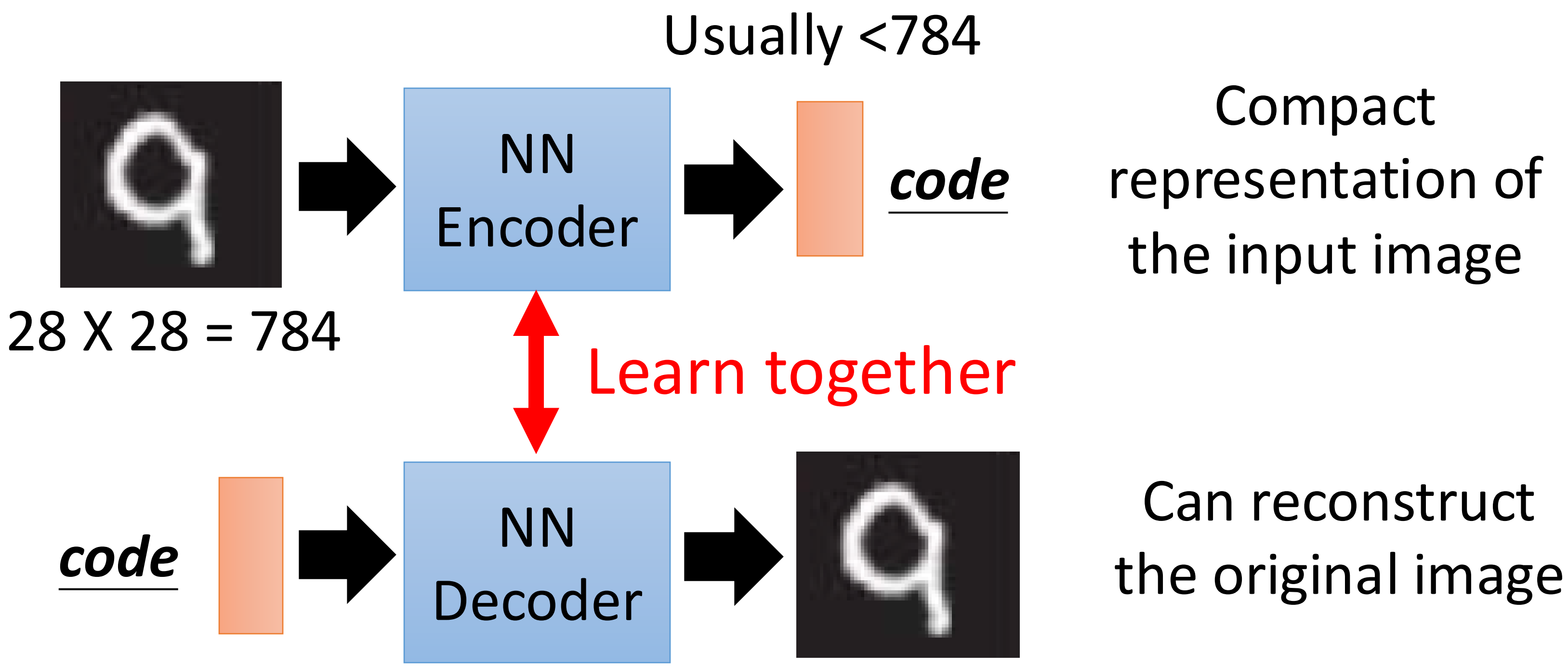
- Informally, ***manifold*** is a subset of points in the high-dim space that locally looks like a low-dim space
- Example: arc of a circle
  - consider a tiny bit of a circumference (2D)
  - => can treat as line (1D)

Manifold Examples

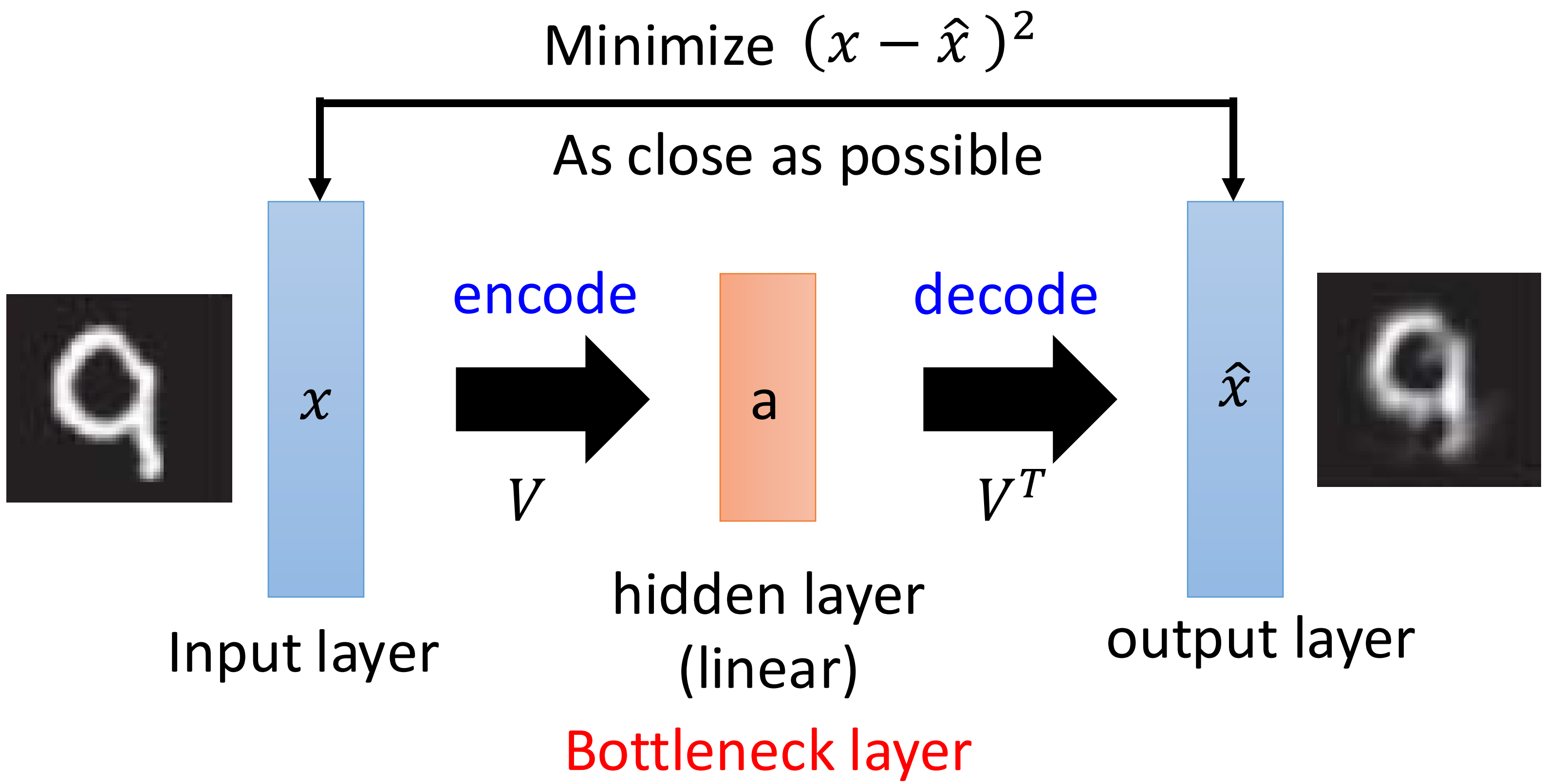


- All three are 2D data embedded in 3D
  - Linear, “S”-shape, “Swiss roll”
- Autoencoder can recover their 2D representation
- PCA: works for the first one, but the last two
  - Linear subspace does not explain it well

Autoencoder: MNIST image



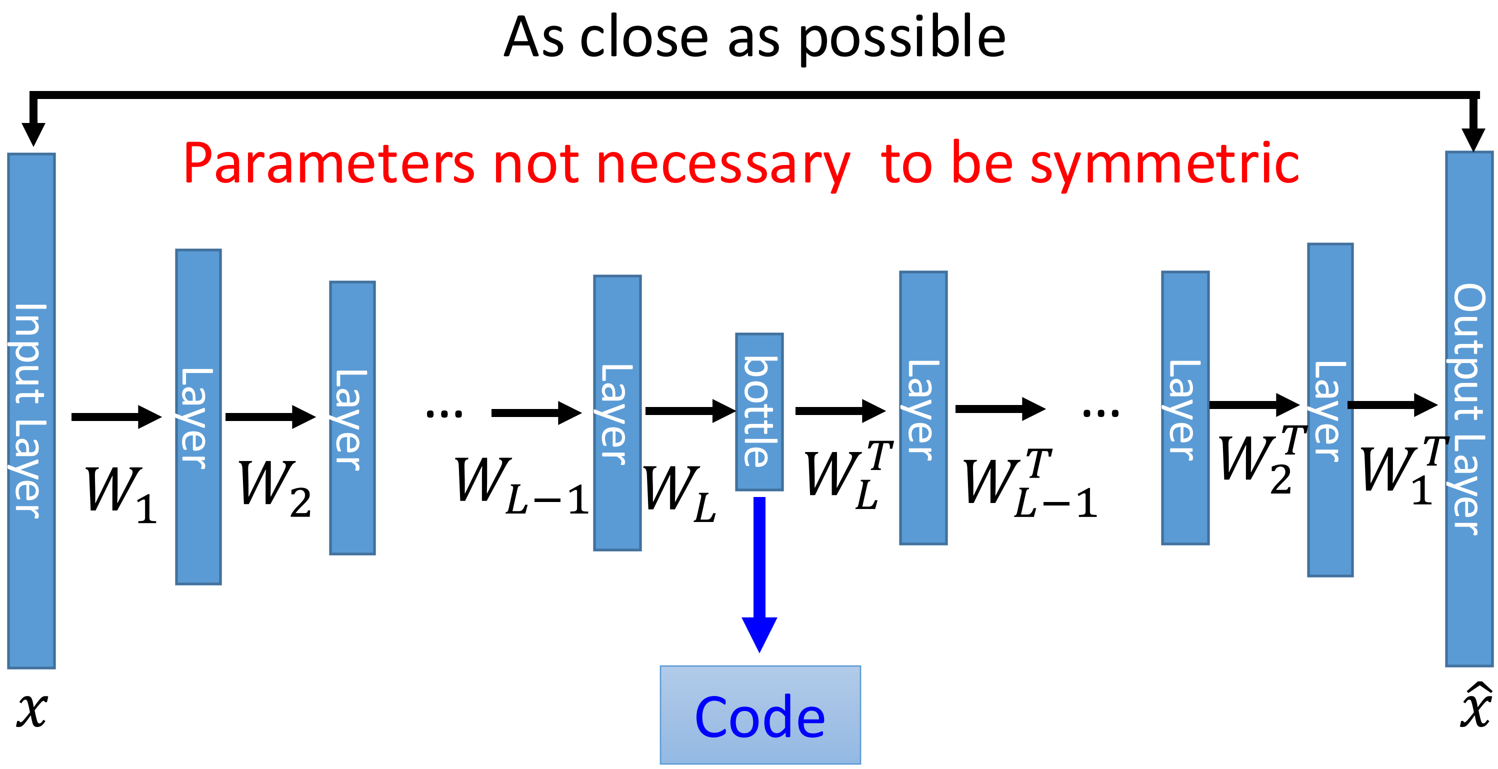
PCA: MNIST image



Output of the hidden layer is the code

Deep Autoencoder

- Auto-encoder can be deeper/have more layers

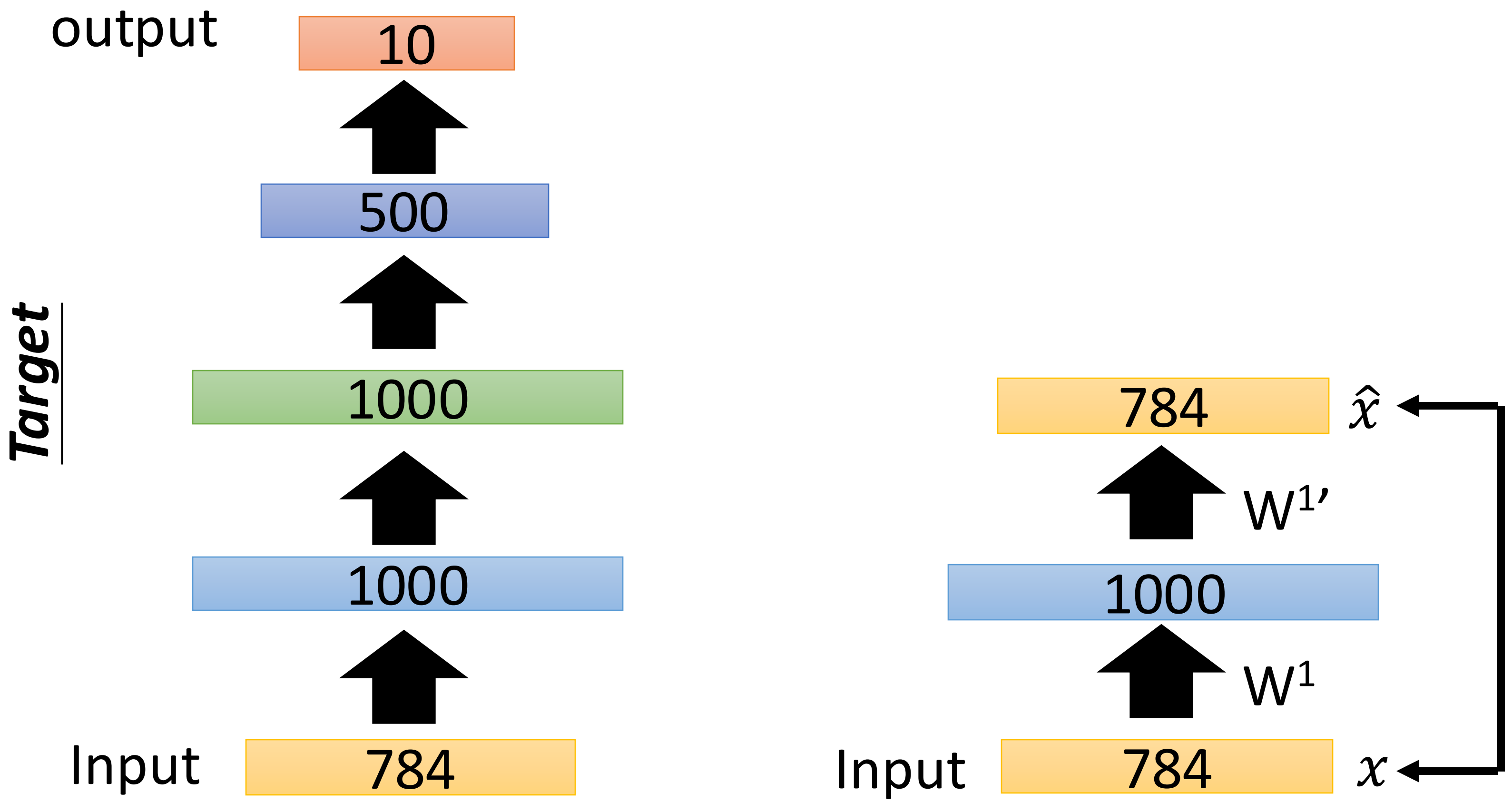


Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507

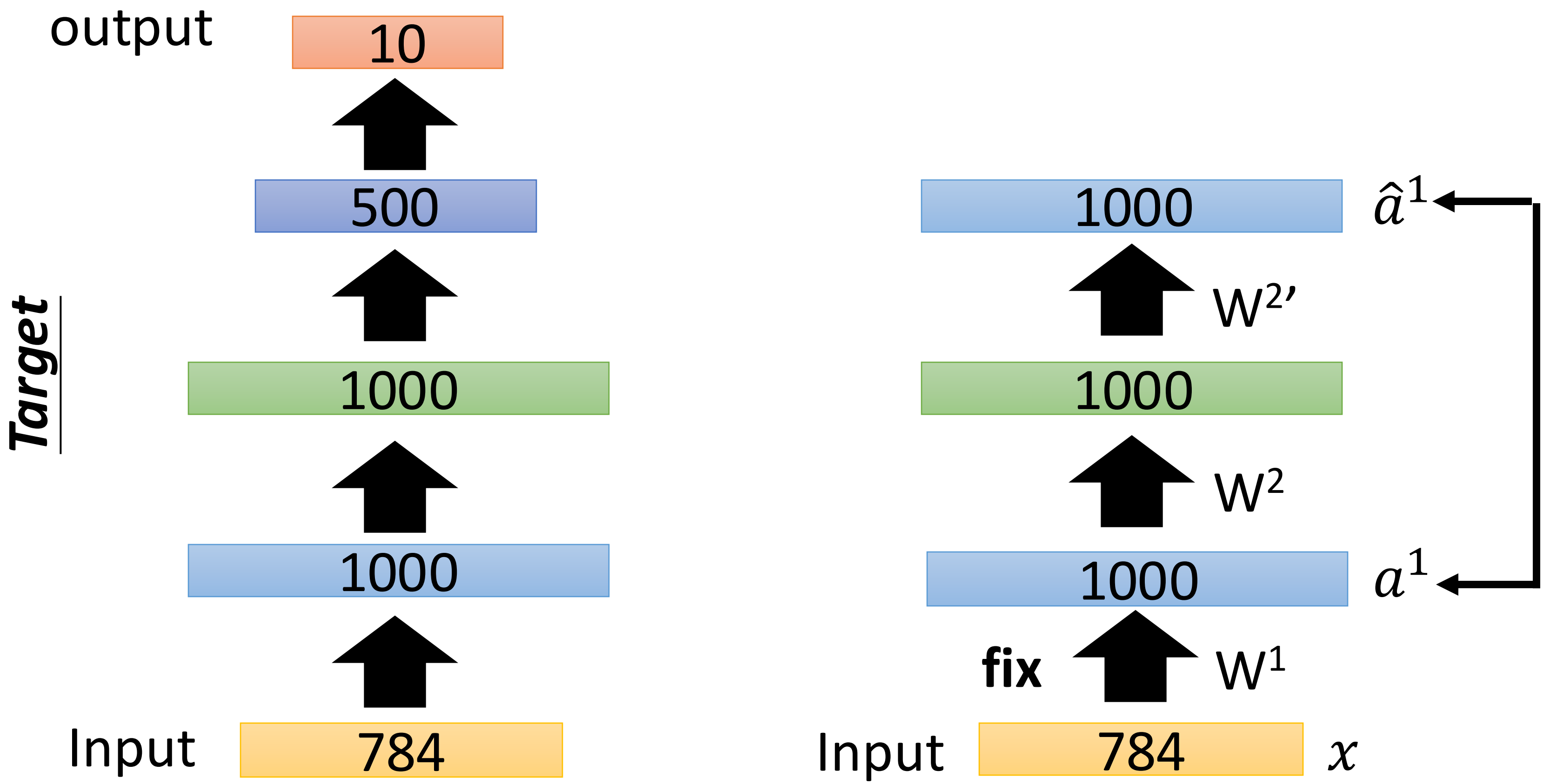
Issues

- Training deep autoencoders is challenging
  - Difficult to reach local minimum
- ***Overfitting*** is an issue in deep autoencoder, as with other deep neural networks
  - Many more model parameters than the training data
- Solutions: *greedy layer-wise pretraining*
- Alternatives:
  - Denoising autoencoder: Robust to noise/missing inputs
  - Sparse autoencoder: Sparsity through regularization
  - Contractive autoencoder: Contractive penalty

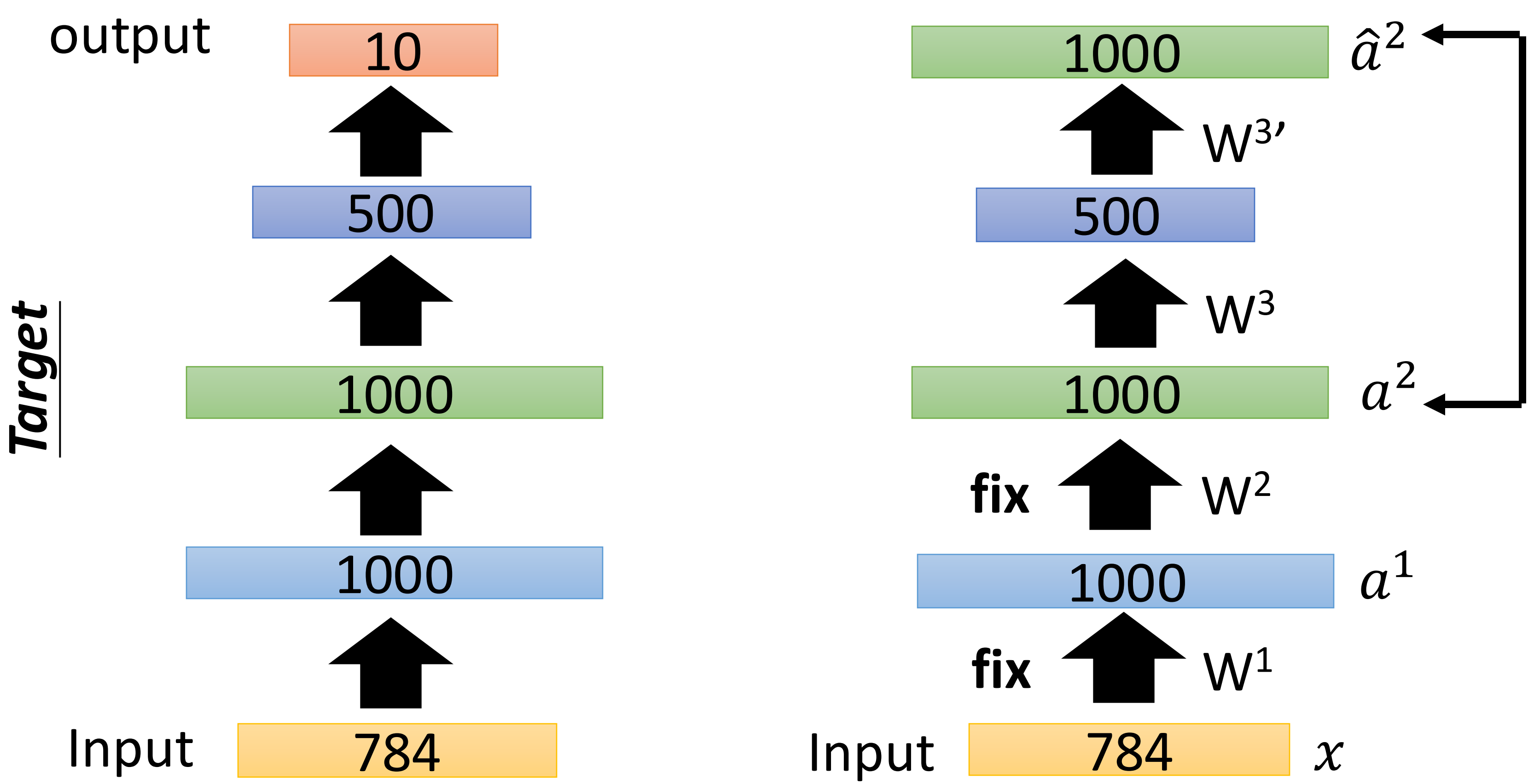
Deep autoencoder via greedy layer-wise pretraining



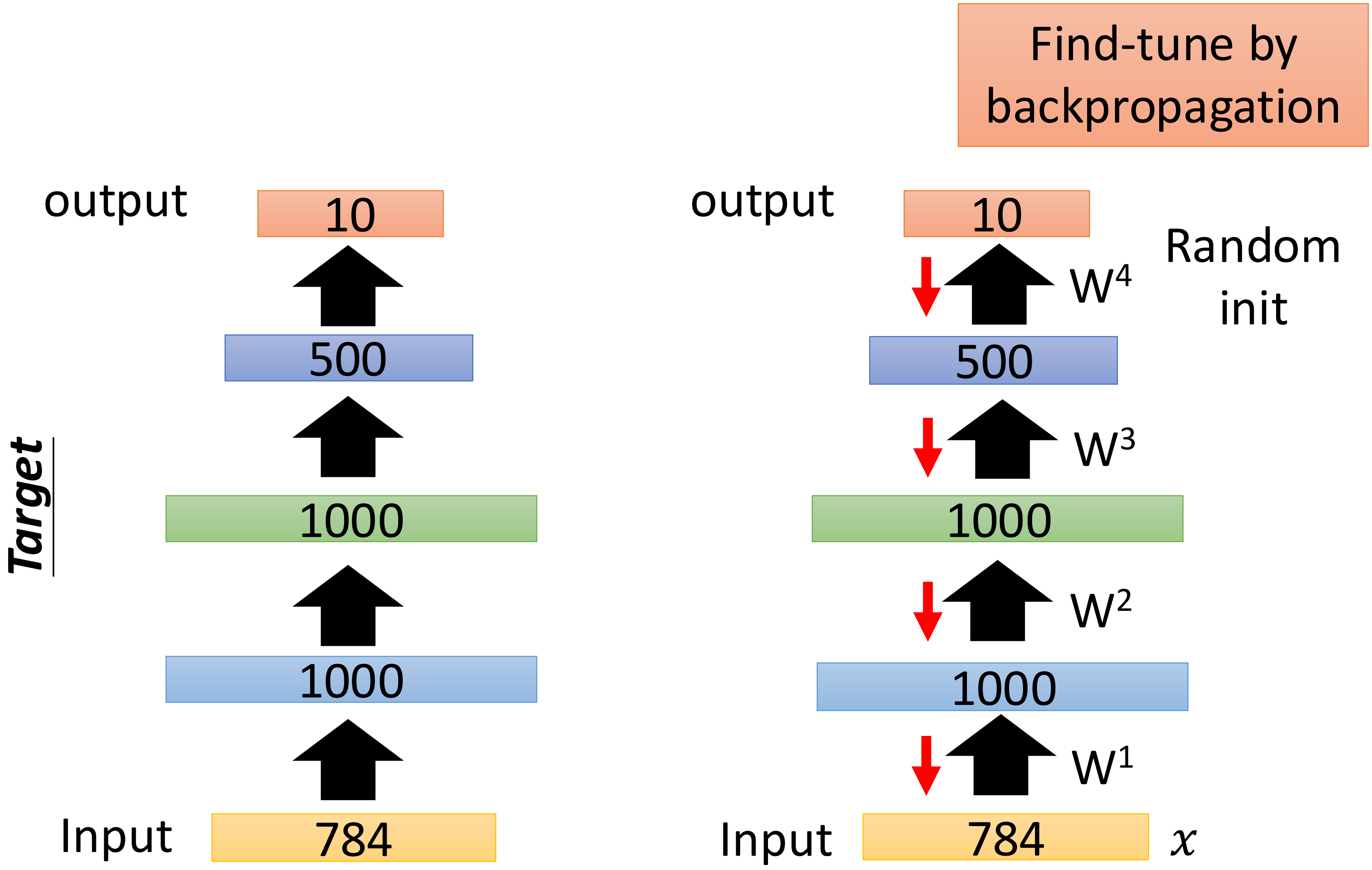
Deep autoencoder via greedy layer-wise pretraining



Deep autoencoder via greedy layer-wise pretraining



Deep autoencoder via greedy layer-wise pretraining



Lecture on Neural Networks

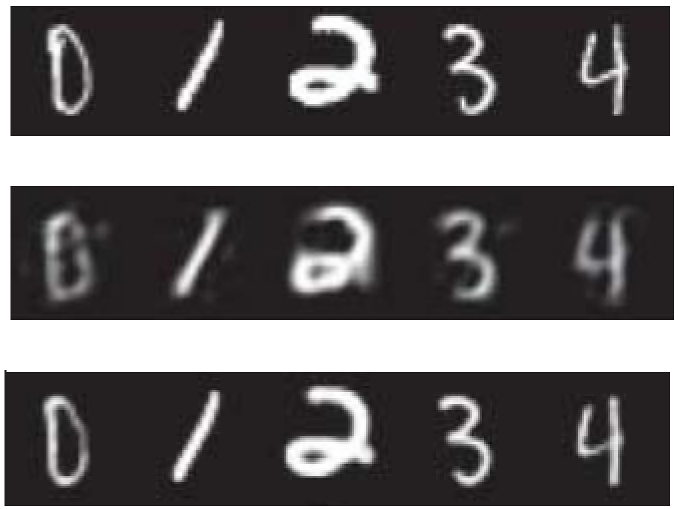
- Neural network basics
- Chain rule
- Back-propagation

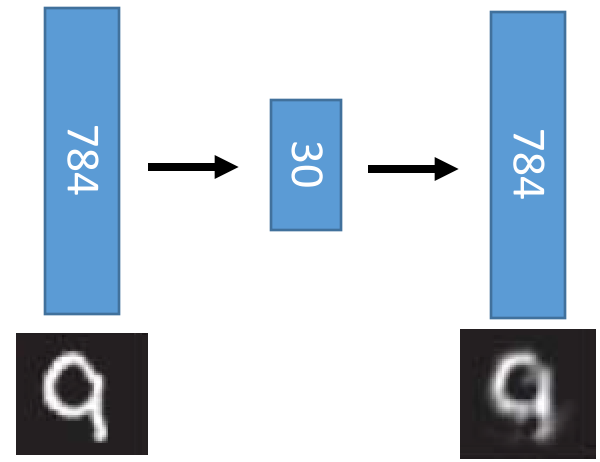
Deep Autoencoder

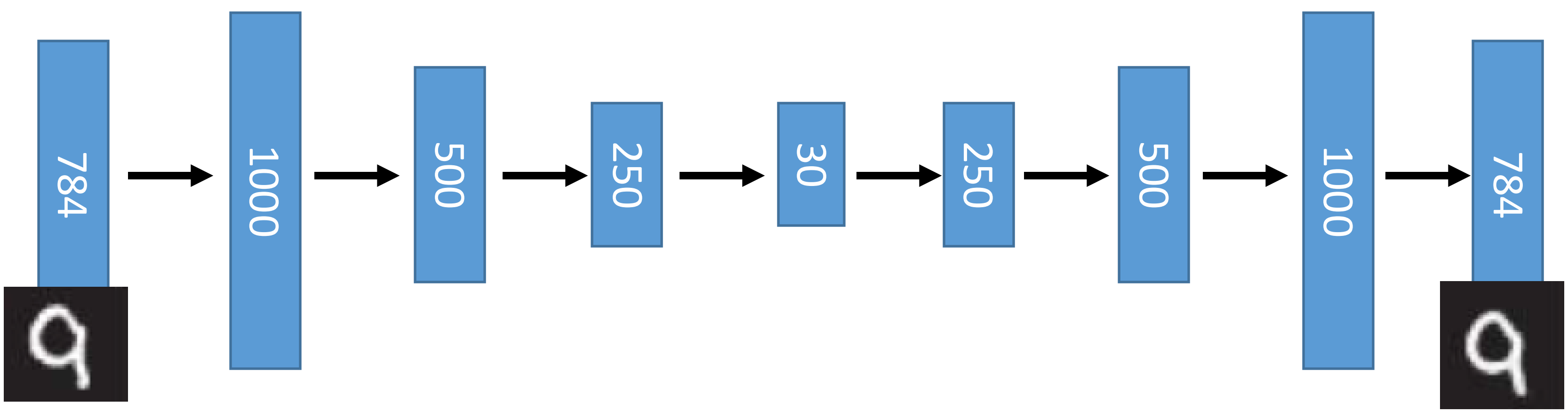
Original Image

PCA

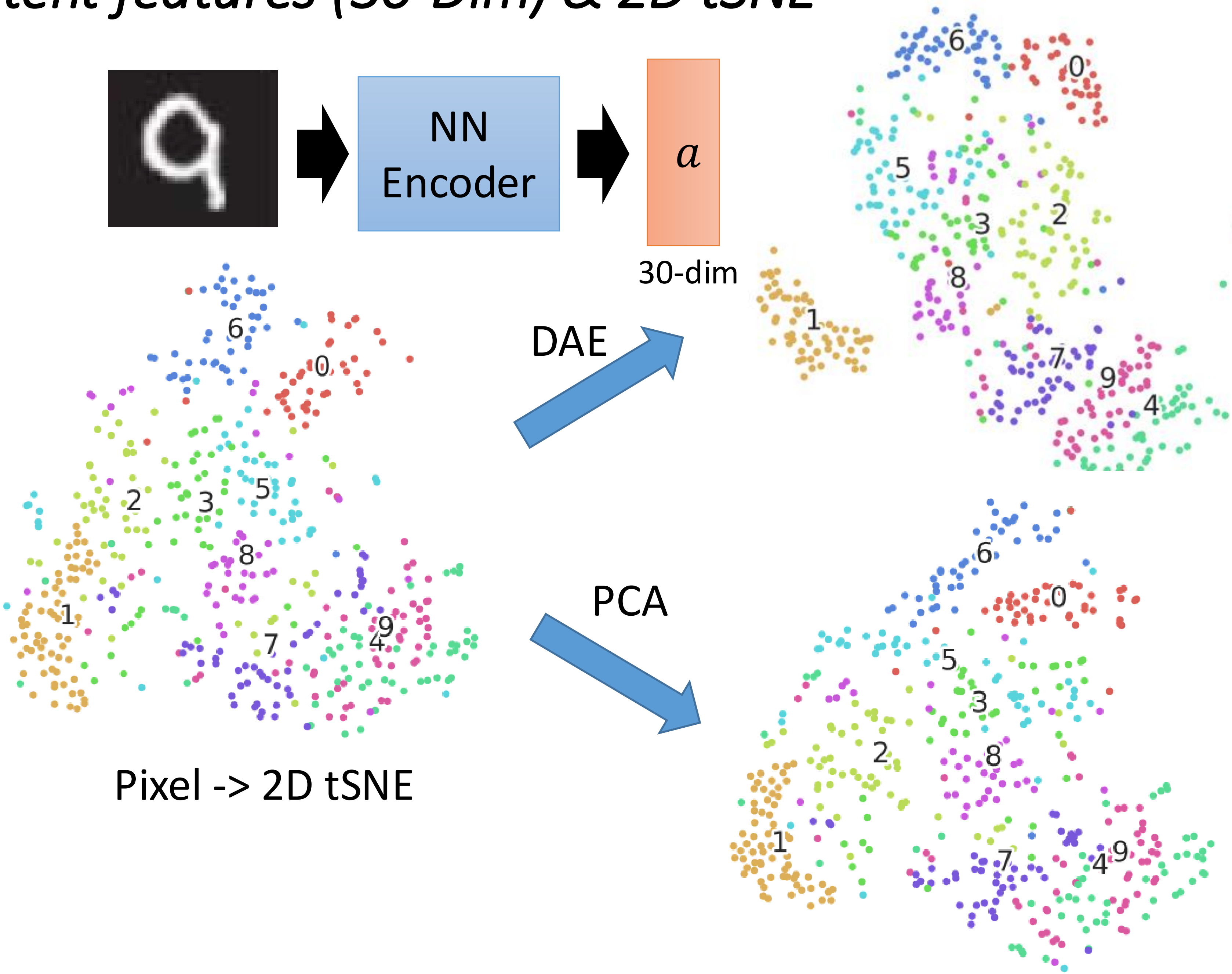
Deep Auto-encoder



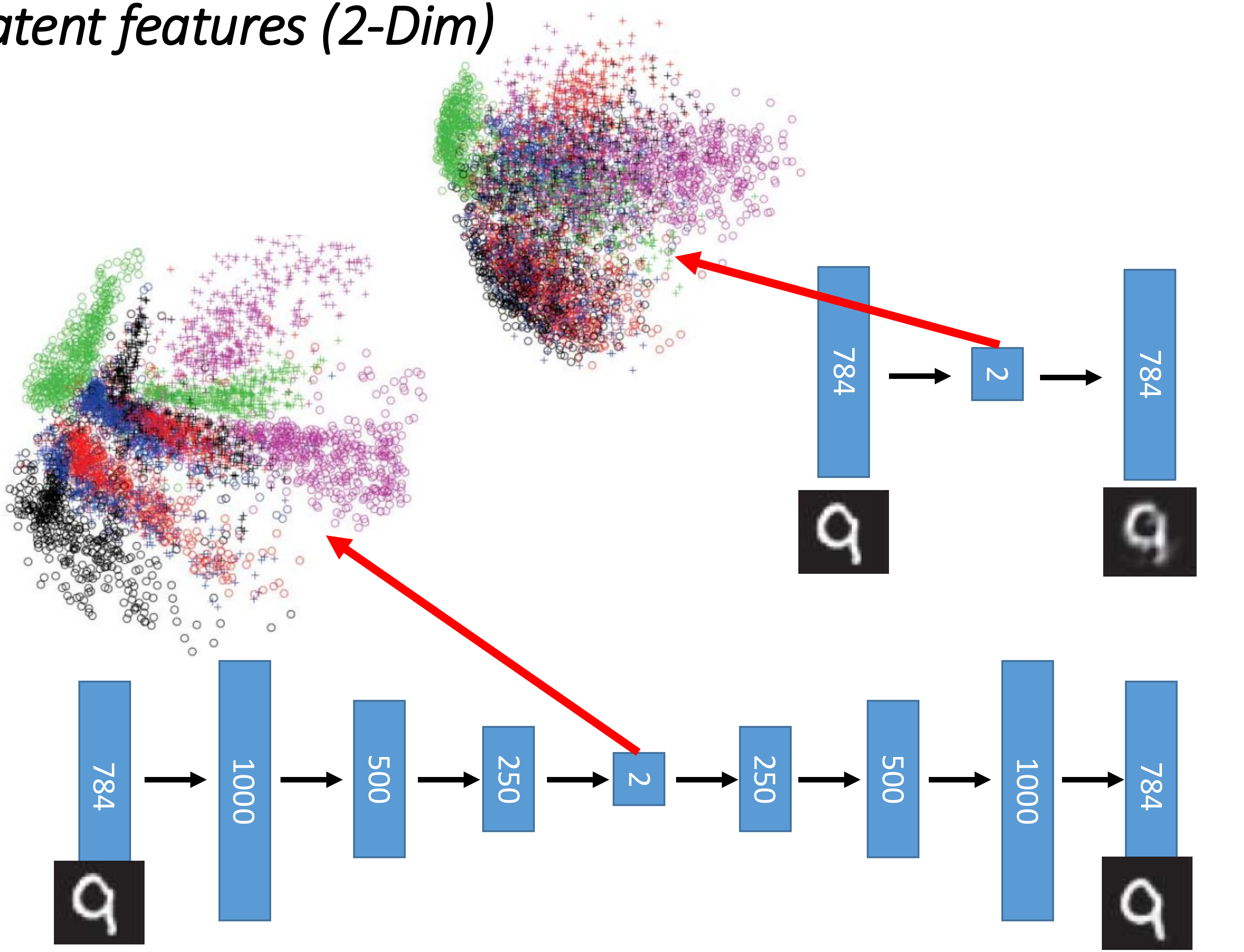




Latent features (30-Dim) & 2D tSNE



Latent features (2-Dim)



Denoising autoencoders

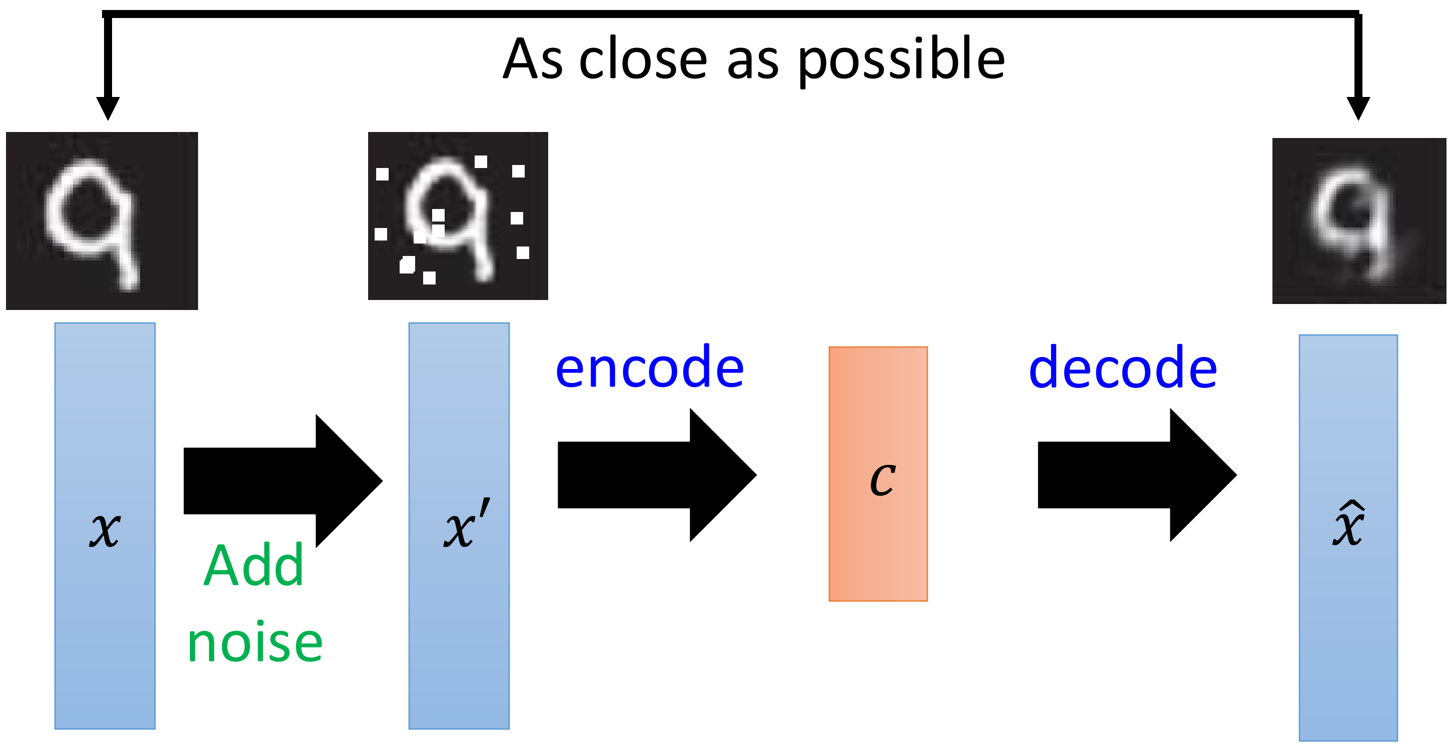
- Basic autoencoders minimizes the loss between the input x and reconstruction input x, i.e., g(f(x))
  - An autoencoder with high capacity can end up learning
    - **Identity function**: x = g(f(x))
  - A possible solution
    - Simply copy/memorize the input instead of learning it

- Denoising autoencoders minimizes the loss between noiseless x and reconstructed noisy input, i.e., g(f(x+w)), w is random noise
  - Prevent copy/memorization
  - Lead to better representations

- Same architecture as autoencoder
  - Only with different training data

Denoising autoencoders

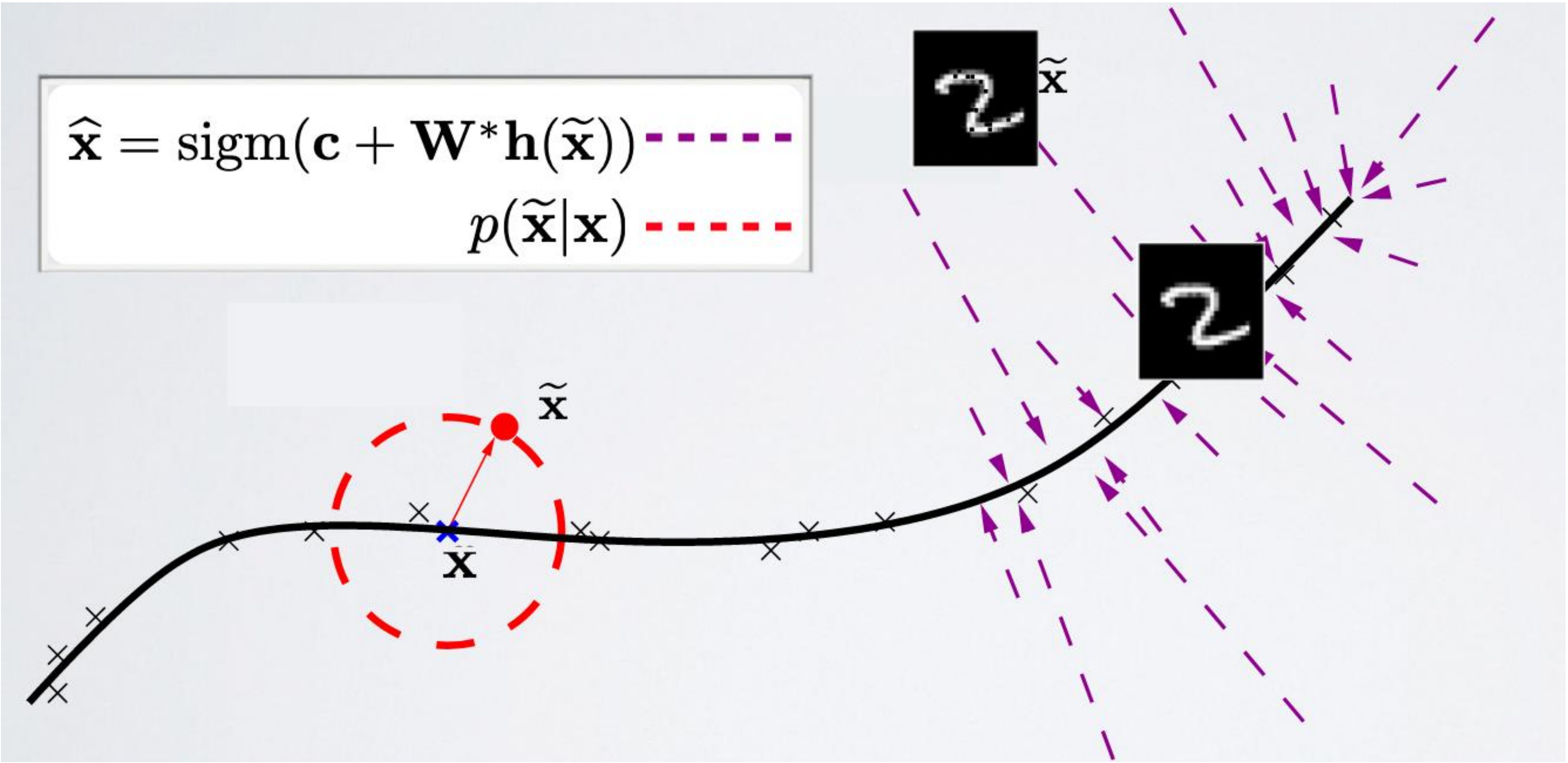
- Input clean image + noise
- Train to reconstruct the clean image



Denoising autoencoders

- Cannot memorize the input-output relationship
  - Due to that the input adds a random noise
- A denoising autoencoder actually learns a projection from a ***neighborhood of training data***
  - And then back onto the training data

Denoising autoencoders



https://ift6266h17.files.wordpress.com/2017/03/14_autoencoders.pdf

Sparse autoencoders

- Construct a loss L to penalize *activations* the network

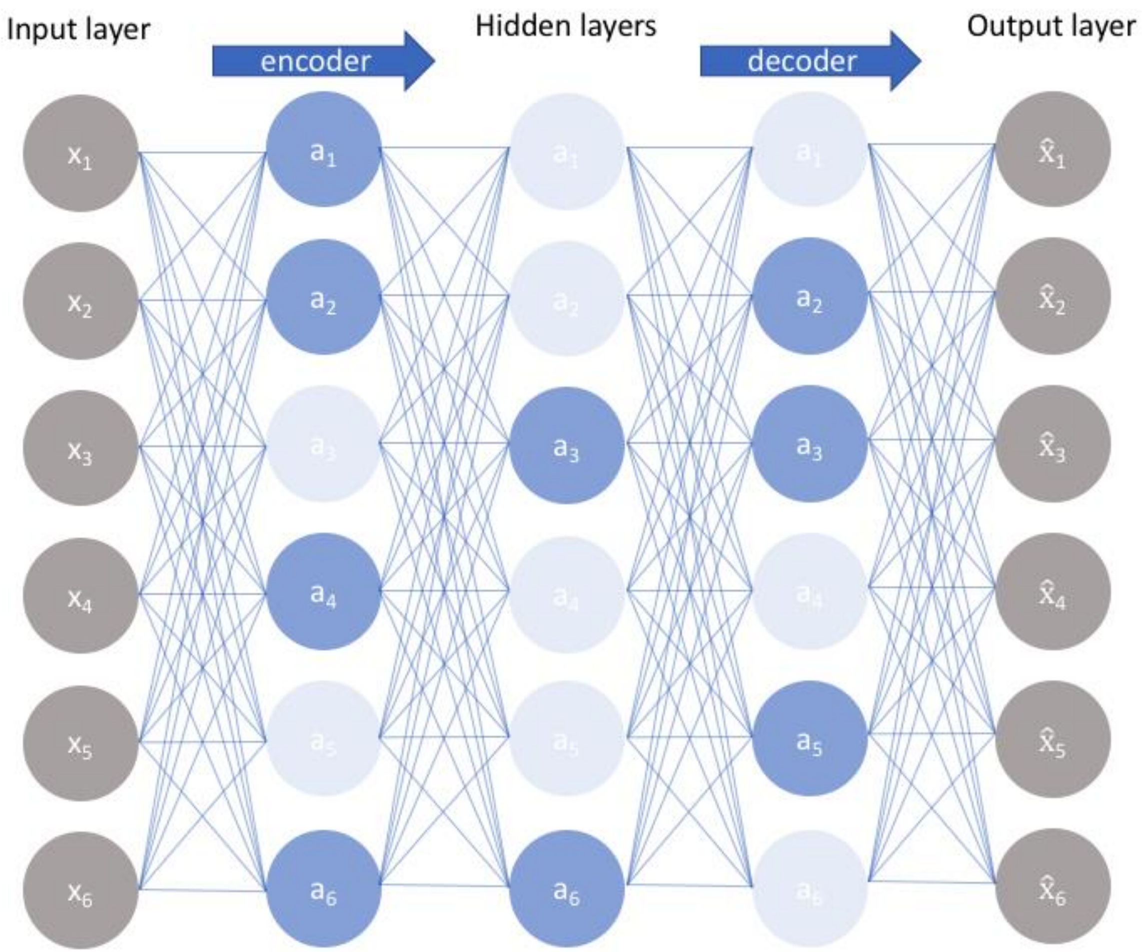
- Selectively activate regions of the network depending on the input data
  - **L1 Regularization**: Penalize the absolute value of the vector of activations *a* in layer *h* for observation

$$\mathcal{L}(x, \hat{x}) + \lambda \sum_i \left| a_i^{(h)} \right|$$

  - **KL divergence:** Use cross-entropy between average activation and desired activation

$$\mathcal{L}(x, \hat{x}) + \sum_j KL\left(\rho || \hat{\rho}_j\right)$$

Sparse autoencoders



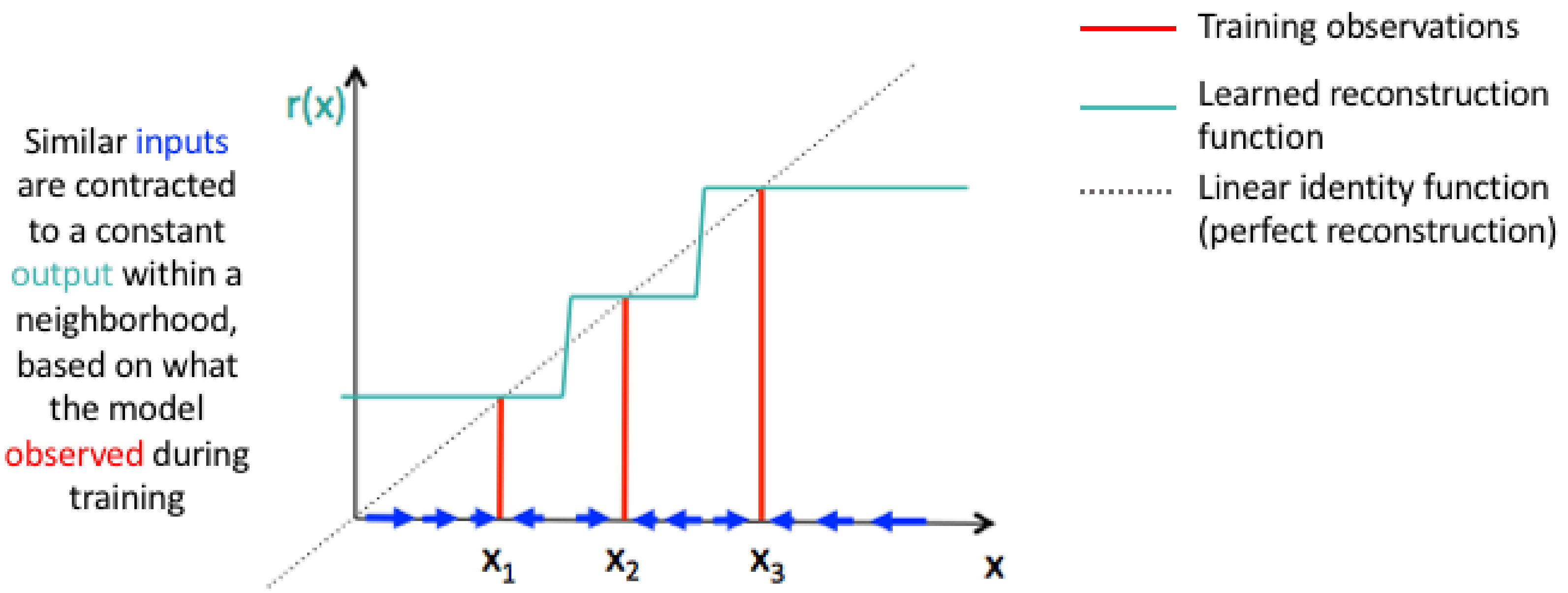
https://www.jeremyjordan.me/autoencoders/

Contractive autoencoders

- Arrange for **similar inputs to have similar activations**.
  - I.e., the *derivative of the hidden layer activations are small* with respect to the input.
- Denoising autoencoders make the *reconstruction function* (encoder+decoder) resist ***small perturbations*** of the input
- Contractive autoencoders make the *feature extraction function* (ie. encoder) resist infinitesimal perturbations of the input

$$\mathcal{L}(x, \hat{x}) + \lambda \sum_i \left\| \nabla_x a_i^{(h)}(x) \right\|^2$$

Contractive autoencoders



https://ift6266h17.files.wordpress.com/2017/03/14_autoencoders.pdf

Autoencoders comparison

Sparse autoencoder

- Prevent overfitting

Denoising autoencoder

- Easy-to-implement: a few more lines of code than regular autoencoder
- no need to compute Jacobian of hidden layers

Contractive autoencoder

- Gradient is deterministic-can use 2nd order optimizers (conjugate gradient, LBFGS, etc.)
- More stable than denoising autoencoder, which uses a sampled gradient

https://ift6266h17.files.wordpress.com/2017/03/14_autoencoders.pdf

Learning More
- Restricted Boltzmann Machine

- Neural networks [5.1] : Restricted Boltzmann machine – definition
  - https://www.youtube.com/watch?v=p4Vh_zMw-HQ&index=36&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH
- Neural networks [5.2] : Restricted Boltzmann machine – inference
  - https://www.youtube.com/watch?v=lekCh_i32iE&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=37
- Neural networks [5.3] : Restricted Boltzmann machine - free energy
  - https://www.youtube.com/watch?v=e0Ts_7Y6hZU&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=38

Learning More
- Deep Belief Network

- Neural networks [7.7] : Deep learning - deep belief network
  - https://www.youtube.com/watch?v=vkb6AWYXZ5I&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=57
- Neural networks [7.8] : Deep learning - variational bound
  - https://www.youtube.com/watch?v=pStDscJh2Wo&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=58
- Neural networks [7.9] : Deep learning - DBN pre-training
  - https://www.youtube.com/watch?v=35MUlYCColk&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=59

Implementation Using Keras

Load data and reshape images

The MNIST Dataset

- n = 60,000 training samples $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$
- Each $\mathbf{x}_i$ is a 28×28 image (reshape to 784-dim vector)

```
print('Shape of x_train_vec: ' + str(x_train_vec.shape))
Shape of x_train_vec: (60000, 784)
```

Building a fully-connected deep Autoencoder (2 Ways)

```
from keras.layers import Dense
from keras import models

model = models.Sequential()
model.add(Dense(100, activation='relu',
                    input_shape=(784,)))
model.add(Dense(20, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(784, activation='relu'))
```

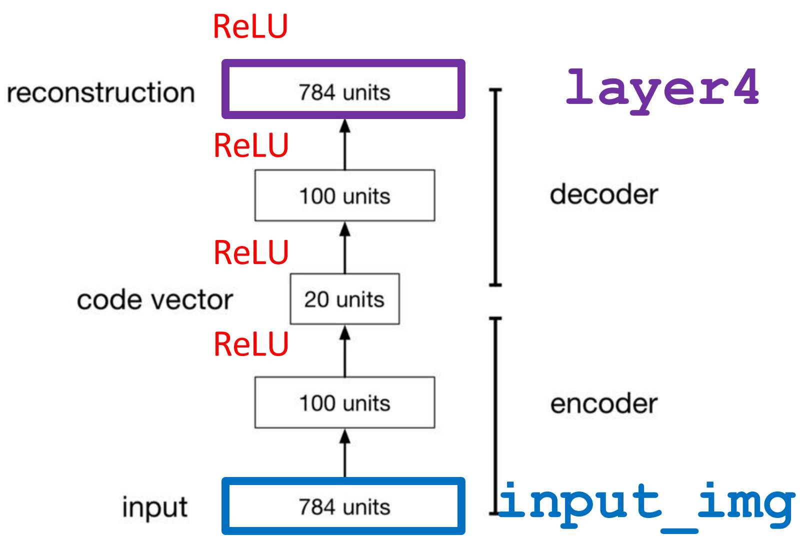
```
from keras.layers import Input, Dense
from keras import models

input_img = Input(shape=(784,))

layer1 = Dense(100, activation='relu')(input_img)
layer2 = Dense(20, activation='relu')(layer1)
layer3 = Dense(100, activation='relu')(layer2)
layer4 = Dense(784, activation='relu')(layer3)

model = models.Model(input_img, layer4)
```

Building a fully-connected deep Autoencoder



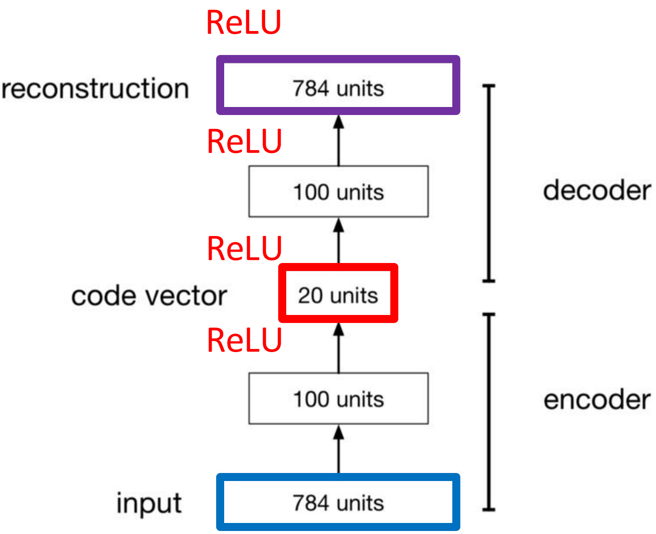

```
from keras.layers import Input, Dense
from keras import models

input_img = Input(shape=(784,))

layer1 = Dense(100, activation='relu')(input_img)
layer2 = Dense(20, activation='relu')(layer1)
layer3 = Dense(100, activation='relu')(layer2)
layer4 = Dense(784, activation='relu')(layer3)

model = models.Model(input_img, layer4)
```

Number of model parameters



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 784)	0
dense_5 (Dense)	(None, 100)	78500
dense_6 (Dense)	(None, 20)	2020
dense_7 (Dense)	(None, 100)	2100
dense_8 (Dense)	(None, 784)	79184

Total params: 161,804
Trainable params: 161,804
Non-trainable params: 0

Train the model

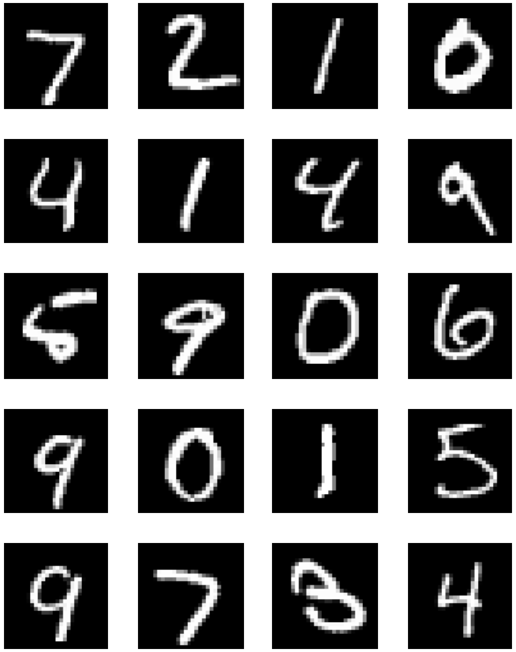
The inputs and targets are the same.

```
model.compile(optimizer='RMSprop', loss='mean_squared_error')
history = model.fit(x_train_vec, x_train_vec, batch_size=128, epochs=50)
```

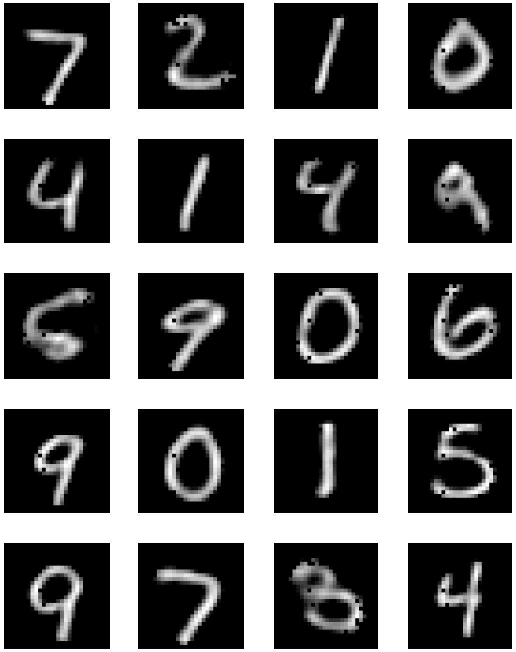
```
Epoch 1/50
60000/60000 [==============================
]                                           - 2s 32us/step - loss: 0.0390
Epoch 2/50
60000/60000 [==============================
]                                           - 2s 30us/step - loss: 0.0281
    :
Epoch 49/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.0145
Epoch 50/50
60000/60000 [==============================] - 2s 37us/step - loss: 0.0145
```

Results on the test Set

Input

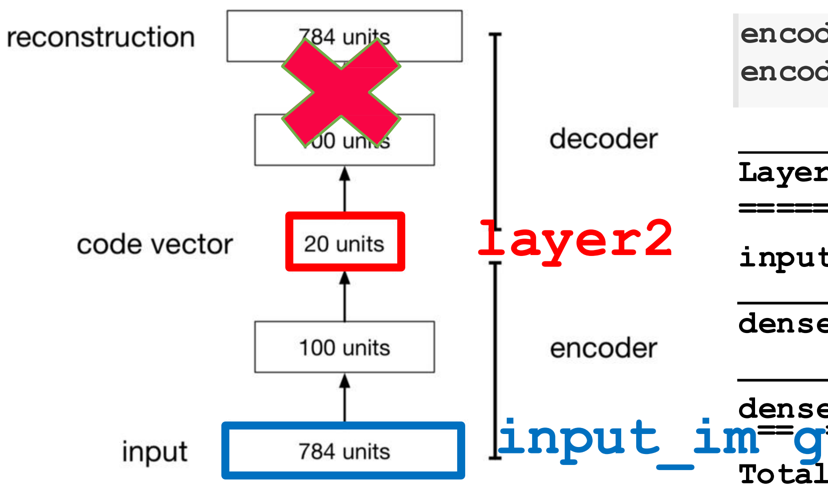


Reconstruction



Dimensionality Reduction

Extract the codes/latent features




```
encoder = models.Model(input_img, layer2)
encoder.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 784)               0
_________________________________________________________________
dense_5 (Dense)              (None, 100)               7850
                                                       0
_________________________________________________________________
dense_6 (Dense)              (None, 20)                2020
=================================================================
Total params: 80,520
Trainable params: 80,520
Non-trainable params: 0
```

Visualize the low-dim codes

Get the low-dim codes.

20-dim

784-dim

```
encoded_test = encoder.predict(x_test_vec)
print('Shape of encoded_test: ' + str(encoded_test.shape))
```

```
Shape of encoded_test: (10000, 20)
```

Project the 20-dim vectors to 2-dim by tSNE

20-dim

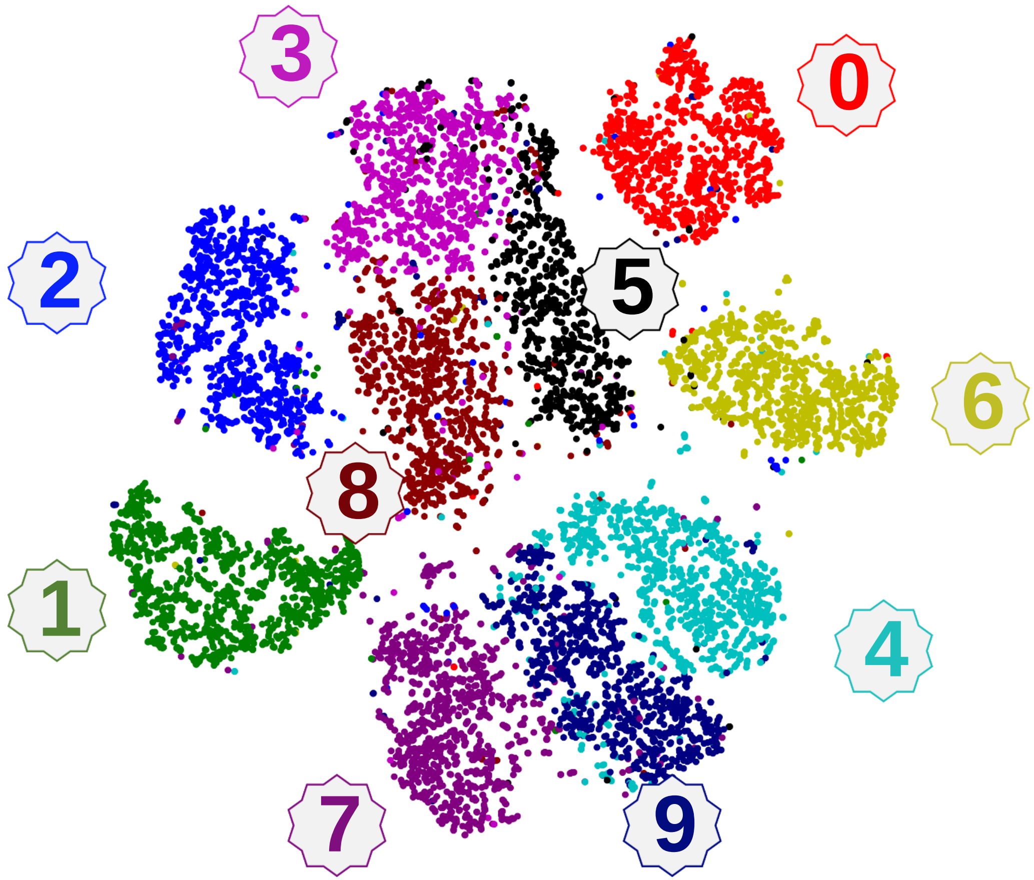
2-dim

```
from sklearn.manifold import TSNE

embedded_test = TSNE(n_components=2).fit_transform(encoded_test)
print(embedded_test.shape)
```

```
(10000, 2)
```

Visualize the low-dim codes

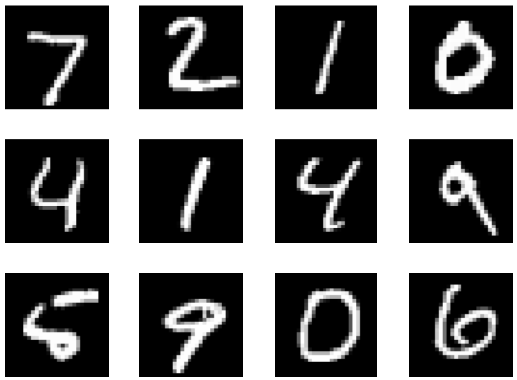


Scatter plot via the tSNE Embedding

Denoising Autoencoder

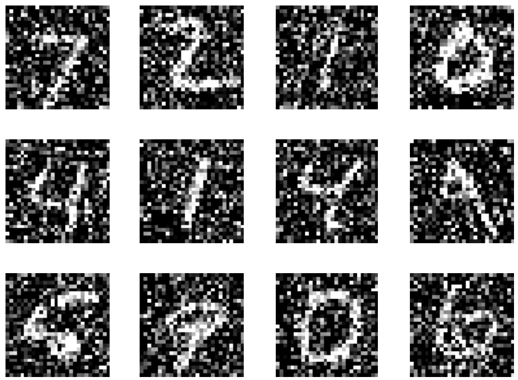
Denoising autoencoder

Original



add noise

Noisy



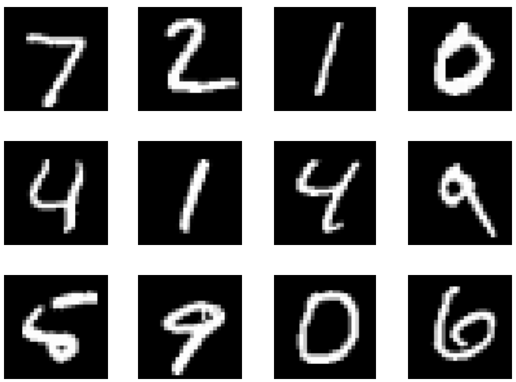
```
import numpy as np

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

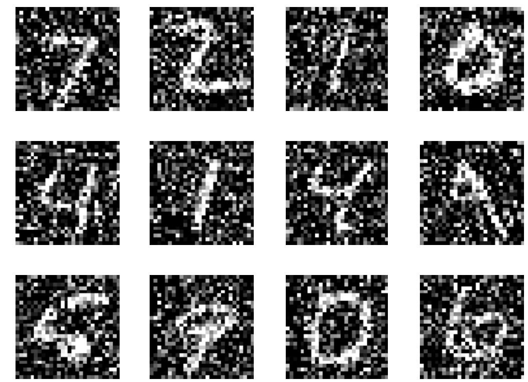
Denoising Autoencoder

Original



add noise

Noisy



Used as targets

Used as inputs

Denoising autoencoder

noisy images as inputs

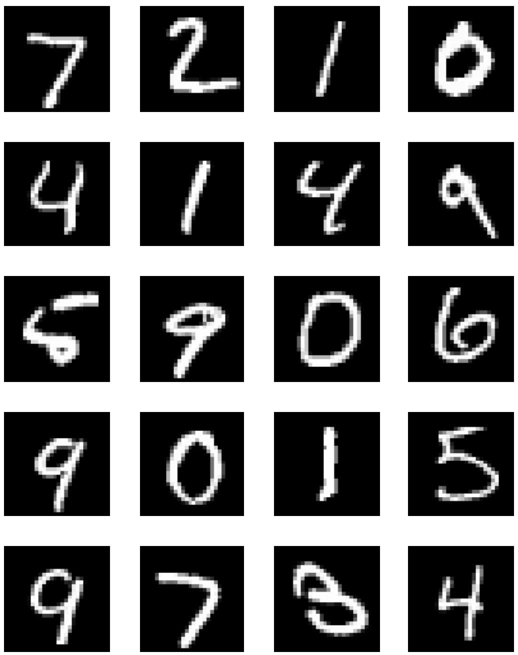
original images as targets

```
model.compile(optimizer='RMSprop', loss='mean_squared_error')
history = model.fit(x_train_noisy, x_train, batch_size=128, epochs=50)
```

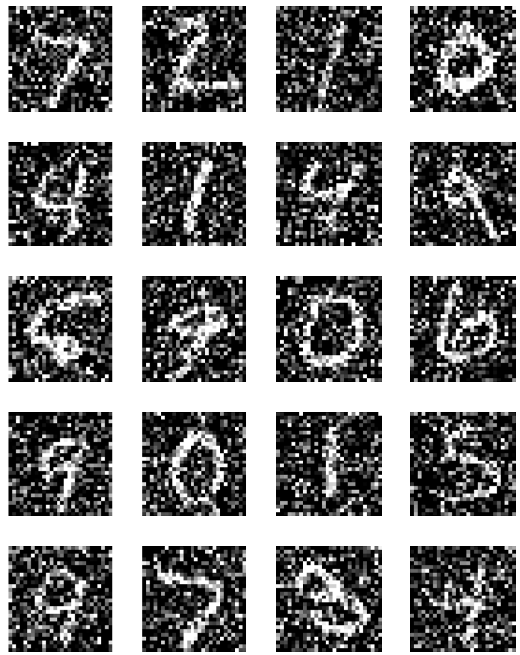
```
Epoch 1/50
60000/60000 [==============================] - 92s 2ms/step - loss: 0.0293
Epoch 2/50
60000/60000 [==============================] - 141s 2ms/step - loss: 0.0162
Epoch 3/50
60000/60000 [==============================] - 130s 2ms/step - loss: 0.0144
    .
    .
    .
Epoch 49/50
60000/60000 [==============================
            ]                               - 143s 2ms/step - loss: 0.0106
Epoch 50/50
60000/60000 [==============================
            ]                               - 122s 2ms/step - loss: 0.0106
```

Results on the test Set

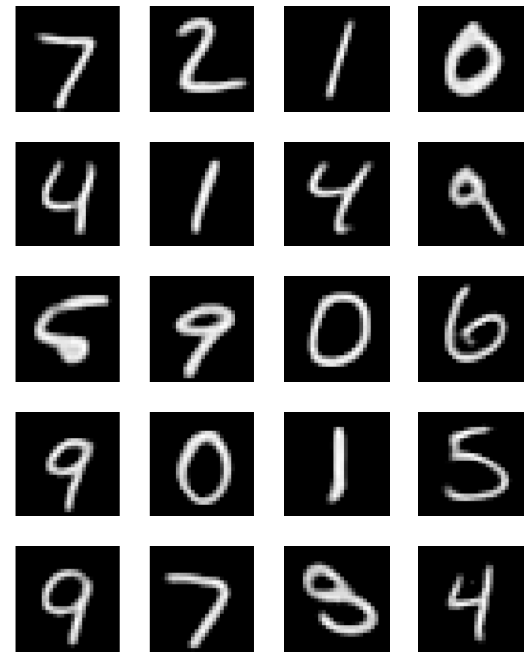
Original



Noisy



Reconstructed



Autoencoder vs denoising autoencoder