# 2026.02.03.화

| | |
|---|---|
| 생성일 | @2026년 2월 3일 오전 9:32 |
| 태그 | DeepLearning |

## 목차



## ▼ 1) CNN (Convolutional Nerual Network)

- 에지 검출, 특징 조합 등을 통해 이미지를 분류하는 AI
- 이미지나 영상 데이터를 처리하는 데 특화된 컴퓨터 비전 딥러닝 구조
- 주요 구성 요소
  - 채널(Channel): 입력 데이터의 깊이 차원
  - 필터(Filter) / 커널(Kernel): 이미지의 국소 영역을 스캔하며 특정 패턴(에지, 질감 등)을 추출하는 작은 가중치 행렬
  - 스트라이드(Strid): 필터가 한 번에 이동하는 픽셀 간격
  - 패딩(Padding): 입력 이미지 가장자리에 데이터를 추가하여 출력 크기를 유지하고 가장 자리 데이터 손실 방지
- 주요 레이어 구조
  - 합성곱 레이어(Convolutional Layer):
  - 활성화 함수(Activation Function):
  - 폴링 레이어(Pooling Layer):
  - 완전 연결 레이어(Fully Connected, Dense Layer):
- 주요 특징
  - 특징 학습: 사람이 특징을 설계하지 않고 스스로 최적의 필터 값을  찾아냄
  - 파라미터 공유: 일반적인 신경망보다 학습해야할 파라미터 수가 훨씬 적음.
  - 공간적 계층 구조: 하위 레이어에 선이나 점을 찾고 상위 레이어일 수록 복잡한 객체를 인식
- 일반 신경망으로는 이미지 처리가 어려운 이유
  - 필요한 매개변수가 매우 많음
  - 공간적 정보 손실 (Flattening)
  - 데이터를 1차원으로 변환해야함
  - 위치 변화에 민감 → 같은 객체여도 위치가 바뀌면 다른 이미지로 인식
- CNN 핵심 아이디어
  - 지역적 연결성
  - 각 뉴런은 작은 부분만 담당
  - 매개변수 공유
  - 하나의 필터를 전체 이미지에 적용
  - 계층적 특징 학습
- 실습 코드

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# 1. 샘플 이미지 생성 (8x8 체스보드 패턴)
sample_image = np.zeros((8, 8))

# 슬라이싱을 이용해 0과 1이 교차하도록 설정
sample_image[1::2, ::2] = 1
sample_image[::2, 1::2] = 1

# 2. sample_image 출력
plt.imshow(sample_image, cmap='gray')
plt.show()

def convolution_2d(image, kernel, stride=1, padding=0):
    """
    2D 컨볼루션 연산 구현

    Args:
        image: 입력 이미지 (H, W)
        kernel: 필터/커널 (K, K)
        stride: 이동 간격
        padding: 패딩 크기
    """
    # 입력 크기
    H, W = image.shape
    K = kernel.shape[0]
    # kernel.shape : vertical_edge 의 크기 (3)
    print(f"K: {K}")

    # 패딩 적용
    image = np.pad(image, padding, mode='constant')
    # np.pad(image, padding, mode='constant') : 이미지의 가장자리를 패딩으로 채움
    # padding : 패딩 크기
    # mode : 패딩 모드 (constant, reflect, edge, wrap)
    # constant : 0으로 채움
    # reflect : 반사 패딩
    # edge : 가장자리 복사
    # wrap : 순환 패딩
    H_padded, W_padded = image.shape

    # 출력 크기 계산
    # 원본 공식 : (H - K + 2*padding) // stride + 1
    # 이미 패딩 적용된 상태이므로 패딩 제외
    # 출력 크기 공식 : (패딩된 크기 - 커널 크기) / 스트라이드 + 1
    out_H = (H_padded - K) // stride + 1
    out_W = (W_padded - K) // stride + 1

    # 출력 배열 초기화
    output = np.zeros((out_H, out_W))
    # 초기화 하는 이유 : 컨볼루션 연산 결과를 저장할 공간을 만들기 위해

    # 컨볼루션 연산
    for i in range(out_H):
        for j in range(out_W):
            # 현재 위치
            h_start = i * stride
            # stride : 이동 간격
            h_end = h_start + K
            w_start = j * stride
            w_end = w_start + K

            # 해당 영역과 커널의 요소별 곱셈 후 합
```

```
            region = image[h_start:h_end, w_start:w_end]
            # region : 현재 위치의 영역
            # kernel : 필터/커널
            output[i, j] = np.sum(region * kernel)
            # np.sum(region * kernel) : 현재 위치의 영역과 커널의 요소별 곱셈 후 합

    return output

# 체스 보드 패턴을 가진 8x8 이미지 생성
# 흑백을 0과 255로 표현
chessboard_size = 8
chessboard = np.zeros((chessboard_size, chessboard_size), dtype=int)
for i in range(chessboard_size):
    for j in range(chessboard_size):
        if (i + j) % 2 == 0:
            chessboard[i, j] = 255 # 흰색
        else:
            chessboard[i, j] = 0 # 검은색

sample_image = chessboard # 체스 보드 이미지를 사용

vertical_edge = np.array([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]
])

# 수직 에지 검출 적용
padding = (vertical_edge.shape[0] - 1) // 2
vertical_edges = convolution_2d(sample_image, vertical_edge, padding=padding)

print(f"원본 크기: {sample_image.shape}")
print(f"결과 크기: {vertical_edges.shape}")

# vertical_edges의 의미
# 수직 방향 변화가 강하게 나오는 부분은 회색으로 검출

# 결과 시각화
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(sample_image, cmap='gray')

plt.subplot(1, 2, 2)
plt.title("Vertical Edges")
plt.imshow(vertical_edges, cmap='gray')

plt.tight_layout()
plt.show()
```

## ▼ 2) 컨볼루션 (Convolution)

- 입력 이미지에서 특징을 추출하는 연산
- 필터를 입력 데이터에 슬라이드 적용하며 국소적인 특징 추출
- 유의미한 공간적 특징을 추출하는 과정
- 공식
  - **$(f*g)(x,y)=\sum\sum f(i,j)\times g(x-i,y-j)$**

- 다양한 필터
  - 수직 / 수평
  - 대각선 / Sobel
  - 블러 / 샤프닝
  - 엠보싱
- 스트라이드: 필터의 이동 간격
  - Strid = 1: 세밀한 특징 추출
  - Strid = 2: 다운 샘플링과 특징 추출을 동시에 수행 (일반적)
  - Strid ≥ 3: 매우 빠른 처리가 필요하거나 실험적 용도
- 패딩: 필터로 인한 원본 이미지 가장 자리 정보 손실 방지, 출력 크기 입력과 같아짐
  - Zero Padding: 가장 자리 픽셀을 0으로 채우기 (일반적)
  - Reflect Padding: 이미지 품질 보존, 자연스러운 경계
  - Edge Padding: 가장 자리 복사, 부드러운 경계, 블러 효과, 이미지 노이즈 제거
  - Wrap Padding: 주기적 패턴 분석
- 특징맵
  - 컨볼루션 연산 결과로 생성되는 2D 배열
  - 입력 이미지에 필터를 적용한 결과
  - 층이 깊어질 수록 풀링에 의해 크기가 작아지고 복잡한 패턴 파악을 위한 채널 수는 많아짐
- 백본 네트워크 (Backbone Network)
  - 특징 추출 담당
  - 범위: 입력층부터 분류기 전까지
  - 종류: ResNet, VGG, EfficientNet
  - 재사용: 전이학습에서 백본만 가져와 사용
- 모델 복잡도를 나타내는 지표
  - 매개변수 수: 학습 가능한 가중치의 총 개수
  - FLOPS: 순전파 1회에 필요한 부동소수점 연산 수
  - 메모리 사용량: 모델 저장 및 추론에 필요한 메모리
  - 추론 시간: 하나의 이미지 분류에 걸리는 시간
- 실습 코드

```
import numpy as np

def full_convolution_demo():
    """전체 컨볼루션 과정 시연"""

    # 5x5 입력 이미지 (위에서 아래로 밝아지는 패턴)
    image = np.array([
        [50,  50,  50,  50,  50],  # 밝은 영역
        [50,  50,  50,  50,  50],  # 밝은 영역
        [100, 100, 100, 100, 100],  # 중간 영역
        [150, 150, 150, 150, 150],  # 어두운 영역
        [150, 150, 150, 150, 150]   # 어두운 영역
    ])

    # 수평 에지 검출 필터 (Horizontal Edge Detection Filter)
    filter = np.array([
        [-1, -1, -1],
        [ 0,  0,  0],
```

```
        [ 1,  1,  1]
    ])

    # 3x3 출력 배열 (특징 맵) 초기화
    output = np.zeros((3, 3))

    print("각 위치별 컨볼루션 계산:")
    for i in range(3):
        for j in range(3):
            # 현재 3x3 영역 추출
            region = image[i:i+3, j:j+3]
            # 3x3 영역 출력
            print(f"현재 영역: {region}")

            # 컨볼루션 계산
            result = np.sum(region * filter)
            output[i, j] = result

            print(f"위치 ({i},{j}): {result}")


    print(f"\n최종 출력:\n{output}")
    print("\n해석:")
    print("- 0에 가까운 값: 에지 없음")
    print("- 큰 음수/양수: 강한 에지 검출")

full_convolution_demo()
```

```
import numpy as np
import matplotlib.pyplot as plt

# 다중 채널 컨볼루션 (함수 정의를 먼저)
def convolution_3d(image, filters, stride=1, padding=0):
    """
    3D 컨볼루션 (다중 채널 입력, 다중 필터)
    """
    K, _, _, C_out = filters.shape
    # K : 커널의 높이
    # C_out : 커널의 출력 채널 수

    image = np.pad(image, ((padding, padding), (padding, padding), (0, 0)))
    H_padded, W_padded, C_in = image.shape

    out_H = (H_padded - K) // stride + 1
    out_W = (W_padded - K) // stride + 1
    output = np.zeros((out_H, out_W, C_out))

    for f in range(C_out):  # 각 출력 채널
        for i in range(out_H):
            for j in range(out_W):
                h_start = i * stride
                h_end = h_start + K
                w_start = j * stride
                w_end = w_start + K

                # 모든 입력 채널에서 연산 후 합
                region = image[h_start:h_end, w_start:w_end, :]
                output[i, j, f] = np.sum(region * filters[:, :, :, f])
```

```
    return output

# RGB 이미지 예시: 32x32 크기의 컬러 이미지 (3개 채널)
rgb_image = np.random.rand(32, 32, 3)

# 필터 설정: 16개의 3x3 필터 (각 필터는 입력 채널 수와 같은 3개 채널을 가짐)
filters = np.random.randn(3, 3, 3, 16)

# 첫 번째 필터의 첫 번째 채널 확인
print(f"filters : {filters[:, :, 0, 0]}")

# --- 필터의 의미 ---
# 예시 : filters[:, :, 0, 0] : 첫 번째 필터 중 빨간색(R) 채널에 대응하는 가중치 행렬
# 예시 : filters[:, :, 1, 0] : 첫 번째 필터 중 초록색(G) 채널에 대응하는 가중치 행렬
# 예시 : filters[:, :, 2, 0] : 첫 번째 필터 중 파란색(B) 채널에 대응하는 가중치 행렬

# --- 실제 응용 ---
# 총 16개의 필터가 있으므로 연산 결과 16개의 출력 채널(특징 맵)이 생성됨
# 다양한 채널의 정보를 통합하여 더 복잡하고 고차원적인 특징을 추출 가능

# 패딩 계산 및 컨볼루션 실행
padding = (filters.shape[0] - 1) // 2
feature_maps = convolution_3d(rgb_image, filters, padding=padding)
print(f"입력: {rgb_image.shape} -> 출력: {feature_maps.shape}")

# 결과
# filters : [[ 0.71455458  1.60690745  1.431584  ]
# [ 1.25201618 -1.24619664  1.06937282]
# [ 0.23047639 -2.19038809 -0.21323507]]
# 입력: (32, 32, 3) -> 출력: (32, 32, 16)

# 시각화
plt.figure(figsize=(15, 5))

# 원본 이미지
plt.subplot(1, 3, 1)
plt.title("Original RGB Image")
plt.imshow(rgb_image)

# 첫 번째 필터의 첫 번째 채널 (3x3)
plt.subplot(1, 3, 2)
plt.title("First Filter (1st channel)")
plt.imshow(filters[:, :, 0, 0], cmap='gray') # 첫 번째 필터의 첫 번째 입력 채널

# 첫 번째 특징맵
plt.subplot(1, 3, 3)
plt.title("First Feature Map")
plt.imshow(feature_maps[:, :, 0], cmap='gray')

plt.tight_layout()
import os
output_path1 = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'convolution_result1.png')
plt.savefig(output_path1, dpi=150, bbox_inches='tight')
print(f"저장됨: {output_path1}")
plt.show()

# 추가: 여러 필터들과 feature maps 시각화
plt.figure(figsize=(16, 8))

# 첫 번째 행: 여러 필터들 표시 (처음 4개)
```

```
for i in range(4):
    plt.subplot(2, 4, i+1)
    plt.title(f"Filter {i+1} (1st channel)")
    plt.imshow(filters[:, :, 0, i], cmap='gray')
    plt.axis('off')

# 두 번째 행: 해당 필터들의 feature maps 표시
for i in range(4):
    plt.subplot(2, 4, i+5)
    plt.title(f"Feature Map {i+1}")
    plt.imshow(feature_maps[:, :, i], cmap='gray')
    plt.axis('off')

plt.tight_layout()
output_path2 = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'convolution_result2.png')
plt.savefig(output_path2, dpi=150, bbox_inches='tight')
print(f"저장됨: {output_path2}")
plt.show()
```

## ▼ 3) 풀링 (Pooling)

- 특징맵의 크기를 줄이는 연산: 차원 축소 (불필요한 데이터를 줄이는 연산)
- 계산량 및 메모리 사용량 감소
- 실습 코드

```
import numpy as np

# 4x4 입력 데이터 생성 (풀링 연산 시연용)
input_data = np.array([
    [1,  3,  2,  4],
    [5,  6,  7,  8],
    [9,  10, 11, 12],
    [13, 14, 15, 16]
])

print("원본 입력:")
print(input_data)
print()

def max_pooling_2d(input_array, pool_size=(2, 2), stride=2):
    """
    2D 최대 풀링 구현

    Args:
        input_array: 입력 배열 (H, W)
        pool_size: 풀링 윈도우 크기
        stride: 스트라이드

    Returns:
        풀링된 배열
    """
    h, w = input_array.shape
    pool_h, pool_w = pool_size

    # 출력 크기 계산 (패딩 없다고 가정)
    out_h = (h - pool_h) // stride + 1
    out_w = (w - pool_w) // stride + 1

    # 출력 배열 초기화
```

```
    output = np.zeros((out_h, out_w))

    # 풀링 연산 수행
    for i in range(out_h):
        for j in range(out_w):
            # 현재 윈도우 영역 계산
            h_start = i * stride
            h_end = h_start + pool_h
            w_start = j * stride
            w_end = w_start + pool_w

            # 2x2 풀링 윈도우 영역 추출
            # 2칸씩 이동하면서 최대값 선택
            # 4x4 이미지에 2x2 풀링 적용 시:
            # 첫 번째 윈도우 : 0~1행, 0~1열
            # 두 번째 윈도우 : 0~1행, 2~3열
            # 세 번째 윈도우 : 2~3행, 0~1열
            # 네 번째 윈도우 : 2~3행, 2~3열

            # 슬라이싱을 통한 윈도우 설정 및 최대값 선택
            window = input_array[h_start:h_end, w_start:w_end]
            output[i, j] = np.max(window)

    return output

def average_pooling_2d(input_array, pool_size=(2, 2), stride=2):
    """
    2D 평균 풀링 구현

    Args:
        input_array: 입력 배열 (H, W)
        pool_size: 풀링 윈도우 크기
        stride: 스트라이드

    Returns:
        풀링된 배열
    """
    h, w = input_array.shape
    pool_h, pool_w = pool_size

    # 출력 크기 계산 (패딩 없다고 가정)
    out_h = (h - pool_h) // stride + 1
    out_w = (w - pool_w) // stride + 1

    # 출력 배열 초기화
    output = np.zeros((out_h, out_w))

    # 풀링 연산 수행
    for i in range(out_h):
        for j in range(out_w):
            # 현재 윈도우 영역
            h_start = i * stride
            h_end = h_start + pool_h
            w_start = j * stride
            w_end = w_start + pool_w

            # 평균값 계산
            # 지정된 윈도우 영역([h_start:h_end, w_start:w_end]) 내의
            # 모든 원소의 산술 평균을 구하여 출력 배열에 저장합니다.
            window = input_array[h_start:h_end, w_start:w_end]
```

```
            output[i, j] = np.mean(window)

    return output

# 최대 풀링 적용
max_pooled = max_pooling_2d(input_data)
print("최대 풀링 결과:")
print(max_pooled)
print()

# 평균 풀링 적용
avg_pooled = average_pooling_2d(input_data)
print("평균 풀링 결과:")
print(avg_pooled)


import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_sample_image

# 샘플 이미지 로드
# sklearn 에서 제공하는 샘플 이미지 로드
# 이미지 크기 : 427x640x3 (높이, 너비, 채널)
image = load_sample_image("flower.jpg")

gray_image = np.mean(image, axis=2) # 그레이스케일 변환

print(f"원본 이미지 크기: {gray_image.shape}")

# 이미지에 풀링 적용
pooled_image = max_pooling_2d(gray_image, pool_size=(4, 4), stride=4)
print(f"풀링 후 이미지 크기: {pooled_image.shape}")

# 시각화
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

axes[0].imshow(gray_image, cmap='gray')
axes[0].set_title(f'Original Image ({gray_image.shape})')
axes[0].axis('off')

axes[1].imshow(pooled_image, cmap='gray')
axes[1].set_title(f'4x4 Max Pooling ({pooled_image.shape})')
axes[1].axis('off')

plt.tight_layout()
plt.show()

# 크기 비교
print(f"데이터 크기 감소: {gray_image.size} -> {pooled_image.size}")
print(f"압축률: {pooled_image.size / gray_image.size:.2%}")
```

## ▼ 4) CNN 아키텍처

- 특징 추출 블록
  - 의미있는 특징 추출
  - 구성: 컨볼루션층 + 활성화함수 + 풀링층
  - 흐름: 입력 이미지 → 컨볼루션 (특징 추출) → 활성화 (비선형성) → 정규화 → 풀링 (축소)
- 분류기

    - 고차원 특징을 클래스 레이블로 매핑
    - 구성: Flatten → Dense → Dropout → Dense
- 클래스 확률
    - 활성화 함수 적용
    - 가장 높은 확률의 클래스가 예측 결과