계획서

[이미지 분석 모델 CNN 구현] 계획서

구성원	팀장 : 우동협 팀원 : 남재현 정채현 최종렬 한서연	
시간	일정 : 매주 화요일 오후 7시 참고사항 : 원활한 진행을 위해 주 1회 이상 대면 활동	
장소	K-CUBE, K-HUB	
교재	혼자 공부하는 머신러닝+딥러닝 외 다양한 참고자료	
프로젝트 주제	<프로젝트 주제> 이미지 분석 모델 CNN에 대한 이해와 실제 코드 구현 저희 조는 CNN에 대해 스터디 진행 후 실제로 CNN 모델을 코드로 구현합니다. 이후 가장 기초적인 데이터와 실제 데이터를 하나씩 선택하여 CNN을 원하는 목적에 맞게 학습시키는 활동을 진행할 예정입니다. 예를 들어, 얼굴을 통해 나이를 예측하거나, 감정을 분류할 수 있는 CNN 모델을 만들어볼 수 있습니다.	
	1회차 (11/12 ~ 11/17)	CNN 스터디를 통한 CNN 이해
	2회차 (11/18 ~ 11/24)	CNN 코드 구현 후 MNIST, CIFAR-10 학습
	3회차 (11/25 ~ 12/01)	원하는 데이터를 통해 CNN 학습
	4회차 (12/02 ~ 12/03)	발표준비
프로젝트 방식	각 주차 별 학습자료를 선정하여 각자 공부 후 배운 내용을 설명하고 질문하는 방식으로 진행할 예정입니다. 프로젝트의 장점은 궁금한 점에 대해 물어보고 같이 고민할 수 있는 팀원이 있다는 것입니다. 이러한 장점을 최대한 살릴 수 있는 방향으로 프로젝트를 진행하고 싶습니다.	
프로젝트 규칙	- 궁금한 점 편하게 물어보기 - 시간 엄수 - 열정 - 원활한 소통	

참고자료

1주차 참고자료

https://wikidocs.net/64066

https://www.youtube.com/watch?si=Ev_lqkEHQV_slg7-&v=aircAruvnKk&feature=youtu.be

AlexNet.pdf

▼ AlexNet

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

SOTA

1 Introduction

Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Until recently, datasets of labeled images were relatively small — on the order of tens of thousands of images (e.g., NORB [16], Caltech-101/256 [8, 9], and CIFAR-10/100 [12]). Simple recognition tasks can be solved quite well with datasets of this size, especially if they are augmented with label-preserving transformations. For example, the current-best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance [4]. But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets. And indeed, the shortcomings of small image datasets have been widely recognized (e.g., Pinto et al. [21]), but it has only recently become possible to collect labeled datasets with millions of images. The new larger datasets include LabelMe [23], which consists of hundreds of thousands of fully-segmented images, and ImageNet [6], which consists of over 15 million labeled high-resolution images in over 22,000 categories.

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs) constitute one such class of models [16, 11, 13, 18, 15, 22, 26]. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feedforward neural networks with similarly-sized layers, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-best performance is likely to be only slightly worse.

1

2주차 참고자료

https://wikidocs.net/62304

남재현

활동현황	완료

NN과 CNN

- 컨볼루션 레이어 (Convolutional Layer)

  입력 데이터에서 지역적(local) 특징을 추출

  kernel을 사용해 이미지의 특정 패턴(edge 등)을 감지하고 연산 결과로 Feature Map 출력

  이미지와 같은 공간적 데이터에서 중요한 정보를 학습

  활성화 함수가 적용될 수 있음

  입력 크기가 커도 모든 뉴런이 연결되지 않기 때문에 파라미터 수가 줄어듬

- 풀링 레이어 (Pooling Layer)

  Feature Map에서 공간적 크기를 줄이고 중요한 정보만 요약. 이때, kernel 사용

  풀링 커널은 학습 가능한 가중치를 가지지 않고, 고정된 연산(최대값, 평균값)을 수행

  지정된 스트라이드(stride) 값만큼 입력 데이터를 이동함. stride=2 라면 kernel은 2칸씩 이동

  중요하지 않은 정보를 제거하니까 overfitting을 예방

- 완전 연결 레이어 (Fully Connected Layer)

  추출된 특징은 local한 패턴에 집중되어 있음. 이를 global 관점에서 종합적으로 분류/회귀 작업을 수행

  풀링 레이어를 지난 특징 맵은 여전히 2D tensor 형태로 남아 있음

  를 1차원 벡터로 변환하여 뉴런 간의 완전 연결을 형성

  CNN의 마지막 단계에서 최종 출력 값 생성

softmax와 같은 활성화 함수로 클래스 확률을 계산함

계층 간의 관계

1. **컨볼루션 레이어**:
    - 저수준 특징(예: 가장자리, 선)을 학습합니다.
    - 여러 계층을 통해 고수준 특징(예: 객체 모양)을 학습합니다.
2. **풀링 레이어**:
    - 컨볼루션 레이어에서 추출된 특징의 크기를 줄이고 가장 중요한 정보를 유지합니다.
    - 네트워크의 공간적 불변성을 증가시킵니다.
3. **완전 연결 레이어**:
    - 컨볼루션과 풀링에서 학습된 정보를 바탕으로 예측을 수행합니다.
    - CNN의 출력이 직접적으로 연결된 분류기로 동작합니다.

NN (Neural Network - Fully Connected Network)

- **구조**:
    - 모든 뉴런이 다음 층의 모든 뉴런과 연결된 완전 연결 구조(fully connected layers)로 이루어져 있음.
    - 데이터의 구조적 정보를 고려하지 않고, 모든 입력 데이터를 1차원 벡터로 취급.
    - 주로 입력 데이터의 순서나 위치 관계를 무시하고 전역적인 패턴을 학습함.
- **특징**:
    - 데이터의 형태(예: 이미지, 텍스트 등)를 고려하지 않음.
    - 입력 데이터의 모든 특징을 동일한 방식으로 학습.

CNN (Convolutional Neural Network)

- **구조**:
  - **컨볼루션 레이어(Convolutional Layers)**: 데이터의 지역적(local) 패턴을 학습하기 위해 필터를 사용하여 이미지와 같은 데이터의 공간적 관계를 학습.
  - **풀링 레이어(Pooling Layers)**: 공간적 크기를 줄이고, 연산량을 줄이기 위해 중요한 특징만 유지.
  - **완전 연결 레이어(Fully Connected Layers)**: 마지막 단계에서 CNN에서 추출된 특징을 기반으로 결정을 내리기 위해 사용.
  - 입력 데이터를 2D 또는 3D 형태(예: 이미지)로 처리하며, 데이터의 구조적 관계를 고려.
- **특징**:
  - **지역적 정보 학습**: 필터를 통해 이미지의 특정 부분(예: 경계, 색상 등)을 학습.
  - **계층적 특징 추출**: 초기 층에서 저수준 특징(선, 가장자리)을 학습하고, 이후 층에서 고수준 특징(물체 형태)을 학습.
  - 파라미터 수를 크게 줄여 연산 효율성 증가.

우동협

활동현황	완료

합성곱 신경망의 구성 요소

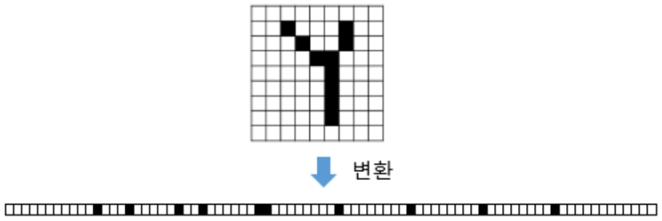
Images Premise

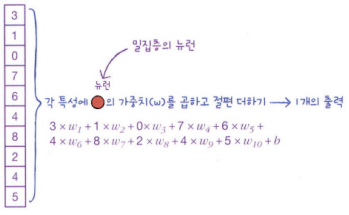
"they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies)."

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Communications of the ACM 60.6 (2017)

합성곱신경망에 대한 이해

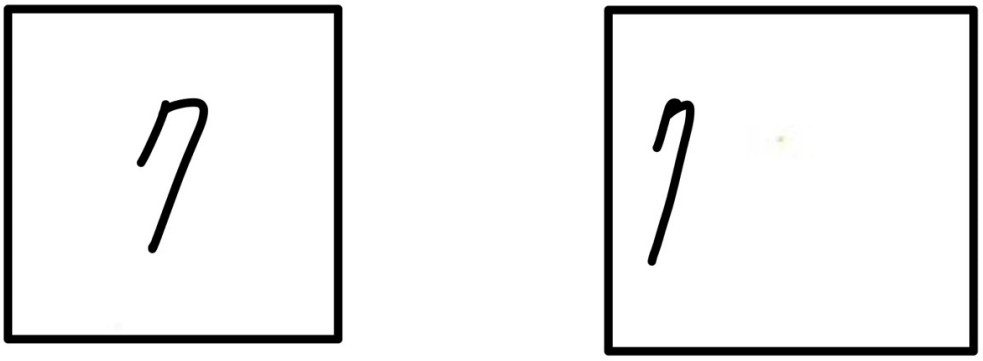
기존의 신경망을 통해 이미지를 처리할 때는 다음과 같은 형태로 처리한다.





자료에서 확인할 수 있듯이 이미지 하나를 하나의 정보(하나의 값)로 처리한다.

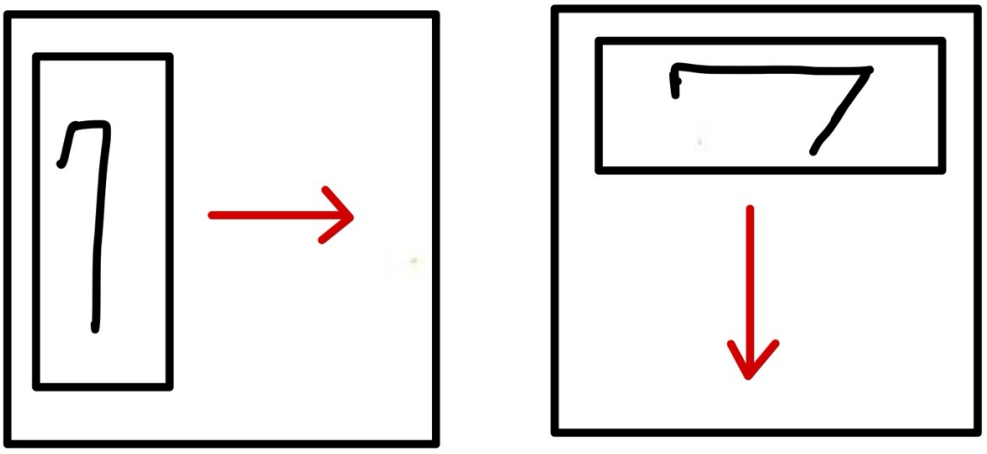
이 말은 즉 아래의 데이터도 다른 가중치를 적용해야 동일한 것으로 인식할 수 있다.



이미지를 평면화시켰을 때, 공간정보가 소실된다는 것은 위 두 7을 전혀 다른 정보로 파악해야한다는 의미!

물론 노드의 수를 늘려 이러한 모든 경우의 수를 포함시킬 수 있다면 좋겠지만, 효율적이지는 않다.

그래서 처음에는 이렇게 접근했을 것 같다.



그 7모양에 맞는 다양한 합성곱 뉴런을 준비한 후 슬라이드하면서 입력 데이터 이해!

→ 동일한 형태, 다른 위치(평행이동 된 대상)에 있는 정보 인식 가능!

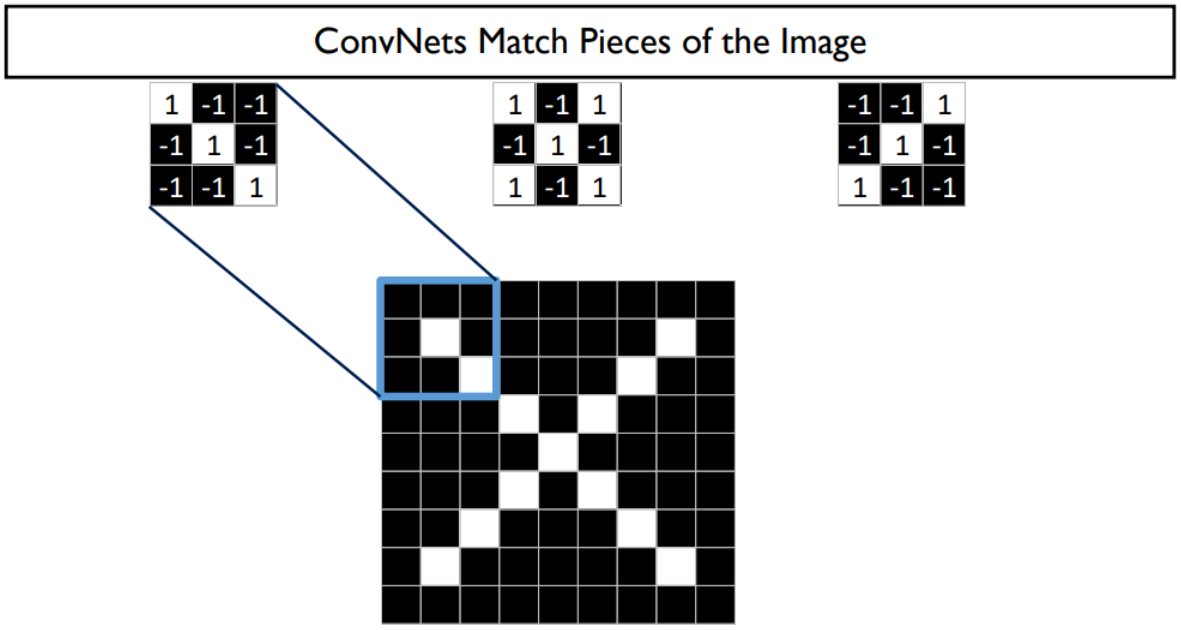
하지만 이러한 형태는 크기, 모양이 다를 경우에는 잘 인식하지 못한다는 어려움이 있다. 물론 다양한 형태(모양)의 합성곱 뉴런을 만들면 되지만, 이러한 방식에서는 입력데이터 7의 이미지 속 크기에 따라 합성곱 신경망의 크기(3X3→ 작은 7, 5X5 → 조금 더 큰 7) 역시 변해야 한다는 한계가 있다.

🤔 합성곱 뉴런의 크기를 달리하는게 왜 문제?

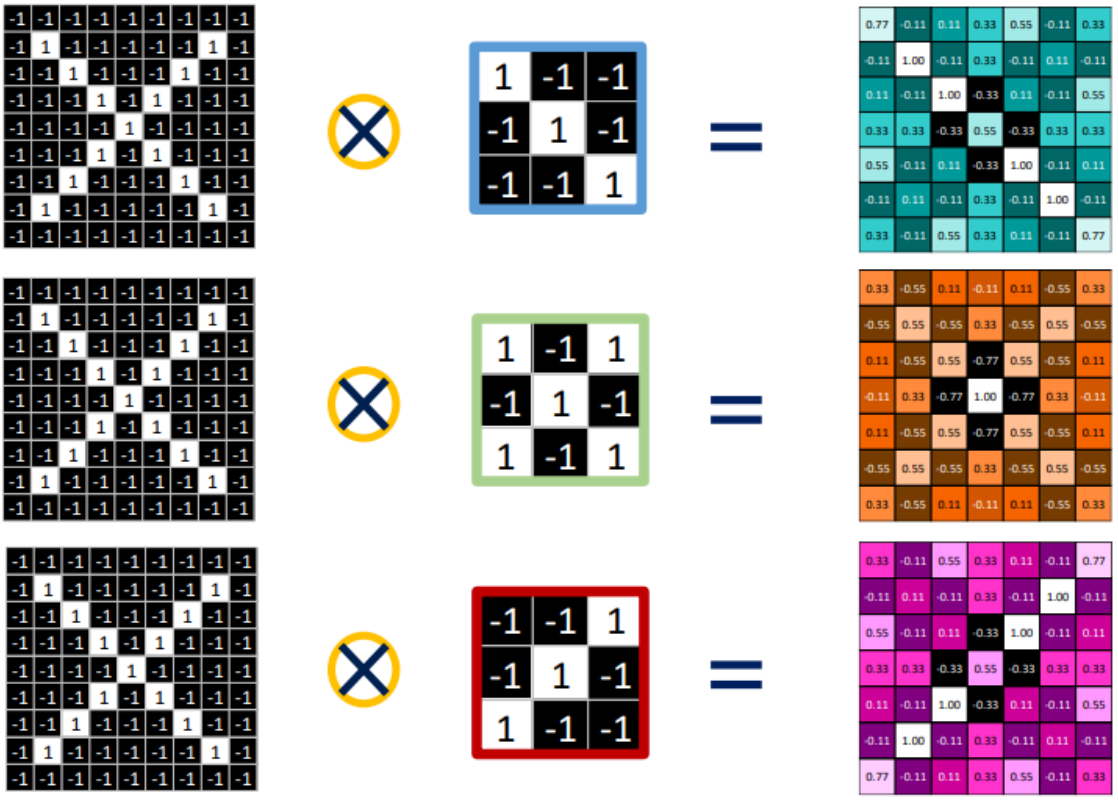
솔직히 하나의 데이터를 다양한 크기의 합성곱(예를 들어, 3X3, 5X5)으로 처리해도 문제는 없다고 생각한다. 출력결과는 stride와 pooling을 통해 동일한 크기로 조절할 수 있다. 하지만 입력데이터 속 대상(예를 들어 7)에 따라 다양한 크기의 합성곱신경망을 만다는 것은
경우의 수가 너무 많다.

따라서 합성곱 뉴런의 크기는 고정해야 하니 인식을 할 때, 작은 단위로 인식을 해보자고 접근할 수 있다. 예를들어, 한글을 ㅣ,ㅡ모양의 결합으로 이해하는 것과 ㄱ,ㄴ,ㄷ,ㄹ 등 모두 따로 하나의 객체로 보는 것은 경우의 수가 너무 많이 차이난다.

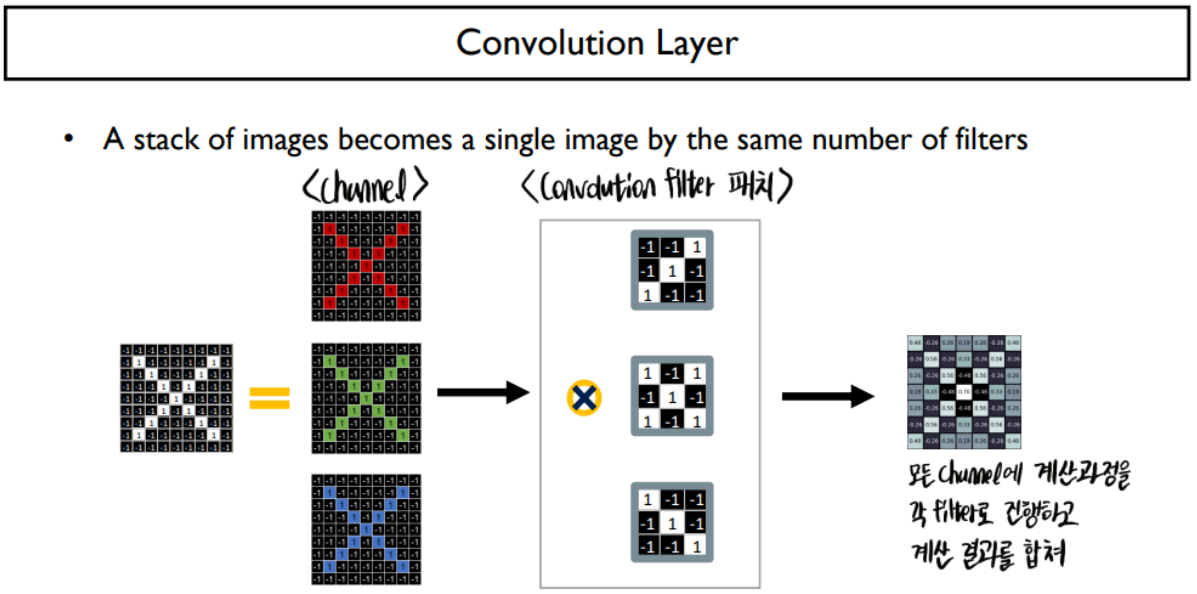
그래서 합성곱은 입력데이터를 작은 단위로 쪼개어 작은 부분부터 이해하는 방식을 사용한다.



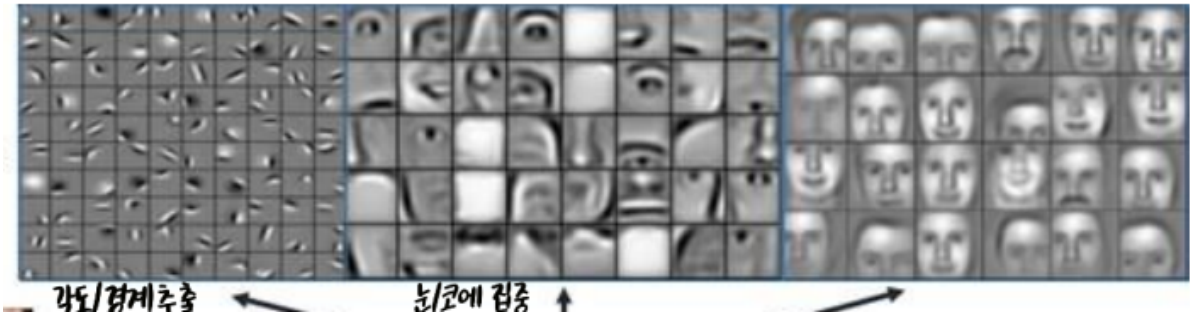
이러한 방식을 통해 각 작은 단위의 위치정보를 파악한다.

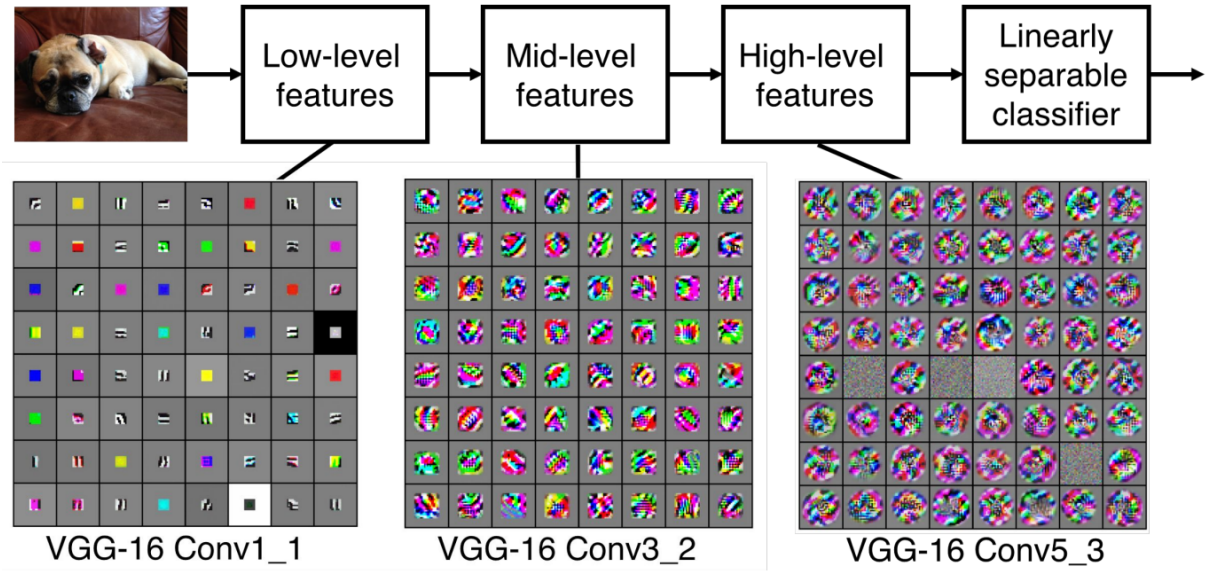


그리고 이들을 합치면 입력데이터의 작은 조각들이 어떻게 서로 인접해있는지 즉 작은 조각들의 부분합을 이해할 수 있다.



이렇게 출력된 결과는 작은 조각들의 결합, 다시말해 조금 더 큰 조각들이 된다. 그리고 다시 더 큰 조각들의 합성곱 신경망을 통해 각 조각들의 위치정보를 구하여 이전보다 더 큰 조각들을 구한다.





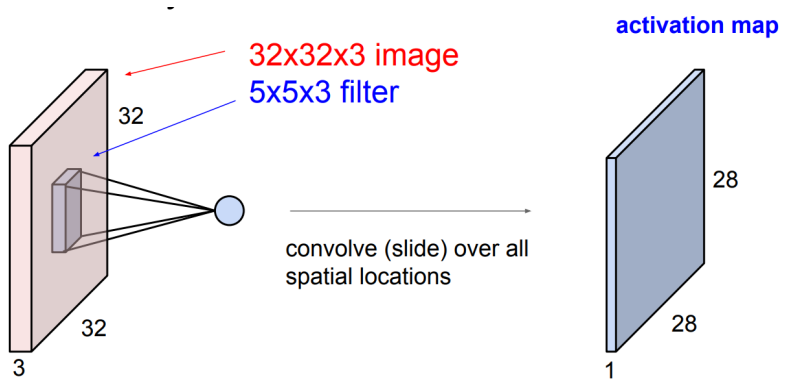
아래처럼 작은 조각들이 점점 합쳐져서 더 큰 조각들이 되는 것을 확인할 수 있다.

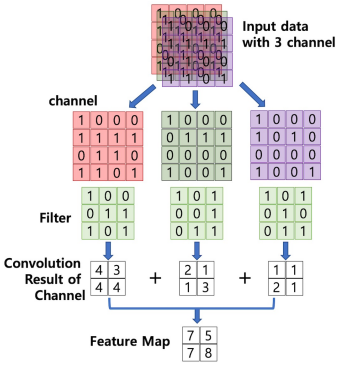
→ 포인트는 **계층에 따라 다른 복잡도의 특징을 학습**한다는 것이다.

🤔 Q. 자료에서 Conv 이미지는 무엇을 의미하나요?

A. 자료의 각 그리드는 하나의 뉴런입니다. 그리고 그리드 속 이미지는 해당 뉴런을 최대로 활성화시킬 수 있는 이미지를 나타냅니다. → 그런 의미에서 뉴런이 어떻게 생겼는지를 의미하기도 합니다.

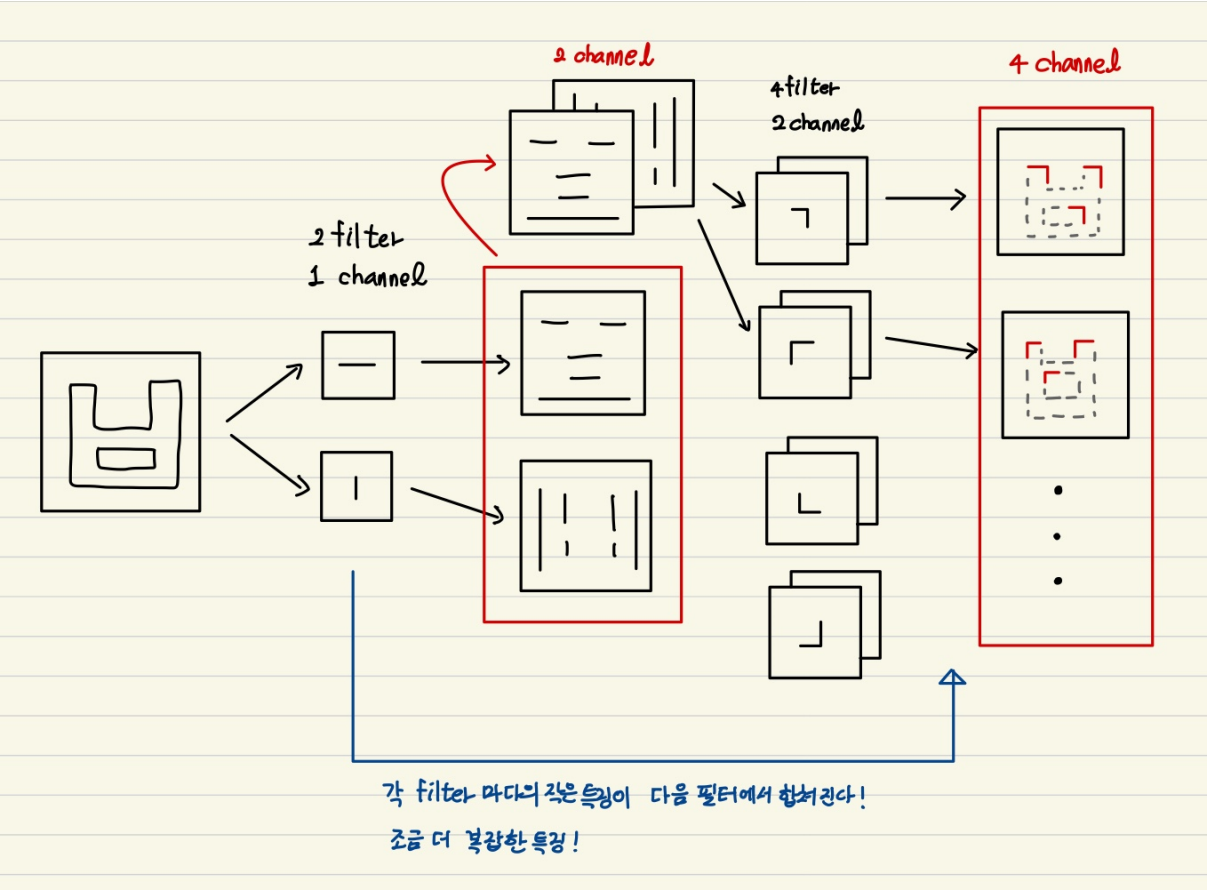
이러한 생각에 대한 근거는 아래와 같다. (채널 간 연산은 합!)





합성곱 신경망은 입력 데이터의 채널 깊이를 하나로 모은다. 이렇게 모인 하나의 채널에 이전 단계 채널 속 특징이 결합해 있다.

직접 해보면 요런 느낌이다!



형태는 저런 식으로 반영하면 될 것 같은데 RGB채널은 조금 고민이다.

1. RGB 역시 하나의 필터 속 여러 채널들의 값이 다르기 때문에 이 값들의 관계를 통해 파악할 수 있다.
2. 개인적인 내 아이디어 : 필터를 아예 따로 분리해서 계속 처리하면 되지 않을까?

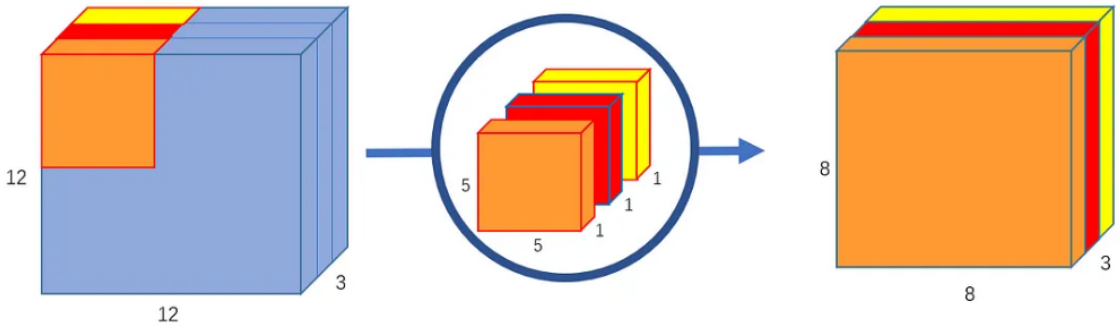


Image 6: Depthwise convolution, uses 3 kernels to transform a 12×12×3 image to a 8×8×3 image

관련논문 : https://arxiv.org/pdf/1610.02357

그 외에 궁금한 점

1. 패딩은 피처맵의 크기를 유지한다고 하는데, 그 외에 이미지 가장자리의 데이터를 더 디테일하게 표현해주는 기능을 한다고 생각한다.
2. 그런데 패딩으로 0을 계속 사용하면 가장자리의 데이터값이 점점 작아지는 문제가 발생한다고 생각한다. → 예를 들어 가장자리가 255에 가까운 값인데 0과 가중합이 되면서 점점 작아지는 문제가 발생한다고 생각한다. → 이를 통해 경계 인식 가능?? 일단 보류
3. 스트라이드를 2로 키우면 데이터를 더 불연속적으로 추출한다고 생각한다. (조금 다른 방식으로도 생각 가능, 가중치 개수)

   → 디테일한 경계를 조금 더 듬성듬성 추출한다. 불필요할 정도로 디테일하게 추출하는 것 방지
4. 풀링 역시 데이터를 더 불연속적으로 만든다고 생각한다.

   → 풀링은 결국 중요한 정보 외의 정보를 없애는 것이다. 구체적으로 표현하면 위치정보를 없애는 것이다. 객체에만 집중해야 할 때 사용!
5. 풀링에도 스트라이드를 1로 사용하면 연속적인 압축이 가능하지 않을까?

   가능하지만 풀링은 단순히 feature map의 크기를 유지시키는 용도로만 사용하는 것은 아니라서 풀링 대용으로 압축한다는 관점을 다른 이야기이다.
6. 풀링을 사용하면 정보 손실이 발생한다고 생각한다. 그럴 바에 패딩을 사용하지 않고, 크기를 줄이는게 더 좋지 않을까?

   → 의도적으로 정보 손실을 일으키는 것!! 위치정보를 없애는 것!!

정채현

활동현황	완료

▼ 1. AI - ML - DL

▼ 1.1. 인공지능 (Artificial Intelligence, AI)

- **사람처럼 학습하고 추론**할 수 있는 능력을 가진 컴퓨터 시스템 기술

1. **좁은 인공지능 (Narrow AI) =약인공지능(weak AI)**

   : 특정 작업에 특화된 AI (ex. 스팸 필터링, 음성 인식)

2. **강한 인공지능 (General AI)**

   : 인간처럼 모든 문제를 해결할 수 있는 일반적인 지능

3. **초인공지능 (Super AI)**

   : 인간 지능을 초월한 수준(이론적 단계)

▼ 1.2. 기계 학습 (Machine Learning, ML)

- 규칙을 명시적으로 프로그래밍하지 않아도, **자동으로 데이터에서 규칙을 학습**하는 알고리즘을 설계하는 분야

1. **지도 학습 (Supervised Learning)**

   : 입력(X)-출력(Y) 쌍이 있는 데이터를 사용 (ex. 주택 가격 예측).

2. **비지도 학습 (Unsupervised Learning)**

   : 레이블이 없는 데이터를 사용하여 패턴을 탐색 (ex. clustering)

3. **강화 학습 (Reinforcement Learning)**

   : 보상을 통해 최적의 행동을 학습 (ex: 알파고)

▼ 1.3. 딥러닝 (Deep Learning, DL)

- ML 중에서 **인공 신경망(Artificial Neural Networks)**을 사용하여 데이터를 학습하는 알고리즘

▼ 1.3.1. 인공 신경망 (Artificial Neural Network , ANN)

- 인간 뇌의 뉴런 구조를 모방한 계산 모델

- **뉴런** : 입력을 받아 가중치와 활성화 함수로 출력 결정.
- **레이어** : 입력층(Input Layer), 은닉층(Hidden Layer), 출력층(Output Layer)으로 구성.
- **가중치 학습** : 역전파(Backpropagation)를 통해 가중치 조정.

▼ 1.3.2. 퍼셉트론(Perceptron)

- **신경망의 가장 기본적인 형태**로, 인공 뉴런(Neural Unit) 하나로 구성
- **단점** : **선형적으로 구분 가능한 문제**만 해결 가능 → 다층 퍼셉트론 개발

▼ 수식

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

- $\sum_{i=1}^{n} w_i x_i + b$
  : 입력값(x)과 가중치(w)의 선형 결합
- **b**

  : bias -모델의 유연성을 높임.
- **f( )**

: **활성화 함수 -** 결과를 특정 범위로 매핑

▼ 1.3.3. 다층 퍼셉트론(MLP, Multi-Layer Perceptron)

- **여러 개의 퍼셉트론을 연결**하여 입력층(Input Layer), 은닉층(Hidden Layer), 출력층(Output Layer)으로 구성된 신경망
- 딥러닝의 가장 기본적인 형태

▼ 수식

$$h(1) = f(W^{(1)}x + b^{(1)})$$
$$h(2) = f(W^{(2)}h^{(1)} + b^{(2)})$$
$$y = f(W^{(3)}h^{(2)} + b^{(3)})$$

▼ 학습과정

1. **순전파(Forward Propagation)**

   : 입력값이 각 층을 거치며 계산되고, 최종 출력값을 생성

2. **손실 함수(Loss Function)**

   : 예측값과 실제값 사이의 오차를 계산

   ex. MSE(Mean Squared Error), 교차 엔트로피(Cross-Entropy Loss)

3. **역전파(Backpropagation)**

   : 출력층에서 시작해 은닉층으로 오차를 전파하며 가중치와 바이어스를 업데이트

   : 기울기(Gradient)를 계산하여 최적화 알고리즘(SGD, Adam 등)으로 가중치를 조정.

▼ 1.3.4. 주요 알고리즘

1. **CNN (Convolutional Neural Networks)**
   - 주로 **이미지 데이터**를 처리하고 분석하는 데 특화

- 입력 데이터의 **공간적 구조(Spatial Structure)**를 유지하면서 특징을 자동으로 추출

2. **RNN (Recurrent Neural Networks)**
   - **순차적 데이터(Sequence Data)**를 처리하는 데 적합
   - 이전 데이터의 정보를 기억하여 다음 데이터 처리에 활용
     →
     시간 종속성(Time Dependency) 고려
3. **GAN (Generative Adversarial Networks)**
   - **생성자(Generator)**와 **판별자(Discriminator)**라는 두 신경망으로 구성
   - **생성자 :** 랜덤한 노이즈 벡터를 입력받아 실제 데이터와 유사한 가짜 데이터 생성
   - **판별자** : 입력 데이터가 실제 데이터인지 생성된 가짜 데이터인지 구별
4. **Transformer**
   - **순차적 데이터를 병렬 처리**하며, **자연어 처리(NLP)**에 특히 강력한 모델

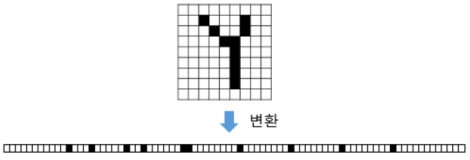
▼ 2. 합성곱신경망 (Convolutional Neural Network, CNN)

- **공간적 구조 보존** : 이미지의 **픽셀 간 관계**를 유지하면서 처리.

  ↔ MLP : 이미지를 1차원 벡터로 변환.
- **지역적 연결성** : 필터(커널)를 통해 이미지의 작은 영역만을 학습.

  → Fully connected 구조에서 유리함.

  ↔ MLP : 1차원 벡터로 변환하면서 데이터가 매우 큼. (28x28 → 784개)
- **파라미터 공유** : 동일한 필터를 이미지 전반에 적용하여 계산 효율성 증가.



공간적 구조 유실

▼ 2.1. CNN 구성 요소

▼ 2.1.1. 입력층 (Input Layer)

- 이미지를 **3D 텐서** 형태로 모델에 입력

  ▼ **텐서 (Tensor)**

  - 데이터의 구조를 표현하는 기본 단위
  - 스칼라(0차원), 벡터(1차원), 행렬(2차원)을 일반화한 개념

  - 이미지 데이터의 3D tensor
    1. **높이 (Height, H)** : 이미지의 세로 픽셀 수
    2. **너비 (Width, W)** : 이미지의 가로 픽셀 수
    3. **채널 (Channel, C)** : 색상 또는 특징 맵의 수
       a. 흑백 이미지 (Grayscale) : 1채널
       b. 컬러 이미지 (RGB) : 3채널

  ex. `224x224x3` 크기의 RGB 이미지

▼ 2.1.2. 컨볼루션 층 (합성곱 층, Convolutional Layer)

- 이미지를 **작은 영역별로 처리**하여 특징(feature)을 추출.

1. **필터(filter, 커널, Kernel)**

   : 작은 크기의 행렬을 사용해 입력 이미지의 **특정 패턴을 감지**

   - **엣지, 텍스처, 코너** 등
   - 필터는 이미지의 모든 위치에 대해 슬라이딩하면서 연산 수행

2. **합성곱 연산(convolution)**

   : 필터와 입력 데이터 간의 **내적(dot product)**을 계산하여 출력값 생성

3. **특징 맵 (Feature Map)**

   : 필터가 이미지의 특정 특징을 감지한 결과값

   → 필터가 여러 개일 경우, 각 필터는 독립적인 특징 맵을 생성

4. **패딩 (Padding)**

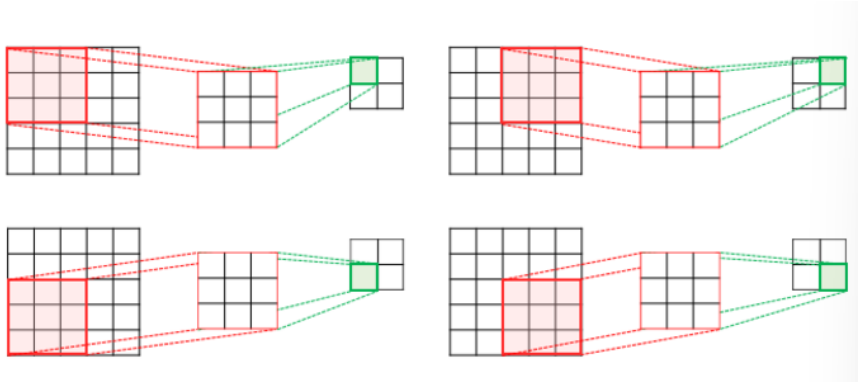
   : 입력 데이터 가장자리에 **0을 추가**하여 합성곱 연산 후, **출력 크기를 조정**하거나 **가장자리 정보를 보존**

   ∵ 합성곱 연산의 결과로 얻은 특성 맵은 입력보다 크기가 작아지므로

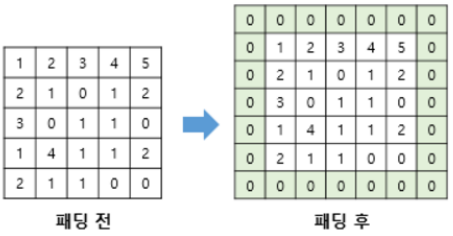
   - **Valid Padding**: 패딩 없이 연산.
   - **Same Padding**: 출력 크기가 입력 크기와 동일.

5. **스트라이드 (Stride)**

   : 필터가 이동하는 간격



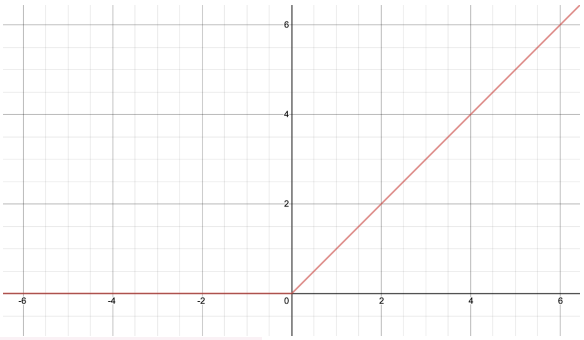
트라이드가 2일 경우에 5 × 5 이미지에 합성곱 연산을 수행하는 3 × 3 커널의 움직임
→ 2 × 2의 크기의 특성 맵



▼ 2.1.3. 활성화 함수 (Activation Function)

- 합성곱 연산 결과에 **비선형성을 부여**하여 **복잡한 패턴을 학습**할 수 있게 만듦.

- **ReLU (Rectified Linear Unit)**:
  - `f(x) = max(0, x)`
  - 음수를 0으로 바꾸고 양수는 그대로 유지.
  - **장점**: 기울기 소실 문제 완화, 연산 간단.



▼ 2.1.4. 풀링 층 (Pooling Layer)

- 이미지의 중요한 특징만 남기고, 덜 중요한 정보를 제거함으로써, 입력 데이터(특성 맵)의 **크기를 축소**

  → 모델의 **계산량 감소**

  → **오버피팅 방지 :** 불변성(Translation Invariance) = 이미지의 이동, 크기 변화, 왜곡에 대해 모델이 강건해짐.

1. **맥스 풀링 (Max Pooling)**

   : 작은 영역에서 **가장 큰 값**을 선택.

   ex. `[[1, 3], [2, 4]] → max = 4`

   → 영역 내에서 가장 중요한 값을 선택하므로, **엣지와 같은 강한 특징**을 더욱 강조

2. **평균 풀링 (Average Pooling)**

   : 작은 영역의 **평균값**을 선택.

   ex. `[[1, 3], [2, 4]] → avg = (1+3+2+4)/4 = 2.5`

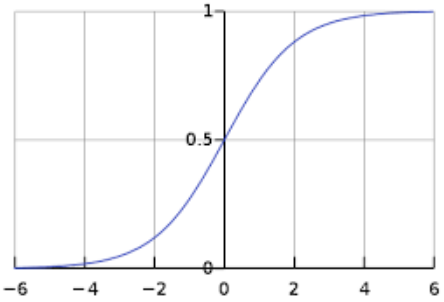
→ 전체적인 **데이터 분포의 요약**에 적합
↔ 중요한 특징이 희석될 수 있음. 노이즈도 함께 평균에 포함될 수 있음.

3. **스트라이드**

   : 필터가 풀링 연산을 수행할 때 이동 간격.

▼ 2.1.5. 완전 연결 층 (Fully Connected Layer, FC Layer)

- 이전 층에서 학습한 특징을 사용하여 **최종적으로 분류(Classification)**를 수행
- 모든 노드가 서로 연결되어 있으며, **Softmax 함수**를 통해 클래스 확률을 출력



▼ 2.1.6. 드롭아웃 (Dropout)

- 뉴런의 일부를 임의로 비활성화하여 **오버피팅 방지**.
- 학습 시 활성화되는 뉴런의 비율을 설정

▼ 2.2. CNN의 학습 방법

- **역전파 (Backpropagation)**
  - 출력층에서 발생한 **오차(error)**를 컨볼루션 층까지 전달하여 가중치 업데이트.
- **손실 함수 (Loss Function)**
  - 학습 모델의 예측값과 실제값 간의 차이를 계산.
  - 예: **교차 엔트로피 손실 (Cross-Entropy Loss)**.
- **최적화 알고리즘**

- **SGD (Stochastic Gradient Descent)**, **Adam** 등을 사용해 가중치 최적화.

▼ 궁금) Kernel - Stride - Pooling 차이

- **왜 커널을 사용하는가?**
  - **국소적 특징 학습**

    : 이미지의 작은 영역을 스캔하여 엣지, 텍스처, 모양 등 저수준 특징을 학습
  - **계층적 특징 추출**

    : 여러 층을 통해 저수준 특징을 조합하여 고수준 특징(객체의 모양, 윤곽)을 학습

- **왜 스트라이드를 사용하는가?**
  - 계산량 감소
  - 특성 맵의 크기를 줄여 메모리 사용량을 최적화

- **왜 풀링을 사용하는가?**
  - 잡음 억제
  - 데이터 압축
  - 위치 변화에 강건

- **왜 각각을 사용하는가?**
  - **커널**
    - 입력 데이터에서 특징을 학습.
    - 합성곱 층에서 **데이터를 이해하는 가장 중요한 도구**.
  - **스트라이드**
    - 계산량을 줄이고, 데이터의 크기를 제어.
    - 합성곱 연산의 효율성을 높이기 위한 도구.

- **풀링**
  - 중요한 정보를 유지하면서 데이터 크기를 줄이고, 모델의 **강건성 (robustness)**을 높임.

⇒ 커널과 스트라이드는 함께 작동하여 **특성 맵 생성과 크기 조정**을 수행

⇒ 풀링은 합성곱 층의 출력에서 **정보를 요약**하고 **잡음을 줄이는 보조적인 역할**

최종렬

활동현황 완료

패딩이 필요한 이유

1.입력값과 출력값의 크기를 같게 하기 위해서

—> 세임 패딩

컨볼루션 연산 수행시 입력 값의 크기가 되는kernel 즉 가중치에 의해 줄어들 수 있다. 이때 패딩을 사용하지 않으면 출력 값이 점점 작아지고 정보의 손실이 온다

패딩을 추가하여 입력값과 출력값의 크기를 동일하게 유지함으로써 데이터가 온전히 보존될 수 있다

그리고 컨볼루션 연산에서 가장자리의 데이터가 연산에 적게 참여하게 되면서 가장자리의 정보를 과소평가 하게 되는 문제가 발생할 수 있다. 가장 **자리의 정보는 중요한 정보를 많이 포함(모든 데이터가 그런건 아니다)**하고 있는데 이것이 출력값에 충분히 반영되게 하기 위해서 아래 그림과 같이 입력값의 가장자리를 둘러싸는 형식으로 0을 추가한다

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

이렇게 되면 합성곱 연산에 가장 자리의 데이터가 기존 보다 더 많이 참여 즉 다른 데이터와 균등하게 반영 되면서 가장 자리의 정보를 출력값에 더 잘 반영할 수 있게 된다.

Q2. Kernel의 가중치를 높게 설정하지 않고 굳이 패딩을 사용하는 이유

CNN의 핵심은 데이터로 부터 가중치를 학습하는 것 —> 이는 이미지 데이터를 학습하면서 최적화됨

그런데 가중치를 임의로 높게 설정하게 되면 이미지 데이터에 따라 충분히 학습되지 못해 가중치의 변화가 제한됨

그래서 편향된 값이 나와 모델 성능이 낮아질 수있다

그리고 가장자리 정보의 중요성은 데이터 마다 다르다

모든 데이터가 항상 가장자리의 값이 가장 중요한 것은 아니다.

가장 자리가 아닌 중심부의 정보가 더 중요할 수도 있다 가중치를 가장자리만 높게 하면 어떤 데이터에는 적절할 수도 있지만 특정 데이터에는 잘못된 결과를 낼 수도 있다

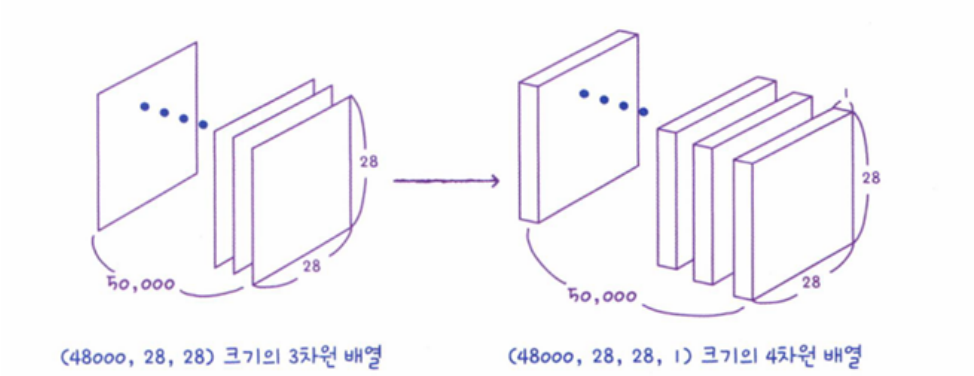
따라서 모든 데이터를 균등하게 처리해서 연산되록 해야한다

8-2 합성곱 신경망을 사용한 이미지 분류

입력 이미지는 항상 깊이(채널)이 있어야 한다. 흑백 이미지의 경우 채널 차원이 없는 2차원 배열이지만 Conv2D 층을 사용하기 위해 마지막에 이 채널 차원을 추가해야 한다.

→ 넘파이 reshape() 메서드를 통해 전체 배열 차원을 그대로 유지하면서 마지막에 차원을 간단하게 추가할 수 있다.

요약하자면 입력 이미지를 cnn이 처리할 수 있는 형식으로 만들기 위해 색 정보를 가진 하나의 차원을 추가하는 것



합성곱 신경망 만들기

손코딩
```
model = keras.Sequential()
model.add(keras.layers.Conv2D(32, kernel_size=3, activation='relu',
          padding='same', input_shape=(28,28,1)))
```

이 과정은 cnn의 첫 번째 합성곱 계층을 정의하는 부분이다.

(합성곱 계층 = cnn에서 가장 기본적인 구성 요소로, 이미지 데이터의 특징을 추출)

Conv2D는 3차원 입력 데이터를 받습니다:

$$\text{입력 형태: (높이, 너비, 채널)}$$

- 예: $28 \times 28 \times 1$ (흑백 이미지), $64 \times 64 \times 3$ (컬러 이미지)

model = keras.Sequential( )

keras의 Sequential api를 활용하여 계층을 차례대로 쌓아 신경망 구조를 구성

model.add(keras.layers.Conv2D(32, kernel_size=3, activation="relu", padding="same", input_shape=(28, 28, 1)))

cnn에서 합성곱 신경망 계층을 추가한다. 이 계층은 입력 이미지에서 특징을 추출한다.

32

합성곱 계층에서 사용할 필터(커널)의 개수이다

결과적으로 출력은 32개의 특성 맵을 생성한다.

kernel_size = 3

필터의 크기를 지정

여기서는 3x3 크기의 필터를 사용

필터는 3x3 영역에서 이미지의 특징을 추출한다

activation="relu”

활성화 함수로 렐루를 사용

렐루는 음수를 0으로 만들고, 양수는 그대로 반환한다. 이는 신경망의 비선형성을 추가하여 복잡한 패턴을 학습하게 한다.

padding = same

패딩 방식을 설정, 입력 이미지와 출력 이미지의 크기를 동일하게 유지한다. 필요시 가장자리에 0을 추가한다.

valid 패딩을 사용하지 않으며, 출력 크기가 입력보다 작아진다.

input_shape=(28, 28, 1)

모델에 입력될 데이터의 형태를 지정한다

28×28: 이미지의 높이와 너비

1: 채널 수(흑백이미지는 1, 컬러이미지는 3)

model.add(keras.layers.MaxPooling2D(2))

최대 풀링 계층을 모델에 추가하는 것을 의미한다. 최대 풀링은 입력 데이터의 크기를 줄이면서 중요한 특징을 추출하고 계산량을 줄인다.

최대 풀링 주요 목적

1. 데이터의 크기 축소
2. 중요한 특징 유지(최대값만 남김)

model = keras.Sequential()
model.add(keras.layers.Conv2D(64,kernel_size=3,activation="relu",padding="same",input_shape=(28,28,1)))

필터의 갯수를 64개로 늘린 것이다.

model • add(keras • layers • FlattenO)
model.add(keras.layers.Dense(100, activation=' relu'))
model•add(keras•layers•Dropout(0•4))
model.add(keras•layers.Dense(10, activation='softmax'))

이 코드는 CNN 모델의
특징 맵을 일렬로 펼친 후(Flatten), 이를 밀집층(Dense Layer)을 통해 처리하고 최종적으로 **출력층**에서 분류 결과를 예측하는 과정을 정의

flatten이 왜 필요할까?

Dense Layer의 입력 요구사항 때문이다. 밀집층 즉 dense layer는 벡터 형태의 데이터를 입력으로 받는다. cnn은 3d 형태 이므로 1d 벡터로 변환해야 밀집층과 연결할 수 있다.

flatten이 없으면 conv2d 계층과 Dense 계층은 서로 연결될 수 없다. dense계층은 1차원 입력을 요구하기 때문에, 3차원 데이터를 전달하면 오류가 발생한다.

model.add(keras.layers.Dense(100, activation='relu'))

dense 은닉층(중간 은닉층)

특성 맵을 일렬로 펼친 후, 이 데이터를 100개의 뉴런을 가진 은닉층에 전달

여기서 은닉층이란

1. 입력 데이터를 가공하고 숨겨진 패턴을 찾는 곳
2. 결과를 만들기 위한 중간 과정3
3. 쉽게 말해, 뭔가를 생각하고 결론을 도출하는 두뇌 역할을 한다.

model.add(keras.layers.Dropout(0.4))

drop out 즉 과적합 방지이다. 훈련 중 무작위로 일부 뉴런을 비활성화(0으로 설정)하여 과적합을 방지. 여기서 0.4는 드롭아웃 비율로 40퍼의 뉴런이 훈련 중 비활성화

model.add(keras.layers.Dense(10, activation='softmax'))

dense(최종 출력층)에 대한 코드다. 여기서 10은 최종적으로 10개의 뉴런을 가지는 출력층을 정의한 것. 각 뉴런은 데이터가 특정 클래스에 속할 확률을 나타낸다.

Softmax 활성화 함수

iii번째 출력 뉴런의 값을 확률로 변환하여, 모든 출력값의 합이 1이 되도록 합니다.

11

예: 이미지가 숫자 7일 확률이 0.8, 숫자 3일 확률이 0.15 등으로 계산.

0.80.8

0.150.15

전체 프로세스 흐름

1. conv2D + pooling: cnn계층에서 이미지 특징을 추출
2. flatten: cnn 계층의 3차원 출력을 1차원 벡터로 전환
3. dense은닉층
4. dropout: 과적합 방지를 위해 일부 뉴런 비활성화
5. dense 출력층: 입력 데이터를 10개의 클래스에 대한 확률로 변환, 활성화 함수를 통해 softmax를 사용해 각 클래스에 속할 확률 계산

파라미터의 수 계산

1. **Dense Layer**:

$$(\text{입력 뉴런 수} \times \text{출력 뉴런 수}) + \text{출력 뉴런 수}$$

2. **Conv2D Layer**:

$$(\text{커널 높이} \times \text{커널 너비} \times \text{입력 채널 수}) \times \text{필터 개수} + \text{필터 개수}$$

Dense Layer 파라미터 공식

$$\text{파라미터 개수} = (\text{입력 뉴런 수} \times \text{출력 뉴런 수}) + \text{출력 뉴런 수}$$

풀링 계층의 파라미터가 없는 이유

풀링은 단순히 입력 데이터의 최대값이나 평균값을 계산하는 연산이기 때문에 학습할 가중치나 편향이 없다.

한서연

활동현황	완료

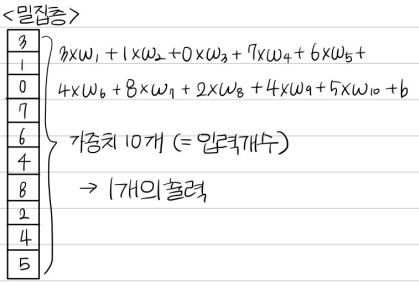
CNN

8-1. 합성곱 신경망의 구성요소

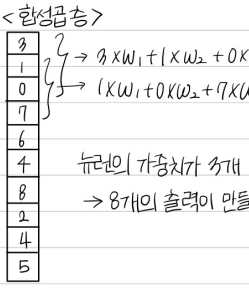
합성곱 convolution

↳ 입력 data에 마법의 도장을 찍어서 유용한 특성만 드러나게 하는것

<밀집층>



<합성곱층>

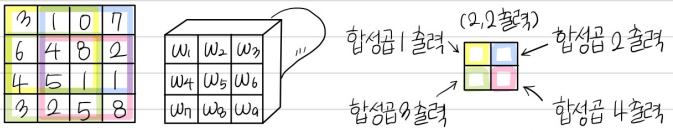


합성곱 신경망(convolutional neural network)에서는 완전연결 신경망과 달리 뉴런을 필터라고 부른다. (뉴런 = 필터 = 커널)

뉴런개수 ← 입력에 곱해지는 가중치

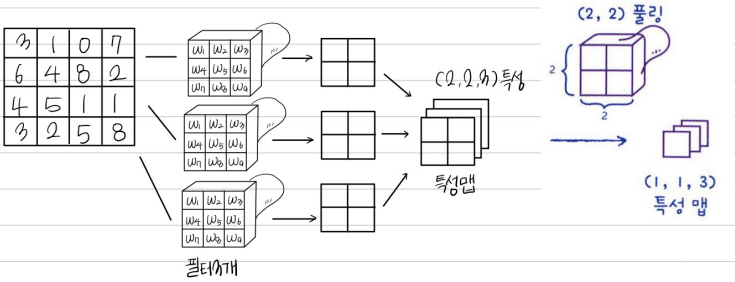
↳ 정의: 1개 이상의 합성곱 층을 쓴 인공신경망

클래스에 대한 확률을 계산하려면 마지막 층에 클래스 개수만큼의 뉴런을 가진 밀집층을 둠

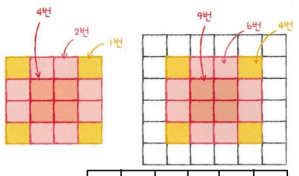


1 → 2 → 3 → 4 : 필터는 모두 4번 움직여 4개 출력을 만듦

특성맵(feature map) : 입력에 놓인 위치에 맞게 2차원으로 배치.



<원소가 커널과 계산되는 횟수>



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

패딩과 스트라이드

Q. 커널 크기는 (3,3)으로 그대로 두고 출력의 크기를 입력과 동일하게 (4,4)로 만들려면?

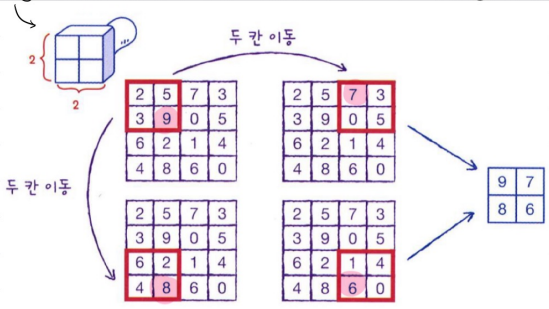
패딩 : (4,4) 입력 배열 주위를 가상의 원소로 채움 ← 세임패딩: 입력주위에 0으로 패딩

밸리드 패딩: 패딩 없이 합성곱 하는것.

스트라이드 : 합성곱 연산중 이동의 크기. 디폴트 = 1

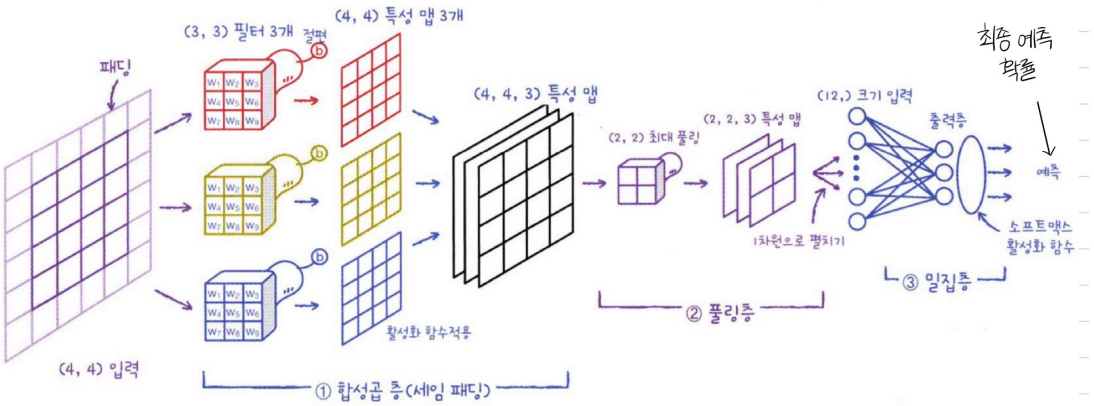
풀링 : 합성곱 층에서 만든 특성맵의 가로세로 크기를 줄임. ← 최대풀링

평균풀링 : 중요한정보를 (평균하여) 희석시킬수 있음



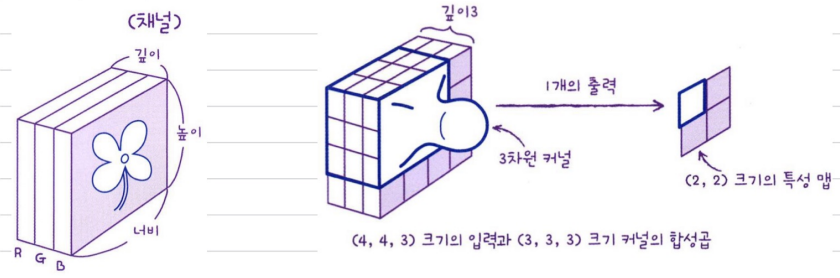
특성맵이 여러개라면 동일한 작업을 반복

합성곱 신경망의 전체 구조



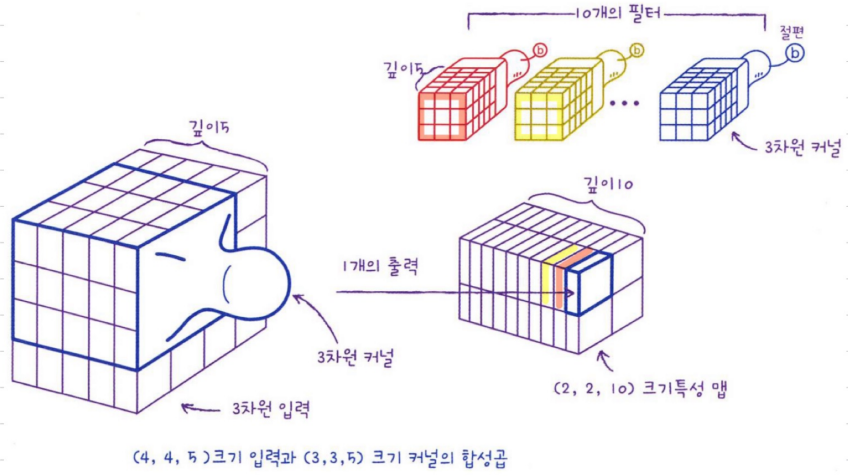
합성곱에서 스트라이드를 크게 해서 특성맵을 줄이는 것보다 풀링층에서 크기를 줄이는 것이 경험적으로 더 나은 성능을 냄.

컬러 이미지를 사용한 합성곱



입력: (4,4,3)
필터의 커널크기(도장): (3,3,3)
⇒ 커널 배열깊이 = 입력깊이

(3,3,3) 영역에 해당하는 27개 원소에 27개 가중치를 곱하고 절편을 구하는 식.
* 입력이나 필터의 차원이 몇개인지 상관없이 항상 출력은 하나의 값.



이렇게 합성곱 신경망은 너비와 높이는 점점 줄어들고 깊이는 점점 깊어지는게 특징.
↓
마지막에 출력층 전에 특성맵을 모두 펼쳐서 밀집층의 입력으로 사용
↓
필터 : 이미지의 어떤 특징을 찾는다.
기본적 특징(직선, 곡선 등)
→ 층이 깊어질수록 구체적 특징을 감지하도록 필터개수↑

어떤 특징이 이미지의 어느 위치에 놓이더라도 쉽게 감지할 수 있도록 너비와 높이차원을 줄여감.

1. 기존 신경망에서 벗어나 CNN을 사용한 이유는 무엇인가요?

기존 신경망(MLP)은 모든 뉴런이 서로 연결된 Fully Connected Layer를 사용합니다. 이 방식은:

- 이미지 데이터의 공간적 구조(위치, 패턴)를 무시.

- 고차원 데이터(예: 이미지)의 경우 계산량이 기하급수적으로 증가.

CNN은 지역적 연결(local connectivity)과 가중치 공유를 활용하여 계산량을 줄이고, 이미지의 공간적/위치적 특징을 효과적으로 학습합니다. 특히, 특징 추출에 강점이 있어 이미지 처리에서 탁월한 성능을 발휘합니다.

11/19스터디에서 추가: 구조가 다르다는건 바라보는 관점이다른것, 가중치 개수가 많아지면 복잡한것을 나타낼수있다

2. CNN에서 합성곱 신경망은 어떤 역할을 하나요?

합성곱(CNN의 핵심 연산)은 특징 추출기 역할을 한다.

- 입력 이미지에서 특정 패턴(예: 경계선, 텍스처, 색상 등)을 탐지.

- 하위 계층에서는 간단한 특징(선, 모서리)을, 상위 계층에서는 복잡한 특징(객체)을 학습.

- 공간 구조 유지: 이미지의 위치 정보를 보존하며, 패턴을 탐지하는 데 효과적

3. 풀링은 어떤 역할을 하나요?

- 차원 축소: 데이터 크기를 줄여 계산량을 감소시킴.

- 특징 요약: 중요한 정보만 남기고, 세부적인 노이즈를 제거.

- 위치 불변성: 특징이 이미지 내의 다른 위치로 이동해도 동일하게 인식하도록 학습.

4. 패딩으로 0을 사용하면 어떤 현상이 발생할까요?

- 경계 정보 보존: 입력 이미지의 경계 부분 특징이 손실되지 않도록 방지.

- 크기 유지: 패딩을 추가하지 않으면 합성곱 연산으로 출력 크기가 작아진다. 이를 방지하기 위해 패딩을 사용.

- 하지만 0 패딩은 실제 데이터가 아닌 값(0)을 추가하기 때문에, 경계 특징이 왜곡될 가능성도 있다.

경계값을 그대로 복제해서 사용할수도 있는데 왜 0을 추가하는지?

- CNN의 패딩에서 0을 추가하는 이유:

수학적 단순화: 0으로 패딩하면 값이 추가되지만, 계산 과정에서는 추가된 값이 영향을 미치지 않으므로 연산이 간단해짐

0이 아닌 값을 패딩으로 사용하면 모델의 복잡성이 증가할 수 있다.

- 결론적으로, CNN에서 0 패딩은 계산 단순화와 데이터 보존 측면에서 가장 효율적인 선택으로 사용.

엣지(가장자리)에 중요한 정보가 있는 이미지라면, 0이 아닌 다른 값을 패딩에 사용할 수도 있다.

예)

1. 주변 픽셀 값을 기반으로 한 패딩:

- 반사 패딩 (Reflect Padding): 이미지 가장자리의 픽셀 값을 반사하여 패딩 영역을 채움.

장점: 가장자리 패턴을 유지하여 자연스러운 경계 처리가 가능.

- 복사 패딩 (Replicate Padding): 가장자리 픽셀 값을 복사하여 패딩 영역을 채움.

장점: 가장자리의 정보 손실이 줄어듦.

2. 평균값 기반 패딩: 이미지의 전체 픽셀 값 평균이나 특정 채널의 평균값으로 패딩 영역을 채움.

장점: 0으로 패딩할 때 생길 수 있는 급격한 경계 변화 문제를 완화.

3. 고정값 패딩: 모든 패딩 값을 동일한 상수로 설정 (예: 255(흰색), -1 등).

- 예: 흰색 배경이 많은 이미지(예: 문서 스캔 이미지)에서는 255로 패딩.

장점: 도메인 지식 활용.

4. 학습 기반 패딩: 패딩 값을 단순히 고정하지 않고, 모델이 데이터에서 가장 적합한 패딩 값을 학습하도록 설계.

- 예: 패딩을 생성하는 네트워크(GAN 또는 Autoencoder)를 사용.

주의할 점:

0이 아닌 값으로 패딩을 설정하면 연산량과 모델의 복잡성이 약간 증가할 수 있다.

잘못된 패딩 값을 선택하면 모델이 왜곡된 정보를 학습할 가능성이 있으므로 데이터 특성에 맞는 패딩 방법을 선택해야 함.

5. 스트라이드를 1과 2로 사용하는 것의 차이는 어떻게 생각해볼 수 있을까요?

- 스트라이드 1: 필터가 한 칸씩 이동하여, 출력 크기가 입력 크기에 비례해 큼,

세밀한 특징 학습 가능, 계산량 증가.

- 스트라이드 2: 필터가 두 칸씩 이동하여, 출력 크기가 줄어듭니다, 계산량 감소, 더 큰 범위의 특징을 추출하지만, 세부 정보가 손실될 가능성.

6. 왜 평균풀링보다 맥스풀링을 많이 사용할까요?

맥스풀링의 장점:

- 더 중요한 특징(가장 강한 활성화)을 강조.

- 노이즈에 덜 민감.

평균풀링의 단점:

- 노이즈가 평균 값에 포함되어 정보가 왜곡될 가능성.

- 특징이 희석되어 분류 성능 저하 가능.

따라서, CNN에서는 맥스풀링이 더 많이 사용

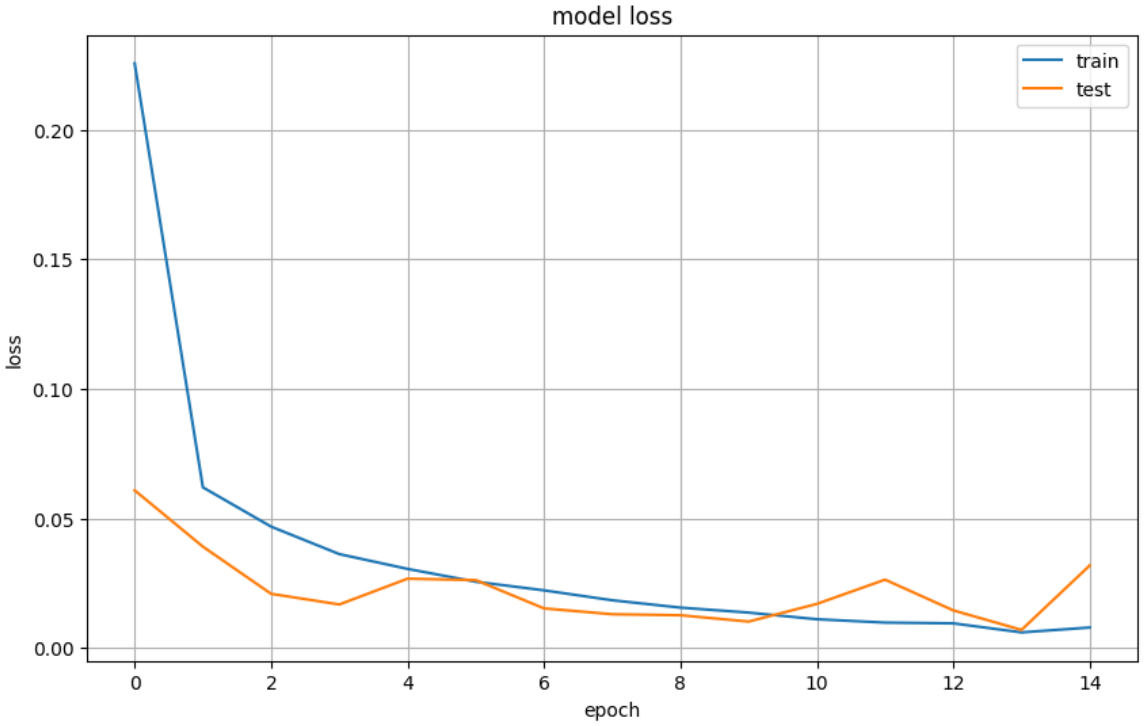
남재현

태그	완료
상태	Conv Size

구조 변경 결과

▼ Kernel size = 3 X 3

- 작고 간단한 Kernel크기로 세밀한 특징을 추출할 수 있음
- 상대적으로 적은 연산량으로 학습이 더 안정적으로 이루어질 가능성이 높음
- MNIST는 작은 데이터셋으로 작은 커널이 더 적합할 수 있음



Accuracy: 0.986799955368042

▼ Kernel size = 5 X 5

- 커널의 크기가 커질수록 더 넓은 영역의 정보를 캡처하고 세밀한 특징 추출이 줄어듦

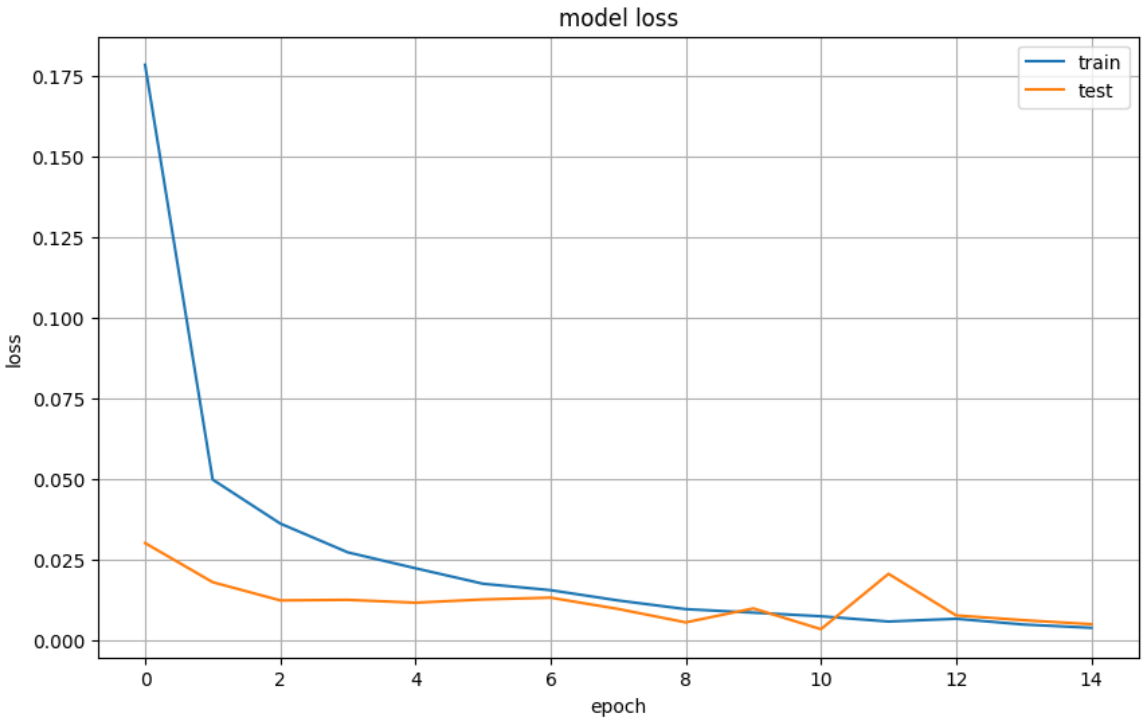
- 3 X 3에 비해 더 많은 파라미터를 학습하므로 모델의 표현력이 높아짐

▼ 결과 분석

3 X 3 Kernel에 비해 높은 정확도 = 우연의 일치일 가능성이 높다

일반적으로 간단한 데이터 셋은 작은 Kernel 크기를 요구함

단, 이번 데이터 셋에서는 5 X 5가 더 적합했던 것



Accuracy: 0.9905999898910522

▼ Kernel size = 7 X 7

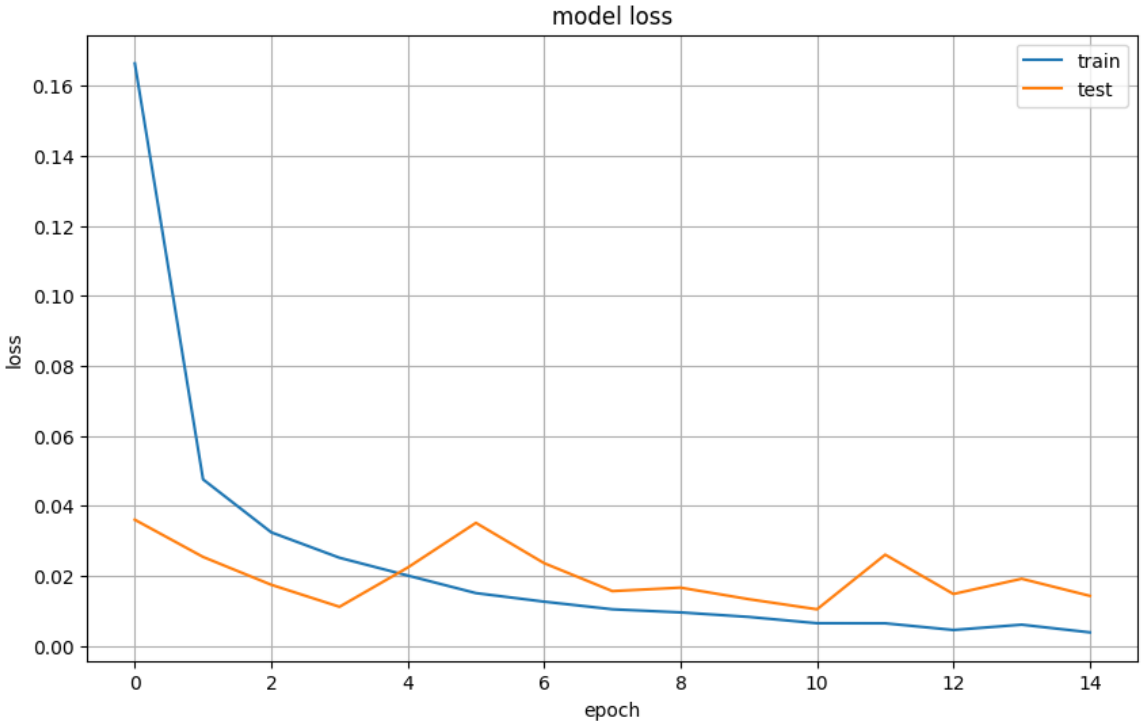
▼ 결과 분석

마찬가지로 5 X 5보다 더 큰 크기의 패턴을 학습함

이 경우는 충분히 넓은 영역의 특징을 학습하면서도 적당한 일반화 수준을 이룬 것

이때 4~6, 10~12 epoch 구간에서 test loss의 일시적인 상승이 관측된. 이는 학습 중 모델이

특정epoch 동안 데이터에 약간의 과적합이 되었지만 이후 안정적으로 수렴했다는 증거



Accuracy: 0.9922999739646912

▼ Kernel size = 9 X 9

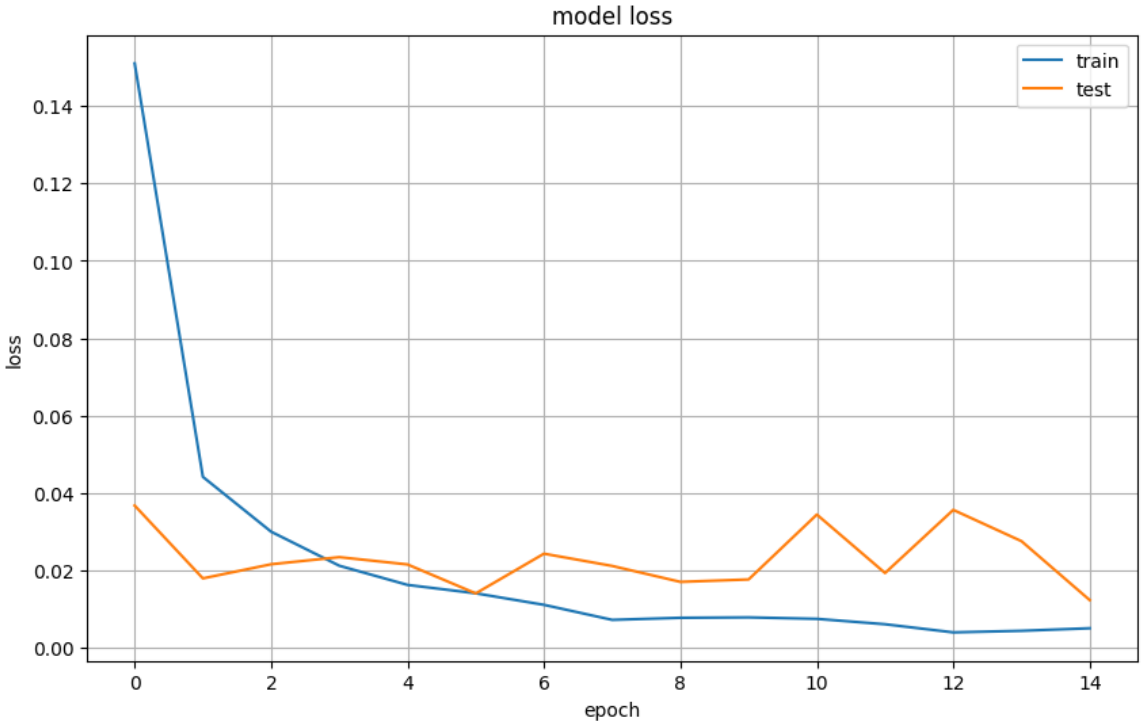
▼ 결과 분석

28 X 28 데이터 셋의 단순한 패턴을 파악하기에 9 X 9가 과하게 크다는 것을 수치로 알 수 있음

더 큰 Kernel을 사용하여 복잡한 패턴을 찾으려고 시도했기 때문에 오히려 부적합을 일으킨 것

단, 초기 loss 감소는 빠르지만 epoch가 진행 될수록 test 손실의 진동이 더 크다

이는 일부 과적합이 발생한 증거가 된다



Accuracy: 0.9879999756813049

우동협

태그	완료
상태	Stride size

합성곱 Stride 조정

목표

합성곱 Stride 조정에 따른 모델 성능 평가 후 Stride의 기능 고민하기

Stride의 기능

커널이 입력데이터 어느 정도 자세하게 인식하는지 처리

이번 튜닝에 대한 내 생각

모델의 구조 변경은 입력데이터의 특성, 모델 사용 목적에 따라 달라진다. 현재 우리가 다루고 있는 MNIST 데이터는 아래와 같은 특징이 있다.

1. 작은 크기
2. 낮은 복잡도

따라서 나는 이번 CNN 구조 변경에서 stride의 크기를 늘리는 것은 입력데이터의 특징을 제대로 못 추출하도록 한다고 생각한다.

구조 변경 특이사항

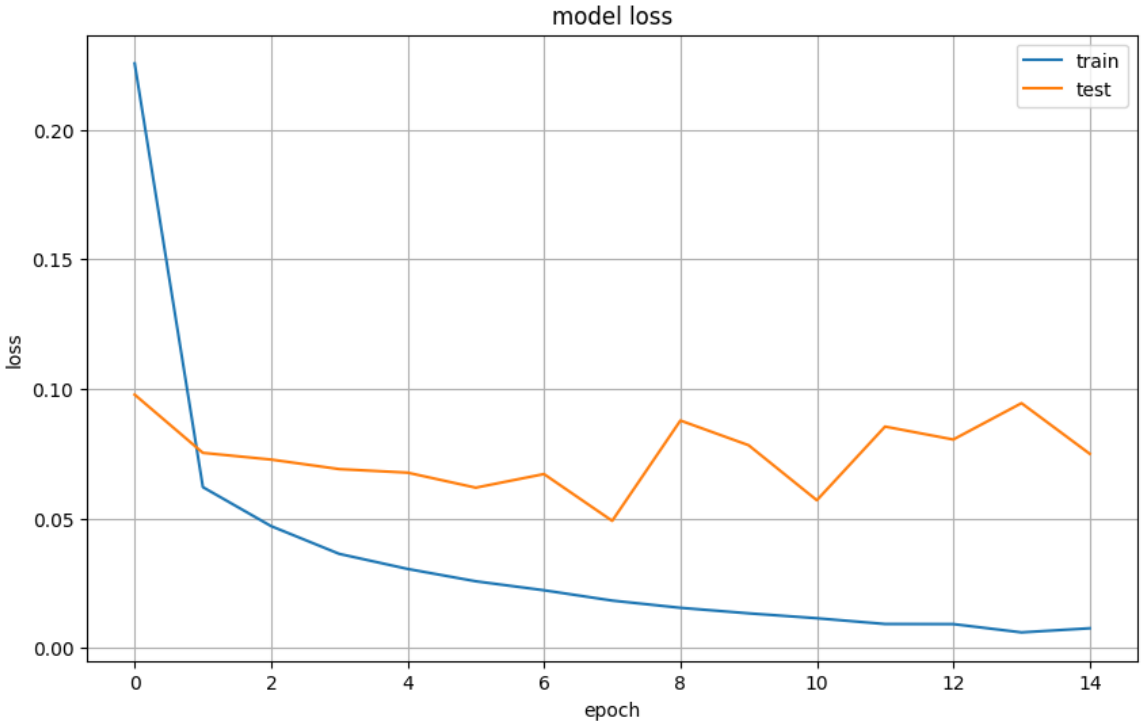
Stride 수를 늘릴수록 크기 유지를 위해 필요한 Padding의 크기가 증가하였다.

또한 입출력 크기를 조절하기 위해 1행와 1열에만 zero padding을 추가하는 작업이 수행되었다.

```
x = pad(x, (1, 0, 1, 0))  # (좌, 우, 위, 아래) 순서, 1행,1열만 zer
```

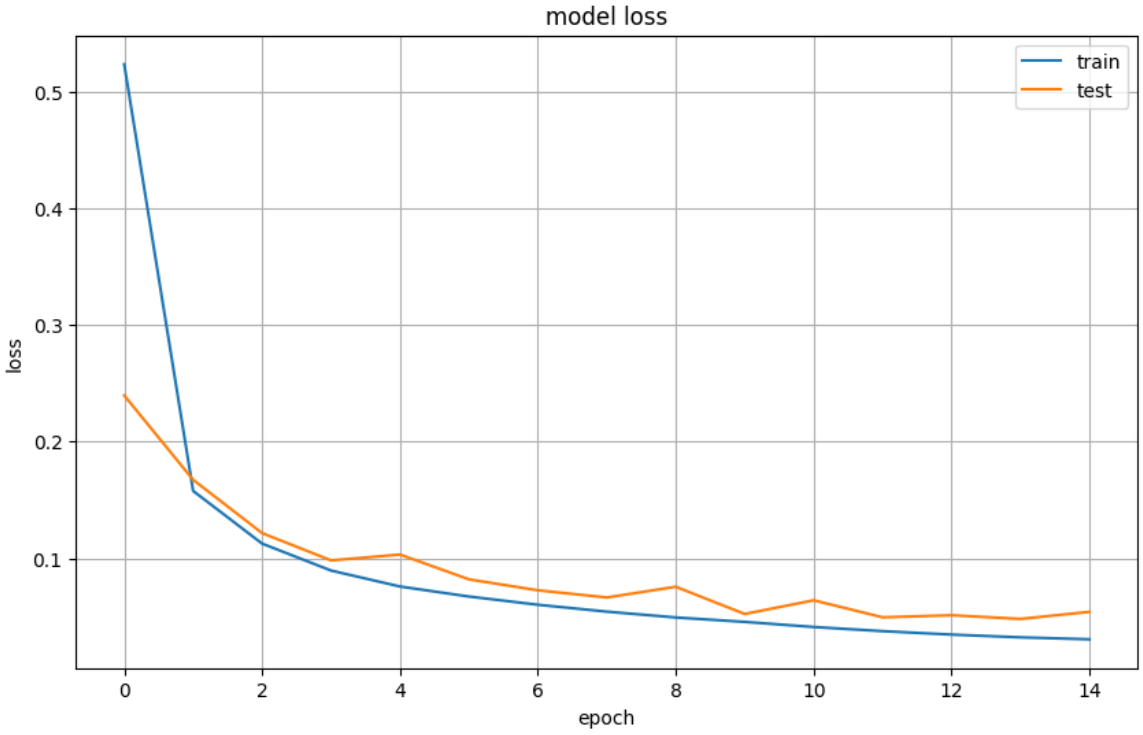
구조 변경 결과

▼ Stride = 1



Accuracy: 0.9881999492645264

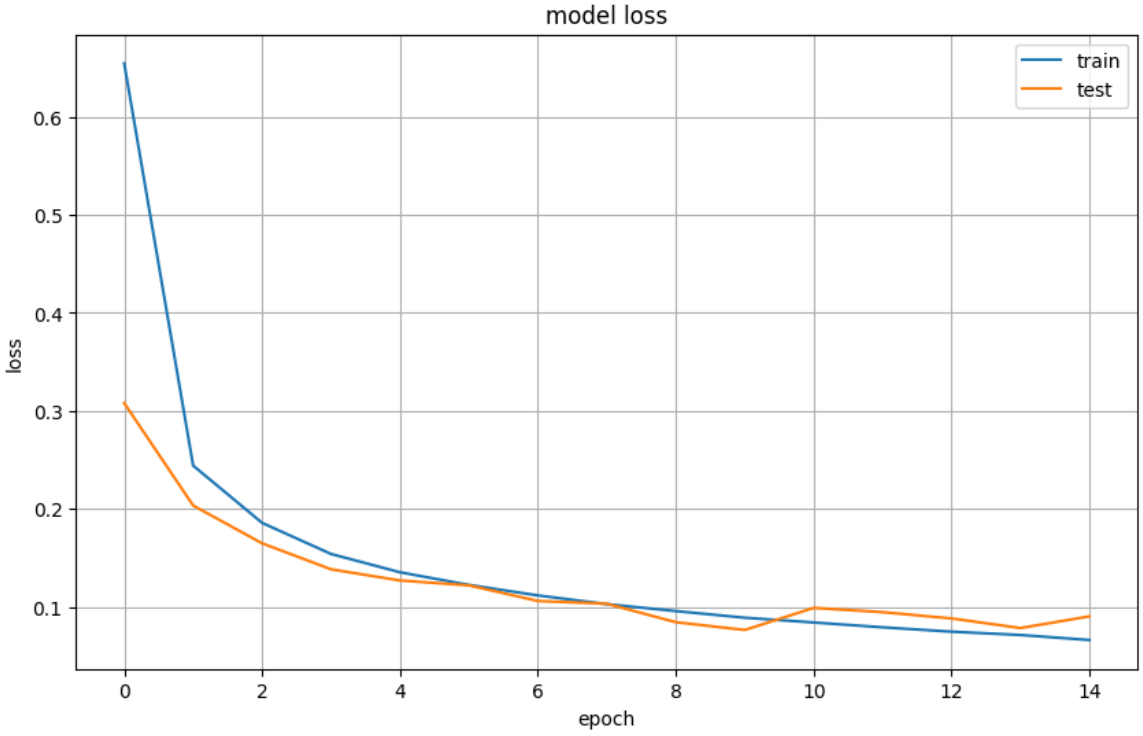
▼ Stride = 2



Accuracy: 0.9803999662399292

▼ Stride = 3

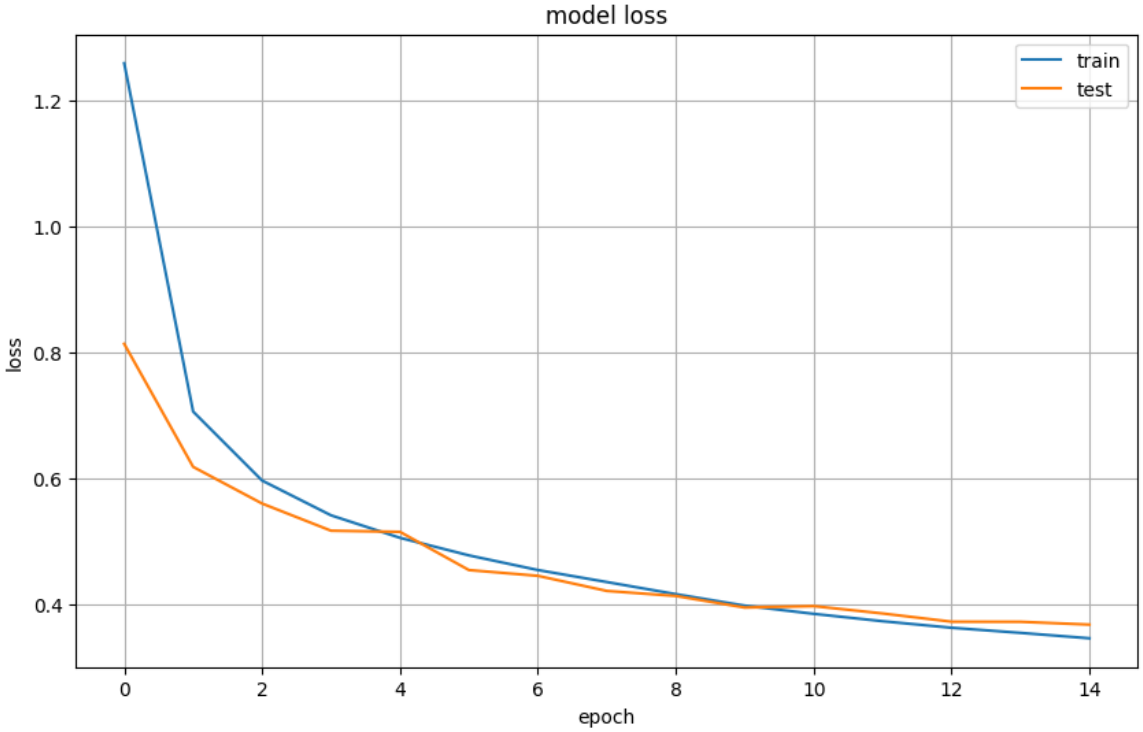
Zero-Padding : Layer1 → 29, Layer2 → 15



Accuracy: 0.9372999668121338

▼ Stride = 4

Zero-Padding : Layer1 → 43, Layer2 → 22



Accuracy: 0.5827000141143799

정채현

태그	완료
상태	Epoch

- 모델 : CNN
- 데이터 : MNIST (train 60,000장, test 10,000장 -> 500장으로 조정)

▼ 💡**구조 변경 요소**💡

1. kernel 크기
2. stride
3. pooling 크기
4. layer수 (kernel layer만 조정)
5. epoch (구조는 아니지만 epoch 수에 따른 모델 학습 관찰)

▼ 🐷 **목표**🐷

좋은 성능을 내는 것이 아니라 구조 변경을 하였을 때, 성능이 변화한다면 왜 이러한 변화가 발생하였는지 생각하는 것!

▼ 🧠 **예상 결과** 🧠

▼ 1. Kernel 크기 (필터 크기)

▼ 1.1. Kernel 역할

- 합성곱 연산에서 **입력 데이터의 작은 영역을 탐지**하는 역할
- 커널 크기는 모델이 학습할 수 있는 **특징의 크기**를 결정

▼ 1.2. 크기 변경의 효과

- **작은 커널**

    - 세밀한 특징(엣지, 텍스처)을 더 잘 학습
    - 작은 특징을 계층적으로 조합해 고수준 특징을 학습 가능. → 작은 커널을 지나치게 많이 사용하면 과적합이 발생할 수 있음.
- **큰 커널**
    - 더 큰 패턴(모양, 윤곽)을 한 번에 학습 가능. → 작은 세부 정보를 놓칠 가능성.

∴ **작은 커널**을 여러 층 쌓아 더 큰 특징을 학습하는 방식이 **일반화**에 더 유리

▼ 2. Stride (스트라이드)

▼ 2.1. 역할

- **커널이 이동하는 간격**
- 출력되는 특성 맵의 크기를 결정합니다.

▼ 2.2. 크기 변경의 효과

- **작은 스트라이드**
    - 커널이 모든 픽셀을 세밀하게 탐색 → 데이터의 **미세한 변화**까지 학습
    - **다양한 입력**을 처리할 수 있는 가능성이 높아짐.

    ↔ 과적합 위험 : **잡음까지 학습**

- **큰 스트라이드**
    - **정보의 일부를 건너뛰어** 데이터의 **요약된 정보**를 학습

      → **잡음에 덜 민감**

      ↔ 복잡한 데이터에서는 일반화 성능이 저하될 수 있음.

▼ 3. Pooling 크기

▼ 3.1. 역할

- 특성 맵을 **다운샘플링하여 계산량을 줄이고 중요한 정보를 요약**하는 역할을 합니다.

▼ 3.2. 크기 변경의 효과

- **작은 풀링 크기 (예: 2×22 \times 22×2)**:
  - 공간 정보를 더 많이 유지.
  - 다운샘플링 효과가 적음.

- **큰 풀링 크기 (예: 4×44 \times 44×4)**:
  - 공간 정보를 크게 축소.
  - 중요한 세부 정보를 손실할 가능성.

▼ 4. Layer 수 (커널 레이어만 조정)

▼ 4.1. 역할

- 레이어 수가 많아지면, 데이터의 저수준 특징(엣지, 텍스처)부터 고수준 특징(모양, 객체)까지 계층적으로 학습할 수 있음.

▼ 4.2. 레이어 수 변경의 효과

- **레이어 수 증가**
  - 더 복잡하고 고차원적인 특징을 학습 가능.
  - 계산량과 파라미터 수 증가 → 학습 시간 증가
  - 과적합 위험
- **레이어 수 감소**
  - 모델이 단순해지고 학습 속도 증가
  - 학습할 수 있는 특징의 한계
  - 복잡한 데이터에서는 성능 저하

▼ 5. Epoch 수

▼ 5.1. 역할

- 데이터셋에 있는 모든 샘플을 한 번씩 사용하여 **모델의 가중치를 업데이트**하는 과정
- **Batch**
  - 학습 데이터 전체를 나눈 작은 그룹(미니배치)
- **Iteration(반복)**

- 한 번의 Batch를 처리하는 과정
- Iteration 수 = 총 데이터 샘플÷Batch 크기

ex :

- 데이터 샘플: 10,000개
- Batch 크기: 64
- Epoch: 5
  - 한 번의 Epoch = 10,000÷64=156 Iterations
  - 5 Epoch = 156×5=780 Iterations

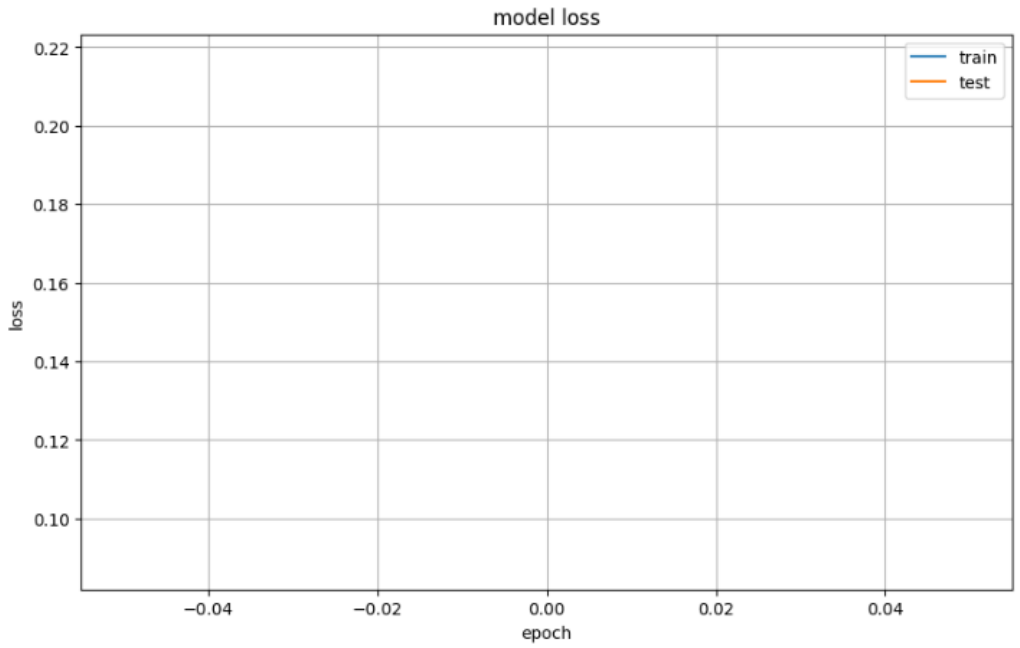
▼ 5.2. 에포크 수 변경의 효과

- **에포크 수가 적으면**
  - 모델이 충분히 학습하지 못해 **과소적합(Underfitting)** 발생.
  - 손실 함수 값이 높은 상태로 학습이 종료.
- **에포크 수가 많으면**:
  - 모델이 충분히 학습 가능.
  - 그러나 과도한 학습은 **과적합(Overfitting)**을 유발할 위험.

▼ 1. Epoch 수 조정 결과

▼ 1. training_epochs = 1

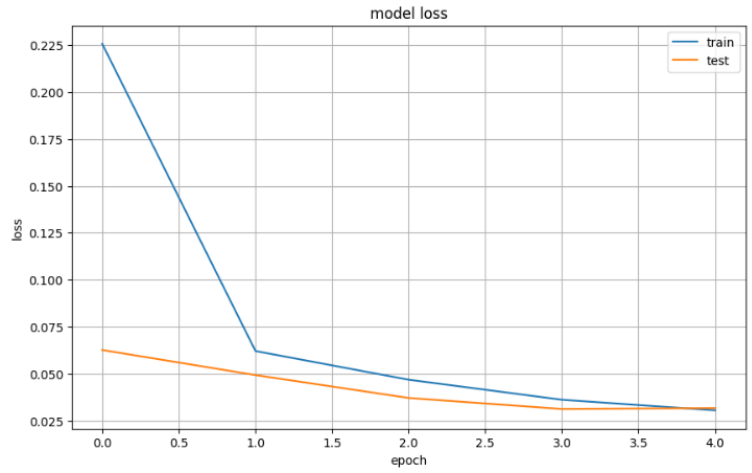
```
[Epoch: 1] Train Loss: 0.2169, Test Loss: 0.0884
```



0.9666000008583069

▼ 2. training_epochs = 5

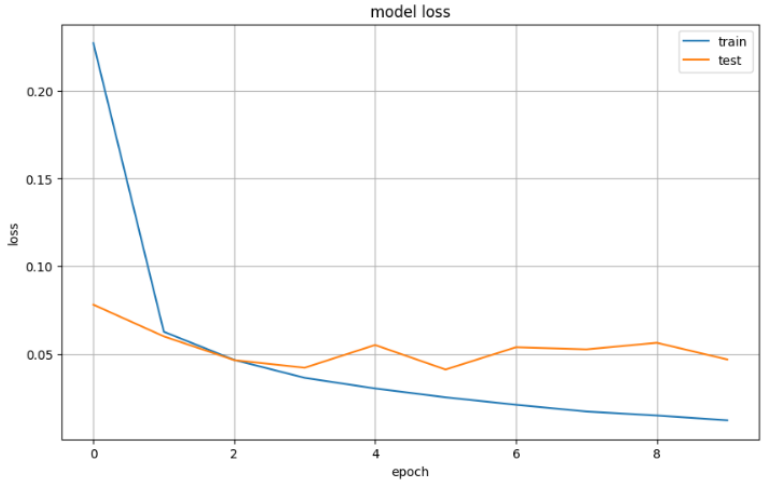
```
[Epoch: 1] Train Loss: 0.2257, Test Loss: 0.0627
[Epoch: 2] Train Loss: 0.0621, Test Loss: 0.0493
[Epoch: 3] Train Loss: 0.0468, Test Loss: 0.0371
[Epoch: 4] Train Loss: 0.0362, Test Loss: 0.0312
[Epoch: 5] Train Loss: 0.0305, Test Loss: 0.0317
```



Accuracy: 0.9765999913215637

▼ 3. training_epochs = 10

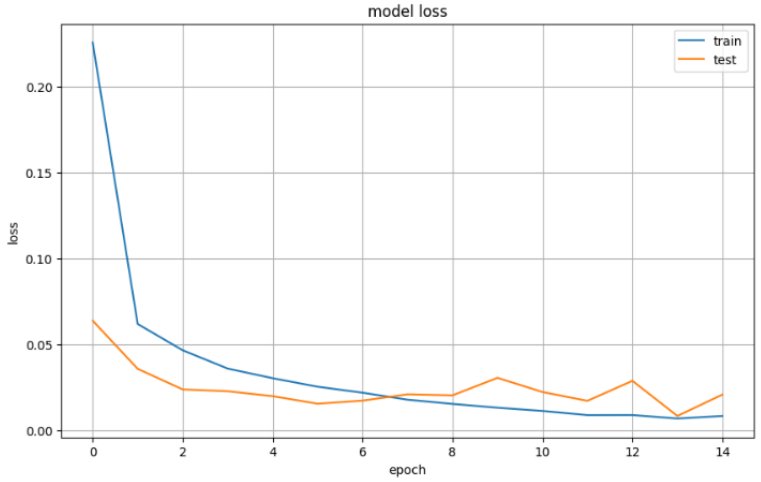
```
[Epoch: 1] Train Loss: 0.2271, Test Loss: 0.0782
[Epoch: 2] Train Loss: 0.0627, Test Loss: 0.0601
[Epoch: 3] Train Loss: 0.0467, Test Loss: 0.0466
[Epoch: 4] Train Loss: 0.0365, Test Loss: 0.0422
[Epoch: 5] Train Loss: 0.0304, Test Loss: 0.0552
[Epoch: 6] Train Loss: 0.0254, Test Loss: 0.0412
[Epoch: 7] Train Loss: 0.0212, Test Loss: 0.0539
[Epoch: 8] Train Loss: 0.0173, Test Loss: 0.0527
[Epoch: 9] Train Loss: 0.0150, Test Loss: 0.0565
[Epoch: 10] Train Loss: 0.0123, Test Loss: 0.0470
```



Accuracy: 0.946899950504303

▼ 4. training_epochs = 15

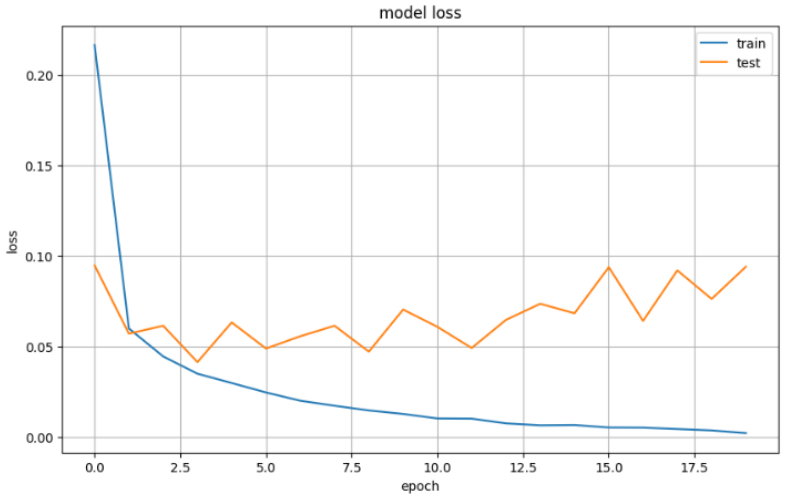
```
[Epoch: 1] Train Loss: 0.2255, Test Loss: 0.0636
[Epoch: 2] Train Loss: 0.0617, Test Loss: 0.0356
[Epoch: 3] Train Loss: 0.0465, Test Loss: 0.0237
[Epoch: 4] Train Loss: 0.0358, Test Loss: 0.0226
[Epoch: 5] Train Loss: 0.0302, Test Loss: 0.0198
[Epoch: 6] Train Loss: 0.0252, Test Loss: 0.0153
[Epoch: 7] Train Loss: 0.0217, Test Loss: 0.0171
[Epoch: 8] Train Loss: 0.0177, Test Loss: 0.0208
[Epoch: 9] Train Loss: 0.0152, Test Loss: 0.0202
[Epoch: 10] Train Loss: 0.0130, Test Loss: 0.0304
[Epoch: 11] Train Loss: 0.0111, Test Loss: 0.0222
[Epoch: 12] Train Loss: 0.0087, Test Loss: 0.0170
[Epoch: 13] Train Loss: 0.0087, Test Loss: 0.0287
[Epoch: 14] Train Loss: 0.0067, Test Loss: 0.0082
[Epoch: 15] Train Loss: 0.0082, Test Loss: 0.0206
```



Accuracy: 0.9883999824523926

▼ 5. training_epochs = 20

```
[Epoch: 1] Train Loss: 0.2153, Test Loss: 0.0946
[Epoch: 2] Train Loss: 0.0598, Test Loss: 0.0569
[Epoch: 3] Train Loss: 0.0443, Test Loss: 0.0613
[Epoch: 4] Train Loss: 0.0348, Test Loss: 0.0412
[Epoch: 5] Train Loss: 0.0297, Test Loss: 0.0631
[Epoch: 6] Train Loss: 0.0245, Test Loss: 0.0487
[Epoch: 7] Train Loss: 0.0199, Test Loss: 0.0555
[Epoch: 8] Train Loss: 0.0171, Test Loss: 0.0613
[Epoch: 9] Train Loss: 0.0145, Test Loss: 0.0470
[Epoch: 10] Train Loss: 0.0126, Test Loss: 0.0702
[Epoch: 11] Train Loss: 0.0101, Test Loss: 0.0607
[Epoch: 12] Train Loss: 0.0100, Test Loss: 0.0490
[Epoch: 13] Train Loss: 0.0074, Test Loss: 0.0645
[Epoch: 14] Train Loss: 0.0063, Test Loss: 0.0734
[Epoch: 15] Train Loss: 0.0065, Test Loss: 0.0682
[Epoch: 16] Train Loss: 0.0051, Test Loss: 0.0936
[Epoch: 17] Train Loss: 0.0050, Test Loss: 0.0640
[Epoch: 18] Train Loss: 0.0043, Test Loss: 0.0918
[Epoch: 19] Train Loss: 0.0035, Test Loss: 0.0761
[Epoch: 20] Train Loss: 0.0020, Test Loss: 0.0939
```



Accuracy: 0.9835000038146973

Epoch	Accuracy
1	0.9666000008583069
5	0.9765999913215637
10	0.946899950504303
15	**0.9883999824523926**
20	0.9835000038146973

▼ 2. 결론

- **학습 손실(Train Loss)**:
  - 모델이 **훈련 데이터**에 대해 손실 함수(예: Cross Entropy Loss)를 줄이는 데 집중

    → Epoch가 증가하면서 손실 값이 점차 감소하며, 데이터의 패턴을 점점 더 잘 학습

    ⇒ **과적합** 가능

- **테스트 손실(Test Loss)**:
  - 모델이 테스트 데이터에 대해 **얼마나 잘 일반화하는지를 표현**

    → Epoch가 너무 많아지면, 모델이 학습 데이터에 과적합하여 테스트 손실이 다시 증가

▼ ⭐ 더 생각해보기

-

epoch 10 vs epoch20

- **손실 함수 & 정확도**

  : 일반적으로 손실이 커질 수록 정확도는 낮아짐.

  ↔ **epoch 10 vs epoch 20**

  - epoch 10 ：비교적 낮은 손실 & 비교적 낮은 정확도
  - epoch 20 : 비교적 높은 손실 & 비교적 높은 정확도

    → **왜 이런 결과가 나온 걸까?**
    : Test data에서
    소수의 오차가 손실을 크게 증가시켰을 수 있음.

  - **손실 함수**
    : 예측 값과 실제 값 사이의
    차이를 기반으로 계산
    → 예측이
    크게 틀린다면, 손실 값에 큰 기여를 함.
    - Cross-Entropy

$$Loss = -\frac{1}{N}\Sigma_{i=1}^{N} y_i log(\hat{y}_i)$$

  - **정확도**
    : 단순히 모델이
    예측을 맞췄는지(1), 틀렸는지(0)을 기준으로 계산

$$Accuracy = \frac{TotalPredictions}{CorrectPredictions}$$

- **결론1. - 정확도와 손실이 모두 낮은 상황**
  i. **Underfitting**

     = 학습 초기의 무작위 예측 - 가중치 초기화 상태 → **큰 오차가 없음.**

- **결론2. - 정확도가 높아도 손실이 커지는 상황**
  i. 모델이 정답을 맞췄지만, **불확실할 때 (확률 값이 낮을 때)**
  ii. 모델이 **소수의 클래스에 대해 일반화하지 못하고 있을 때**
  iii. Test, train **데이터 분포가 크게 다를 때**

최종렬

태그	완료
상태	Pooling Layer

pooling

이미지에서 중요한 정보를 요약하고 공간 크기를 줄이는 기법

적절한 풀링 방식 찾기

pooling 종류별 테스트 결과

(1) max pooling

—> 정확도: 0.9871999621391296

(2) average pooling

—> 정확도: 0.98679995536804

(3) min pooling

→ 정확도: 0.8597999811172485

(픽셀 값에 음수 적용)

MNIST데이터의 구조적 특징

28 × 28 크기의 흑백 이미지로 구성

숫자는 흰색(밝은 픽셀 값, 일반적으로 255)로 나타나며, 배경은 검정색(어두운 픽셀 값,0)이다.

→ 배경 숫자간의 명확한 대비가 존재

데이터의 희소성

대부분의 픽셀 값이 0으로 채워져 있다.

숫자 모양을 나타내는 픽셀만 유의미한 값을 가지며, 전체 이미지에서 이는 매우 작은 비율을 차지한다.

max pooling 과 average pooling비교

맥스 풀링

각 필터 내에서 **가장 큰 픽셀 값을 선택**함

이는 주로 윤곽선, 경계, 밝은 특징과 같은 중요한 정보를 강조

(숫자의 중요한 정보인 윤곽선만 남고, 불필요한 배경 정보는 효과적으로 제거된다.)

평균 풀링

필터 내의 **모든 픽셀 값을 평균화**한다.

(윤곽선을 완벽히 보존하지는 못하더라도, 경계 역역의 전반적인 패턴과 모양을 유지한다. 이 방법은 윤곽선이 부드럽게 표현되더라도, 전체적인 숫자의 형태를 파악하기에 충분)

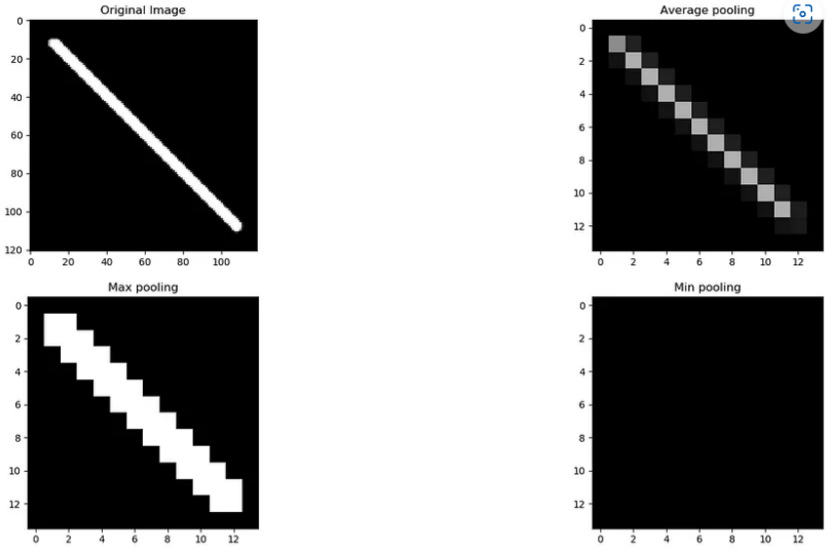
이미지의 전체적인 정보를 부드럽게 만들지만, **숫자의 뚜렷한 특징이 약화될 수 있다.**

최소 풀링

mnist 데이터에서 **가장 작은 픽셀 값(최소 값)을 선택**한다.

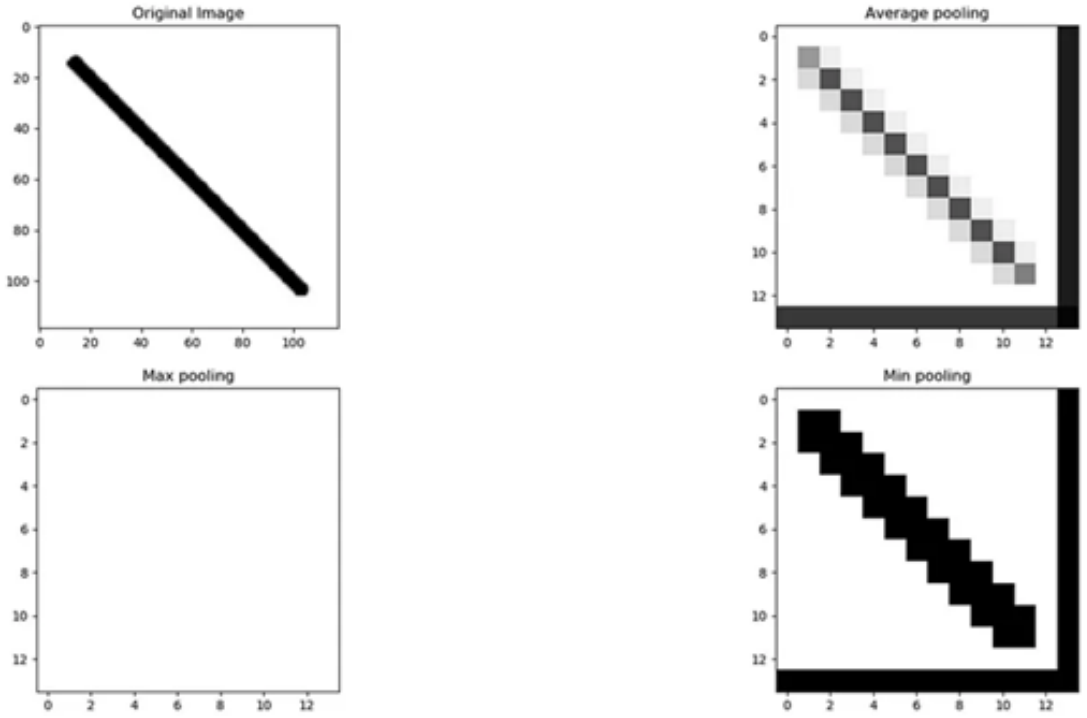
그런데 여기서 배경은 모두 흑색으로 0으로 된 데이터가 많다. 최소 풀링 방식을 사용하게 되면 풀링 연산 과정에서 0이 선택되는 비율이 매우 높아져 숫자의 특징을 담은 값들이 작은 값 0 즉 흑색으로 대체되어 정보의 손실이 발생한다.

결과적으로 숫자의 모양, 윤곽선, 경계와 같은 중요한 특징이 약화되거나 사라지게 된다.



Max pooling gives better result for the images with black background and white object (Ex: MNIST dataset)

배경이 흰색 숫자가 흑색인 경우



결론: MNIST 데이터와 같은 흑백 이미지에서 숫자와 배경의 대비가 명확한 경우, MAX Pooling이 가장 효과적인 방식이다.

적절한 Kernel size 및 stride size 찾기

1. MNIST 데이터 특성

- **이미지 크기**: 28×28 픽셀로 고정된 크기.
- **숫자 윤곽**: 이미지 중앙에 숫자가 위치하고, 흑백 대비가 뚜렷함.
- **단순한 구조**: 숫자의 윤곽선 정보를 유지하면서 배경 노이즈를 제거하는 것이 중요.

풀링 후 출력의 크기를 결정하는 공식

$$\text{Output Size} = \left\lfloor \frac{\text{Input Size} - \text{Kernel Size}}{\text{Stride}} + 1 \right\rfloor$$

Kernel size와 stride의 역할

kernel size: 풀링 할 영역의 크기를 정의

큰 크기 → 더 많은 정보를 압축

그러나 **세부 정보가 손실**될 가능성이 크다.

작은 크기 → **디테일을 유지**

그러나 **크기 축소 효과는 작아** 계산량이 증가

stride: 풀링 영역이 이동하는 간격

클수록 → 공간 축소 효과가 크며 계산량이 감소

너무 큰 크기 → 세부 정보를 손실할 위험이 있음

kernel size 선택

작은 크기(2x2)

mnist처럼 작은 이미지에서 윤곽선을 유지하면서 크기를 줄이는 데 적합

큰 크기(3x3 이상)

큰 이미지에서는 적합하나 mnist에서는 적합하지 않음

더 많은 정보를 요약하지만 28x28같은 작은 이미지에서는 숫자의 세부 정보를 잃을 수 있음

→ 2×2크기가 적절

stride 선택

stride < kernel (2×2, stride=1)

풀링 영역이 겹치며 더 자세한 정보를 보존

세부 정보는 더 잘 유지되지만, mnist와 같은 간단한 데이터에서는 불필요하게 계산량이 늘어날 수 있음

stride size = kernel size(2×2, stride=2)

겹치지 않는 영역에서 풀링을 수행

mnist에서는 28 x 28 → 14 × 14로 축소하여 계산량을 줄이고 중요한 윤곽선을 유지

stride > kernel size (2×2, stride >3):

공간을 빠르게 줄이지만, 풀링하지 않은 부분의 정보 손실 가능성이 크다.

mnist와 같은 작은 이미지에서는 권장되지 않음

MNIST와 같은 28x28 사이즈의 이미지 데이터에는 kernel 사이즈는 주로 2x2와 3x3이 쓰이고 stride는 1,2,3이 쓰임

모든 경우의 수 테스트 결과

kernel size (2,2) & stride size (1) → 0.971299946308136

kernel size (2,2) & stride size (2) → 0.9868999719619751

kernel size (2,2) & stride size (3) → 0.976099967956543

kernel size (3,3) & stride size (1) → 0.9365999698638916

kernel size (3,3) & stride size (2) → 0.9847999811172485

kernel size (3,3) & stride size (3) → 0.9681999683380127

결론: kernel size (2,2)와 stride size(2)가 가장 적절한 조합이다. 28x28크기인 MNIST 이미지를 14x14로 축소하여 계산량을 줄이고, 작은 영역에서도 숫자의 윤곽선을 유지하며 배경 노이즈를 제거할 수 있다. 그리고 stride(2)는 겹치지 않는 영역에서 풀링을 수행하여 중복 계산을 방지하면서 중요한 특징을 추출할 수 있다. 즉 효율성과 정확도를 모두 만족시키는 조합이다.

한서연

태그	완료
상태	Number of Layer

레이어 수 조정

목표

CNN에서 합성곱 레이어 수를 조정하여 모델의 성능 변화를 평가하고, 레이어 수 증가 또는 감소가 학습과 일반화 성능에 미치는 영향을 분석한다.

CNN의 레이어 (Layer)란 무엇인가?

레이어(Layer)는 **신경망의 기본 구성 단위**로, 데이터가 입력된 후 학습 과정을 거쳐 다음 단계로 전달되는 과정을 정의합니다. CNN에서 레이어는 크게 합성곱(Convolution), 활성화 함수(Activation), 풀링(Pooling)으로 구성되며, 각 레이어는 특정한 역할을 수행합니다.

CNN에서 레이어의 역할

1. **합성곱 레이어 (Convolutional Layer)**
   - 이미지 데이터에서 **국소적 특징**(예: 가장자리, 곡선 등)을 추출하는 역할.
   - 입력 이미지에 커널(필터)을 적용해 특징 맵(Feature Map)을 생성.
   - **핵심 파라미터**:
     - **필터 크기**: 커널의 높이와 너비(예: 3x3, 5×5).
     - **필터 수**: 생성될 특징 맵의 개수(출력 채널 수).
     - **Stride**: 필터가 이동하는 간격.
     - **Padding**: 경계 부분에서 크기를 유지하기 위한 추가 여백.
2. **활성화 함수 (Activation Function)**
   - 각 뉴런의 출력을 비선형 함수로 변환해 복잡한 패턴 학습 가능.
   - CNN에서는 주로 **ReLU**(Rectified Linear Unit) 함수 사용:

    - 계산량이 적고, 기울기 소실(Vanishing Gradient) 문제 완화.

3. **풀링 레이어 (Pooling Layer)**
    - 특징 맵의 크기를 줄여 계산량을 줄이고, 중요한 정보만 유지.
    - **Max Pooling**: 구역 내 최대값 선택.
    - **Average Pooling**: 구역 내 평균값 계산.
4. **전결합층 (Fully Connected Layer)**
    - 합성곱 레이어와 풀링 레이어에서 추출된 특징 벡터를 Flatten(1차원 변환)해 최종 출력 클래스로 매핑.
    - CNN의 마지막 단계에서 전체 이미지를 학습된 특징으로 분류.

레이어 수가 성능에 미치는 영향

- **적은 레이어 수 (얕은 네트워크)**
    - 간단한 데이터셋(예: MNIST)에는 적합.
    - 학습이 빠르고 과적합 가능성이 낮음.
    - 고차원 특징을 학습하지 못해 복잡한 데이터에는 성능 한계.
- **많은 레이어 수 (깊은 네트워크)**
    - 복잡한 데이터셋(예: CIFAR-10, ImageNet)에서 고차원 특징 학습 가능.
    - 그러나 레이어가 많아질수록 학습 시간이 증가하고, 과적합 가능성도 높아짐.

CNN에서 레이어 설계 시 고려할 점

1. **데이터셋의 복잡도**
    - 단순 데이터셋(MNIST): 2~3개의 레이어로 충분.
    - 복잡한 데이터셋(CIFAR-10, ImageNet): 10개 이상의 레이어 필요.
2. **계산 리소스**
    - 레이어 수가 많아질수록 계산량 증가.
    - 제한된 리소스에서는 레이어 수와 파라미터를 적절히 조정해야 함.
3. **과적합 방지**

- 깊은 모델에서는 과적합 가능성이 높으므로 Dropout, 데이터 증강, 정규화 등을 사용.

레이어 튜닝에 대한 생각

레이어 수를 늘리거나 줄이는 것이 항상 성능 향상으로 이어지지는 않는다.

특히 우리가 다루고 있는 **MNIST 데이터셋**은 다음과 같은 특징을 가진다:

1. 비교적 **작은 이미지 크기** (28x28)
2. **복잡도가 낮은 데이터** (단순한 숫자 형태)

따라서 **레이어 수를 너무 많이 늘리는 경우**, 모델이 불필요하게 복잡해지고, 학습 시간은 증가하나 성능은 오히려 감소할 가능성이 높다고 생각했다.

반면, 레이어를 줄이는 경우, 고차원적인 특징을 학습하지 못해 성능이 크게 저하될 수 있다.

구조 변경 특이사항

- **레이어 수 증가(3, 4개)**

  레이어 수가 증가할수록 모델의 용량(파라미터 수)이 급격히 늘어났다.

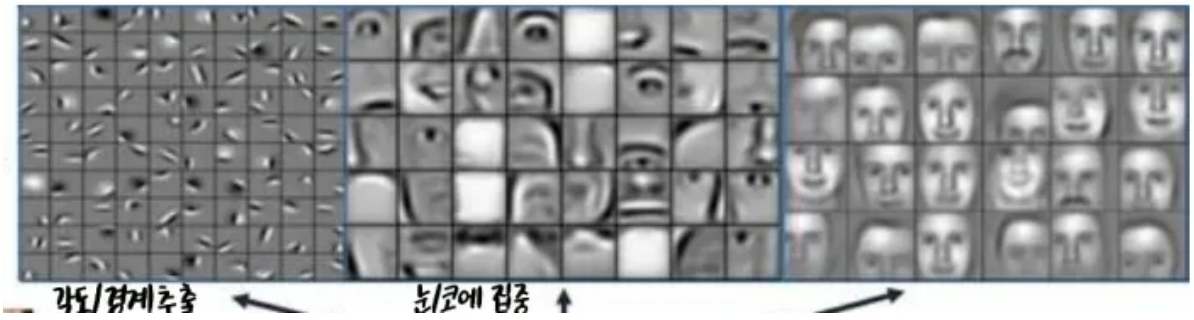
  이에 따라 학습 속도가 느려졌으며, 과적합 가능성이 높아졌다.

  특히 4개의 레이어를 사용했을 때는 **Flatten 후 출력 크기**를 조정해야 했으며, Fully Connected Layer의 입력 크기도 함께 변경되었다.

- **레이어 수 감소(1개)**

  단일 레이어를 사용했을 때, 모델의 학습 용량이 제한되었고, Test Loss가 크게 증가했다.

  MNIST 데이터의 패턴을 충분히 학습하지 못한 결과로 보인다.



구조 변경 결과

레이어1

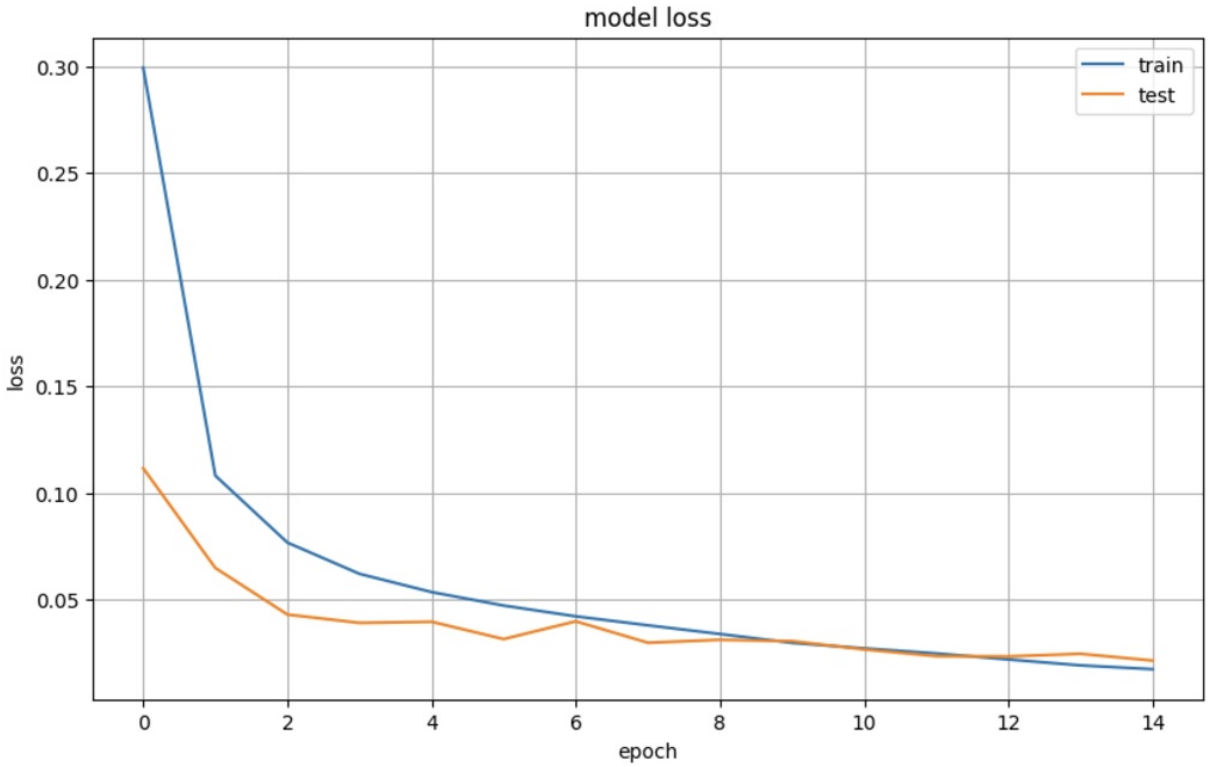
```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # Layer 1 (유일한 레이어)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, pa
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Fully Connected Layer
        # Flatten 이후의 크기: 14x14x32 (1번의 MaxPool로 인해 크기
        self.fc = torch.nn.Linear(14 * 14 * 32, 10, bias=True

        # Xavier Initialization
        torch.nn.init.xavier_uniform_(self.fc.weight)

    def forward(self, x):
        out = self.layer1(x)
        out = out.view(out.size(0), -1)  # Flatten
        out = self.fc(out)
        return out
```



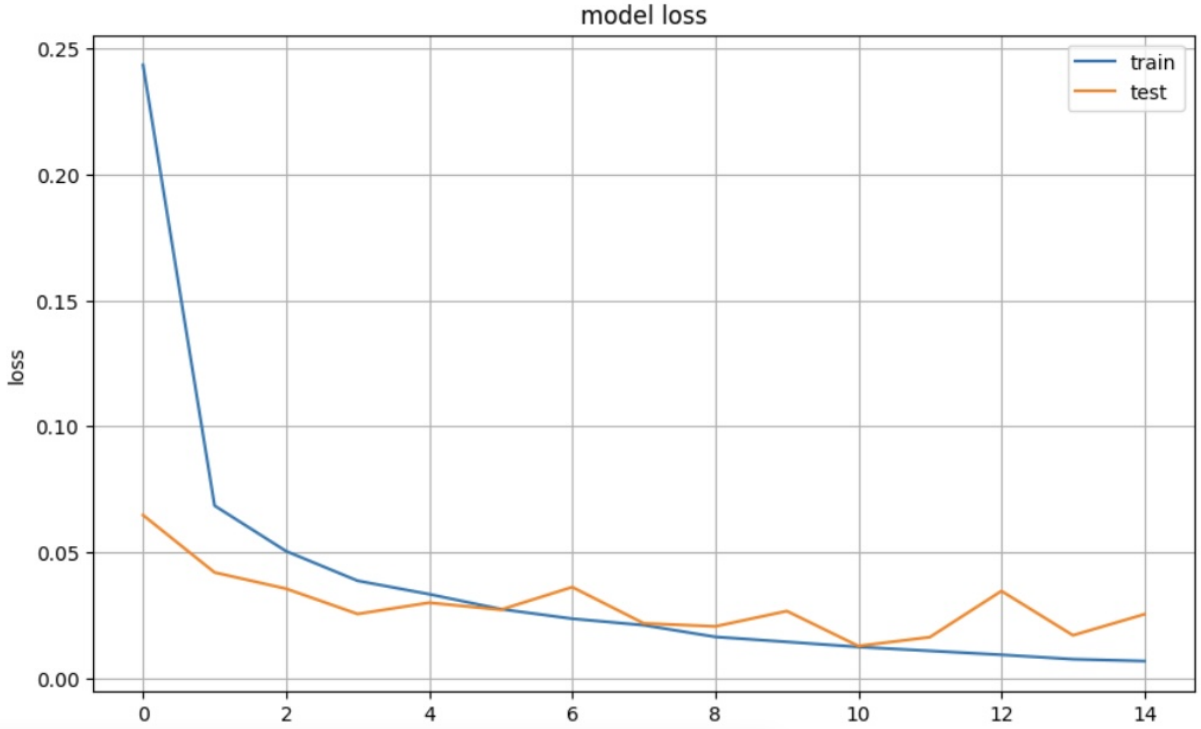
구조 요약

1. **Layer 1**
   - 입력 채널: `1` (MNIST는 흑백 이미지)
   - 출력 채널: `32` (32개의 특징 맵 생성)
   - 커널 크기: `3x3`
   - 풀링 크기: `2x2` → **출력 크기**: `14 x 14 x 32` (이미지 크기가 28x28에서 MaxPool로 절반으로 줄어듦).
2. **Fully Connected Layer**
   - Flatten된 크기: `14 x 14 x 32 = 6272`
   - 출력 크기: `10` (0~9까지의 숫자 클래스).

레이어2

```
# layer 2 (추가)
self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1,
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))
```

```
        # 전결합층 7x7x64 inputs -> 10 outputs
        self.fc = torch.nn.Linear(7 * 7 * 64, 10, bias=True)
```



구조 요약

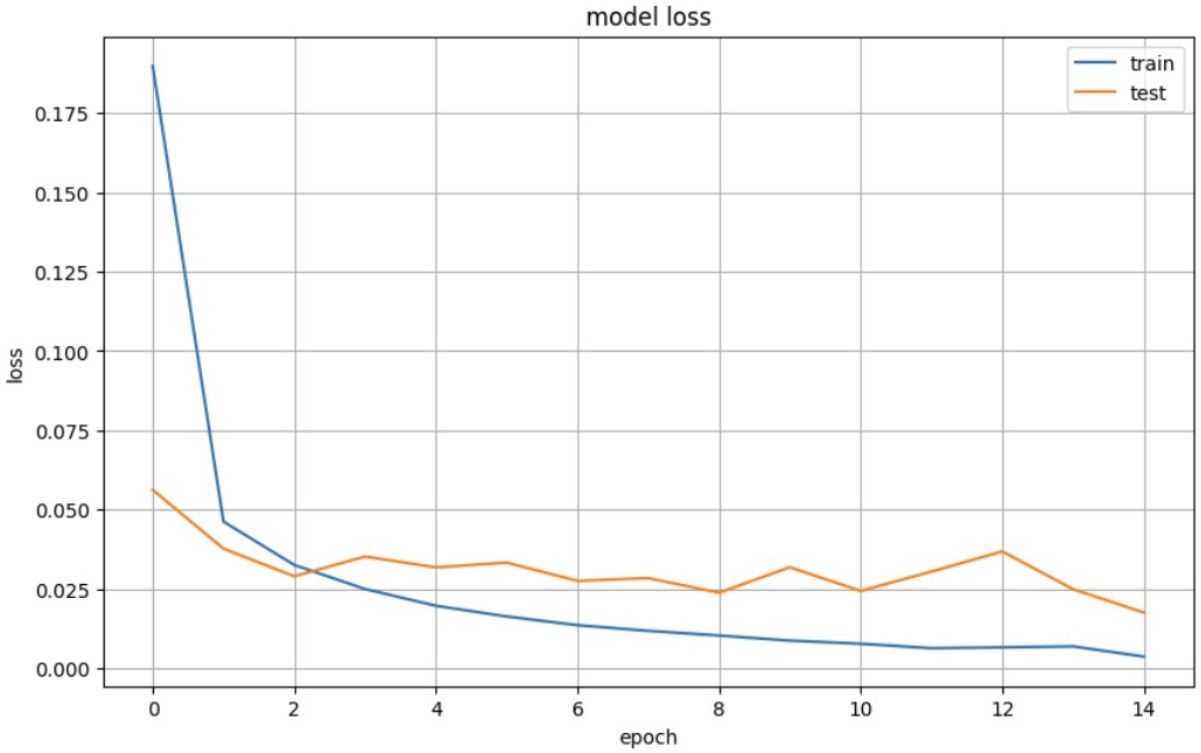
1. **Layer 2**
   - 입력 채널: `32`
   - 출력 채널: `64` (64개의 특징 맵 생성)
   - 커널 크기: `3x3`
   - 풀링 크기: `2x2` → **출력 크기**: `7 x 7 x 64` (14×14 → MaxPool로 절반 감소).
2. **Fully Connected Layer**
   - Flatten된 크기: `7 x 7 x 64 = 3136`
   - 출력 크기: `10` (0~9까지의 숫자 클래스).

레이어3

```
# Layer 3 (추가된 레이어)
self.layer3 = torch.nn.Sequential(
```

```
    torch.nn.Conv2d(64, 128, kernel_size=3, stride=1, padding
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2)
)

# Fully Connected Layer
# Flatten 이후의 크기: 128 채널, 3번의 MaxPool로 인해 3x3 이미지
self.fc = torch.nn.Linear(3 * 3 * 128, 10, bias=True)
```

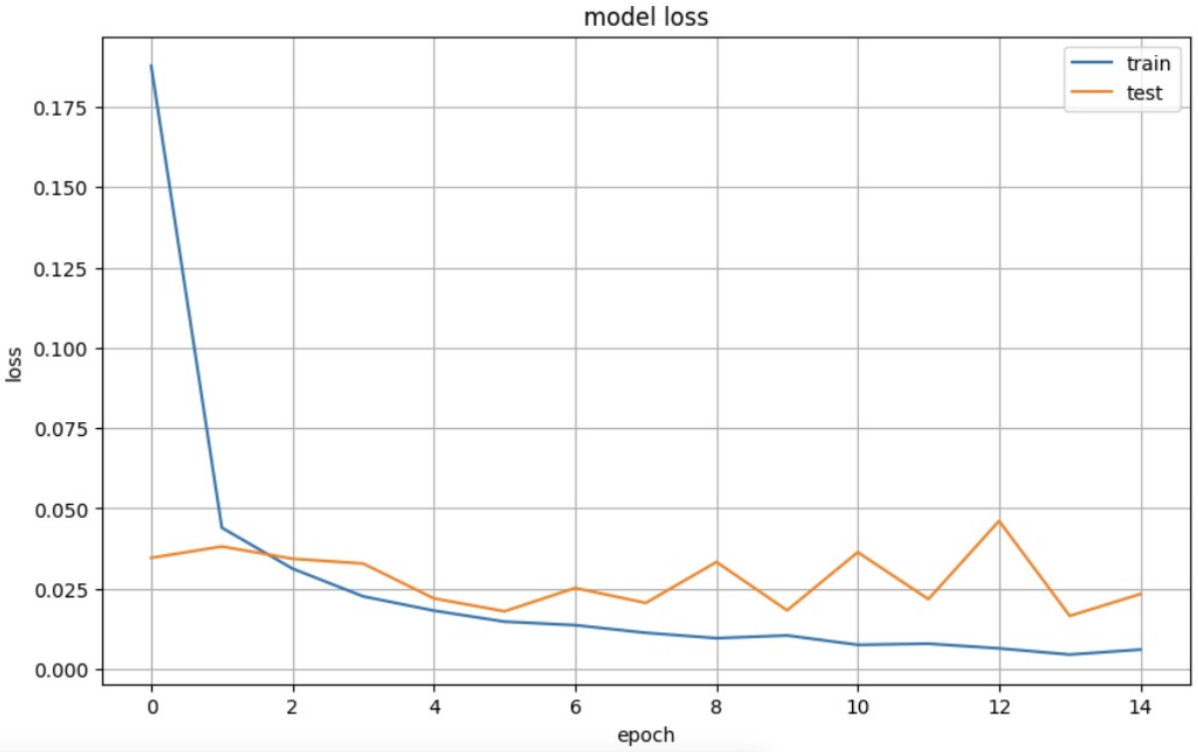


1. **Layer 3**
    - 입력 채널: `64`
    - 출력 채널: `128`
    - 커널 크기: `3x3`
    - 풀링 크기: `2x2` → **출력 크기**: `3 x 3 x 128`.
2. **Fully Connected Layer**
    - Flatten된 크기: `3 x 3 x 128 = 1152`
    - 출력 크기: `10`.

레이어4

```
# Layer 4 (추가)
self.layer4 = torch.nn.Sequential(
    torch.nn.Conv2d(128, 256, kernel_size=3, stride=1, paddin
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2)
    )

# Fully Connected Layer
# Flatten 이후의 크기: 1x1x256 (4번의 MaxPool로 인해 크기 줄어듦)
self.fc = torch.nn.Linear(1 * 1 * 256, 10, bias=True)
```



1. **Layer 4**
   - 입력 채널: `128`
   - 출력 채널: `256`
   - 커널 크기: `3x3`
   - 풀링 크기: `2x2` → **출력 크기**: `1 x 1 x 256`.
2. **Fully Connected Layer**

- Flatten된 크기: `1 x 1 x 256 = 256`
- 출력 크기: `10`.

결과 정리

모델 구조	Train Loss	Test Loss	Accuracy	특이사항
1 Layer	0.0173	0.0214	0.6881	단순 모델, 고차원 특징 학습 부족
2 Layers	0.0068	0.0254	0.9877	적당한 복잡도, 안정적 성능
3 Layers	0.0036	0.0175	**0.9919**	가장 높은 성능
4 Layers	0.0060	0.0233	0.9747	과도한 복잡도, 성능 저하 발생

결론 및 내 생각

1. **2~3개의 레이어**가 MNIST 데이터셋에서 가장 적합한 구조로 나타났다.
2. **레이어 수를 늘리는 것**은 복잡한 데이터셋(CIFAR-10 등)에서는 필요할 수 있지만, MNIST처럼 단순한 데이터셋에서는 성능 저하로 이어질 수 있다.
3. 모델 설계는 항상 **데이터의 특성과 모델의 목적**을 고려해야 하며, 지나치게 단순하거나 복잡한 모델은 모두 비효율적이라는 것을 확인했다.