CH2. Mathematical Building Blocks of Deep-learning (2)

Il-Youp Kwak
Chung-Ang University

Data representations for neural networks

Tensor is a container for data

0D tensors (Scalars)

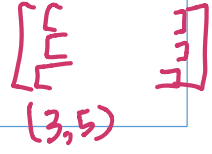
```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

1D tensors (Vectors)

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

2D tensors (Matrices)

```
>>> x = np.array([[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]])
>>> x.ndim
2
```



3D tensors and higher-dimensional tensors

```
>>> x = np.array([[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]],
[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]],
[[5, 78, 2, 34, 0],
[6, 79, 3, 35, 1],
[7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Key attributes for *tensors*

x.ndim

Number of axes (rank): can be checked by .ndim for numpy objects

dimension

Shape: tuple of integers that describes how many dimensions the tensor has along each axis. Can be checked by .shape for numpy objects

Data type (dtype): type of the data in tensor. Ex: float32, uint8, float64, **and so on.**

Key attributes for *tensors*

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

>>> print(train_images.ndim)
3

>>> print(train_images.shape)
(60000, 28, 28)

>>> print(train_images.dtype)
uint8
```

→ integer ✓

Displaying Digits

```
digit = train_images[4]

import matplotlib.pyplot as plt
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

visualization

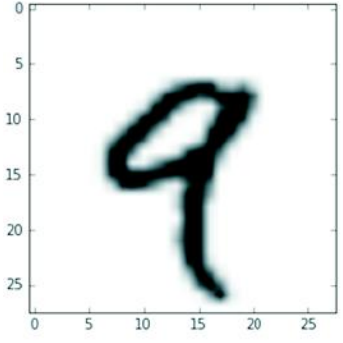


Figure 2.2 The fourth sample in our dataset

chapter02_mathematical-building-blocks.i

File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents

+ Code + Text Copy to Drive

RAM Disk Editing

```
x.ndim
```

Key attributes

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
train_images.ndim
```

```
train_images.shape
```

```
train_images.dtype
```

→ 3

→ (60000, 28, 28)

→ 'uint8' → $2^8 = 256$

Displaying the fourth digit

```
import matplotlib.pyplot as plt
digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

```
train_labels[4]
```

Manipulating tensors in NumPy

✓ 1s completed at 1:16 PM

Manipulating tensors in Numpy

Tensor slicing

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape)
(90, 28, 28)
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

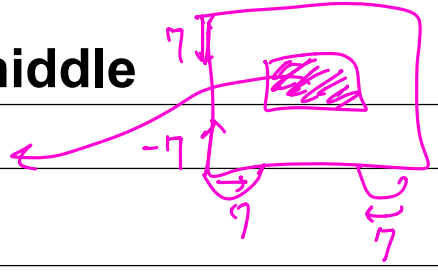
Manipulating tensors in Numpy

Select 14×14 pixels in bottom-right corner of all images

```
my_slice = train_images[:, 14:, 14:]
```

Crop 14 × 14 pixels centered in the middle

```
my_slice = train_images[:, 7:-7, 7:-7]
```




```
plt.imshow(my_slice[4], cmap=plt.cm.binary)
plt.show()
```

chapter02_mathematical-building-blocks.i

File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents

+ Code + Text Copy to Drive

RAM Disk Editing

```
[47] train_labels[4]
```

```
9
```

Manipulating tensors in NumPy

Loading...

```
my_slice = train_images[10:100]
my_slice.shape
```

```
[ ] my_slice = train_images[10:100, :, :]
my_slice.shape
```

```
[ ] my_slice = train_images[10:100, 0:28, 0:28]
my_slice.shape
```

```
[ ] my_slice = train_images[:, 14:, 14:]
```

```
[ ] my_slice = train_images[:, 7:-7, 7:-7]
```

The notion of data batches

```
[ ] batch = train_images[:128]
```

```
[ ] batch = train_images[128:256]
```

0s completed at 1:30 PM

The notion of data batches

The 1st axis of data tensors in deep-learning will be sample axis

(60000, . ,)

Deep-learning models don't process an entire dataset at once

이 개수(sample 축의 데이터 개수 기준)로 batch 사이즈 결정.

```
# 1st batch
batch = train_images[:128]
# 2nd batch
batch = train_images[128:256]
# nth batch
batch = train_images[128 * n:128 * (n + 1)]
```

Real-world examples of data tensors

Vector data – 2D tensor of shape (samples, features)

Timeseries data or sequence data – 3D tensors of shape (samples, timesteps, features)

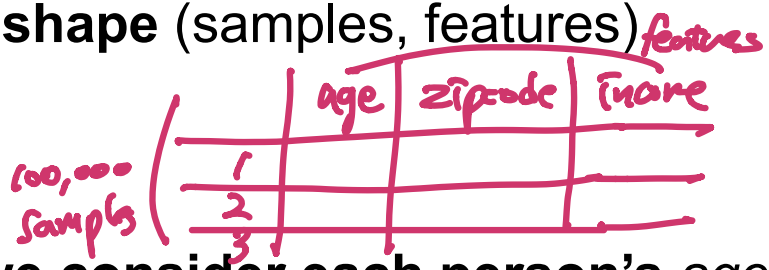
Images – 4D tensors of shape (samples, height, width, channels) **or** (samples, channels, height, width)

RGB

Video – 5D tensors of shape (samples, frames, height, width, channels) **or** (samples, frames, channels, height, width)

Vector Data

Vector data – 2D tensor of shape (samples, features)



Example:

Dataset of people, where we consider each person's *age*, *ZIP code*, **and** *income*.
Each person can be characterized as a vector of 3 values
Entire dataset of 100,000 people can be stored in a 2D tensor of shape (100000, 3).

Timeseries data or sequence data

Timeseries data or sequence data – 3D tensors of shape (samples, timesteps, features)

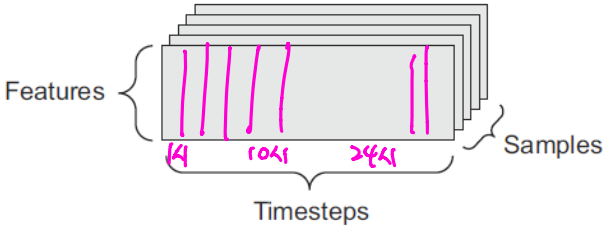


Figure 2.3 A 3D timeseries data tensor

Dataset of stock prices. Every minute, we store the *current price*, *the highest* and *the lowest price* in the past minute. Encoded as a 2D tensor of shape (390, 3) for 390 minutes in a trading day

250 days' data stored in 3D tensor of shape (250, 390, 3).

Image data

Images – 4D tensors of shape (samples, height, width, channels) **or** (samples, channels, height, width)

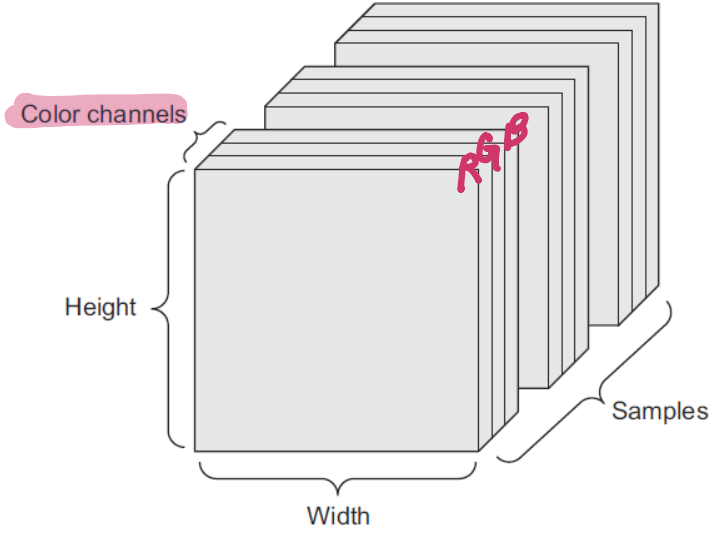


Figure 2.4 A 4D image data tensor (channels-first convention)

Video data

Video – 5D tensors of shape (samples, frames, height, width, channels) **or** (samples, frames, channels, height, width)

Example:

A 60-second, 144 × 256 YouTube video clip sampled at 4 frames per second would have 240 frames. A batch of four such video clips would be stored in a tensor of shape (4, 240, 144, 256, 3)

The gears of neural networks: tensor operations

Tensor operations **applied to tensors of numeric data.**

Example:

keras.layers.Dense(512, activation='relu')

Unpack this

$\sigma(W_1 x + b_1)$ (512×900, 900×1, 512×1)

output = relu(dot(W, input) + b)

Three tensor operations:

- ***dot product*** **(dot) between input tensor and a tensor W**
- ***addition*** **(+) between the resulting 2D tensor and vector b**
- ***relu operation***: **relu(x) is max(x, 0).**

Element-wise operations

Ex) relu and addition

(1, 2, 3) + (0, 0, 1) = (1, 2, 4)

Naïve implementation of an element-wise relu operation:

```
def naive_relu(x):
    assert len(x.shape) == 2, 'x.ndim() must 2'
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)

    return x
```

print

shape 2이상인

(1 2 3
4 5)

relu(x) = max[x, 0]

⇒ X는 0 보다 큰값과 0인 값으로 구성됨

chapter02_mathematical-building-blocks.i

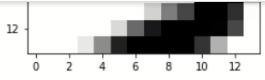
File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents

+ Code + Text Copy to Drive

RAM Disk Editing



x1 = np.array([[1,-2,3],[2,3,-1]])
x1.ndim
x1.shape
naive_relu(x1)
↳ array([[1,0,3],[2,3,0]])

The notion of data batches

```
[ ] batch = train_images[:128]
```

```
[ ] batch = train_images[128:256]
```

```
[ ] n = 3
    batch = train_images[128 * n:128 * (n + 1)]
```

Real-world examples of data tensors

Vector data

Timeseries data or sequence data

Image data

~~y = x~~

y = x.copy()

얕은 복사!!
original 이 바뀔 수 있다!!

0s completed at 1:36 PM

Element-wise operations

Ex) relu and addition

Naïve implementation of addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape

    x = x.copy()   ## avoid overwriting
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

Element-wise operations

Element-wise relu:

```
import numpy as np
z = np.maximum(z, 0.)
```

naive coding 보다 시간↓

Element-wise add:

```
z = x + y
```

Broadcasting

What happens with addition when the shapes of the two tensors being added differ?

X . Y ⇒ dim 작은 Tensor repeated

dimension 다름

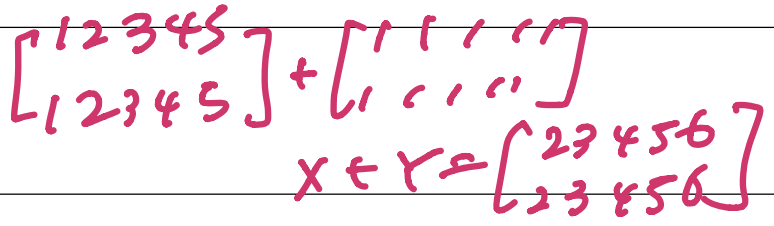
Broadcasting consists of two steps:

1. **Axes (called broadcast axes) are added to the smaller tensor to match the ndim of the larger tensor.**
2. **The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.**

Broadcasting

Example:

```
import numpy as np
x = np.array([1,2,3,4,5])
y = np.array([[1,1,1,1,1], [1,1,1,1,1]])
```



```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]

    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a 2D Numpy tensor.

y is a Numpy vector.

Avoid overwriting the input tensor.

Broadcasting

if one tensor has shape *(a, b, … n, n + 1, … m)* and the other has shape *(n, n + 1, … m)*. The broadcasting will then automatically happen for axes *a* through *n - 1*.

```
import numpy as np

x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))

z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).

y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

chapter02_mathematical-building-blocks.i

File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents

+ Code + Text Copy to Drive

RAM Disk

Editing

[73] x1

```
array([[1, 0, 3],
       [2, 3, 0]])
```

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {0:.2f} s".format(time.time() - t0))
```

```
t0 = time.time()
for _ in range(1000):
```

0s completed at 2:16 PM

Tensor dot

Dot product in math

$(a, b, c) \cdot (d, e, f)$

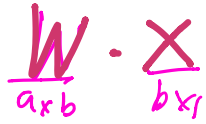
$= adt + be + cf$

```
import numpy as np
z = np.dot(x, y)
```

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are NumPy vectors.

Matrix vector dot




```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a NumPy matrix.

y is a NumPy vector.

The first dimension of x must be the same as the 0th dimension of y!

This operation returns a vector of 0s with the same shape as y.

Matrix vector dot

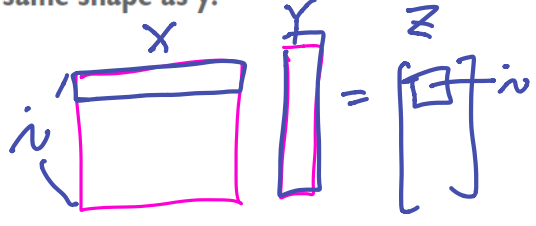
```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a NumPy matrix.

y is a NumPy vector.

The first dimension of x must be the same as the 0th dimension of y!

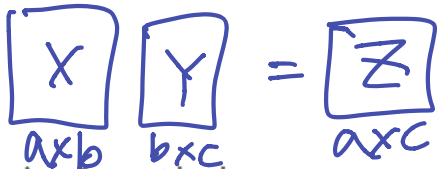
This operation returns a vector of 0s with the same shape as y.



Could also use the previous codes

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Dot product for 2D tensors




```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

x and y are NumPy matrices.

The first dimension of x must be the same as the 0th dimension of y!

This operation returns a matrix of 0s with a specific shape.

Iterates over the rows of x . . .

. . . and over the columns of y.

The gears of neural networks: Tensor operations

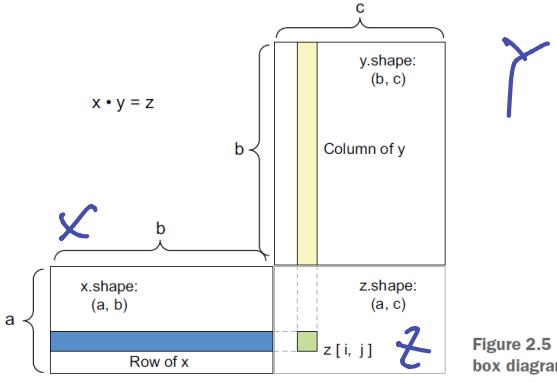


Figure 2.5 Matrix dot-product box diagram

Tensor dot

More generally, you can take the dot product between higher-dimensional tensors, following the same rules for shape compatibility as outlined earlier for the 2D case:

(a, b, c, d) . (d,) -> (a, b, c)
(a, b, c, d) . (d, e) -> (a, b, c, e)

Tensor reshaping

we preprocessed the digits data before feeding it into our network:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Try it:

```
x = np.array([[0., 1.], [2., 3.], [4., 5.]])

x = x.reshape((6, 1))

x = x.reshape((2, 3))
```

chapter02_mathematical-building-blocks.i

File Edit View Insert Runtime Tools Help Cannot save changes

Share

Table of contents

+ Code + Text Copy to Drive

RAM Disk Editing

```
 1.87539025, 1.41756671, 1.11338379, 0.96418242, 1.21331544],
[1.31749392, 0.94878296, 1.91253185, 0.38539614, 0.40926378,
 1.57273443, 1.31628261, 1.66346262, 1.51518779, 0.81068861]])
```

```
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

Tensor product

```
[ ] x = np.random.random((32,))
    y = np.random.random((32,))
    z = np.dot(x, y)
```

```
[ ] def naive_vector_dot(x, y):
        assert len(x.shape) == 1
        assert len(y.shape) == 1
        assert x.shape[0] == y.shape[0]
        z = 0.
        for i in range(x.shape[0]):
            z += x[i] * y[i]
        return z
```

```
[ ] def naive_matrix_vector_dot(x, y):
        assert len(x.shape) == 2
        assert len(y.shape) == 1
        assert x.shape[1] == y.shape[0]
```

0s completed at 2:19 PM

Geometric interpretation of tensor operations

Vector as an arrow

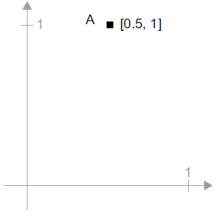


Figure 2.6 A point in a 2D space

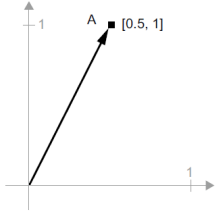


Figure 2.7 A point in a 2D space pictured as an arrow

Sum of two vectors

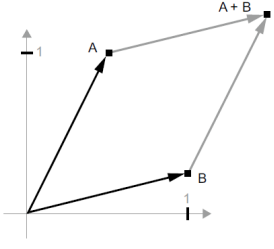


Figure 2.8 Geometric interpretation of the sum of two vectors

Translation as vector addition

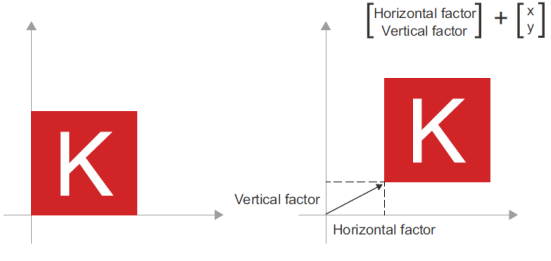


Figure 2.9 2D translation as a vector addition

2D rotation

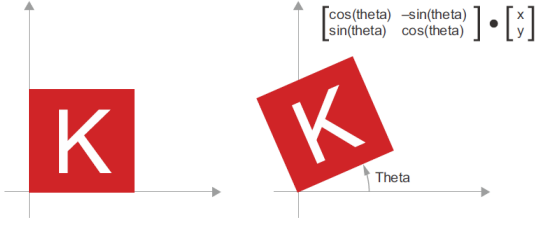


Figure 2.10 2D rotation (counterclockwise) as a dot product

Geometric interpretation of tensor operations

2D scaling

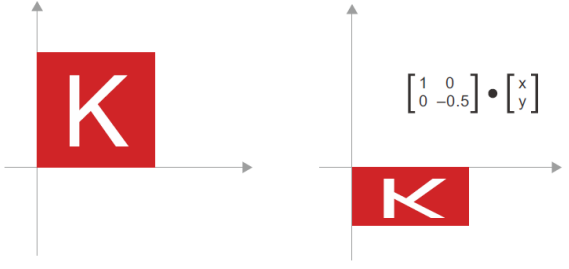


Figure 2.11 2D scaling as a dot product

Affine transform

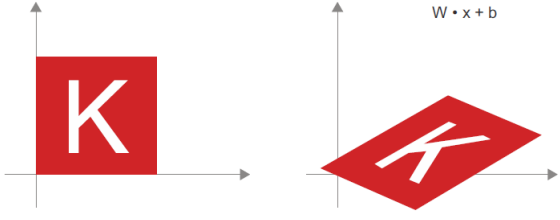


Figure 2.12 Affine transform in the plane

Affine transform followed by relu

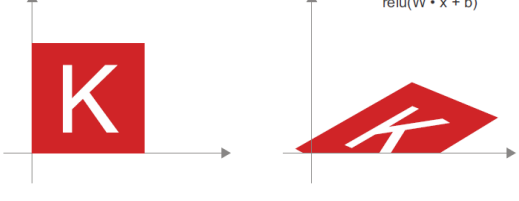


Figure 2.13 Affine transform followed by `relu` activation

Geometric interpretation of deep learning

Neural networks consist entirely of chains of tensor operations

These tensor operations are just simple geometric transformations of the input data

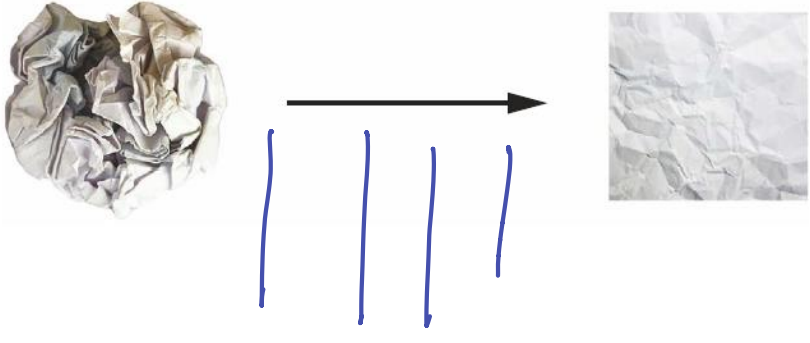


Figure 2.14 Uncrumpling a complicated manifold of data

Thank you! ☺