Lecture Six: Dimensionality Reduction (DR) and Manifold Learning (MAL)

COMP3032 Machine Learning

Why DR: too many features

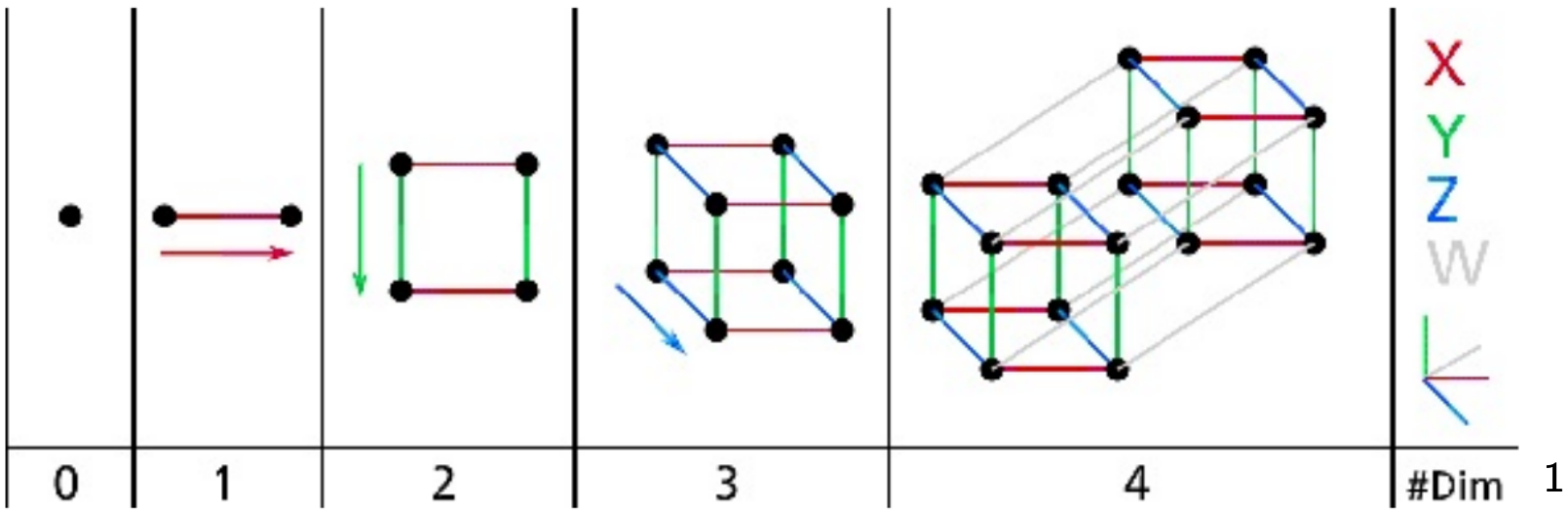
- Many Machine Learning problems often involve thousands, sometimes millions of features
- make training extremely slow
- increased computation complexity
- make it more difficult to find a good solution
- referred to as the curse of dimensionality

Why DR: visualization

- extremely useful for data visualization
- people are used to living in three dimensional space
- easier to understand two or three dimensional objects than four dimensional and above
- make it possible to plot a condensed view of a high-dimensional training set
- acquire some insights by detecting some patterns such as clusters
- essential to communicate to people who are not data scientists

Why DR: visualization

From 1D to 4D



[1] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

Why DR: speed up

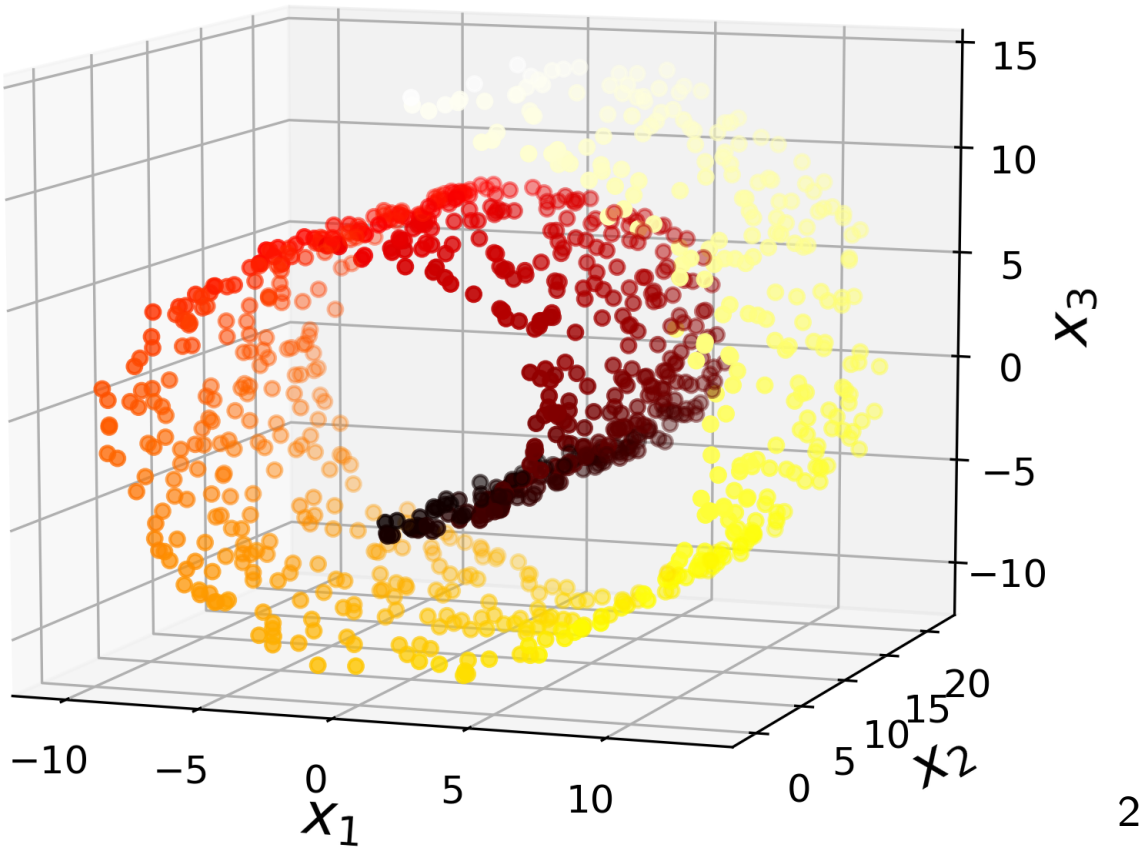
- it is often possible to reduce the number of features considerably
- while preserving as much relevant information as possible
- speed up training
- cause some information loss

Manifold Learning (MAL)

Plays an important role in analyzing complex high-dimensional data by providing a way to represent it in a more interpretable and manageable form.

- data is often collected in high-dimensional spaces
- challenge to visualize, analyze, and model
- aim to discover the underlying structure of high-dimensional data
- find lower-dimensional representations of the data
- while preserving its essential structure

Swiss roll



Swiss roll

[2] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

What is MAL

The technique of modelling the *manifold* where the training instances lie

- this modelling process is called *manifold learning*
- adopted by many DR algorithms
- rely on the *manifold hypothesis*
  - real-world high-dimensional datasets mostly lie closer to a much lower dimensional manifold
  - an important assumption

General definition

Definition

Suppose $d < n$, a d-dimensional manifold is a *subset* of the n-dimensional space that locally resembles a d-dimensional hyperplane.

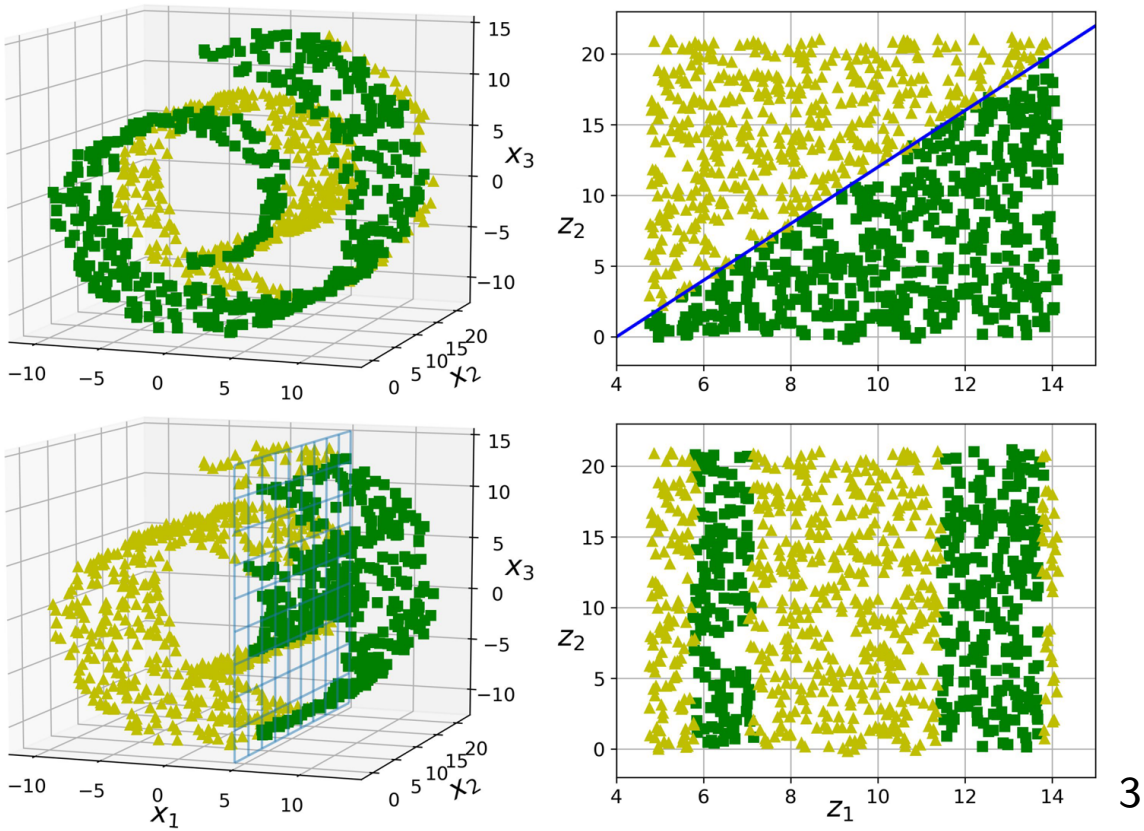
- Swiss roll is an instance of *2D manifold*
- A *2D manifold* is a 2D shape that can be *bent* and *twisted* in a *higher*-dimensional space
- $d = 2$, $n = 3$
- locally resembles a 2D plane, but rolled in the third dimension

Manifold Learning

An implicit assumption

- it is simpler to model the task (e.g. classification or regression) in the lower-dimensional space of the manifold
- e.g. the Swiss roll, split into two classes, 3D and 2D, as shown in the figure
- true
  - in the original 3D space, the decision boundary look very complex
  - in the 2D unrolled manifold space, the decision boundary is simpler, a straight line
- not always true
  - in original 3D space, $x_1 = 5$
  - in unrolled 2D space, 4 lines

Manifold Learning



[3]

The decision boundary in different dimensions

[3] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

PCA: main idea

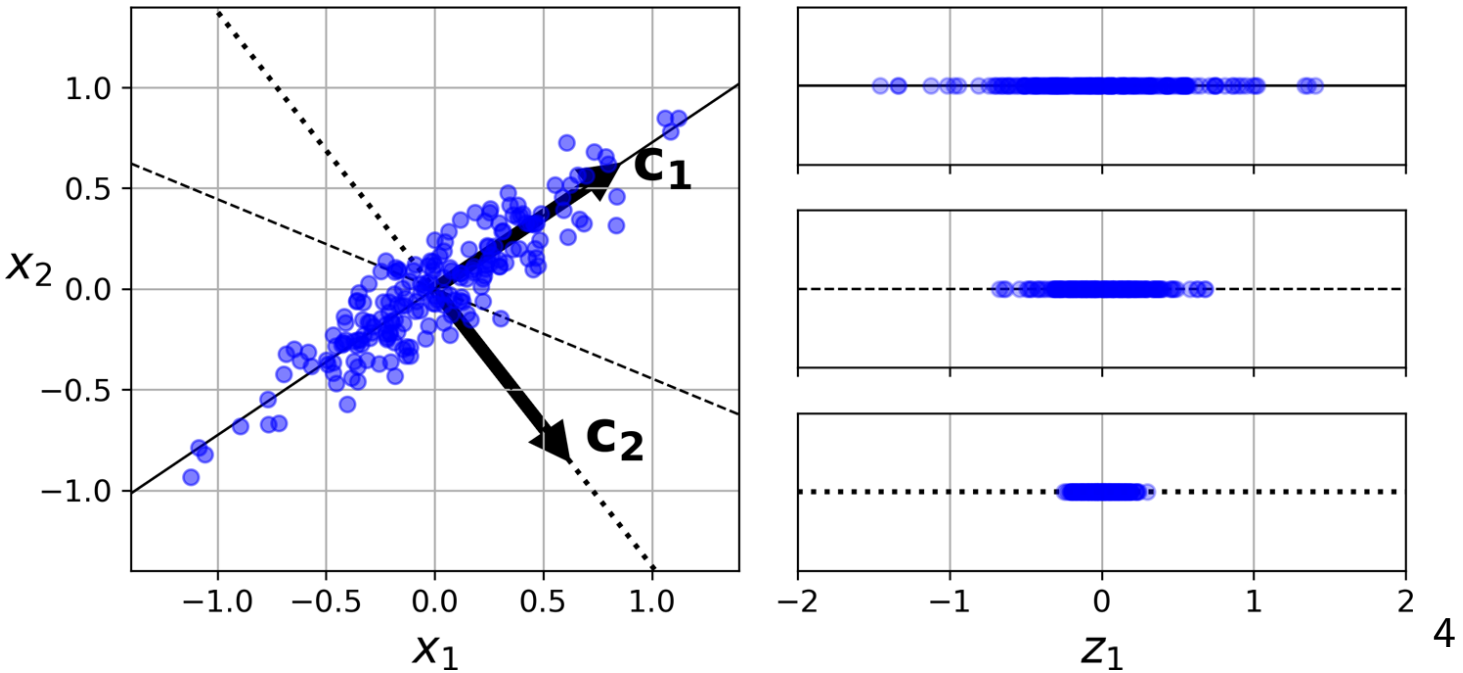
- *Principal component analysis* (PCA) is currently the most popular dimensionality reduction algorithm
- *main idea*:
  - find the hyperplane that lies closest to the dataset
  - *project* the dataset onto the hyperplane

PCA: find the hyperplane

How to find that hyperplane?

- An example: a simple 2D dataset is represented on the left in the figure
- three different axes (i.e., 1D hyperplanes)
- on the right is the result of the projection of the dataset onto each axe
  - the projection onto the solid line preserves the maximum variance
  - the projection onto the dotted line preserves little variance
  - the projection onto the dashed line preserves medium variance.

Preserving the Variance



- solid line: most variance.
- dashed line: medium variance.
- dotted line: least variance.

Select the axis with the greatest possible amount of variance.

[4] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

PCA: find the hyperplane

how to find that hyperplane?

- select the hyperplane (axis) that *preserves* the *greatest* ammount of *variance*
- likely lose less informaiton
- it has the *minimum* mean squared distance between the *original* dataset and the *projection*
- simple idea

PCA: find the hyperplane

- in the example, it is the solid line
- then find a second axis, orthogonal to the first one
- preserves the largest amount of remaining variance
- the dotted line
- for a higher-dimensional dataset, PCA continues to find a third axis, orthogonal to both previous axes, and a fourth, so on
- as many axes as the number of dimensions in the dataset

PCA: principal components

The unit vector that defines the i^{th} axis is called the i^{th} principal component (PC)

- the first PC is *unit vector* c_1
- the second PC is *unit vector* c_2
- all PCs are *orthogonal* to each other
- The number of PCs is the same as the number of dimensions in the feature space

Singular Value Decomposition (SVD)

How to find the principal components of a training set?

- The standard matrix factorisation called *singular value decomposition* (SVD) can find all the PCs
- the SVD decomposes the training set matrix X into the dot products of three matrices:

$$X = U \cdot \Sigma \cdot V_T,$$

where

$$V = (c_1 \;\; c_2 \;\; \dots \;\; c_n)$$

contains all the PCs that we need

Singular Value Decomposition (SVD)

Use the NumPy's `linalg.svd()` function:

```
import numpy as np

x1 = 2 * np.random.rand(100, 1) # generate linear random data
x2 = 4 + 4 * x1 + np.random.randn(100, 1)
x3 = x1**2 + x2**2

X = np.c_[x1, x2, x3]

X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

Singular Value Decomposition (SVD)

- It is necessary to centre the dataset when using PCA
- $c1$ and $c2$ are the first two principal components
- `np.linalg.svd(X_centered)` returns:

  - `U`, `s` and $\texttt{V}_t$,

  - corresponding to the SVD decomposition.

Projecting Down to d Dimensions

Once the PCs are obtained

- the dimensionality of the dataset can be reduced down to d-dimensions
- by projecting the dataset onto the hyperplane corresponding to the **first** d principal components
- this hyperplane ensures the *preservation* of the *maximum variance*

Projecting Down to d Dimensions

- the actual projection down to d dimensions is achieved by the following equation
- i.e. computing the dot product of the training set (X) by the matrix containing the first d columns (principal components) of V (W_d):

$$X_{d\text{-proj}} = X \cdot W_d$$

Projecting Down to d Dimensions

Consider again the previous Python example:

```
W2 = Vt[:2].T
X2D = X_centered.dot(W2)
```

- get the plane defined by the first two PCs
- project onto the plane as defined by the first two principal components

Projection via `fit_transform()` Method

- The `PCA` class in `scikit-learn` already implements the PCA using SVD decomposition
- PCA already centres the data by default

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
pca.components_

array([[ 0.01243499, 0.06065888, 0.99808109],
[-0.32259327, -0.94454256, 0.0614242 ]])
```

The `components_` variable

- The principal components can be accessed using the `components_` variable
- `components_` is the transpose of W_d
- contains one row for each of the first d principal components
- returns PCs as *horizontal vectors*
- each vector has the *same dimension* as the *initial* feature space

Explained Variance Ratio

- the *explained variance ratio* of each PC is another piece of useful information
- it quantifies the proportion of the dataset's variance that lies along the axis of each PC
- can be accessed via the `explained_variance_ratio_` of the `PCA` class

```
pca.explained_variance_ratio_

# array([9.99877079e-01, 9.80841942e-05])
```

Choosing the Right Number of Dimensions

How ro determine the *optimal number* of dimensions to reduce down to?

- a strategy is to choose the number of dimensions that *adds up* to a *sufficiently significant* portion of the variance (e.g., 95%)
- another is down to 2 or 3 for data visualization

Choosing the Right Number of Dimensions

Consider the MNIST dataset:

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.target = mnist.target.astype(np.uint8)
X = mnist["data"]
y = mnist["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y)

pca = PCA()
pca.fit(X_train)

cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1

pca = PCA(n_components=d)
X_reduced = pca.fit_transform(X_train)
```

Choosing the Right Number of Dimensions

- train the `PCA` instance on `X_train`
- performs PCA without reducing dimensionality
- get the cumulative sum of explained variance ratio
- computes the minimum number of dimensions required to preserve 95% of the training set's variance
- extract only the first d PCs
- train again

Choosing the Right Number of Dimensions

- alternatively, use the `PCA()` object initialisation
- "`n_components=0.95`" specifies the overall variance ratio to preserve
- it is a float between 0.0 and 1.0

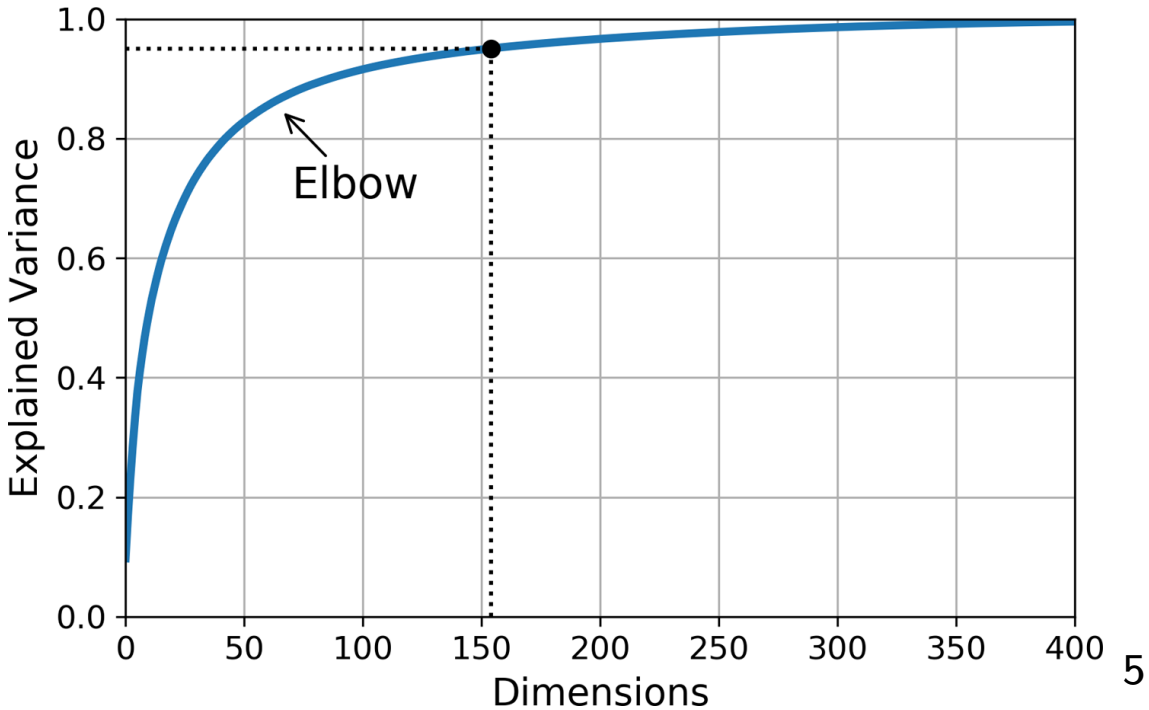
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Choosing the Right Number of Dimensions

Another option, plot the explained variance as a function of the number of dimensions:

- plot `cumsum` as a function of its index, i.e., the number of dimensions
- usually has an elbow in the curve, where the explained variance's growth slows down
- down to about 100 dimensions don't lose too much explained variance

Plot the explained variance



[5]

Explained variance as a function of the number of dimensions

[5] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

PCA for Compression

- After dimensionality reduction, the training set takes up much less space
- apply PCA to the MNIST dataset that retains 95% of the explained variance
- each instance has only over 150 features, instead of the original 784 features
- dataset is now less than 20% of its original size
- Most of the variance is preserved
- a reasonable compression ratio
- size reduction can greatly speed up a classification algorithm

Inverse the transformation

- it is possible to inverse transformation of the PCA projection
- decompress the reduced dataset to 784 features
- by applying the inverse transformation of the PCA projection
- not exactly the same (due to the projection loss), but should be close to the original data
- reconstruction error: the mean squared distance between the original and the reconstructed data (compressed and then decompressed)
- The equation of the inverse transformation

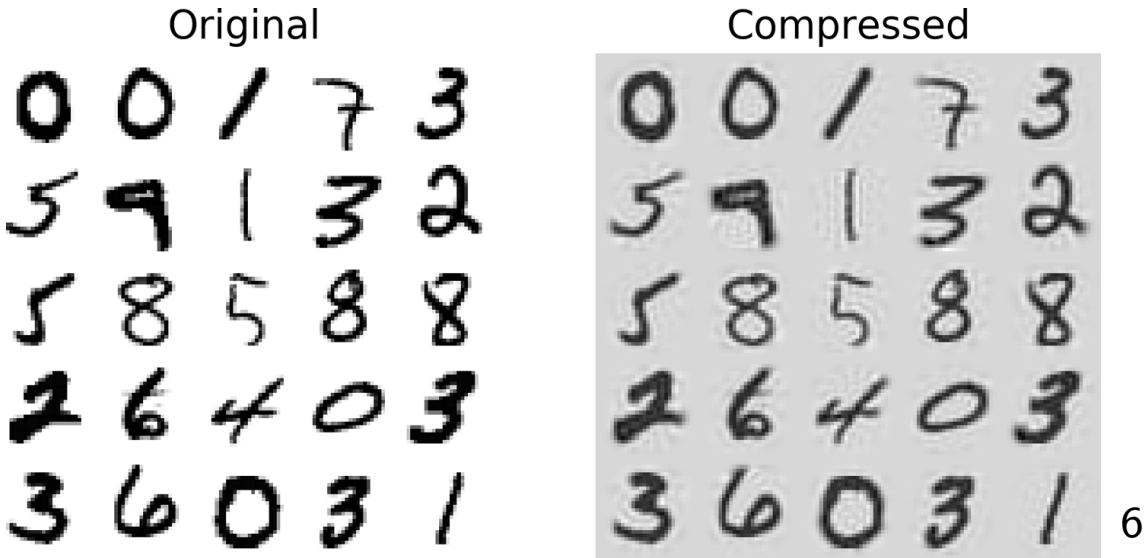
$$X_{d\text{-recovered}} = X_{d\text{-proj}} \cdot W_d^T$$

Inverse the transformation

The *inverse_transform*() method

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression



[6]

MNIST compression that preserves 95% of the variance

[6] From Géron, A. (2019) *hands on machine learning with scikit learn and tensor flow*.

Incremental PCA

Incremental PCA (IPCA) algorithm does not require to fit in memory the whole training set

- the training set is split into *mini-batches*
- feed an IPCA algorithm one mini-batch at a time
- useful for large training sets

Incremental PCA

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)

for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Incremental PCA

- the code splits the MNIST dataset into 100 mini-batches
- using NumPy's array_split() function
- feeds them to IncrementalPCA class to reduce the dimensionality down to 154 dimensions
- must call the partial_fit() method with each mini-batch (not fit())

Linear Discriminant Analysis (LDA)

Main idea

- a supervised method for DR for classification problems
- during training it learns the most distinct axes between the classes
- these axes define a hyperplane onto which to project the data
- the projection keeps classes as far apart as possible
- useful for reducing dimensionality before running another classification algorithm