ML Day24 (Sobel Filtering)

▼ **1-Dimensional Window Extraction**

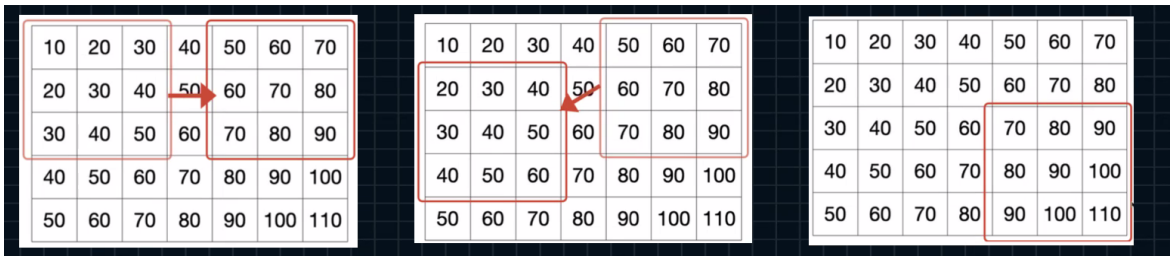
- window란?
  - 10개의 원소를 가진 1차원 벡터가 주어졌을 때, 3칸의 window가 차례대로 지나가면서 데이터를 3개씩 뽑는 과정

  - window의 갯수(L_ : Prime L) ⇒ L : 데이터의 갯수, W: window 1개의 칸의 갯수 **(중요)**

```
import numpy as np

data = 10*np.arange(1, 11)
L = len(data)
W = 3
print(data, '\n')

L_ = L - W + 1
for idx in range(L_):
    print(data[idx : idx + W])  # [0 : 0 + 3] 부터 [7 : 7 + 3]까지 slicing
```

```
[ 10  20  30  40  50  60  70  80  90 100]

[10 20 30]
[20 30 40]
[30 40 50]
[40 50 60]
[50 60 70]
[60 70 80]
[70 80 90]
[ 80  90 100]
```

▼ **2-Dimensional Window Extraction**




```
import numpy as np

data1 = 10*np.arange(1, 8).reshape(1, -1)
data2 = 10*np.arange(5).reshape(-1, 1)
data = data1 + data2
print(data, '\n')

H, W = data.shape         # data.shape = (5, 7)
F = 3                     # 임시적으로 window size를 의미
H_ = H - F + 1
W_ = W - F + 1

for h_idx in range(H_):
    for w_idx in range(W_):
        print(data[h_idx : h_idx + F,
              w_idx : w_idx + F])
    print()
```

```
[[ 10  20  30  40  50  60  70]
 [ 20  30  40  50  60  70  80]
 [ 30  40  50  60  70  80  90]
 [ 40  50  60  70  80  90 100]
 [ 50  60  70  80  90 100 110]]

[[10 20 30]
 [20 30 40]
 [30 40 50]]
[[20 30 40]
 [30 40 50]
 [40 50 60]]
[[30 40 50]
 [40 50 60]
 [50 60 70]]
[[40 50 60]
 [50 60 70]
 [60 70 80]]
[[50 60 70]
 [60 70 80]
 [70 80 90]]
```

▼ data1 / shape(1, 7)

	0	1	2	3	4	5	6
0	10	20	30	40	50	60	70

▼ data2 / shape(5, 1)

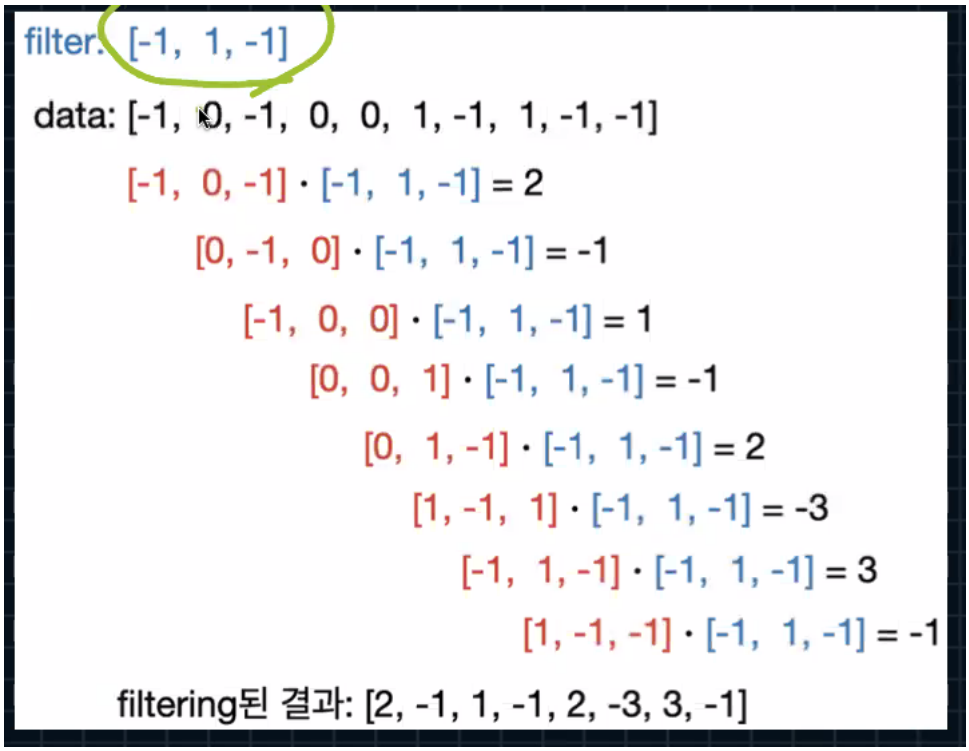
	0
0	0
1	10
2	20
3	30
4	40

▼ data = data1 + data2 / shape(5, 7)

	0	1	2	3	4	5	6
0	10	20	30	40	50	60	70
1	20	30	40	50	60	70	80
2	30	40	50	60	70	80	90
3	40	50	60	70	80	90	100
4	50	60	70	80	90	100	110

▼ **1-D Correlation**

- window를 쭉 뽑으면서, 내가 원하는 패턴이 있는 부분을 추출하는 연산

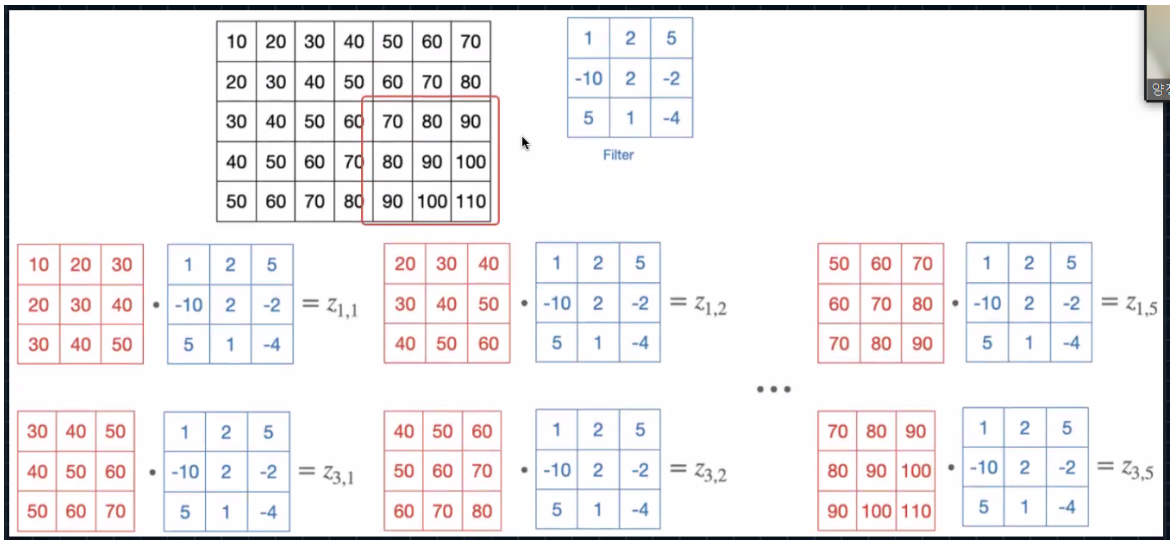


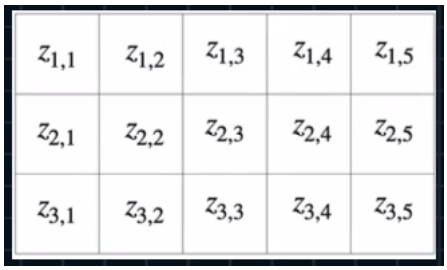
- data에서 window를 뽑고, filter와 내적(dot product : 원소끼리 곱한 후 더함)한 결과를 저장
- window와 filter가 같을 때 가장 큰 값(내적의 값이 가장 큰)을 출력
- window와 filter가 반대 vector일 때 가장 작은 값 출력

```
import numpy as np

np.random.seed(0)

data = np.random.randint(-1, 2, (10, ))
filter_ = np.array([-1, 1, -1])
print(f"{data = }")
print(f"{filter_ = }")

L = len(data)
F = len(filter_)

L_ = L - F + 1                              # L_ = 총 window의 갯수
filtered = []
for idx in range(L_):                       # 총 window의 갯수만큼 for문으로
    window = data[idx : idx + F]            # window는 data[idx : idx + F]로 계속 바뀌게
    filtered.append(np.dot(window, filter_))  # np.dot (vector내적 및 행렬 곱 함수)
    # Notice
    # window는 for문을 지나며 항상 바뀌는 부분
    # filter_는 항상 [-1, 1, -1]로 일정
filtered = np.array(filtered)
print("filtering result:", filtered)
```

```
data = array([-1,  0, -1,  0,  0,  1, -1,  1, -1, -1])
filter_ = array([-1,  1, -1])
filtering result: [ 2 -1  1 -1  2 -3  3 -1]
```

▼ **2-D Correlation**






```
import numpy as np

data1 = 10*np.arange(1, 8).reshape(1, -1)
data2 = 10*np.arange(5).reshape(-1, 1)

data = data1 + data2
filter_ = np.array([[1, 2, 5],
                    [-10, 2, -2],
                    [5, 1, -4]])
print(data, '\n')

H, W = data.shape      # height(5), width(7)
F = filter_.shape[0]   # 임시적으로 window size를 의미
H_ = H - F + 1         # 3
W_ = W - F + 1         # 5

filtered_data = np.zeros(shape=(H_, W_))   # filtered_data : shape=(H_, W_)인 형태로 모든 원소가 0인 행렬
for h_idx in range(H_):
    for w_idx in range(W_):
        window = data[h_idx : h_idx + F,
                      w_idx : w_idx + F]
        z = np.sum(window * filter_)

        filtered_data[h_idx, w_idx] = z

print(filtered_data)
```

```
[[ 10  20  30  40  50  60  70]
 [ 20  30  40  50  60  70  80]
 [ 30  40  50  60  70  80  90]
 [ 40  50  60  70  80  90 100]
 [ 50  60  70  80  90 100 110]]

[[-30. -30. -30. -30. -30.]
 [-30. -30. -30. -30. -30.]
 [-30. -30. -30. -30. -30.]]
```

▼ 1-D Correlation Exercise

Hint!!

1. 최종 행렬의 곱셈 먼저 살펴보기
2. window를 쌓은 행렬만들기 → 행렬의 곱셈을 통해서, 모든 window의 연산을 한 번에 구할 수 있다는 아이디어
3. Indexing Array 만들기
4. Index array가 만들어지는 원리

- row의 개수 ⇒ window의 개수
  - $L' = L - F + 1$
- column의 개수 ⇒ window의 원소의 개수, filter의 원소의 개수
  - F

```
# 한솔이 code (np.tile을 이용)
import numpy as np
np.random.seed(0)
data = np.random.randint(-1, 2, (10, ))
filter_ = np.array([-1, 1, -1]).reshape(-1,1)
L,F = len(data) , len(filter_)

# np.tile을 이용해 data를 L+1만큼 반복하여 늘어뜨린 후, (-1, L+1)형태로 reshape하고 [:L-F+1,:F] slicing.
data = np.tile(data, reps=L+1).reshape(-1,L+1)[:L-F+1,:F]
print(np.matmul(data,filter_))      # np.matmul함수를 통해 행렬들의 내적을 구함
```

```
# 선생님 code
import numpy as np
data = np.random.randint(-1, 2, (10,))
filter_ = np.array([-1, 1, -1])
L, F = len(data), len(filter_)             # 각각 L과 F에 len(data), len(filter_)를 대입
L_ = L - F + 1                             # prime L(L_)은 L - F + 1
filter_idx = np.arange(F).reshape(1, -1)   # np.arange(F) -> 0, 1, 2(index값)로 이루어진 (1, 2)행렬
window_idx = np.arange(L_).reshape(-1, 1)  # np.arange(L_) -> 0 ~ 7(index)로 이루어진 (8, 1)행렬
idx_arr = filter_idx + window_idx          # filter_idx + window_idx => (8, 3)형태인 idx_arr행렬 생성
print(filter_idx, window_idx, idx_arr)
window_mat = data[idx_arr]                 # index값을 가진 (8, 3) index_arr를 data에 집어넣어 각index값에 해당하는 값들이 위치하도록 window_m
print(window_mat.shape, filter_.shape)     # (8, 3), (3,)

# 행렬의 곱셈을 이용하기 위해서 filter_를 (3, 1)로 바꿔줄 필요가 있음
correlations = np.matmul(window_mat, filter_.reshape(-1, 1))

# 만약 결과를 vector처럼 다뤄야 하면
correlations = correlations.flatten()
print(correlations)
```

▼ filter_idx

▼ window_idx

0
1
2
3
4
5
6
7

▼ idx_arr = filter_idx + window_idx

0	1	2
1	2	3
2	3	4
3	4	5
4	5	6
5	6	7
6	7	8
7	8	9

▼ window_mat = data[idx_arr]

-1	-1	0
-1	0	-1
0	-1	0
-1	0	1
0	1	-1
1	-1	-1
-1	-1	1
-1	1	-1