# 기본 신경망

```python
import torch
import torch.nn as nn
import torch.optim as optim

# 1. 데이터 준비
x = torch.tensor([[1.0], [2.0], [3.0], [4.0]])   # 입력
y = torch.tensor([[2.0], [4.0], [6.0], [8.0]])   # 정답 (y = 2x)

model = nn.Linear(1, 1)  # 2. 모델 정의 (Linear: y = wx + b)

# 3. 손실 함수와 옵티마이저 정의
criterion = nn.MSELoss()                              # 평균제곱오차
optimizer = optim.SGD(model.parameters(), lr=0.01)  # 확률적 경사하강법

# 4. 학습 루프
for epoch in range(10):

    y_pred = model(x)    # (1) 순전파 - 예측
    loss = criterion(y_pred, y)   # (2) 손실 계산
    optimizer.zero_grad()   # (3) 기울기 초기화 (이전 값 누적 방지)
    loss.backward() # (4) 역전파 - 기울기 계산
    optimizer.step() # (5) 옵티마이저로 가중치 업데이트
    # 로그 출력
    print(f"Epoch {epoch+1}: loss={loss.item():.4f}")

# 5. 최종 결과 확인
print("₩n학습 후 가중치, 절편:")
print("w =", model.weight.item())
print("b =", model.bias.item())

```

## 표준 딥러닝 학습 단계 (PyTorch 기준)

1. **데이터 준비** (Data Preprocessing & Loading) :
   import torchvision.transforms.v2 as transforms
- **입력 이미지 준비** → Tensor **변환**(C, H, W): 원본 이미지를 읽고 전처리 적용

2. **데이터 로딩** (Batching): (C,H,W) → (Batch, C,H,W)**로 변환**
   import torch.utils.data as data

3. **모델 및 학습 환경 설정**
- **모델 준비**: 신경망 아키텍처(예: SimpleFCN, CNN)를 정의하고 초기화
- **Loss (손실함수)**:
- **Optim 준비**:

4. **순전파 및 손실 계산** (Forward Pass)
5. **역전파 및 가중치 업데이트** (Backward Pass & Optimization)
6. **반복 및 조정**

```
10
11 # 3. 손실 함수와 옵티마이저 정의
12 criterion = nn.MSELoss()
13 optimizer = optim.SGD(model.parameters(), lr=0.01)
14
15 # 4. 학습 루프
```

Learning Rate(LR)은 "**너무 빨리 맞추지 않게, 천천히 제대로 맞추게 하는 안전장치**".

- **lr없음(=1로고정)** : 너무큰폭으로 이동, 손실이 폭발하거나 NaNB발행
- **lr너무큼**: 비슷하게 발산, 학습실패
- **lr적절함**: 천천히 최솟점이동,학습성공

### Learning Rate Scheduling (학습률 스케줄링)

**lr(learning rate, 학습률)** 은 훈련 내내

**고정값으로 두기보다는**
**"점진적으로 조정"함**

- 왜 계속 학습률을 변경해야하는가?

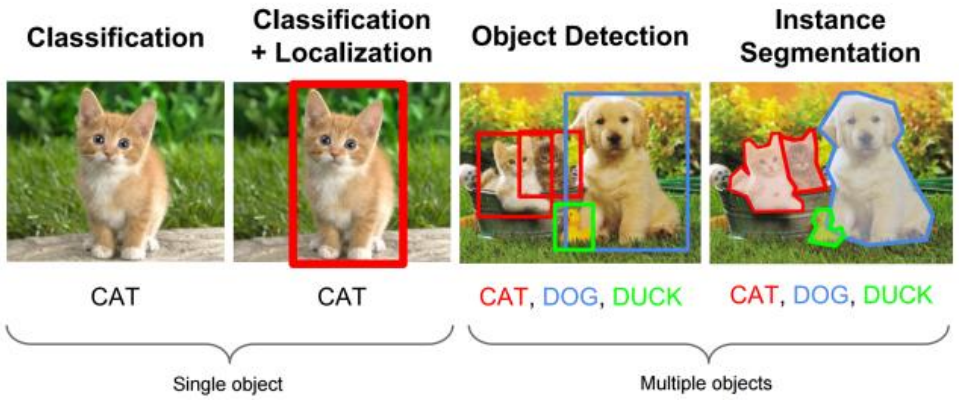
| 시점 | 현상및 조치 |
|---|---|
| **초기 학습 초반** | 모델이 랜덤 상태<br>→ 방향만 대충 잡는 중<br>**[필요조치]**<br>**lr 크게** → 빠르게 수렴 방향 탐색 |
| **중반 이후** | 이미 손실이 꽤 줄어듦<br>**[필요조치]**<br>**lr 작게** → 세밀하게 조정 |
| **후반** | 최소점 근처 → 큰 lr이면 튀거나 진동<br>**[필요조치]**<br>**lr 더 작게** → 미세한 조정, 안정화 |

| 구분 | 기본 아이디어 | 주요 하이퍼파라미터 | 장점 | 단점 | 주 사용처 |
|---|---|---|---|---|---|
| **SGD (Stochastic Gradient Descent)**<br>단순히 현재 기울기 방향으로 이동 | w ← w - lr * grad | lr | 구현 간단, 메모리 적음 | 수렴 느림, 지역 최소에 갇힐 수 있음 | 기본 베이스라인, 단순 모델 |
| **Momentum**<br>이전 이동 방향(속도)을 고려 | v = βv + lr * gradw -= v | lr, β(보통 0.9) | 진동 줄이고 수렴 빠름 | 최적점 근처에서 overshoot 가능 | CNN, RNN |
| **Nesterov Accelerated Gradient (NAG)**<br>이동 예측 후 보정 | v = βv + lr * grad(w - βv) | lr, β | 더 빠른 수렴, 안정적 | 구현 복잡 | CNN, RNN |
| **AdaGrad**<br>파라미터별 적응형 학습률 | w ← w - lr/√(G+ε) * grad | lr, ε | 희소데이터에 강함 | 학습률이 너무 빨리 감소 | NLP, 추천 시스템 |
| **RMSProp**<br>AdaGrad의 감쇠 적용 | $E[g^2] = \beta E[g^2]+(1-\beta)g^2$w -= lr/√(E[$g^2$]+ε)*g | lr, β, ε | 안정적, RNN에 강함 | β 조정 필요 | 순환신경망, 시계열 |
| **Adam (Adaptive Moment Estimation)**<br>Momentum + RMSProp 결합 | m = $\beta_1$m + (1-$\beta_1$)gv = $\beta_2$v + (1-$\beta_2$)$g^2$w -= lr * $\hat{m}$ / (√$\hat{v}$ + ε) | lr, $\beta_1$, $\beta_2$, ε | 대부분 잘 작동, 빠른 수렴 | 오버피팅 가능, 일반화 약함 | 대부분의 딥러닝 |
| **AdamW**<br>Adam + 정규화 개선 (Weight Decay 분리) | Adam과 동일 + w -= lr*weight_decay*w | lr, $\beta_1$, $\beta_2$, weight_decay | Adam의 과적합 개선 | 약간 느릴 수 있음 | Vision, NLP (BERT, ViT) |
| **AdaDelta**<br>AdaGrad의 누적 문제 개선 | 변화량으로 비율 조정 | ρ, ε | 학습률 튜닝 거의 불필요 | 복잡도 높음 | CNN, RNN |
| **Nadam (Nesterov + Adam)**<br>Adam + Nesterov 모멘텀 | $\hat{m}$ 계산 시 look-ahead 사용 | lr, $\beta_1$, $\beta_2$ | 빠른 초기 수렴 | 약간 불안정할 수 있음 | RNN, LSTM |
| **Lion (2023, Google)**<br>부호(sign) 기반 업데이트 | w -= lr * sign(m) | lr, $\beta_1$, $\beta_2$ | 경량, 빠른 수렴 | 안정성 검증 중 | 대형 언어모델 (PaLM, Gemini) |
| **LAMB**<br>Layer별 Adaptive scaling | r = $\hat{m}$ / (√$\hat{v}$ + ε)w -= lr * η * r | lr, $\beta_1$, $\beta_2$ | 대형 배치 학습 적합 | 구현 복잡 | BERT 대규모 훈련 |
| **Adafactor**<br>메모리 효율 Adam | 2차 모멘트 근사 | lr, β | 메모리 절약, Transformer 친화적 | 구현 난이도 | T5, Transformer |
| **RAdam (Rectified Adam)**<br>Adam의 초반 불안정성 보정 | rectified term 추가 | lr, $\beta_1$, $\beta_2$ | 초기 안정성↑ | 조금 느림 | Transformer, CNN |

# CNN- FC층

CNN 모델 (Convolutional Neural Network, 합성곱 신경망)은 주로 이미지, 비디오와 같은 **시각 데이터**를 처리하고 분석하기 위해 개발된 딥러닝 알고리즘입니다.

인간의 시신경 구조를 모방하여 설계되었으며, 특히 **이미지의 공간 정보(픽셀 간의 위치 관계)**를 유지하면서 학습하는 것이 특징입니다. 이는 **기존의 완전 연결 신경망(Fully Connected Network)이 이미지를 1차원 데이터로 변환할 때 정보가 손실되는 문제**를 해결합니다.




부성순강사(bakpak@empas.com)

## 용어이해

**완전 연결층 (Fully Connected Layer)**, 또는 밀집층 (Dense Layer)은 인공 신경망(ANN)의 가장 기본적인 구성 요소 중 하나입니다.

CNN (합성곱 신경망)과 같은 복잡한 구조에서 완전 연결층은 보통 **모델의 마지막 부분**에 위치하여, 이전 계층에서 추출된 모든 특징(Feature)을 기반으로 입력 데이터를 최종적인 카테고리(클래스)로 **분류**하거나 **회귀** 값을 출력하는 역할을 담당합니다. (wx+b를 수행함)

## 파이썬 코드

| 프레임워크 | 명칭 | 의미 |
| --- | --- | --- |
| **PyTorch** | `nn.Linear` | 선형 변환을 수행하는 완전 연결층 |
| **TensorFlow/Keras** | `Dense` | 밀집된 연결을 가진 완전 연결층 |

```
self.fc = nn.Linear(32 * 8 * 8, 10)
```

부성순 bakpak@empas.com

# Pytorch에서의 FC층 미리 보기

- x = x.view(-1, 32 * 8 * 8): Flatten (**평탄화**) 과정

3차원 형태의 특징 맵(채널32, 높이8, 너비8)을 완전 연결 계층의 입력으로 사용하기 위해 1차원 벡터로 쭉 **펼치는** 역할을 함

- self.fc : **완전 연결 계층**

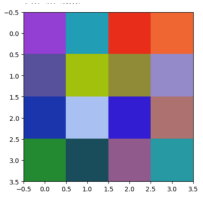
이 계층이 학습하는 wx+b는 앞서 추출된 특징들을 기반으로 입력 이미지가 최종적으로 어떤 클래스(범주)에 속하는지를 **판단하고 분류**하는 데 사용됨

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc = nn.Linear(32 * 8 * 8, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * 8 * 8)
        x = self.fc(x)
        return x

model = SimpleCNN()
```

(1) 합성곱층(특징만 찾음)

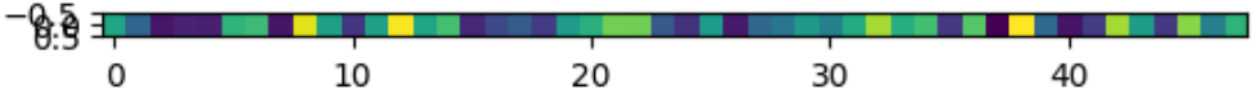
(2) 완전연결층 (w,b를 만듦)

1



2

```
torch.Size([1, 48])
<matplotlib.image.AxesImage at 0x797452e3c0e0>
```




부성순 bakpak@empas.com

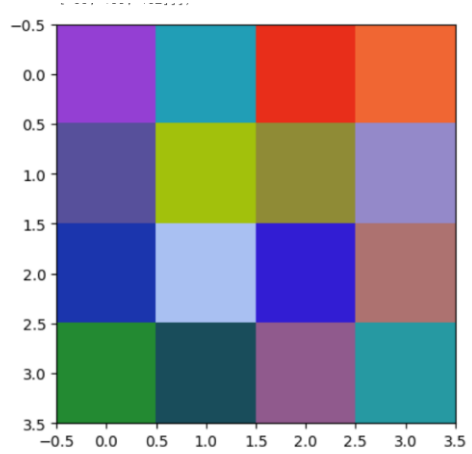
# [1-1] 이미지의 특징

Red Green blue

```
1 img=[[[148,63,211], [ 33,158,182],[232,46,26],[240,102,51]],
2       [[ 87,80,155], [162,193,12],[143,139,54],[148,137,201]],
3       [[ 27,53,173], [169,192,242], [50,29,211], [173,114,112]],
4        [[ 35,138,50], [ 25,77,91], [144,90,141],[38,153,162]]]
5 import numpy as np
6 imgArray=np.array(img)
7 print(imgArray.shape)
8 plt.imshow(imgArray)
9
10 imgArray
```

(4, 4, 3)

높이,너비,채널




### . 채널 (Channel) 🌈

**채널**은 이미지의 특정 색상 성분이나 정보 유형을 저장하는 개별적인 2차원 행렬입니다.

- **역할:** 이미지를 구성하는 **색상 요소**를 분리하여 저장합니다.
- **RGB 이미지:** 가장 일반적인 컬러 이미지(예: JPEG)는 빨간색(Red), 녹색(Green), 파란색(Blue)의 세 가지 채널로 이루어져 있습니다. 이 세 채널의 밝기 값을 조합하여 최종 픽셀의 색상이 결정됩니다.
- **흑백 이미지: 밝기(Grayscale)** 정보만을 담는 하나의 채널로 구성됩니다.
- **PyTorch 순서:** 딥러닝 프레임워크인 PyTorch는 연산 효율성을 위해 데이터를 **(채널, 높이, 너비)** 순서로 처리하는 것을 기본으로 합니다.

### . 숫자 255의 의미 💡

숫자 255는 일반적으로 이미지 픽셀이 가질 수 있는 **최대 밝기**를 나타냅니다.

• **8비트 표현:**

대부분의 디지털 이미지는 각 채널의 픽셀 값을 **8비트(Bit) 부호 없는 정수**로 저장합니다.
8비트는 2^8 = 256가지의 상태를 표현할 수 있습니다.

• **값의 범위:**

따라서 픽셀 값의 범위는 **0(가장 어두움/색상 없음)**부터 **255(가장 밝음/최대 강도)**까지입니다.

디지털 화면(모니터, 스마트폰, TV 등)은 **RGB (Red, Green, Blue)** 색상 모델을 사용하며, 이는 빛을 섞어 색을 만드는 방식(가산 혼합)입니다.

- 이 모델에서 **각각의 숫자**는 세 가지 기본색(빨강, 초록, 파랑) **빛의 강도**를 의미
- (0,0,0)은 색이 없음
- (255,255,255)는 세 가지 기본색의 빛이 **가장 강하게 모두 섞인 상태**이며, 이는 **흰색**으로 인식

• **정규화:**

딥러닝 모델에 이미지를 입력할 때는 이 0~ 255 사이의 정수형 데이터를0.0~부터 1사이로 변경하는 실수(float)로 변환하는 **정규화** 과정을 거칩니다. 이는 학습의 안정성과 효율성을 높이기 위해 필수적입니다.
(다양한 정규화 기법이 있음)

부성순 bakpak@empas.com

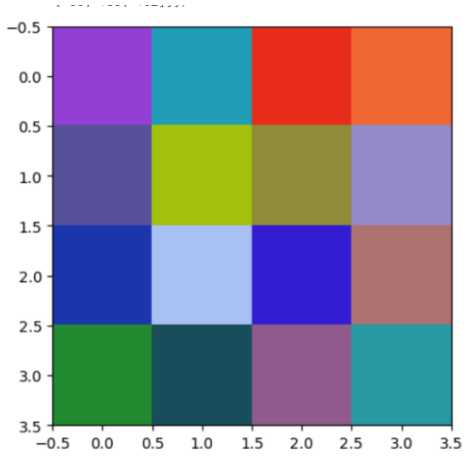
# [1-2] pytorch용 입력용으로 변환: 높이(H),너비(W),채널(C)-> Torch용 채널(C),높이(H),너비(W)로 변환

Red Green blue

```
1 img=[[[148,63,211], [ 33,158,182],[232,46,26],[240,102,51]],
2       [[ 87,80,155], [162,193,12],[143,139,54],[148,137,201]],
3       [[ 27,53,173], [169,192,242], [50,29,211], [173,114,112]],
4        [[ 35,138,50], [ 25,77,91], [144,90,141],[38,153,162]]]
5 import numpy as np
6 imgArray=np.array(img)
7 print(imgArray.shape)
8 plt.imshow(imgArray)
9
10 imgArray
```

(4, 4, 3)

높이,너비,채널




Tasnforms에서 C,H,W로 변환

```
1 import torch
2 import torchvision.transforms.v2 as transforms
3
4 transform = transforms.Compose([
5     transforms.ToImage()
6     # transforms.ToDtype(torch.float32, scale=True)
7 ])
8
9 input_tensor_chw = transform(imgArray)
10 print(input_tensor_chw.shape)
11 print(input_tensor_chw)
```

채널모습
이해의 편의성을
위해 스케일 안함

```
torch.Size([3, 4, 4])
Image([[[148,  33, 232, 240],
        [ 87, 162, 143, 148],
        [ 27, 169,  50, 173],
        [ 35,  25, 144,  38]],

       [[ 63, 158,  46, 102],
        [ 80, 193, 139, 137],
        [ 53, 192,  29, 114],
        [138,  77,  90, 153]],

       [[211, 182,  26,  51],
        [155,  12,  54, 201],
        [173, 242, 211, 112],
        [ 50,  91, 141, 162]]], )
```

채널,높이,너비

Red

Green

Blue

부성순 bakpak@empas.com

## 참고) PyTorch의 C,H,W 순서 사용이유

### 1. 연산 효율성: 합성곱 연산 (Convolution)에 최적화

딥러닝, 특히 CNN(합성곱 신경망)에서 가장 많이 수행되는 연산은 합성곱(Convolution)입니다. 이 연산은 필터가 각 채널을 독립적으로 슬라이딩하며 진행됩니다.

- **GPU 및 메모리 접근:** (C, H, W) 순서로 데이터를 저장하면, 텐서의 메모리 배치가 **채널별**로 연속적으로 저장될 가능성이 높습니다.
- **성능 이점:** GPU가 합성곱 연산을 수행할 때, 여러 필터가 한 번에 다음 채널의 데이터를 읽어와야 합니다. 채널이 앞에 위치하면, 메모리상에서 해당 채널의 데이터가 가깝게 배치되어 데이터 접근 속도(Cache Locality)가 빨라져 연산 효율성이 극대화됩니다.

### 2. 라이브러리 관습: Caffe 및 Torch의 영향

PyTorch의 선조인 Torch (Lua)와 초기 딥러닝 프레임워크인 **Caffe**는 텐서를 (C, H, W) 순서로 처리하는 것을 표준으로 삼았습니다.

- **역사적 계승:** PyTorch는 이러한 기존 라이브러리의 방식을 계승하여 텐서 차원 순서를 `(B, C, H, W)` (Batch, Channel, Height, Width)로 설정했으며, 이는 **NVIDIA의 CUDA 라이브러리**와도 연산 구조가 잘 맞습니다.

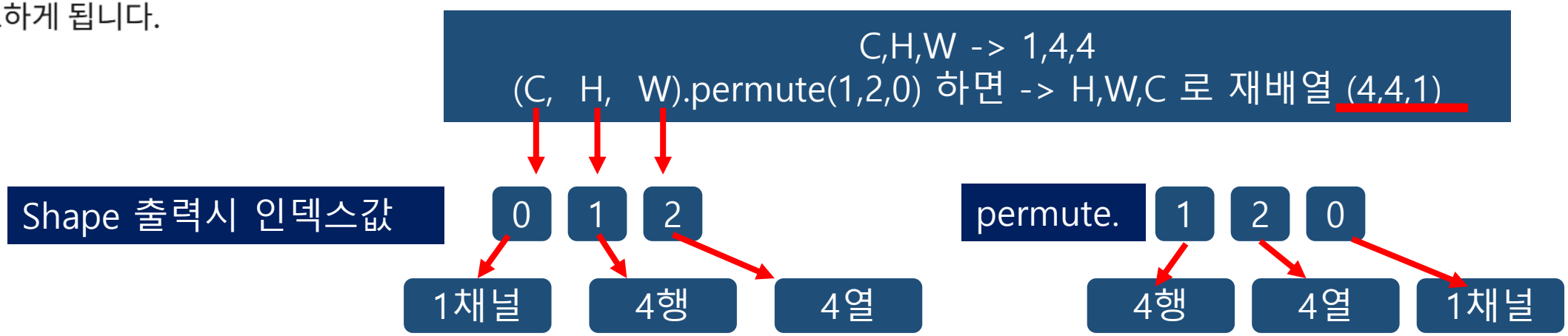
부성순 bakpak@empas.com

## 참고) PyTorch의 C,H,W 순서 사용이유

### 🆚 Matplotlib 및 NumPy의 (H, W, C) 순서와의 차이

반면에 Matplotlib, OpenCV, NumPy 등 일반적인 이미지 처리 라이브러리들은 이미지를 **(높이, 너비, 채널)** $(\mathbf{H} \times \mathbf{W} \times \mathbf{C})$ 순서로 처리합니다.

- **인간 친화적:** 이 순서는 인간이 이미지를 볼 때 높이와 너비를 주요 차원으로 인식하는 방식에 더 직관적입니다.
- **변환의 필요성:** 이 때문에 PyTorch 모델에 이미지를 입력하거나 모델의 출력을 시각화할 때, `permute(2, 0, 1)`나 `permute(1, 2, 0)`와 같은 **차원 변환(Transpose)** 작업이 필연적으로 필요하게 됩니다.



# [1-3] 모델 학습에 input할 Batch,C,H,W로 변환

```python
import torch
import torchvision.transforms.v2 as transforms

transform = transforms.Compose([
    transforms.ToImage()
    # transforms.ToDtype(torch.float32, scale=True)
])

input_tensor_chw = transform(imgArray)
print(input_tensor_chw.shape)
print(input_tensor_chw)
```

```
torch.Size([3, 4, 4])
Image([[[148,  33, 232, 240],
        [ 87, 162, 143, 148],
        [ 27, 169,  50, 173],
        [ 35,  25, 144,  38]],

       [[ 63, 158,  46, 102],
        [ 80, 193, 139, 137],
        [ 53, 192,  29, 114],
        [138,  77,  90, 153]],

       [[211, 182,  26,  51],
        [155,  12,  54, 201],
        [173, 242, 211, 112],
        [ 50,  91, 141, 162]]], )
```

```python
input_tensor_bchw = input_tensor_chw.unsqueeze(0)
print(input_tensor_bchw.shape)
input_tensor_bchw
```

```
torch.Size([1, 3, 4, 4])
tensor([[[[148,  33, 232, 240],
          [ 87, 162, 143, 148],
          [ 27, 169,  50, 173],
          [ 35,  25, 144,  38]],

         [[ 63, 158,  46, 102],
          [ 80, 193, 139, 137],
          [ 53, 192,  29, 114],
          [138,  77,  90, 153]],

         [[211, 182,  26,  51],
          [155,  12,  54, 201],
          [173, 242, 211, 112],
          [ 50,  91, 141, 162]]]])
```

```python
import torch.utils.data as data

print(input_tensor_chw.shape)
y_label=2 #라벨 2라고 가정하고
dataset = [(input_tensor_chw, torch.tensor(y_label))]
data_loader = data.DataLoader(dataset, batch_size=1, shuffle=False)
for images, labels in data_loader:
  print(images.shape)
```

```
torch.Size([3, 4, 4])
torch.Size([1, 3, 4, 4])
```

데이터 로더기가 B,C,H,W로 변환

부성순 bakpak@empas.com

# [1-4] 모델 학습 전체프로세스

## 1. 데이터준비

```
import numpy as np
img=[[[148,63,211], [ 33,158,182],[232,46,2
     [[ 87,80,155], [162,193,12],[143,139,5
     [[ 27,53,173], [169,192,242], [50,29,2
      [[ 35,138,50], [ 25,77,91], [144,90,1
import numpy as np
imgArray=np.array(img)
print(imgArray.shape)
plt.imshow(imgArray)

```

## 2. 텐서변환 C,H,W

```

transform = transforms.Compose([
    transforms.ToImage(),
    transforms.ToDtype(torch.float32, scale=True)
])

input_tensor_chw = transform(imgArray)
```

## 3.배치단위 변환 B,C,H,W

```
################################
# 데이터 로더기로 변환함
################################3
import torch.utils.data as data

print(input_tensor_chw.shape)
y_label=2 #라벨 2라고 가정하고
dataset = [(input_tensor_chw, torch.ten
data_loader = data.DataLoader(dataset,
```

## 4. 모델구성

```

class SimpleFCN(nn.Module):
    def __init__(self):
        super(SimpleFCN, self).__init__()

        input_features = 4 * 4 * 3

        self.fc1 = nn.Linear(in_features=input_features, out_features=2)
        self.fc2 = nn.Linear(in_features=2, out_features=3)

    def forward(self, x):
        x = x.view(-1, 4 * 4 * 3)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

## 5. 실행

```
model = SimpleFCN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

NUM_EPOCHS=2
for epoch in range(NUM_EPOCHS):
    for i, (inputs, targets) in enumerate(data_loader):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        print(f"[Epoch {epoch+1}/{NUM_EPOCHS}, Step {i+1}] Loss: {loss.item():

print(" 학습 완료! ")
```

```
[Epoch 1/2, Step 1] Loss: 0.644446
[Epoch 2/2, Step 1] Loss: 0.640309
 학습 완료!
```

부성순 bakpak@empas.com

# CNN- Conv2

## Conv2D란?

Conv2D (2D Convolution, 2차원 합성곱)는 CNN (합성곱 신경망)의 핵심을 이루는 계층으로, 주로 이미지와 같은 **2차원 그리드 데이터**의 **특징을 추출**하는 데 사용되는 연산입니다.

이름에서 '2D'는 필터(커널)가 입력 데이터의 **높이(Height)와 너비(Width) 차원**을 따라 2차원적으로 움직이며 연산을 수행한다는 의미입니다.

## 💡 Conv2D의 주요 특징

- **가중치 공유 (Parameter Sharing)**: 하나의 필터가 이미지의 모든 위치에 동일하게 적용됩니다. 이 덕
  에 모델이 학습해야 할 파라미터(가중치) 수가 획기적으로 줄어들고, 이미지 내 물체의 위치가 바뀌어
  (위치 불변성) 특징을 인식할 수 있게 됩니다.
- **지역값 (Local Receptive Field)**: 각 뉴런은 입력 이미지의 전체가 아닌 필터 크기만큼의 **지역적인 영**
  만 바라보고 특징을 추출합니다. 이로 인해 이미지의 공간적 관계가 보존됩니다.

## 🔍 Conv2D의 작동 원리

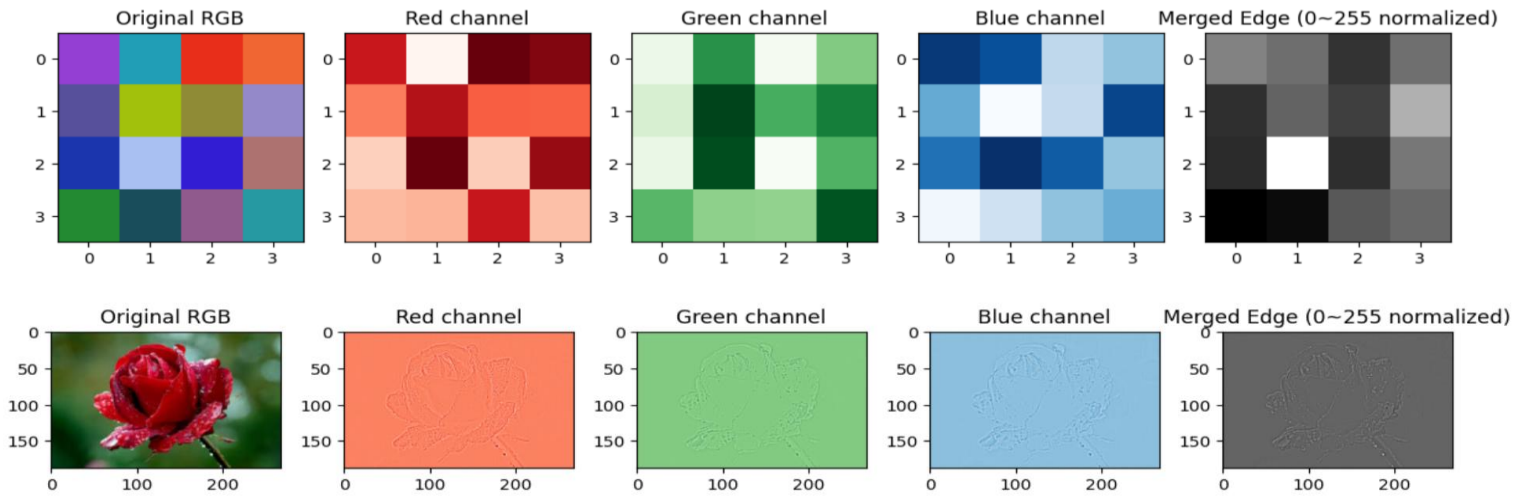
Conv2D는 다음 세 가지 핵심 요소를 사용하여 입력 이미지의 공간적 특징을 파악합니다:

1. **필터 (Filter 또는 Kernel)**:
   - 학습 가능한 작은 행렬로, 입력 데이터 위를 이동하며 **특정 패턴이나 특징** (예: 수직선, 대각선, 특정 색상 패턴)을 감지하는 역할을 합니다.
   - 필터의 깊이(Depth)는 항상 입력 데이터의 채널 수와 일치합니다.
2. **합성곱 연산 (Convolution Operation)**:
   - 필터가 이미지 위를 일정 간격(Stride)으로 이동하면서, 필터의 원소와 이미지의 해당 영역 픽셀 값을 **곱한 후 모두 더하는** 연산입니다.
   - 이 연산의 결과는 해당 영역에 필터가 감지하려는 특징이 얼마나 강하게 존재하는지를 나타냅니다.
3. **특징 맵 (Feature Map)**:
   - 필터가 전체 입력 이미지를 훑으면서 생성하는 출력 행렬입니다.
   - 하나의 필터는 하나의 특징 맵을 생성하며, 이 특징 맵은 다음 계층의 입력으로 사용됩니다.

부성순강사(bakpak@empas.com)

# 합성곱망에서의 Conv2D의 이해 - OpenCV에서의 합성곱

```
22 # ------------------------------------------------------
23 #  RGB 채널 분리
24 # ------------------------------------------------------
25 r, g, b = imgArray[:,:,0], imgArray[:,:,1], imgArray[:,:,2]
26
27 # ------------------------------------------------------
28 #  엣지 필터 (Laplacian 계열)
29 # ------------------------------------------------------
30 kernel = np.array([[-1,-1,-1],
31                    [-1, 8,-1],
32                    [-1,-1,-1]], dtype=np.float32)
33
34 r_edge = cv2.filter2D(r, -1, kernel)
35 g_edge = cv2.filter2D(g, -1, kernel)
36 b_edge = cv2.filter2D(b, -1, kernel)
37
38 # ------------------------------------------------------
39 #  채널별 결과 평균 (PyTorch Conv2d와 동일)
40 # ------------------------------------------------------
41 merged_edge = (r_edge + g_edge + b_edge) / 3.0
42
```




이 커널은

**"밝기 변화가 있는 부분만 두드러지게 만드는 역할"**

, 즉 **엣지(윤곽선) 검출**

부성순 bakpak@empas.com

# 합성곱망에서의 Conv2D의 이해

## 커널 정의

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

이 커널은

**"밝기 변화가 있는 부분만 두드러지게 만드는 역할"**

, 즉 **엣지(윤곽선) 검출**

**입력 이미지 주변 3×3 영역 (픽셀 블록)**

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

## 합성곱 연산(convolution)

$$\text{Output}(x,y) = \sum_{i=-1} \sum_{j=-1} K[i,j] \times I(x+i, y+j)$$

이때 중심 e=200, 나머지 주변은 대략 50 근처 값

- 주변보다 중심이 훨씬 밝으므로 **큰 양수(≈ 흰색)**
- 만약 중심이 주변보다 어두우면 결과는 **음수(≈ 검정)**
- 중심과 주변이 비슷하면 **0 근처(회색)**

## 합성곱 연산(convolution)

$$\text{Output} = (-1)a + (-1)b + (-1)c + (-1)d + (8)e + (-1)f + (-1)g + (-1)h + (-1)i$$

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 50 & 52 & 49 \\ 51 & 200 & 53 \\ 48 & 49 & 52 \end{bmatrix}$$

커널*이미지

| | | |
|---|---|---|
| -50 | -52 | -49 |
| -51 | 1600 | -53 |
| -48 | -49 | -52 |

합=1195

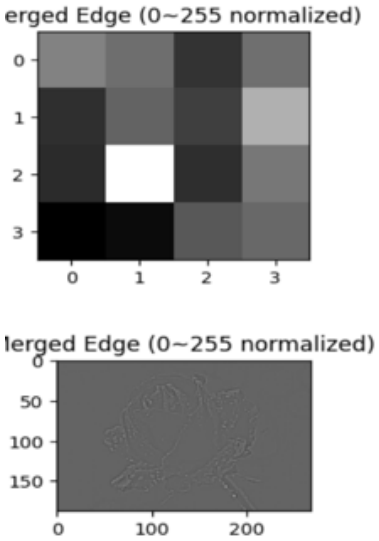
부성순 bakpak@empas.com

# 합성곱망에서의 Conv2D의 이해 - OpenCV에서의 합성곱

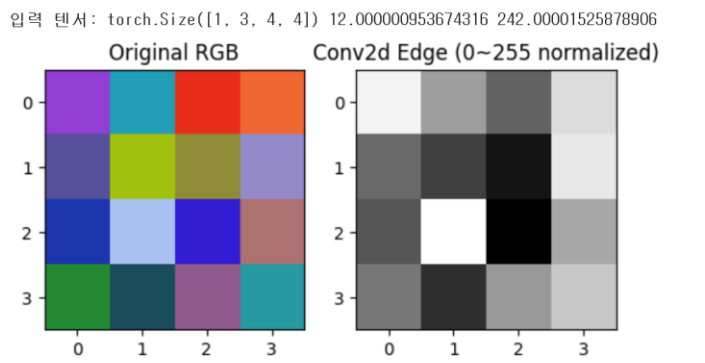
```
28 x = transform(img_np).unsqueeze(0)  # shape: (1, 3, 4, 4)
29 print("입력 텐서:", x.shape, x.min().item(), x.max().item())
30
31 # ----------------------------
32 #  Conv2d 정의 (padding=1)
33 # ----------------------------
34 conv = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3, bias=False, padding=1)
35
36 kernel = torch.tensor([[-1., -1., -1.],
37                        [-1.,  8., -1.],
38                        [-1., -1., -1.]])
39
40 with torch.no_grad():
41     conv.weight[:] = kernel.repeat(3,1,1).unsqueeze(0) / 3  # 3채널 평균
42
```

커널값 무작위가 아닌 강제로 할당

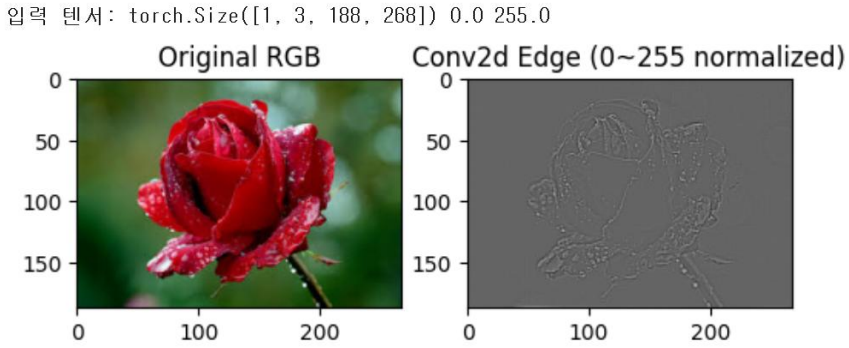
OpenCV



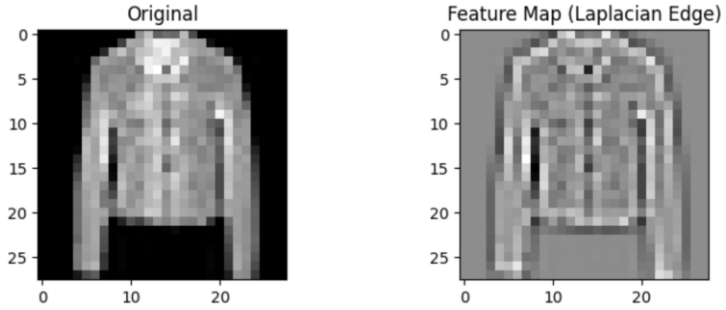

Torch




Torch



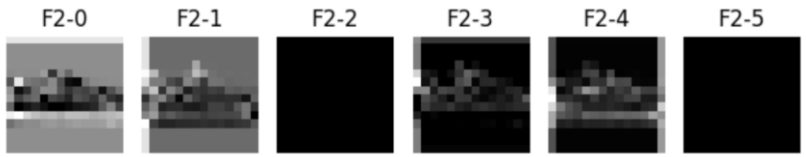

Torch




부성순 bakpak@empas.com

# [3] fashion_mnist를 이용한 pytorch CNN_ 특징맵

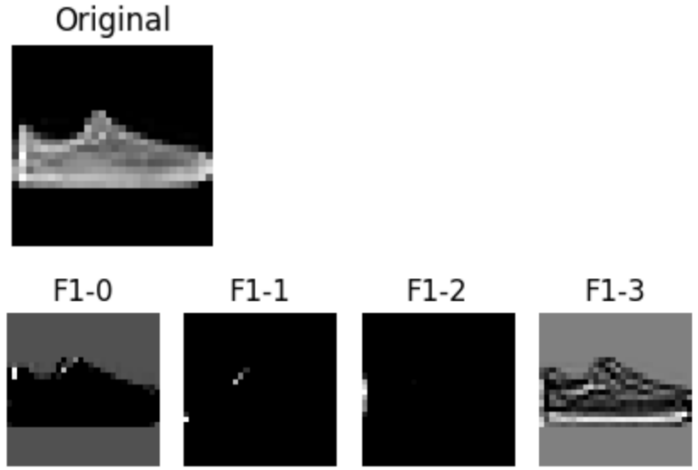
```
4 class SimpleCNN(nn.Module):
5     def __init__(self):
6         super().__init__()
7         self.conv1 = nn.Conv2d(1, 4, 3, padding=1)   # 1채널 → 4채널
8         self.conv2 = nn.Conv2d(4, 8, 3, padding=1)   # 4채널 → 8채널
9         self.pool = nn.MaxPool2d(2, 2)
10        self.relu = nn.ReLU()
11    def forward(self, x):
12        x1 = self.relu(self.conv1(x))    # conv1 출력
13        x2 = self.pool(self.relu(self.conv2(x1)))  # conv2 출력 (pool 포함)
14        return x1, x2
15
16 model = SimpleCNN()
17
```




**시각적으로 보면:**

- F2-0, F2-1 : 신발 형태 전체를 더 넓은 시야로 인식
- F2-2 : 거의 검정 → 반응 거의 없음
- F2-3 ~ F2-5 : 신발의 밑창, 윤곽선, 그림자 등에 강한 반응

➡️ 즉, **Conv2d 깊어질수록 feature map이 "전체 형태"나 "큰 윤곽" 중심으로 바뀜**




**시각적으로 보면:**

- F1-0 : 신발 윗부분의 명암 차이를 잡아내고 있음
- F1-1, F1-2 : 거의 반응이 없는 검은 화면 → 필터가 그 패턴에 덜 반응
- F1-3 : 신발 전체 형태(밑창, 윤곽선)를 어느 정도 포착

➡️ 즉, "무작위 Conv2d라도 이미 픽셀 주변의
작은 패턴(엣지, 밝기 대비)"에 반응하기 시작

훈련을 시키면 이 필터들이 "엣지 → 윤곽 → 질감" 순으로 점점 정제됨

부성순 bakpak@empas.com

# Conv2D에서 가장 중요한건

**Conv2d에서 가장 중요한 건 "패턴을 인식할 수 있게 만드는 가중치(필터, kernel)"**

## 🔹 1️⃣ 핵심은 "필터(커널, weight)"

`Conv2d`는 단순히 픽셀을 더하거나 평균내는 연산이 아니라,
**작은 커널(보통 3×3, 5×5 등)** 을 이미지 위로 움직이면서
각 위치의 **패턴(엣지, 질감, 형태)** 에 대한 "민감도"를 계산합니다

## 🔹 2️⃣ 왜 "가중치(필터)"가 제일 중요하냐

- CNN이 **이미지를 이해**하려면
  단순히 픽셀 값이 아니라 **픽셀 간의 관계(공간적 구조)** 를 학습해야 합

## 시각적으로 보면

- 첫 Conv2d → 엣지, 명암, 방향 (낮은 수준 특징)
- 두 번째 Conv2d → 질감, 형태
- 세 번째 Conv2d → 물체의 부분 (신발 밑창, 옷깃 등)
- 마지막 레이어 → 클래스(신발, 셔츠 등)에 직접 연관된 패턴

이런 "**패턴의 위계적 학습(Hierarchical Feature Extraction)**"
을 가능하게 하는 게 Conv2 핵심인 **weight(커널)** 들입니다.

- 그외 Conv2d 요소

| 요소 | 역할 | 왜 중요한가 |
|---|---|---|
| stride | 필터 이동 간격 | 너무 크면 세부 정보 손실, 너무 작으면 계산 많음 |
| padding | 테두리 보정 | 출력 크기 유지 및 가장자리 정보 보존 |
| activation (ReLU) | 비선형성 추가 | 단순 선형 필터를 복잡한 패턴 탐지기로 바꿔줌 |
| pooling | 지역 요약 | 위치 변화에 강건(translation invariance)하게 함 |
| **채널 수** (out_channels) | 필터 개수 | 다양한 패턴을 동시에 학습하게 함 |

부성순 bakpak@empas.com

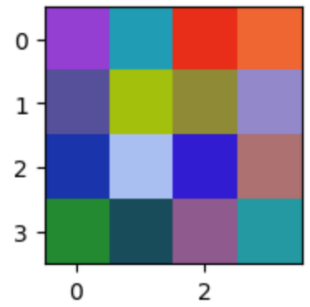
# CNN의 완전 연결층 미리보기

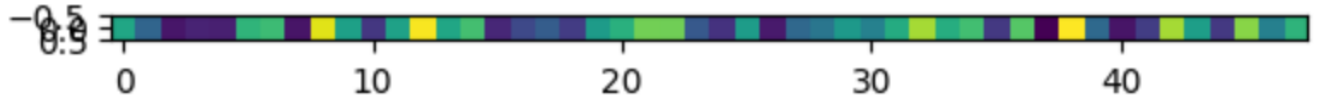
```python
import torch

# 1. 입력 텐서 (가상의 특징 맵) 정의
# (배치 크기: 1, 채널 수: 3, 높이: 4, 너비: 4)
# 이 텐서는 '특징 추출 단계'를 거친 후의 중간 결과라고 가정합니다.
# rand는 0~1사이값만 무작위로 만듦
input_tensor = torch.rand(1, 3, 4, 4)
print(input_tensor)

print(f"**변환 전 텐서 모양:** {input_tensor.shape}")
print("-" * 30)

# 2. view(-1, ...) 연산 적용 (Flatten)
# -1 : 배치 크기는 그대로 유지 (PyTorch가 알아서 계산)
# 3 * 4 * 4 : 나머지 차원을 모두 곱하여 1차원 벡터로 만듦
flattened_size = 3 * 4 * 4
output_tensor = input_tensor.view(-1, flattened_size)

# 3. 결과 확인
print(f"**변환 후 텐서 모양:** {output_tensor.shape}")
print(f"1차원 벡터의 크기 (3 * 4 * 4): {flattened_size}")
print(output_tensor)
```

1

2

```
tensor([[[[0.5815, 0.3409, 0.1042, 0.1390],
          [0.1276, 0.6371, 0.6641, 0.0994],
          [0.9100, 0.5619, 0.1958, 0.5647],
          [0.9409, 0.5797, 0.6783, 0.1477]],

         [[0.2473, 0.3151, 0.2086, 0.5424],
          [0.6192, 0.7567, 0.7522, 0.3017],
          [0.1812, 0.5446, 0.1129, 0.3535],
          [0.4006, 0.5367, 0.4457, 0.6019]],

         [[0.8268, 0.6091, 0.6774, 0.1970],
          [0.7131, 0.0490, 0.9487, 0.3579],
          [0.1010, 0.2131, 0.8264, 0.5533],
          [0.1995, 0.7888, 0.4406, 0.6352]]]])
**변환 전 텐서 모양:** torch.Size([1, 3, 4, 4])
------------------------------
**변환 후 텐서 모양:** torch.Size([1, 48])
1차원 벡터의 크기 (3 * 4 * 4): 48
tensor([[0.5815, 0.3409, 0.1042, 0.1390, 0.1276, 0.6371, 0.6641, 0.0994, 0.9100,
         0.5619, 0.1958, 0.5647, 0.9409, 0.5797, 0.6783, 0.1477, 0.2473, 0.3151,
         0.2086, 0.5424, 0.6192, 0.7567, 0.7522, 0.3017, 0.1812, 0.5446, 0.1129,
         0.3535, 0.4006, 0.5367, 0.4457, 0.6019, 0.8268, 0.6091, 0.6774, 0.1970,
         0.7131, 0.0490, 0.9487, 0.3579, 0.1010, 0.2131, 0.8264, 0.5533, 0.1995,
         0.7888, 0.4406, 0.6352]])
```




부성순 bakpak@empas.com

# CNN의 완전연결층 이해 : Conv2D 가 없을때 완전 연결층(FC Layer의 근본적인 문제점)

## 1. 엄청난 파라미터(가중치) 수 증가와 계산량 폭증

• **문제**: 완전 연결층은 **입력 뉴런의 모든 값**이 **다음 계층의 모든 뉴런**과 연결
따라서 이미지의 해상도가 조금만 커져도 모델이 학습해야 할 **가중치(파라미터)** 수가
기하급수적으로 늘어납니다. (10채널*4행*4열)⇒160개파라미터 (10채널*40*40)⇒1600개의 파람
• **결과**: 모델의 **학습 속도가 매우 느려지고**, 막대한 메모리가 필요하며, 실제 서비스 환경에 배포가 거의 불가능

## 2. 공간적 정보 손실 (Spatial Information Loss)

- **문제**: 이미지를 완전 연결층에 입력하기 위해서는 2차원 또는 3차원 데이터를 강제로 1차원 벡터로 평탄화(Flatten)해야 함
- **결과**: 이미지에서 픽셀들이 서로 **인접해 있다는 공간적 관계** (예: 눈과 코가 가까이 있다는 정보)가 완전히 무시되고 단지 긴 배열의 일부로 취급됩니다. 이는 이미지의 **특징을 효과적으로 추출하고 이해하는 데 큰 장애물**이 됩니다.

## 3. 위치 불변성 (Translation Invariance) 학습 불가

- **문제**: 완전 연결층은 입력 데이터의 **특정 위치**에 있는 특정 픽셀에만 반응하도록 가중치를 학습합니다.
- **결과**: 만약 훈련 데이터에서 고양이의 눈이 이미지의 **중앙**에 있었다면, 고양이가 이미지의 **왼쪽 구석**으로 이동할 경우, 모델은 이를 **전혀 다른 특징**으로 인식하고 정확하게 분류하지 못할 가능성이 높습니다. CNN의 **가중치 공유(Parameter Sharing)** 개념이 없기 때문에, 물체의 위치가 바뀌면 모델의 성능이 급격히 저하됩니다

CNN은 **합성곱(Convolution)** 연산을 통해 **파라미터 수를 줄이면서(효율성), 지역적 특징을 보존하고(공간 정보), 위치에 상관없이 동일한 특징을 감지(위치 불변성)** 할 수 있도록 하여, 완전 연결층이 가진 근본적인 한계를 극복합니다.

부성순 bakpak@empas.com