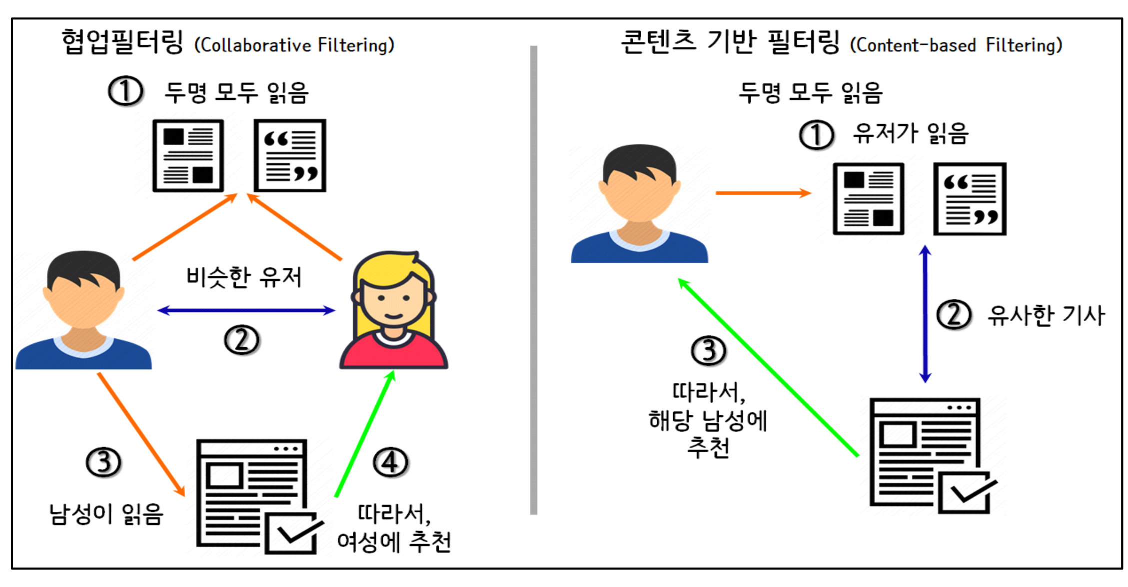
15 인공지능 개요

1. 인공지능이란 무엇일까?

가. 인공지능의 개념

1) 인공지능(AI)은 'Artificial Intelligence' 의 약자로 '인공지능' 이라 번역됨
2) 흔히 '로봇' 을 인공지능이라 오해하는 경우가 많음
3) 인공지능은 '지능을 가진 컴퓨터', '컴퓨터상에서 인간의 지능을 재현한 것', '컴퓨터상에서 인간의 지능을 재현하기 위한 기술' 등을 의미함
4) 그렇다면 인간의 지능은 무엇을 의미하는 것일까?
   - 계산하는 능력을 의미한다면 이미 컴퓨터는 인간의 능력을 뛰어넘음
   - 컴퓨터는 본질적으로 계산기인데 이를 인공지능이라 부를 수 있을까?
   - 그렇지 않음. 우리는 데스크 톱이나 노트북 등 컴퓨터 자체를 인공지능, 인공지능 컴퓨터라 부르지 않음
   - 인공지능은 계산하는 기능뿐만 아니라 창조적인 무언가를 해낼 수 있는 기능이 있어야 함
5) 그렇다면 무엇을 할 수 있어야 인공지능이라 불릴 수 있을까?
   - 무엇을 어떻게 할 수 있어야 인공지능인지에 대한 명확한 기준은 없음
   - 다만 사람이 알려준 것 이상의 일을 할 수 있어야 함
   - 사람이 알려준 것 이상의 일을 할 수 있다는 것은 컴퓨터가 스스로 어떠한 기준이나 근거(알고리즘)를 바탕으로 판단할 수 있다는 것을 의미함
   - 예: 추천 알고리즘 → 협업 필터링, 콘텐츠 기반 필터링



6) 인공지능과 인공지능 기술은 어떻게 다른 것일까?

- 우리 일상생활과 함께 하는 인공지능 기술을 사람들은 인공지능이라고 부르고 있음. 실제로 존재하지 않는 인공지능을 인공지능이라고 부르는 것에서부터 착각이 발생함
- 인공지능 기술은 인공지능을 실현하기 위해 개발되고 있는 다양한 기술임 (예: 자연어 처리 기술, 음성인식 기술, 영상 처리 기술 등)
- 인공지능은 인공지능 기술 개발의 목적이고, 인공지능 기술은 인공지능을 구현하기 위한 방법임
- 현 수준의 인공지능은 인공지능 기술(Technology) 혹은 약한 인공지능임

● **인간적 사고** “컴퓨터가 생각하게 하는 흥미로운 새 시도. 문자 그대로의 완전한 의미에서 마음을 가진 기계.” (Haugeland, 1985) “인간의 사고, 그리고 의사결정, 문제 풀기, 학습 등의 활동에 연관시킬 수 있는 활동들.” (Bellman, 1978)	● **합리적 사고** “계산 모형을 이용한 정신 능력 연구.” (Charniak 및 McDermott, 1985) “인지와 추론, 행위를 가능하게 하는 계산의 연구.” (Winston, 1992)
● **인간적 행위** “사람이 지능적으로 수행해야 하는 기능을 수행하는 기계의 제작을 위한 기술.” (Kurzweil, 1990) “현재로서는 사람이 더 잘하는 것들을 컴퓨터가 하게 만드는 방법에 대한 연구.” (Rich 및 Knight, 1991)	● **합리적 행위** “계산 지능은 지능적 에이전트의 설계에 관한 연구이다.” (Poole 외, 1998) “인공지능은 … 인공물의 지능적 행동에 관련된 것이다.” (Nilsson, 1998)
강인공지능	약인공지능

7) 우리 주변에 존재하는 인공지능

- 휴대전화 추천 문구 기능 → ‘안녕하세요?’의 ‘안’이라는 글자만 입력해도 ‘안녕하세요?’라는 추천 문구를 제안해 줌
- 영어를 한국어로 번역해주는 기능 → 구글 번역기, 네이버 파파고 등
- 추천해주는 기능 → 인터넷 쇼핑 시 물건 추천, 유튜브 영상 추천 등
- 음성인식 기능(음성을 문자로 변환), 음성 변환 기능(문자를 음성으로 변환)
- 휴대전화로 사진 찍을 EO 안면을 인식하는 기능
- 이 밖에도 우리 주변에 존재하는 인공지능의 종류와 기능은 매우 다양함

8) 인공지능의 실체는?

- 새로운 개념이 아니라, 목적을 달성하기 위한 프로그램들의 집합체임
- 즉, 프로그램은 결국 SW이고, SW를 만드는 가장 기본은 인간의 두뇌에서 추상화의 과정을 거쳐 만들어지는 ‘알고리즘’이라고 할 때, 현재의 인공지능은 프로그램이며 알고리즘일 뿐임

9) 인공지능이 잘하는 일과 못하는 일

- 인공지능은 무언가를 분류하는 일을 잘함(예: 메일함에 도착한 메일이 스팸 메일인지 아닌지를 분류하는 작업). 그러나 인공지능의 분류 작업의 결과는 수학에서의 확률과 통계의 영역에 속하는 추측일 뿐, 실제로 그렇다는 것을 의미하는 것은 아님
- 인공지능은 사고의 유연성이 요구되는 일이나 창조적인 일을 잘 못함
- 인공지능은 정확한 답을 알지 못함. 인공지능은 추측하거나 예측하는 것일 뿐임. 하지만 사람이 정답인지 알 수도 없는 문제를 예측하고 판단할 수 있다는 것은 놀라움. 즉, 인공지능은 정확한 정답을 모르지만, 가능한 정답에 근사한 값을 예측하여 내놓을 수는 있음
- 인공지능 로봇 도로보군의 도쿄대학 입학 도전 → 수학, 역사 과목은 만점에 가까운 성적을 거두었으나 영어, 국어 과목은 반타작 내외에 가까운 성적을 기록함 → 인공지능이 잘 할 수 있는 일과 할 수 없는 일에 대한 이해가 필요함

2. 인공지능 발전의 역사

가. 인공지능은 최근에서야 생겨났다고 생각하는 사람들이 많지만, 사실 인공지능의 역사는 꽤나 긴 편임(1950년대 ~ 현재)

나. 역사적인 다트머스 회의(1956) → 존 맥카시가 인공지능이란 용어를 처음 사용했으며, 이에 대한 공동의 연구를 10인의 연구자에게 제안함

다. 즉, 과거에 추구했던 인공지능은 강인공지능이나, 두 차례 인공지능의 겨울과 같은 기술적 한계에 부딪혀 현재 추구하는 인공지능은 약인공지능이라 정리할 수 있음

"학습이나 지능의 특징들은 원칙적으로 기계에게 시뮬레이션 할 수 있도록 만들어져 있다. 기계가 추상적 개념으로부터 언어를 사용하고 현재 인류에 당면한 문제를 해결하고 스스로 향상시키는 방법을 찾기위한 시도가 이루어질 것이다. 신중하게 선정된 과학자 집단이 여름동안 함께 연구한다면 이러한 문제들 중 하나 이상에서 상당한 진전을 이룰 수 있다고 생각한다.
- 인공지능에 관한 다트머스 여름 연구 프로젝트 제안 (McCarthouse et al, 1955)

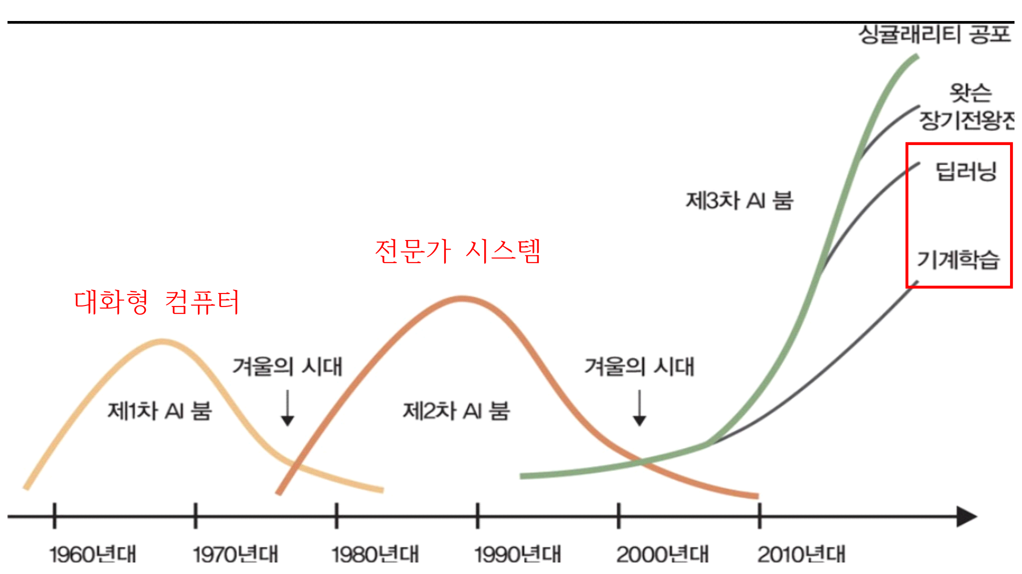
와... 패기보소!

과거에 추구했던 인공지능 = 강인공지능

VS.

papago

현재 추구하는 인공지능 = 약인공지능



라. 1세대 열풍의 인공지능, 대화형 컴퓨터

1) 엘리자(ELIZA, 1966)

- MIT 컴퓨터공학과 교수 요제프 바이첸바움(Joseph Weizenbaum)이 탄생시킨 최초의 대화형 컴퓨터 프로그램임
- 음성이 아닌 문자로 대화할 수 있는 컴퓨터로, 널리 쓰이는 챗봇의 시초임
- 주로 질문에 답변하는 정신치료사 역할을 하도록 설계된 대화 시뮬레이션 DOCTOR가 가장 유명함
- 실제로 대화의 내용을 이해하고 대답하는 프로그램이 아니었기 때문에 인공지능이라 불리기에는 성능 면에서 부족한 면이 많았음

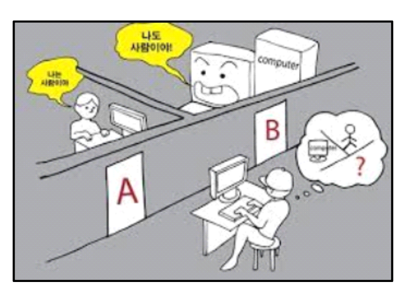
```
  Welcome to
                    EEEEEE  LL      IIII  ZZZZZZZ   AAAAA
                    EE      LL       II       ZZ   AA   AA
                    EEEEE   LL       II      ZZZ   AAAAAAA
                    EE      LL       II     ZZ     AA   AA
                    EEEEEE  LLLLLL  IIII  ZZZZZZZ  AA   AA

  Eliza is a mock Rogerian psychotherapist.
  The original program was described by Joseph Weizenbaum in 1966.
  This implementation by Norbert Landsteiner 2005.


ELIZA: Is something troubling you ?
YOU:   Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:   They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:   Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:   He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:   It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

http://psych.fullerton.edu/mbirnbaum/psych101/Eliza.htm

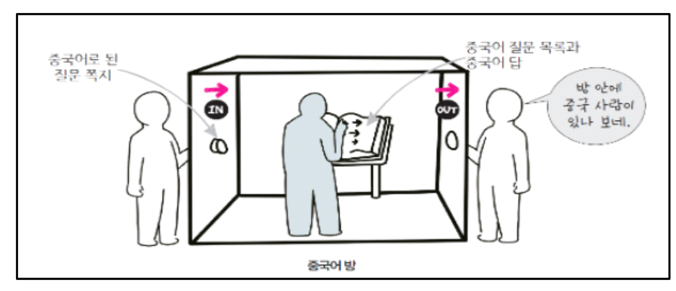
2) 튜링 테스트(Turing test)

- 심판의 역할을 하는 사람이 대화 상대를 보지 않은 채로 이야기를 나눈 뒤, 이때 인공지능을 사람이라고 판단할 경우 테스트를 통과하는 인공지능이라고 말할 수 있음

- 최근까지 이 튜링 테스트를 통과한 컴퓨터는 없었지만, 2014년에 처음으로 ‘유진 구스트만’이 튜링 테스트를 통과함
- 5분 정도 대화할 수 있으면 통과할 수 있으므로 진정한 의미에서의 인공지능이라 부르기엔 부족한 점이 많음



튜링 테스트



중국어 방

마. 2세대 열풍의 인공지능, 전문가 시스템

1) 전문가 시스템이란?

- 특정한 분야에서 특정 기능만을 수행하는 시스템을 뜻함
- 예: 자연스러운 대화로 호텔 예약을 도와주는 시스템, 특정 병명의 질환을 진단해내는 시스템 등
- 실제 사람과 같은 지능을 가지고 모든 것을 판단하는 인공지능이 아니라, 목적에 따라서 일부 기능만을 수행하는 좁은 범위의 인공지능임
- 사람처럼 유연하게 대응하거나 예측할 수 없었기 때문에 1세대 열풍의 인공지능처럼 2세대 열풍의 인공지능도 인공지능이라 불릴만한 성능을 가지지 못함
- 그러나 전문가 시스템은 현재 우리 실생활에서도 유용하게 쓰이고 있는 인공지능 기술임

바. 3세대 열풍의 인공지능, 머신러닝과 딥러닝

1) 3세대 열풍의 인공지능은 현재의 인공지능으로써, 머신러닝(Machine Learning, 기계학습)과 딥러닝(Deep Learning, 심층학습) 기술의 등장으로 인해 촉발됨

2) 머신러닝이란?

- 머신러닝이란 용어를 처음 사용한 아서 사무엘(1959)은 컴퓨터가 명시적으로 프로그램되지 않고도 학습할 수 있도록 하는 연구 분야라 정의함
- 즉, 인공지능을 만드는 방법론으로 컴퓨터가 데이터에서 규칙이나 패턴 등을 스스로 찾아내는 기술
- 머신러닝 기술을 통해 컴퓨터가 규칙에 부합하지 않는 미지의 데이터를 판단하고 이에 대응할 수 있게 됨
- 머신러닝에서의 ‘학습’이란 사람이 제공한 데이터를 바탕으로 컴퓨터가

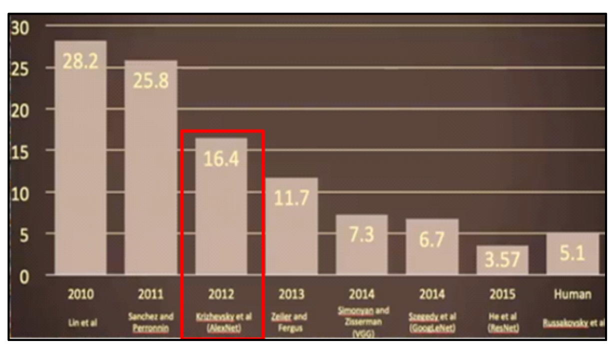
더 나은 규칙으로 수정해나가는 일련의 작업을 뜻함

전통적인 프로그래밍

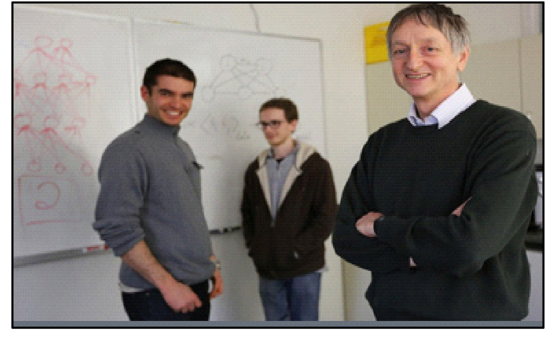
머신러닝

3) 딥러닝이란?

- 머신러닝의 한 분야로 뇌의 신경망의 작동원리를 본 따 컴퓨터(기계)에 구현한 학습기법임
- 컴퓨팅 파워의 향상, 많은 양의 데이터 축적, 알고리즘의 개선에 힘입어 2012에 ILSVRC라는 이미지 인식(image recognition) 경진대회에서 토론토 대학의 슈퍼비전 팀이 딥러닝으로 압도적인 성능 차이를 보이며 1위를 차지하게 되면서 역사의 전면에 등장함



딥러닝, 2012 이미지 인식 대회 우승


딥러닝 4대 천왕

- 머신러닝에서는 연구자가 컴퓨터의 학습에 교사 역할을 담당하며, 교사인 연구자가 정성스럽게 데이터를 손질하여 컴퓨터에 제공하는 것이 중요하지만, 딥러닝에서 연구자는 손질하지 않은 상태의 데이터를 그대로 컴퓨터에게 제공함
- 즉, 컴퓨터가 학습을 스스로 판단하고 진행하면서 앞으로의 상황을 예측하게 된 것임. 그런데도 학습한 결과의 성능이 기존의 머신러닝보다 높아짐
- 이러한 장점을 바탕으로 딥러닝은 사회 각 분야에서 다양하게 활용되기 시작함
- 딥러닝이 지닌 문제점 → 컴퓨터가 어떠한 알고리즘으로 분류하고 예측하는지 사람들이 분석할 수 없음(딥러닝의 블랙박스화 문제)
- 이로 인해 인공지능이 판단한 이유와 근거를 파악할 수 있는 신뢰할만한 인공지능 출현의 사회적·기술적 요구가 발생함 → 설명 가능한 인공지능(Explainable 인공지능: XAI) 분야의 출현

3. 머신러닝의 개념과 종류

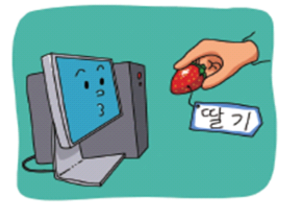
가. 등장 배경

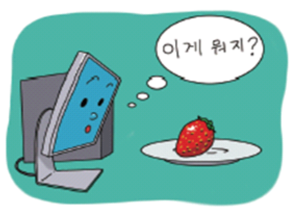
1) IT 기술이 발달하고 인터넷이 생활화되면서 사람들이 만들어내는 데이터(Data)의 종류가 다양(Variety)해짐

2) 그 용량(Volume) 또한 과거와는 비교할 수 없을 만큼 커졌으며, 데이터의 생성 속도(Velocity)까지 굉장히 빨라짐

3) 기술의 발전으로 인해 데이터를 보유하는 데 필요한 비용이 줄어들어 사람들이 과거에는 가질 수 없었던 빅데이터를 저렴한 비용으로 손쉽게 가질 수 있게 됨

나. 머신러닝이란 사람의 학습 과정을 그대로 기계(컴퓨터)에게 적용한 것임





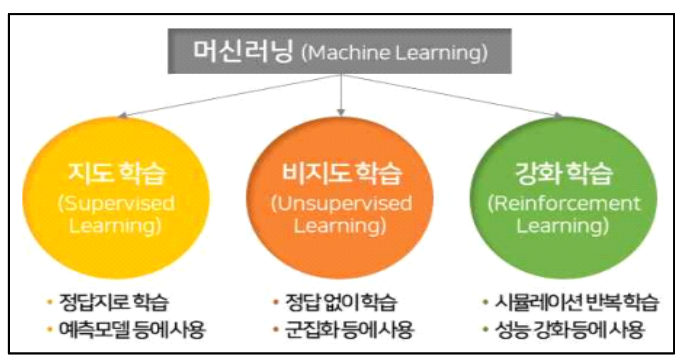
사람의 학습 과정　　　　　컴퓨터의 학습 과정

다. 기계(컴퓨터)가 학습하기 위해 필요한 준비물

1) 컴퓨터(PC, 노트북)

2) 데이터(수치화된 경험): 교재

라. 머신러닝의 종류



1) 지도학습(Supervised Learning)

- 사람이 데이터에 라벨을 붙인 뒤, 기계(컴퓨터)에게 이를 학습시키는 방법
- 기계(컴퓨터)는 사람이 라벨을 붙여 준 데이터(학습 데이터)를 학습하게 되고, 사람에게 지도를 받아(Supervised) 학습하는 것이 되기 때문에 ‘지도학습’이라는 명칭을 붙임
- 컴퓨터가 데이터를 학습하여 규칙을 파악하는 단계인 학습 단계, 파악한 규칙을 활용해 새로운 데이터를 예측하는 예측 단계로 크게 이루어짐

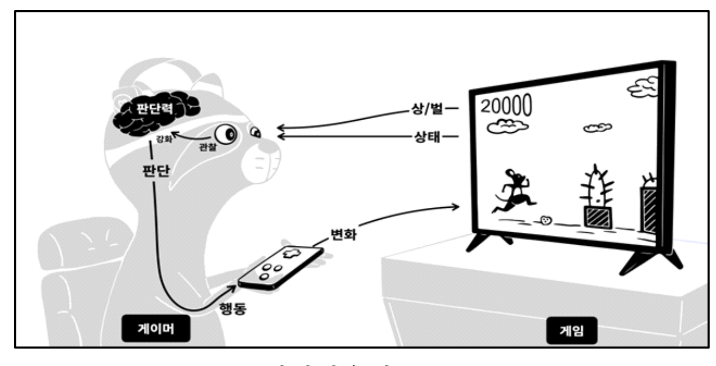
- 지도학습으로 해결할 수 있는 문제에는 크게 분류와 회귀가 있음
- 분류는 출력이 몇 가지 클래스(범주)로 주어지는 것이고, 회귀는 출력이 연속된 실수로 주어짐

2) 비지도학습(Unsupervised Learning)

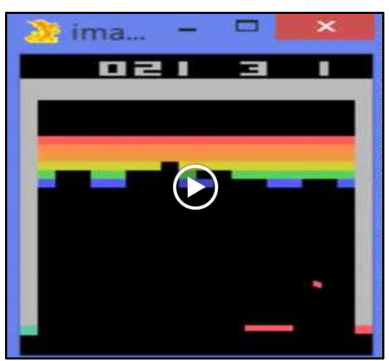
- 지도학습과는 달리 레이블(정답)이 없음
- 레이블(정답)이 없다는 것은 기계(컴퓨터)가 사람으로부터 지도를 받지 않기 때문에 '비지도학습'이라는 명칭을 붙이게 됨
- 레이블(정답)이 없으므로 이것이 '무엇이다'라고 분류할 수 없지만, 데이터를 비슷한 특징을 가진 것들로 군집화(Clustering: 데이터를 그룹화 하는 작업) 할 수 있음
- 라벨(정답)을 알 수 없거나 정답이 없는 데이터를 사용할 때 사용
- 클러스터가 어떠한 특징을 통해 성립되었는지는 사람이 유추해야 함

3) 강화학습(Reinforcement Learning)

- 특정 상태에서 취할 수 있는 다양한 행동을 평가한 후, 더 좋은 행동을 기계(컴퓨터)가 스스로 학습해가는 방법을 의미함
- 알파고(AlphaGo)로 인하여 화제가 되었던 바둑, 장기 등의 게임이나 로봇 동작 제어 분야에서 높은 성능을 발휘하고 있음
- 지도학습처럼 라벨(정답)을 제공하진 않지만, 선택할 수 있는 행동의 목록과 선택한 행동이 바람직한지 아닌지를 판단하는 기준은 사람이 직접 제공함
- 기계(컴퓨터)는 지정된 범위 내에서 시행착오를 통해 학습해 나가게 됨
- 로봇, 바둑과 장기, 아타리 게임, 벽돌 깨기 게임처럼 규칙이 정해져 있고 평가 기준을 제공할 수 있는 문제에는 적합함
- 구성 요소: 에이전트, 상태, 행동, 보상, 정책



강화학습의 요소



벽돌 깨기 게임

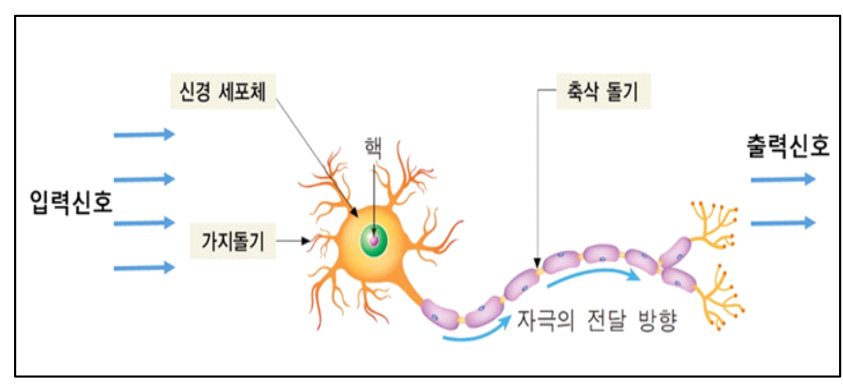
4. 퍼셉트론으로 살펴보는 인공신경망의 원리

가. 인공신경망(딥러닝)

1) 머신러닝의 한 분야인 인공신경망은 과거 두 차례의 겨울을 맞이하며 그 수명이 다해가는 듯 했으나 이미지 인식 분야에서 괄목할만한 성능의 개선을 이루어 내면서 딥러닝이라는 이름으로 다시 한번 도약하게 됨

2) 최근에는 머신러닝이라고 하면 딥러닝을 떠올리는 사람도 많지만, 모든 머신러닝이 딥러닝에 해당하지는 않음. 딥러닝은 머신러닝의 기법 중 하나임

3) 딥러닝의 기원이 되는 퍼셉트론(Perceptron)은 프랭크 로젠블랏(1957)에 의해 처음으로 고안되었는데, 생물학적 신경계(말초 신경의 구심성 뉴런 즉, 감각뉴런)의 원리를 단순화해서 이를 수학적으로 모델링(Modeling) 한 것임

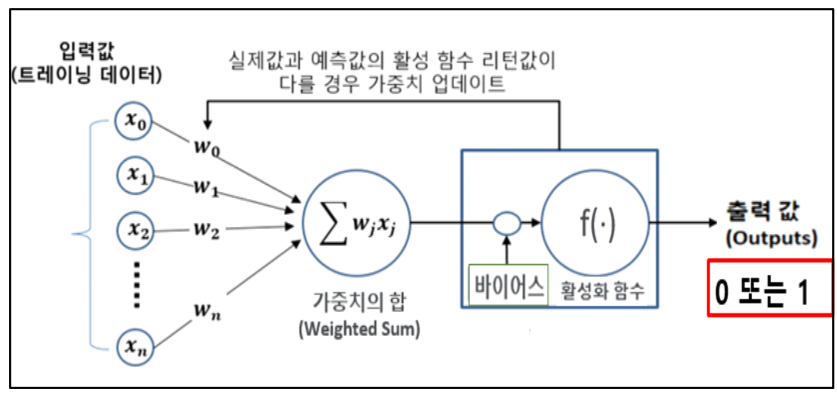


생물학적 신경계

4) 사람은 신경망을 통해 학습하고, 감정을 느끼며 생각하는데 사람의 뇌와 같이 컴퓨터에 신경망을 구현하는 기술을 인공신경망이라 함

5) 단층 퍼셉트론

- 여러 개로 입력된 수치를 가중치와 각각 곱한 후 모두 더한 값을 활성화 함수로 보낸 후, 활성화 함수를 통과한 값이 0 또는 1중 무엇을 출력해낼지를 결정하는 구조를 지님



단층 퍼셉트론

● 가중치(W)란? 가중치와 관련된 입력값이 얼마나 중요한지를 나타내는 지표

● X0과 관련된 가중치 W0의 값이 크면 클수록 X0이라는 입력값이 중요하므로 X0값을 결과에 더 많이 반영하게 됨

● 퍼셉트론의 가중합 수식:

$$y = (X1 \times W1) + (X2 \times W2) + \ldots + (Xn \times Wn) - bias = \sum_{i=1}^{n} Xi \times Wi - bias$$

· y>=0 이면 출력값 = 1, y<0 이면 출력값 = 0

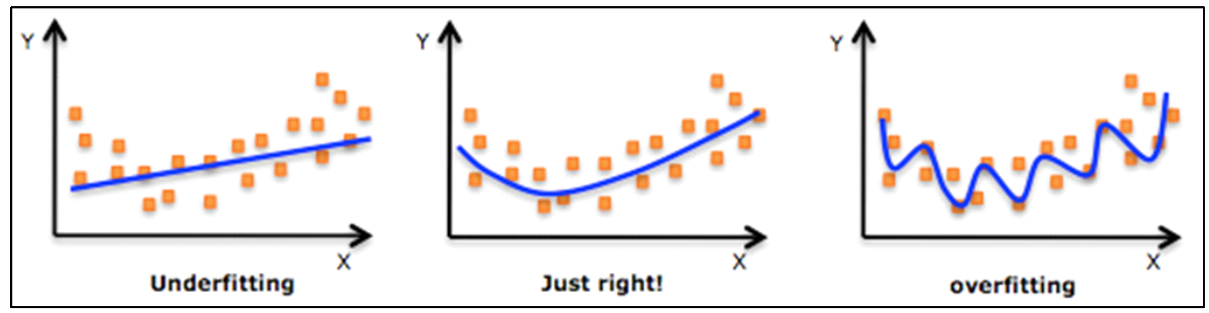
● 바이어스(Bias, 편향)란?

· 퍼셉트론의 출력값을 0과 1중 무엇으로 할지 결정하는 기준

· 예) '입력값의 합계가 100 이상일 때에는 1, 100 미만일 때는 0을 출력한다' 에서 판정 기준 100이 바이어스 값

· 바이어스의 값이 높을수록 모델이 간단해져 과소적합(Underfitting)의 위험이 발생

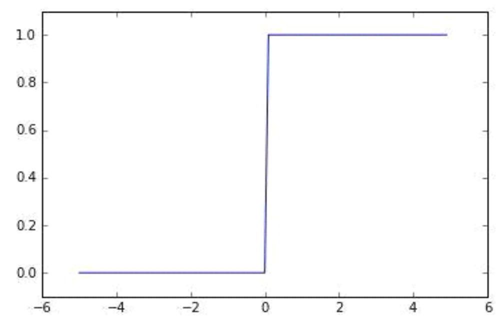
· 바이어스의 값이 낮을수록 학습 데이터에만 잘 들어맞는 모델이 되어 과대적합(Overfitting)의 위험이 있으며, 모델은 더욱 복잡해짐



● 활성화 함수(Activation Function)란?

· 입력신호의 총합(각각의 입력값과 가중치 값을 곱한 후 모두 더한 값)을 판정 기준에 따라 출력 신호로 변환하는 함수

· 단층 퍼셉트론에서는 계단 함수였으나 이후 시그모이드 함수로 변경됨

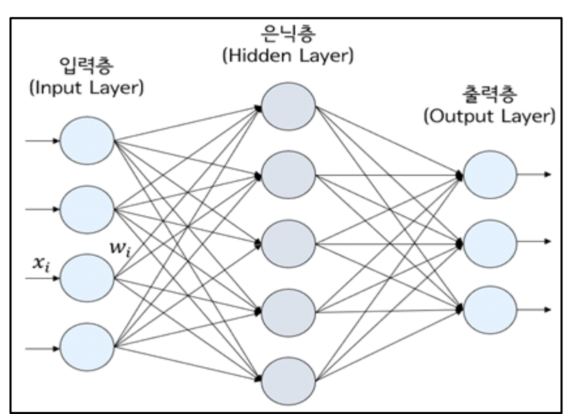


· 퍼셉트론에서는 계단(Step) 함수를 사용함

· 입력이 0보다 크면 1을 출력함(발화함)

· 입력이 0보다 작으면 0을 출력함(발화하지 않음)

6) 다층 퍼셉트론



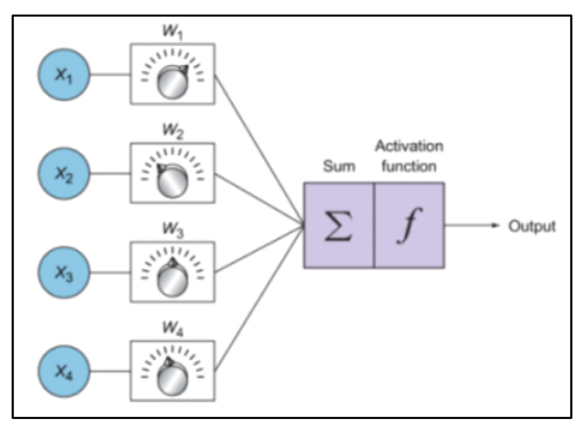
· 하나의 노드인 퍼셉트론을 기본 소자로 여러 개의 퍼셉트론을 병렬로 이어 만든 것

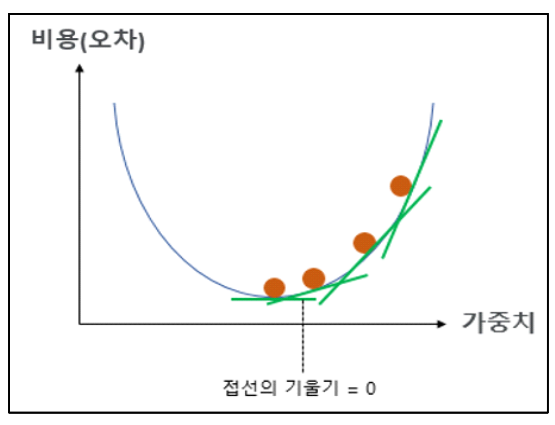
· 은닉층이 2층 이상인 신경망을 다층 퍼셉트론이라 함(= 뉴럴넷, 심층신경망)

· 인공신경망은 학습 데이터를 활용해 가중치와 바이어스의 값을 조정하면서 규칙과 패턴을 학습함

7) 퍼셉트론은 어떻게 학습을 할까?

- 퍼셉트론은 실수를 통해 배우는 시행착오 전략으로 학습을 함
- 가중치를 볼륨 손잡이처럼 조절하며 손실 함숫값을 최소가 되게 함





- 뉴런이 입력의 가중합을 계산한 뒤, 활성화 함수에 입력하여 예측값을 결정함 (이 과정을 순방향 계산이라고 함)
- 예측값과 실제 레이블값을 비교하여 오차를 계산함
- 오차에 따라 가중치를 조정함
- 오차가 아주 작아지도록 이 과정을 반복하여 최적의 가중치를 알아냄 (=학습)

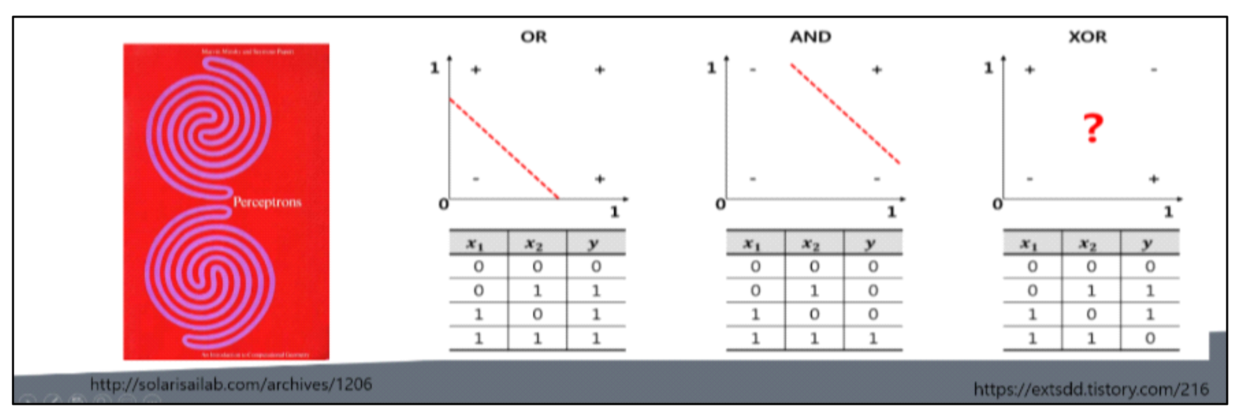
● 인공신경망의 겨울과 극복의 역사에 대해 살펴봅시다.

<<1차 겨울: XOR 문제>>

· 직선 하나로 분리할 수 있는 선형 데이터셋은 하나의 퍼셉트론으로도 문제를 해결할 수 있음

· 그러나 복잡한 데이터셋인 비선형 데이터셋은 하나의 퍼셉트론으로는 문제를 해결할 수 없으며, 현실 세계에는 이러한 데이터가 대다수임 → 성능 저하

예: XOR 문제(입력 값이 서로 다르면 1, 같으면 0을 출력) 해결 불가능



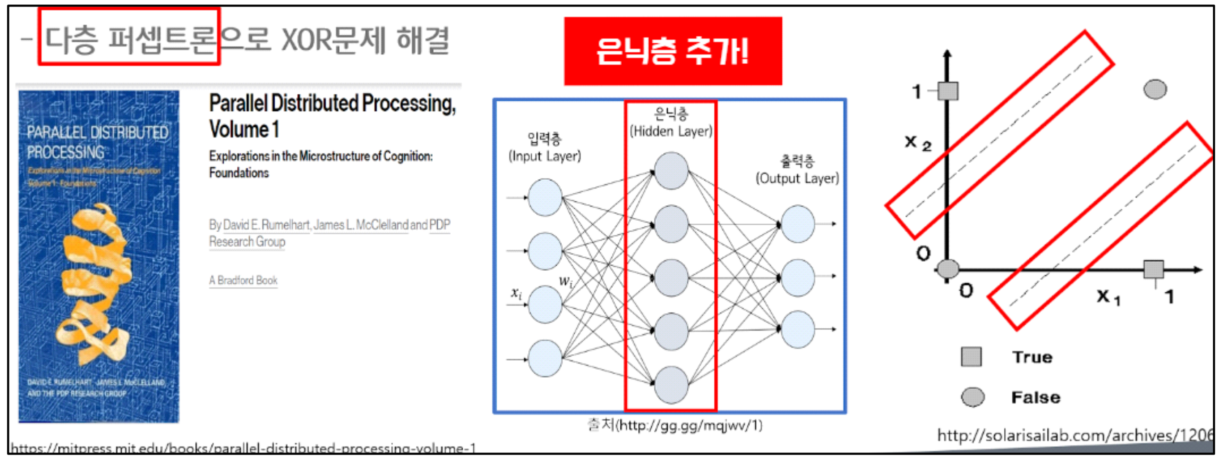
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

<<1차 겨울의 극복: 다층 퍼셉트론>>

· 은닉층(특징이 실제로 학습이 되는 곳)을 추가하여 여러 개의 퍼셉트론을 엮은 다층 퍼셉트론으로 XOR 문제를 비롯한 좀 더 복잡한 문제를 해결할 수 있게 됨



· 실습 URL: playground.tensorflow.org → 신경망 설계와 하이퍼파라미터 튜닝 (성능이 낮다면 층수와 노드 수를 늘리고, 학습 데이터에는 높은 성능을 보이지만 새로운 데이터에는 정확도가 떨어지는 과적합이 일어난다면 층수를 줄임)

<<2차 겨울: 다층 퍼셉트론 학습의 문제>>

· 그런데 은닉층이 추가되어 구조가 복잡해지니 학습을 어떻게 시켜야 할지에 대한 아이디어가 없었음

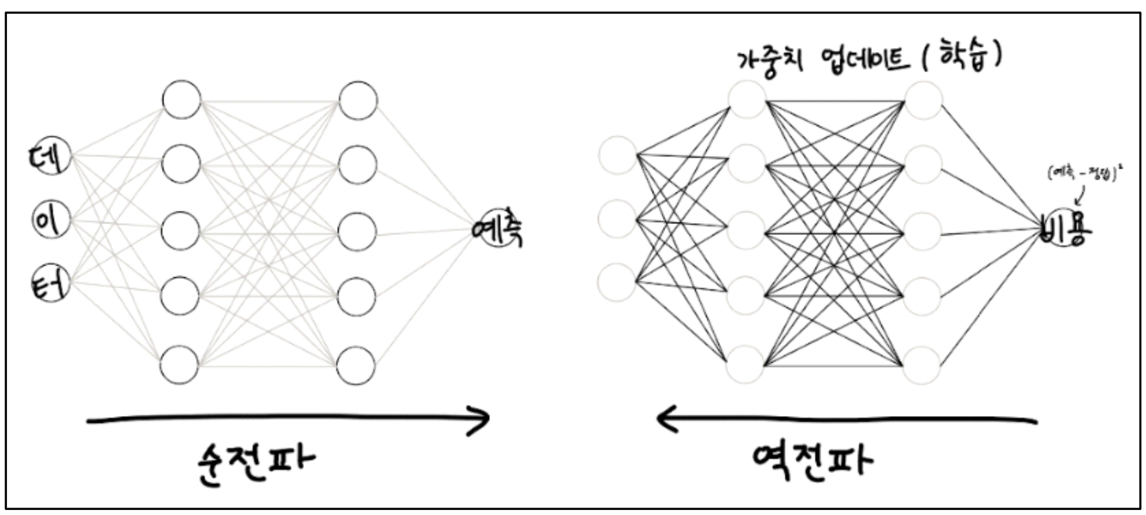
· 즉, 모델이 데이터를 입력으로 받아 층별로 계산을 한 뒤 예측값을 내놓는 과정인 순전파(forward propagation)는 큰 문제가 없었지만,

· 그 예측과 정답과의 거리를 계산하고 평균을 내어 “비용“을 계산한 뒤, 모델이 더 좋은 예측을 할 수 있는 방향으로 (비용이 적어지는 방향으로) 가중치들을

업데이트하는 방법에 대한 아이디어가 없었음

<<2차 겨울의 극복: 역전파 알고리즘>>

· 가중치를 조정하는 과정인 역전파(backpropagation)로 해결함 → 다층 퍼셉트론의 학습이 비로소 잘 됨

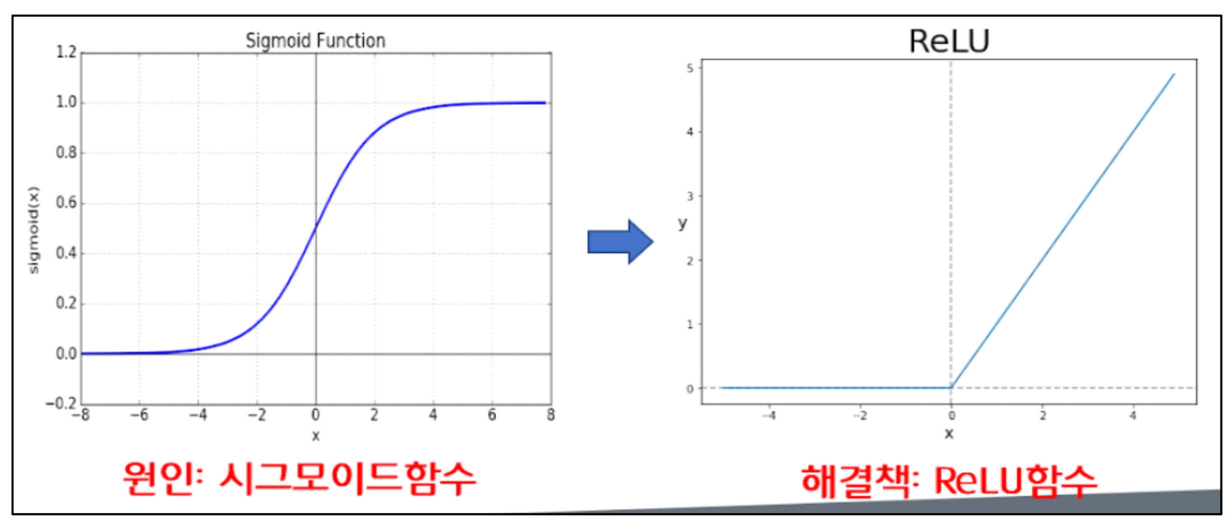


<<3차 겨울: 기울기 소실의 문제>>

· 그런데 층을 깊게 쌓으니까 역전파 과정에서 기울기가 사라지는 현상이 발생해 학습이 잘 안 됨 → 현실 세계의 문제에 괄목할 만한 성능을 보여주지 못함

<<3차 겨울의 극복: ReLU 함수>>

· 주로 사용하던 활성화 함수인 시그모이드 함수를 ReLU 함수로 바꾸어 사용 → 깊은 층의 신경망도 학습이 비로소 잘 됨(활성화 함수에 대해서는 다음 장에서 더 자세히 살펴봄)



· 이 외에도 컴퓨터 계산 속도의 향상, 알고리즘의 개선, 빅데이터의 등장으로 인해 인공신경망은 다양한 머신러닝 기법 중 주류가 됨
· 양치기 소년과도 같았던 역사를 지우기 위해 퍼셉트론, 인공신경망 등의 용어를 버리고, 현재는 딥러닝이라는 새로운 이름으로 부르고 있음

5. 딥러닝의 장점과 단점

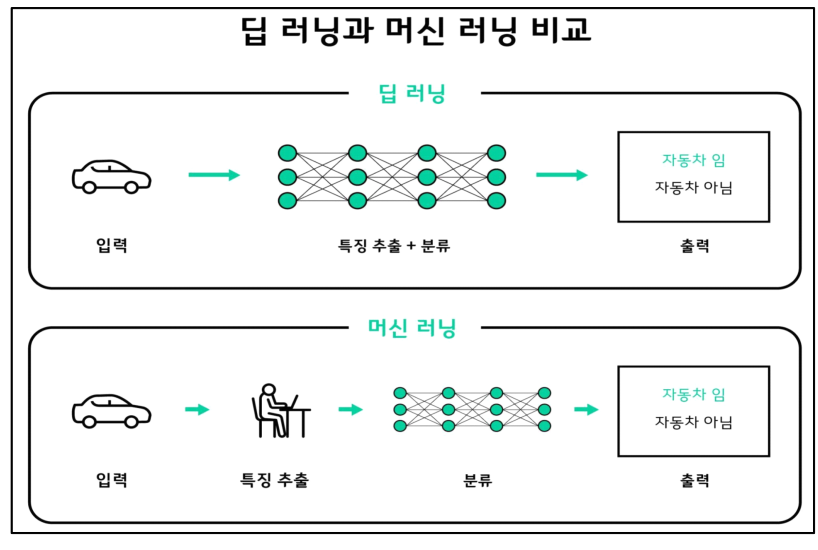
가. 장점

1) 사람이 아무런 힌트를 주지 않아도 정확한 데이터를 충분한 양만큼 제공하면 결과가 훌륭함
2) 머신러닝보다 전문가의 기술이 덜 필요하여 초보자에게 더 적합함

나. 단점

1) 딥러닝을 통해 추출해 낸 규칙의 블랙박스화 현상이 발생함
2) 머신러닝의 방법보다 더 대규모의 데이터가 필요함(부족한 데이터 문제)

6. 머신러닝과 딥러닝의 비교



· 머신러닝은 입력 데이터의 특징 추출을 사람이 담당하며, 이를 바탕으로 기계가 판단함
· 딥러닝은 입력 데이터의 특징 추출과 분류를 기계가 담당함. 즉, 머신러닝에 비해 사람의 수고가 덜 함

	머신러닝	딥러닝
훈련 데이터 셋의 크기	Small	Large
특징 추출	Yes	No
사용 가능한 분류기의 수	Many	Few
훈련 시간	Short	Long

7. 인공지능, 머신러닝, 딥러닝의 관계

가. 인공지능은 가장 포괄적 개념으로, 사람이 해야 할 일을 기계가 대신할 수 있는 모든 자동화에 해당함

나. 머신러닝은 인공지능을 구현하기 위한 방법으로, 규칙을 프로그래밍하지 않고, 데이터로부터 패턴을 기계가 스스로 학습하는 방법에 해당함

다. 딥러닝은 머신러닝 방법 중 인공신경망을 기반으로 한 모델로, 데이터의 특징 추출 및 판단까지 기계가 한 번에 수행하는 신경망 네트워크를 의미함

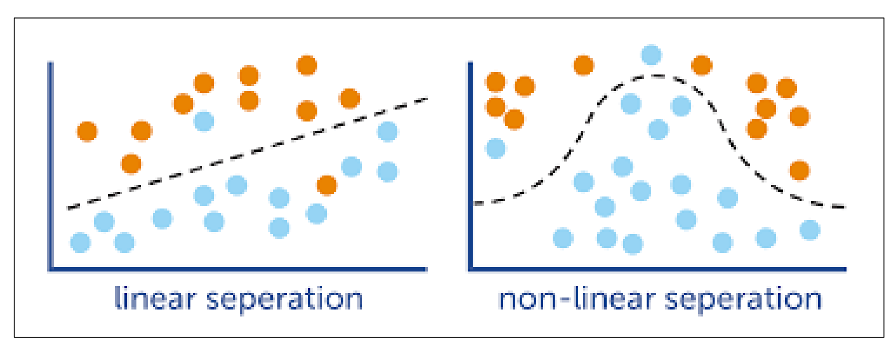
인공지능 (Artificial Intelligence)
- 가장 포괄적 개념
- 인간 지능과 연결된 인지 문제를 해결하는 학문

머신러닝 (Machine Learning)
- 인공지능 구현의 대표적 방법 중 하나

딥러닝 (Deep Learning)
- 머신러닝 방법 중 하나
- 인공 신경망(Artifical Neural Network) 이용

16 딥러닝과 인공신경망

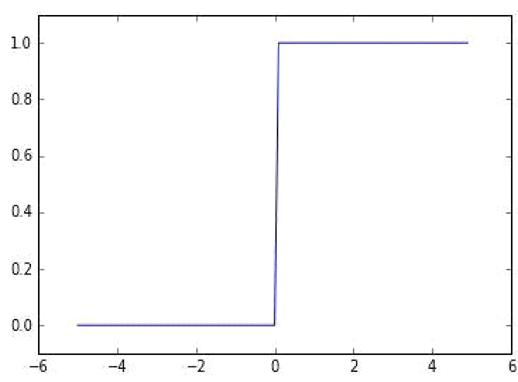
1. 여러 가지 활성화 함수
   가. 신경망 구조 설계에는 뉴런의 활성화 함수를 결정하는 것도 포함됨
   나. 활성화 함수는 각 퍼셉트론 내에 계산의 마지막 단계에 배치되어 뉴런의 발화 여부를 결정함
   다. 신경망에서 굳이 활성화 함수를 사용하는 이유는?
      1) 활성화 함수는 전이 함수 또는 비선형성이라고도 하는데, 그 이유는 활성화 함수가 가중합의 선형 결합을 비선형 모델로 만들기 때문임
      2) 아래 그림처럼 무 자르듯 일직선으로 구분할 수 없다는 점이 복잡계의 현실이며, 둥글게 곡선을 그릴 줄 알아야 세밀하게 분리할 수 있음



      3) 신경망에서 활성화 함수가 없다면 입력과 가중치의 곱셈 값을 모두 더하는 계산만 있어 선형 변환만 가능함. 즉, 은닉층이 아무리 깊고 복잡해도, 활성화 함수가 없으면 결국 곱셈과 덧셈의 향연이 되므로 하나의 선형 연산이 될 뿐임
         - 선형의 연산을 갖는 층을 수십개 쌓아도, 결국 하나의 선형 연산으로 나타낼 수 있음
         - 예: 입력값 x=2 => y=2x 통과 => y=4 출력 / 이 4를 다시 입력값으로 => y=4x+4 => y=20 출력
         - 최초의 입력값 2로 출력값 20을 만들었는데, 여러 층을 통과시키지 않더라도, y=10x라는 하나의 선형의 함수로 정리가 되어 버림. 즉, 딥러닝에서 층을 쌓은 혜택을 얻고 싶다면 활성화 함수 중에서도 비선형의 활성화 함수를 사용해야 함
      4) 딥러닝 모델은 활성화 함수 덕분에 이미지, 사진 등의 비선형 데이터를 잘 표현하고 다룰 수 있게 됨

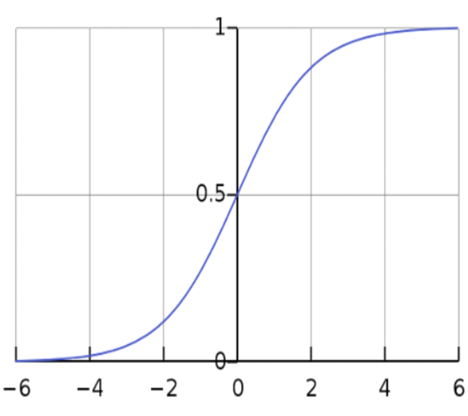
라. 활성화 함수의 종류

1) 계단 함수(Step function)



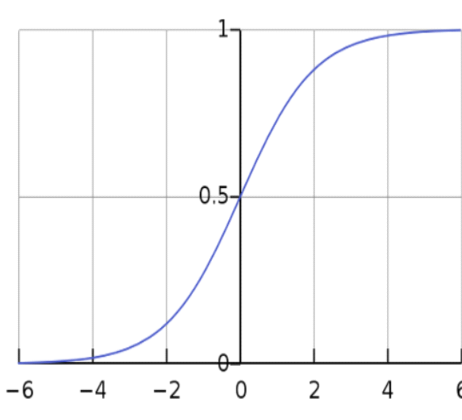
· 계단 함수는 0과 1, 두 가지 값만 출력함
· 입력이 0보다 크면 1을 출력함(발화함)
· 입력이 0보다 작으면 0을 출력함(발화하지 않음)
· 참 또는 거짓, 스팸메일 또는 스팸메일 아님, 합격 또는 불합격 등의 결과를 예측하는 이진 분류 문제에 주로 사용됨

2) 시그모이드 함수(Sigmoid function)



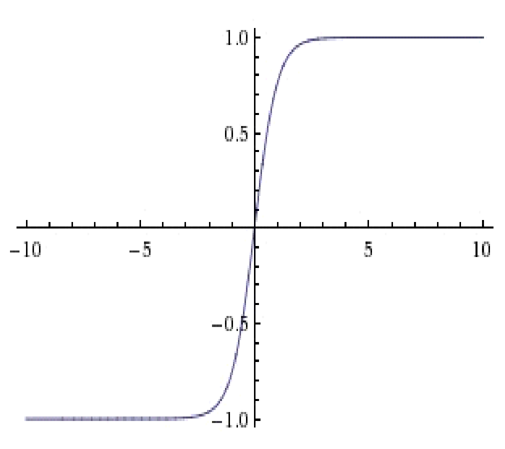
· 0과 1사이 구간의 값을 출력함
· 이진 분류에서 두 클래스의 확률을 구할 때 주로 사용함(출력층에서만 주로 사용)
· x값이 작거나 커질수록 기울기가 완만하게 0으로 수렴되어가는 특징이 있고, 이러한 점이 선형의 함수보다 현실을 더 잘 반영한다 여겼기 때문에 약방의 감초처럼 많이 활용함
· 기울기 소실 현상이 발생함

3) 소프트맥스 함수(Softmax function)



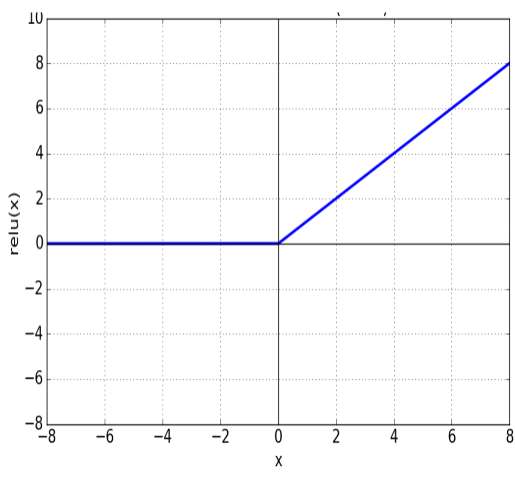
· 시그모이드 함수의 일반형으로, 3개 이상의 클래스를 대상으로 한 분류에서 각 클래스의 확률을 구할 때 사용함
· 소프트맥스 함수의 출력값은 입력값을 0과 1 사이의 값으로 변환하며, 총합은 항상 1임
· 클래스가 2개인 이진 분류에도 소프트맥스 함수가 유효하지만, 이 경우에는 시그모이드 함수와 동일하게 작동함

4) 하이퍼볼릭 탄젠트 함수(tanh function)



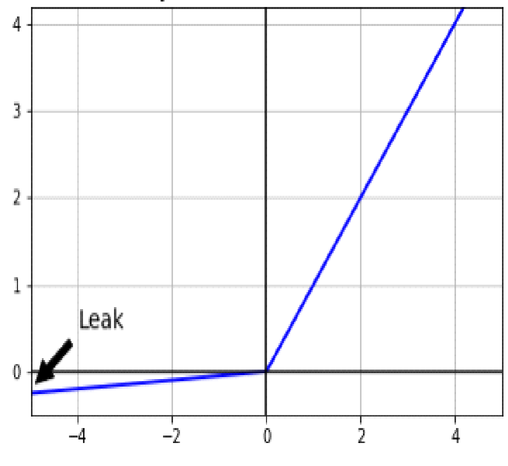
- -1 ~ 1 사이의 값을 출력함
- 시그모이드 함수와 거의 유사하며, 시그모이드 함수에 비해 은닉층에서 더 좋은 성능을 냄
- 왜? 데이터의 평균을 0으로 만들었기 때문에 데이터를 중앙에 모으는 효과가 있어 다음 층의 학습에 유리하기 때문임
- 하지만 시그모이드 함수가 갖고 있는 x값이 작거나 커질수록 기울기가 소실되는 문제는 고스란히 안고 있음

5) 렐루 함수(ReLU function)



- 입력값이 0보다 크면 입력값과 동일한 값을 출력하고, 0보다 작으면 발화하지 않음
- 다양한 상황에서 잘 작동하고, 은닉층에서 tanh 함수나 시그모이드 함수보다 더 높은 성능을 보여 현재 최고 성능의 활성화 함수로 평가받고 있음
- 은닉층에서 어떤 활성화 함수를 사용할지 모르겠으면 ReLU를 사용하면 된다고 여겨짐

6) 누설 렐루 함수(Leaky ReLU function)



- x가 음수일 때 기울기가 0이라는 ReLU 함수의 단점을 극복하기 위해 만든 ReLU 함수의 변종임
- x<0인 구간에서 함숫값이 0이 되지 않도록 작은 기울기(약 0.01)로 함숫값을 음수로 만들어 줌
- 상당히 드물게 사용되나 대부분의 경우 ReLU보다 높은 성능을 보임

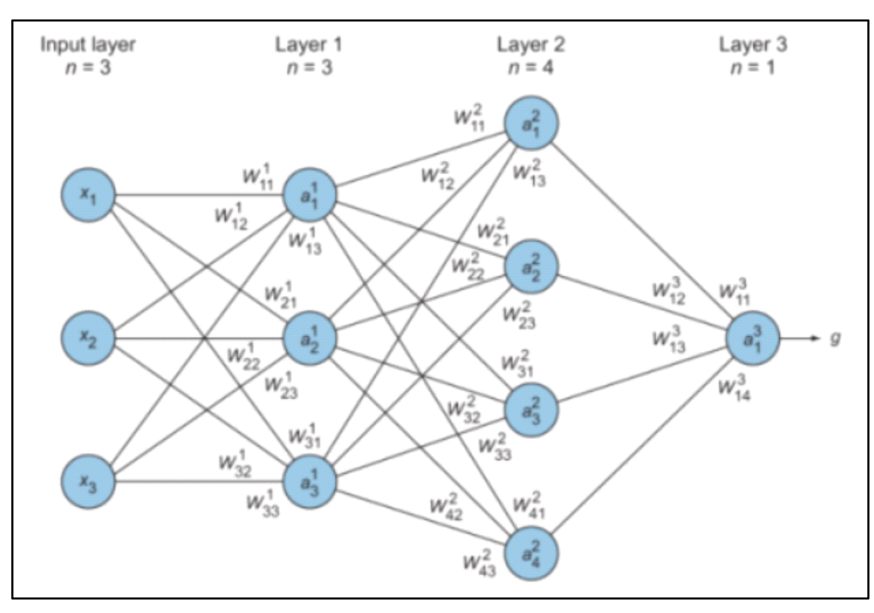
● 활성화 함수의 종류가 많은데 어떤 걸 골라 써야 하나요?

· 활성화 함수의 선택이 중요하기는 하나, 그렇게 어렵진 않음
· 일단 정해진 원칙대로 결정한 다음 모델을 튜닝하는 과정에서 필요에 따라 수정해나가면 됨
· 대략 예시가 될 수 있는 기준은 다음과 같음
 - 은닉층: 은닉층의 대부분의 경우에는 ReLU 함수를 사용하면 됨
 - 출력층: 대개의 경우에는 소프트맥스 함수가 적합하며, 이진 분류의 경우에는 시그모이드 함수도 좋은 선택임. 만약, 회귀 문제일 경우에는 활성화 함수를 사용하지 않아도 됨(굳이 비선형성을 도입할 필요가 없이 우리가 원하는 연속 값을 얻을 수 있으므로)

2. 오차 함수를 이용한 신경망 학습

가. 순방향 계산

1) 특징의 선형 결합을 활성화 함수에 통과시키는 계산 과정을 순방향 계산(Feedforward process) 이라고 함
2) 입력층에서 출력층 방향으로 정보가 흘러가기 때문에 순방향 계산이라고 부르며, 이 과정은 가중합과 활성화 함수를 연이어 계산하는 과정을 반복하는 형태로, 신경망의 각 층을 지나 예측에 이르는 과정임
3) 3개의 층을 가진 신경망을 예로 들어 순방향 계산을 살펴보자



3개의 층을 가진 간단한 구조의 신경망

- 첫 번째 층에서의 계산

$$a_1^{(1)} = \sigma(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3)$$
$$a_2^{(1)} = \sigma(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)} x_3)$$
$$a_3^{(1)} = \sigma(w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2 + w_{33}^{(1)} x_3)$$

- 두 번째 층에서의 계산

$$(a_1^{(2)},\ a_2^{(2)},\ a_3^{(2)},\ \text{and}\ a_4^{(2)})$$

- 마지막 세 번째 층에서의 예측 결과 계산

$$\hat{y} = a_1^{(3)} = \sigma(w_{11}^{(3)} a_1^{(2)} + w_{12}^{(3)} a_2^{(2)} + w_{13}^{(3)} a_3^{(2)} + w_{14}^{(3)} a_4^{(2)})$$

4) 굉장히 많은 노드와 층을 가진 신경망의 경우, 계산은 매우 복잡해지게 되는데 행렬을 이용하면 여러 개의 입력을 한 번에 계산할 수 있음
   - 이런 방법으로 큰 규모의 계산을 빠르게 수행할 수 있으며, 특히 넘파이 같은 도구를 활용하면 코드 한 줄로도 가능함

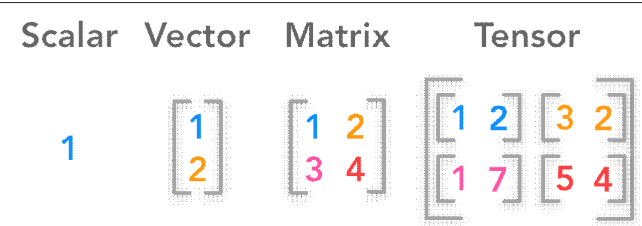
$$\hat{y} = \sigma \underbrace{\begin{bmatrix} w_{11}^3 & w_{12}^3 & w_{13}^3 & w_{14}^3 \end{bmatrix}}_{W^{(3)},\ \text{Layer 3}} \cdot \sigma \underbrace{\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \\ w_{41}^2 & w_{42}^2 & w_{43}^2 \end{bmatrix}}_{W^{(2)},\ \text{Layer 2}} \cdot \sigma \underbrace{\begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \end{bmatrix}}_{W^{(1)},\ \text{Layer 1}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{\text{Input vector}}$$

   - 위의 그림을 오른쪽부터 보면 입력을 쌓아 벡터를 만들고, 입력 벡터를 첫 번째 층의 가중치 행렬과 곱해 시그모이드 함수를 적용하고, 그 결과를 다시 곱하는 과정임

● 행렬 계산은 어떻게 하나요?

· 행렬의 차원에 대한 정의
  - 스칼라(Scalar): 하나의 숫자

- 벡터(Vector): 숫자의 배열
- 행렬(Matrix): 2차원 배열
- 텐서(Tensor): n차원 배열 (n > 2)



· 스칼라곱과 행렬곱

- 스칼라곱: 행렬의 모든 요소에 스칼라값을 곱하는 것으로, 스칼라곱은 행렬 크기에 영향을 미치지 않음

$$2 \cdot \begin{bmatrix} 10 & 6 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 10 & 2 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 3 \end{bmatrix}$$

- 행렬곱: (행1 × 열1) × (행2 ×열2)와 같이 행렬을 곱하려면 열1과 행2의 수가 일치해야 함

$$A \times B = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \end{pmatrix}$$

· 전치(Transposition)

- 열벡터에 전치를 적용하면 행벡터가 되고, 행벡터에 전치를 적용하면 열벡터가 됨(예; m×n → n×m)

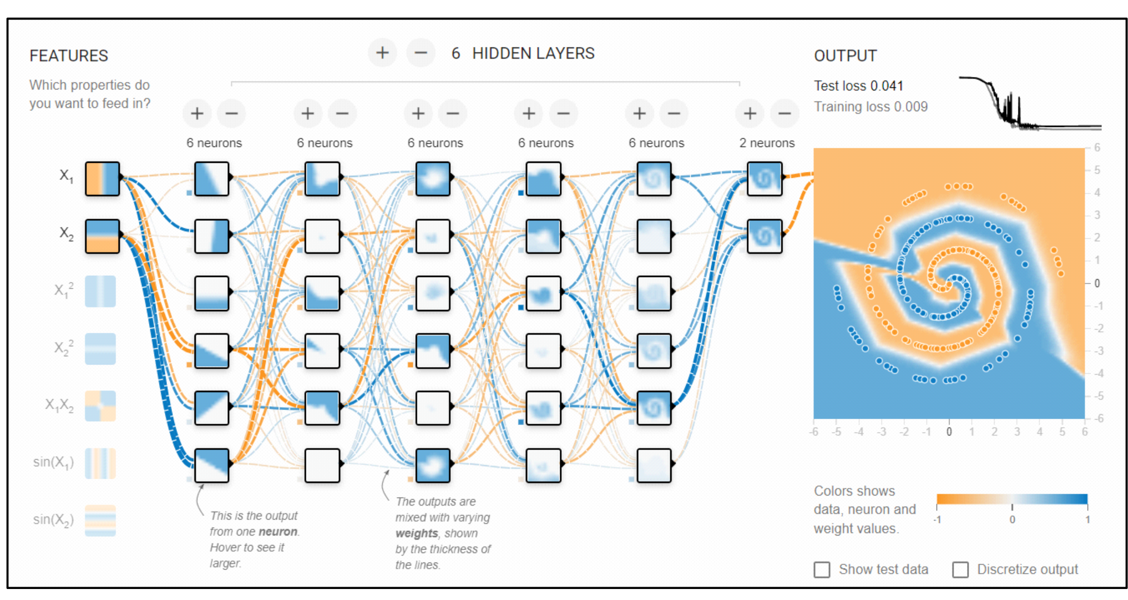
$$A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \Rightarrow A^T = [2 \;\; 8]$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \\ 1 & -1 \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 4 & -1 \end{bmatrix}$$

5) 특징 학습

- 은닉층의 각 노드는 각 층에서 학습된 새로운 특징임
- 즉, 각 층에서의 순방향 계산이 이루어질 때마다 신경망이 패턴을 학습해 새로운 특징들로 변환되며, 이러한 특징은 신경망이 부리는 마술임
- 각 층에서 새로 만들어진 특징들은 볼 수도 없고, 제약할 수도 없고, 이해할 수도 없는 것이기 때문에 이들을 은닉층이라고 부름



은닉층의 각 노드들은 각 층에서 새로 학습된 특징임

- 첫 번째 층에서는 선이나 모서리 같은 기본적인 특징이 학습되며, 두 번째 층에서는 꼭짓점 같은 조금 더 복잡한 특징이 학습됨
- 이런 식으로 마지막 층이 되면 원이나 나선 같은 데이터 셋과 부합하는 도형이 학습됨
- 즉, 신경망은 은닉층을 통해 데이터에 포함된 패턴을 인식하고 다시 패턴의 패턴, 패턴의 패턴의 패턴과 같은 식으로 추상화된 패턴을 인식함

나. 오차 함수

1) 오차 함수(Error function)란?

- 신경망의 예측 결과가 바람직한 출력과 비교해서 얼마나 동떨어졌는지를 측정하는 수단임
- 예: 손실 값이 크면 모델의 정확도가 낮으며, 손실 값이 작으면 모델의 정확도가 높다는 의미임. 손실이 클수록 정확도를 개선하기 위해 모델을 더 많이 학습시켜야 함
- 오차 함수는 비용 함수(Cost function) 또는 손실 함수(loss function)라고도 부르며, 통상 같은 의미로 사용됨

2) 오차 함수가 왜 필요한가?

- 최적화의 문제는 오차 함수를 정의하고, 파라미터(가중치)를 조정해서 오차 함수가 계산하는 오차를 최소가 되도록 하는 문제임
- 즉, 오차를 최소화하는 최적의 파라미터(가중치)를 찾는 것이 최적화 문제의 최종목표이며, 오차를 최소로 만드는 과정을 오차 함수 최적화라고 함
- 오차 함수는 신경망의 예측 결과가 얼마나 정답과 동떨어졌는지, 원하는 성능보다 얼마나 부족한지 알기 위해 사용하는 것

3) 오차의 값이 언제나 양수(+)여야 하는 이유는?

- 첫 번째 예측의 오차가 10이었고, 두 번째 예측의 오차가 -10이었다면 평균 오차는 0이 될 것임
- 하지만 오차가 0이면 신경망이 완벽하게 정답을 맞혔다는 의미인데, 실상은 그렇지 않음. 즉, 모든 예측의 오차는 양수여야 평균을 계산할 때 오차끼리 서로 상쇄하는 일이 발생하지 않음
- 빗나간 방향은 중요하지 않으며, 중요한 것은 목표물에서 얼마나 떨어져 있는가임

4) 평균제곱오차(Mean Squared Error, MSE)

- 출력값이 실수인 회귀문제에서 널리 사용함
- 단순히 (예측값 - 실제값)을 하는 것 대신 다음 식과 같이 각 데이터의 오차를 제곱해서 평균을 구함

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$$

- 차이가 커질수록 제곱 연산으로 인해 값이 뚜렷해지며 제곱으로 인해 오차가 양수이든 음수이든 누적 값을 증가시킴
- 최적 값에 가까워질 경우 이동 거리가 다르게 변화하기 때문에 최적값에 수렴하기가 용이하나, 제곱하기 때문에 1 미만의 값은 더 작아지고, 그 이상의 값은 더 커지는 값의 왜곡이 있을 수 있음
- 평균제곱오차의 변종 중, 오차의 절댓값의 평균인 평균절대오차(Mean Absolute Error)도 있음

$$MAE = \frac{1}{N}\sum_{i=1}^{n}\left|\hat{y_i} - y_i\right|$$

5) 교차 엔트로피(Cross Entropy)

- 두 확률 분포 간의 차이를 측정할 수 있다는 특성 때문에 주로 분류 문제에서 많이 사용함
- 예: 개가 그려진 이미지를 세 가지 클래스 중 하나로 분류하려고 할 때 이미지의 실제 확률 분포는 다음과 같음

```
1 Probability(cat)        P(dog)        P(fish)
2       0.0                 1.0           0.0
```

- 즉, 고양이일 확률 0%, 개일 확률 100%, 물고기일 확률 0%가 됨
- 그리고 머신러닝 모델의 예측 결과로 얻은 확률 분포가 아래와 같았다고 가정함

```
1 Probability(cat)        P(dog)        P(fish)
2       0.2                 0.3           0.5
```

- 실제 확률 분포와 예측 확률 분포는 얼마나 가까울까? 교차 엔트로피는 이 거리를 평가할 수 있음
- y가 대상 확률 분포, p가 예측 확률 분포, m이 클래스 수일 때 교차 엔트로피는 다음과 같이 정의됨

$$E(W,b) = -\sum_{i=1}^{m} y_i \log(p_i)$$

- 위 예제의 손실 계산은 다음과 같음
- E = - (0.0 * log(0.2) + 1.0 * log(0.3) + 0.0 * log(0.5)) = 1.2
- 신경망이 학습을 통해 예측이 좀 더 정확해졌다고 가정해보자. 예를 들어 신경망에 개가 찍힌 이미지를 입력했을 때, 처음에는 30%의 확률이었는데, 학습 후 50%로 확률이 올라갔다고 할 경우,

```
1 Probability(cat)        P(dog)        P(fish)
2       0.3                 0.5           0.2
```

- 다시 손실 값을 계산해보자

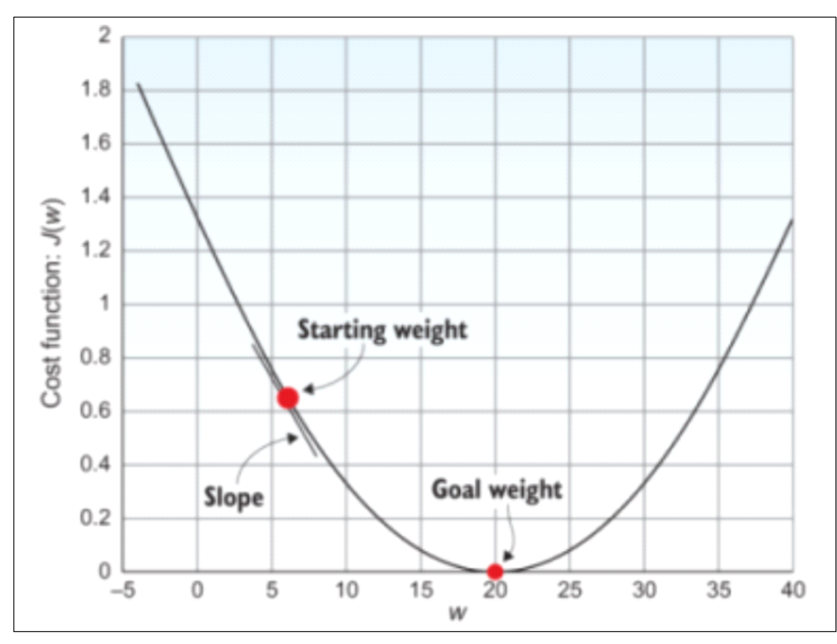
- E = - (0.0*log(0.3) + 1.0*log(0.5) + 0.0*log(0.2)) = 0.69
- 신경망의 예측이 조금 더 정확해지니 손실 값이 1.2에서 0.69로 감소함
- 이상적인 결과로 신경망의 예측 결과가 개일 확률이 100%로 나왔다면 손실 값은 0이 됨

● 여기서 잠깐!

- 오차 함수의 계산은 직접 계산할 필요가 없음
- 하지만 원리를 이해해야 신경망을 설계할 때 내부 동작을 고려할 수 있음
- 딥러닝 프로젝트에서는 주로 라이브러리를 사용하며, 이들 라이브러리에서는 일반적으로 파라미터 형태로 오차 함수를 선택하여 사용함

6) 오차와 가중치의 관계

- 신경망이 학습을 하려면 오차 함수의 함숫값을 가능한 최소가 되게 해야 함(0이 이상적임)
- 오차가 작을수록 모델이 출력한 예측값의 정확도가 높은 것임
- 오차값을 최소가 되게 하려면 변수(파라미터)인 **가중치**를 조작해야 함. 신경망은 **가중치를 조정**하는 방식으로 학습함
- 가중치의 변화에 대한 오차 함숫값의 변화를 그래프로 나타내면 다음과 같음



- 가중치는 무작위 값으로 초기화되며, 우리가 할 일은 곡선을 따라 내려가 오차가 최소가 되는 목표지점에 도달하는 것임

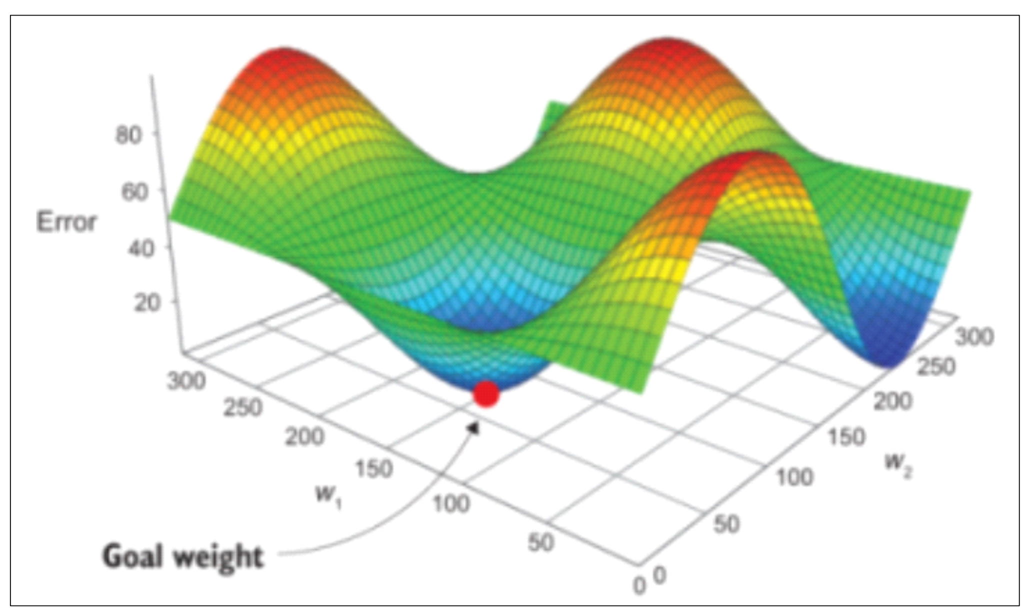
- 이 목표지점에 해당하는 최적 가중치(= 오차가 최소가 되게 하는 가중치로 최적화)를 찾아가는 과정은 최적화 알고리즘을 이용해 반복적으로 가중치를 수정해나가는 과정임

다. 최적화 알고리즘

1) 최적화는 어떤 값을 최소화하거나 최대화하는 것으로 문제를 바라보는 관점임
2) 오차 함수의 가장 큰 이점은 신경망의 학습을 오차를 최소화하는 최적화의 문제로 바꿀 수 있다는 것임
3) 최적화란, 어떤 값이 최소(혹은 최대)가 되도록 파라미터를 수정하는 것임

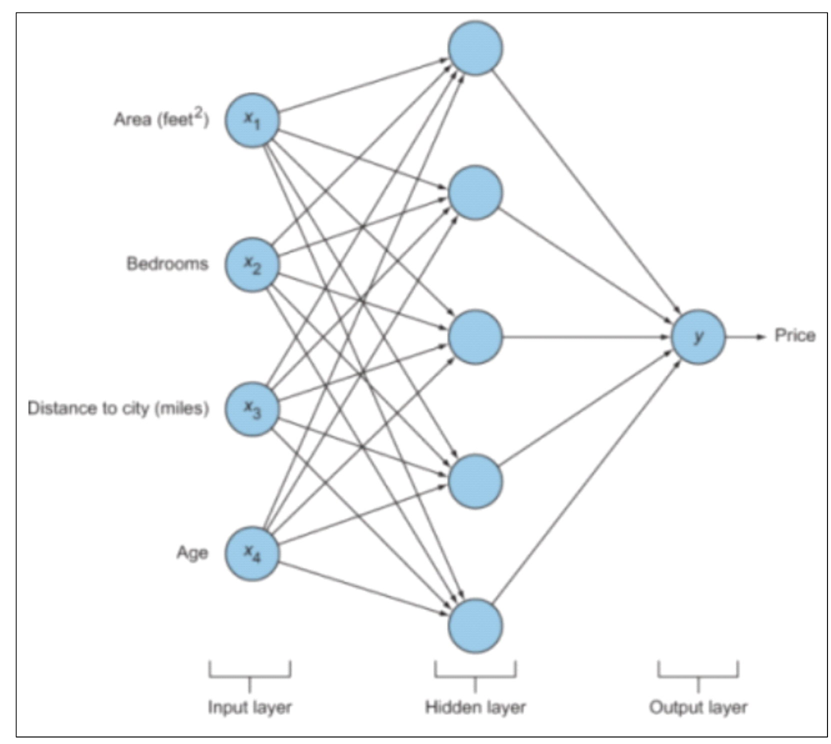
● Tip
- 신경망에서 오차 함숫값을 최적화하려면 가중치와 편향을 수정해가며 최적 가중치를 찾아야 함

4) 단일 퍼셉트론의 가중치에 대한 오차 함수의 변화는 2차원 그래프로 단순하게 나타낼 수 있으나, 가중치가 2개라면 3차원 평면 그래프를 아래와 같이 얻음



5) 여기서 신경망의 구조가 복잡해질수록 가중치의 개수는 수백에서 수천에 이르게 됨
6) 사람의 감각으로는 3차원 이상의 공간을 이해할 수 없으므로 가중치가 10개나 되는 오차 함수의 그래프를 시각화하는 것은 불가능함(그러므로 여기서는 2차원 또는 3차원으로 표현된 오차 평면을 이용해서 설명함)

7) 최적화 알고리즘은 왜 필요할까? 가중치의 모든 가능한 값을 시도해보며 오차의 최소 지점을 찾으면 되지 않을까?



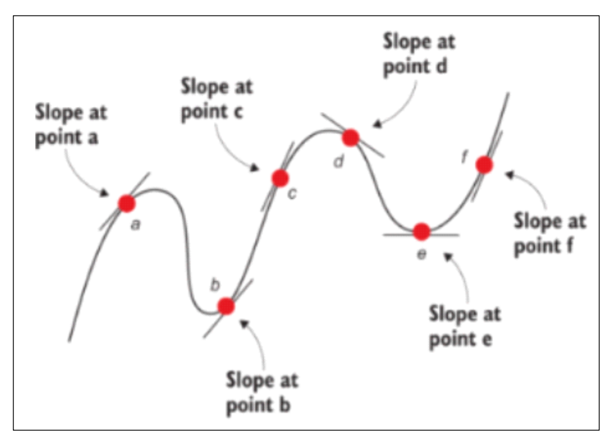
- 위 그림을 보면 입력층과 출력층 사이에 20개의 에지(가중치)가 있고, 여기에 더해 은닉층과 출력층 사이에 다시 5개의 가중치가 더 있으므로 모두 25개의 가중치를 최적화해야 함
- 만약, 1,000가지 값에 대해 가능한 모든 값을 시도해보며 오차가 최소가 되는 지점을 찾는다고 하면 $1{,}000^{25}$ 즉, 10^{75} 조합을 계산해보아야 함
- 이를 계산하려면 우주가 존재했던 시간보다 더 긴 시간이 필요하므로 모든 값을 시도해보는 방식은 현실적으로 사용하기 어려움
- 그러나 최적화 알고리즘이면 몇 분이면 최적화가 가능함

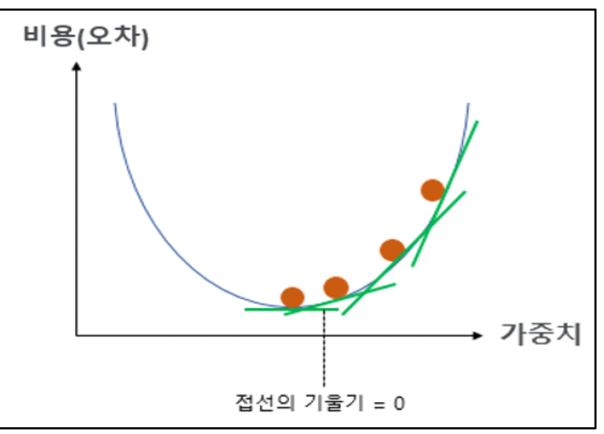
● 신경망에서 가장 널리 사용되는 최적화 알고리즘, 경사 하강법

- 경사 하강법에는 배치 경사 하강법, 확률적 경사 하강법, 미니배치 경사 하강법 등 몇 가지 변종이 존재함

8) 배치 경사 하강법

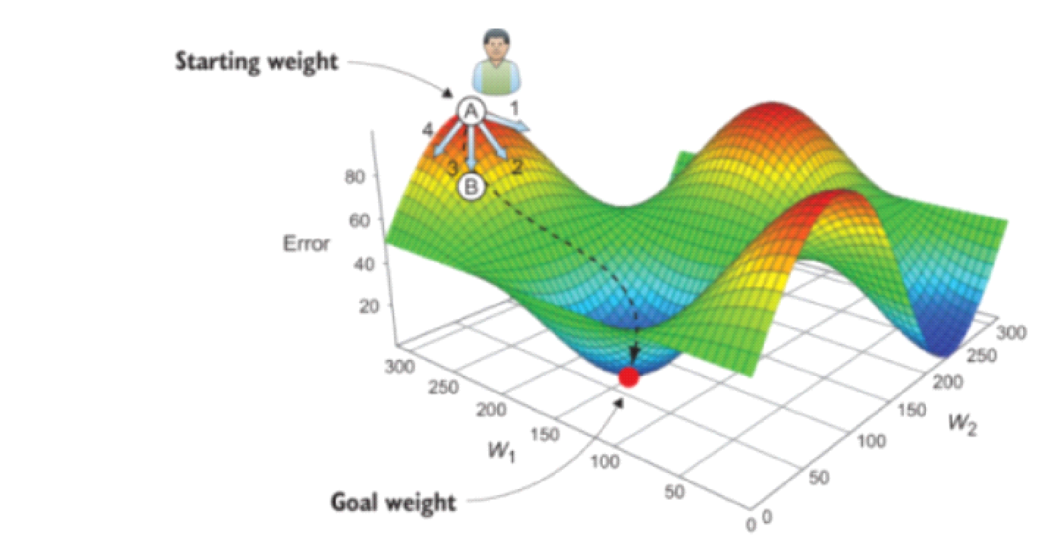
- 경사(=미분)는 곡선의 특정한 지점에서 곡선에 대한 접선이 갖는 변화율 또는 기울기를 나타내는 함수를 의미함
- 즉, 간단히 말하면 곡선의 기울기나 가파름을 나타냄





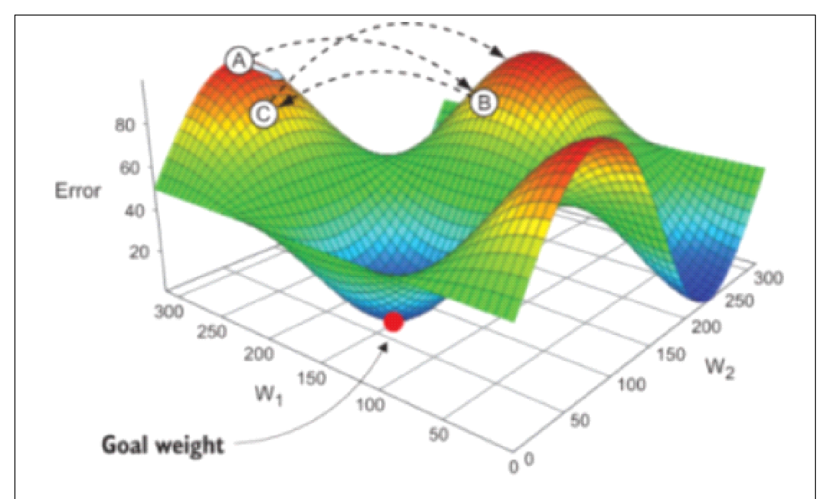
- 경사 하강(Gradient descent)은 가중치를 반복적으로 수정하며 오차 함수의 최저점에 도달할 때까지 오차 함수의 언덕을 내려가는 과정임
- 즉, 가중치의 초깃값에 대해 미분을 구해 오차 함수의 기울기를 계산하고, 이 방향으로 가중치를 수정하는데, 오차의 최소점에 도달할 때까지 이 과정을 반복함

● 경사 하강법의 원리에 대해 알아보자!



- 가중치의 초깃값은 무작위로 결정한 A임
- 우리의 목표는 오차 함수 그래프의 언덕을 내려가 오차 함숫값이 최소가 되는 가중치 w1과 w2에 도달하는 것임
- 오차 함수 그래프의 언덕을 내려가려면 한 걸음마다 걸음의 방향(경사), 보폭(학습률)을 알아내야 함

- 언덕 아래로 내려가려면 경사가 가장 가파른 방향으로 걸음을 내딛어야 함(이 경사는 오차 함수의 미분이며, 어느 방향이 가장 가파른지 1,2,3,4 중 판단해야 함)
- 3을 선택하면 지점 B에 다다르게 되며, 언덕을 충분히 내려갈 때까지 (순방향 계산 및 오차 계산 후 가장 가파른 방향을 찾아 이동하는) 과정을 반복함 → 이렇게 경사를 통해 걸음의 방향을 결정함
- 학습률은 경사 하강법 수행 중 가중치를 수정할 때 이동할 보폭에 해당하며, 그리스 문자 알파(α)로 나타냄
- 학습률의 값이 크면 오차 함수의 언덕을 큰 보폭으로 내려가므로 학습이 빠르게 진행되며, 값이 작으면 학습 속도가 느려짐
- 그렇다고 학습률을 너무 크게 설정하면 오차값이 진동하기만 할 뿐, 감소하지 않음



- 즉, 학습률이 아주 작으면 오차의 최소점에 도달할 수 있지만 학습 시간이 오래 걸리며, 학습률이 너무 크면 오차가 진동하여 학습이 잘 진행되지 않음(보통 학습률은 0.1 또는 0.01을 초깃값으로 설정한 후, 점차 낮춰감)

- 방향(경사)과 보폭(학습률)을 곱하면 각 단계의 가중치 변화량을 계산할 수 있음. 이때, 기호가 마이너스인 이유는 우리는 경사를 따라 내려가야 하기 때문임(경사의 반대 방향으로 이동)

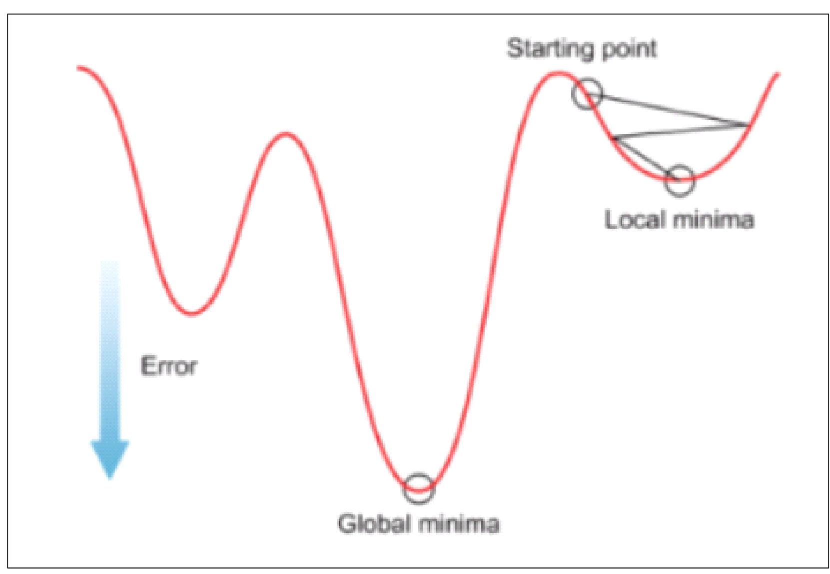
$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

보폭 × 방향

$$w_{next-step} = w_{current} + \Delta w$$

● 배치 경사 하강법의 문제점

- 첫 번째, 손실 함수 중에는 단순한 사발 모양이 아닌 경우도 많음. 구멍이나 벼랑 등 최소점에 접근하는 것을 어렵게 하는 특이한 지형들이 존재함
- 여러 개의 지역 최소점을 가진 경우, 우리의 목표인 전역 최소점에 도달하기 어려울 수 있음



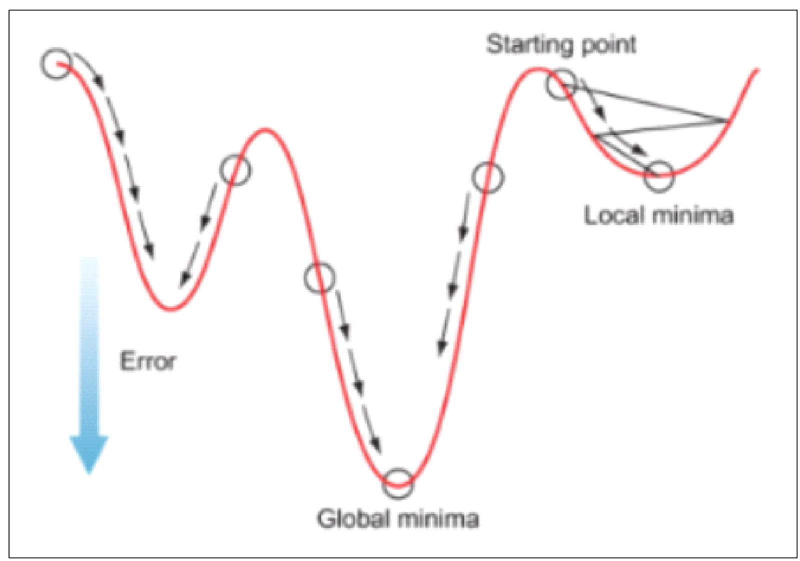
- 두 번째, 경사를 계산하기 위해 매번 훈련 데이터 전체를 사용한다는 점이 문제임
- 훈련 데이터 수가 1억개이면 가중치를 한 번 수정하기 위해 1억 개의 손실 값을 합하여 평균을 내야 하는데, 계산 비용이 너무 크며 속도도 느려짐

※ 훈련 데이터를 하나의 배치(batch)로 사용하기 때문에 배치 경사 하강법이라 부름

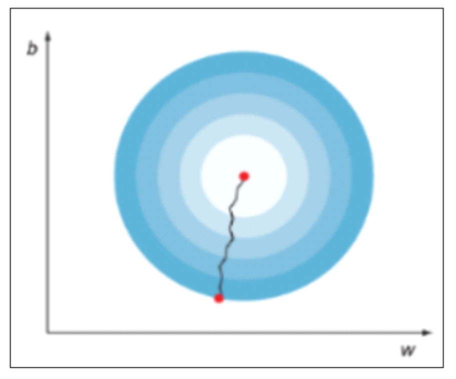
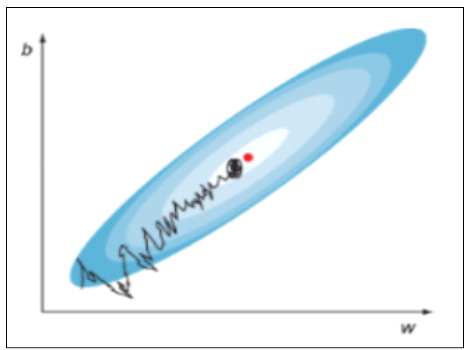
→ 이러한 문제들을 해결하기 위해 고안된 것이 확률적 경사 하강법임

9) 확률적 경사 하강법

- 무작위로 데이터 점을 골라 데이터 점 하나를 이용해 가중치를 수정하는 방법임
- 가중치에 다양한 시작점을 만들 수 있고, 여러 지역 극소점을 발견할 수 있음
- 이렇게 찾은 여러 지역 극소점 중 가장 작은 값을 전역 최소점으로 삼음



- '확률적' 은 '무작위' 로 선택했다는 의미임
- 경사 하강법이 전체 훈련 데이터에 대해 손실과 경사를 계산하여 가중치를 수정하는데 비해 확률적 경사 하강법은 훈련 데이터 중 하나를 무작위로 선택하여 이 데이터에 대한 경사를 계산함

경사 하강법	확률적 경사 하강법
1. 훈련 데이터 전체를 입력함 2. 경사를 계산함 3. 가중치를 수정함 4. n번의 에포크 동안 반복함  ※ 오차 함수를 따라 하강하는 경로가 매끄러움	1. 훈련 데이터를 무작위로 섞음 2. 데이터를 하나 선택해서 입력함 3. 경사를 계산함 4. 가중치를 수정함 5. 또 다른 데이터를 하나 선택해서 입력함 6. n번의 에포크 동안 반복함  ※ 오차 함수를 따라 하강하는 경로가 진동하는 패턴을 보임

● 왜 전역 최소점으로 내려가는 경로가 지그재그일까?

- 확률적 경사 하강법에서는 훈련 데이터 하나를 대상으로 가중치를 수정하므로 계산 속도는 빠르지만 각각의 가중치 수정이 정확하게 전역 최소점을 향하지 않기 때문임
- 보통 전역 최소점에 다가가는 정도로 충분하므로 이 점은 실제로 문제가 되지는 않음
- 거의 대부분 확률적 경사 하강법이 배치 경사 하강법보다 빠르고 높은 성능을 보임

10) 미니배치 경사 하강법
- 배치 경사 하강법과 확률적 경사 하강법의 절충안임
- 경사를 계산할 때, 모든 훈련 데이터나 하나의 훈련 데이터를 사용하는 대신 훈련 데이터를 몇 개의 미니배치(32, 64, 128, 256 정도로 설정하며 256이 가장 흔함)로 분할한 다음, 이 미니배치로부터 경사를 계산하는 방법임
- 배치 경사 하강법에 비해 가중치 수정 횟수가 더 많은 만큼 더 적은 반복 횟수에서 가중치가 수렴하며, 확률적 경사 하강법에 비해서도 벡터 연산을 할 수 있어 계산 효율이 좋음

● 배치 크기와 에포크란?

- 에포크는 학습 과정에서 전체 훈련 데이터가 한 번씩 입력되는 동안을 의미함
- 배치 크기는 경사를 한 번 계산하기 위해 입력되는 훈련 데이터의 수를 의미함
- 예: 훈련 데이터 수가 1,000이고 배치 크기가 256이면 1 에포크는 크기가 256인 배치 3개와 232인 배치 1개로 구성됨

3. 역전파 알고리즘

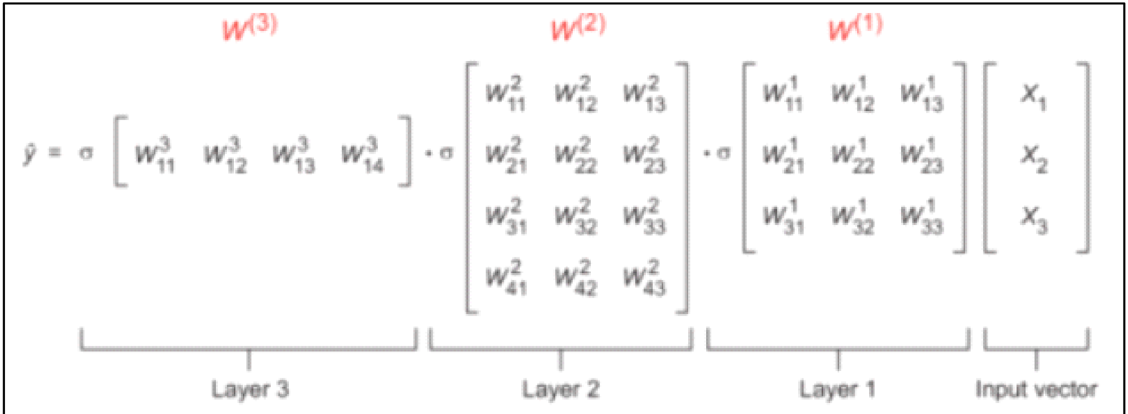
가. 역전파(backpropagation)란?

1) 가중치 수정을 위해 가중치에 대한 오차의 미분을 출력층부터 첫 번째 층까지 전달하는 것을 말함
2) 역전파 계산을 통해 오차 함수의 언덕을 내려오는 길을 한 걸음 내딛게

되며, 결국에는 오차 함수의 최소점에 다다르게 됨

● 지금까지 배운 신경망 학습

- 순방향 계산: 가중합을 계산하고, 이를 활성화 함수에 넣어 출력값 계산



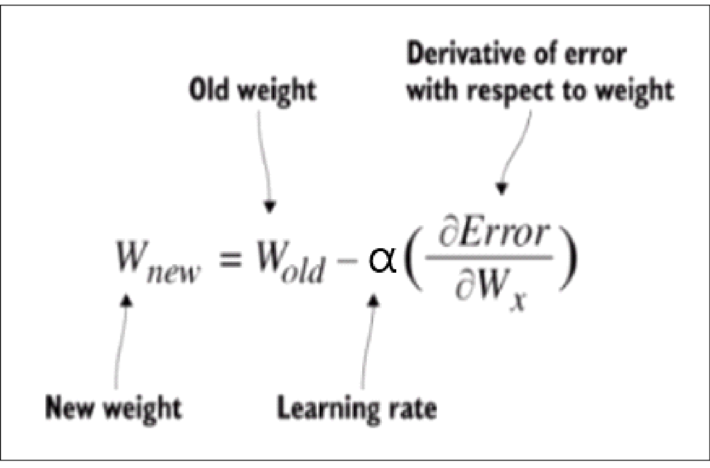
- 출력값과 정답을 비교해서 오차 함수 또는 손실 함수를 계산함

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

- 경사 하강법 알고리즘을 사용해서 △w를 계산하고 오차 함숫값을 최적화함

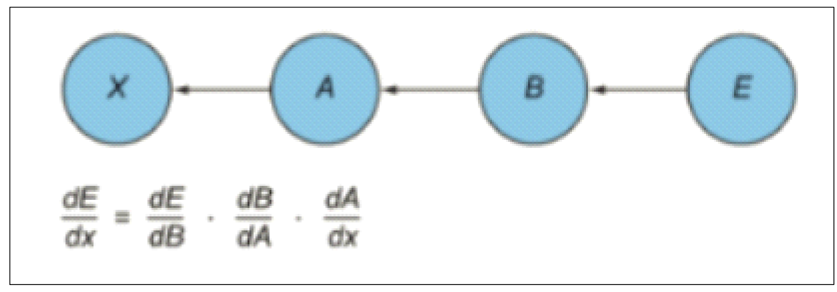
$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

- △w를 신경망 전체에 반대 방향으로 역전파하며, 가중치를 수정함

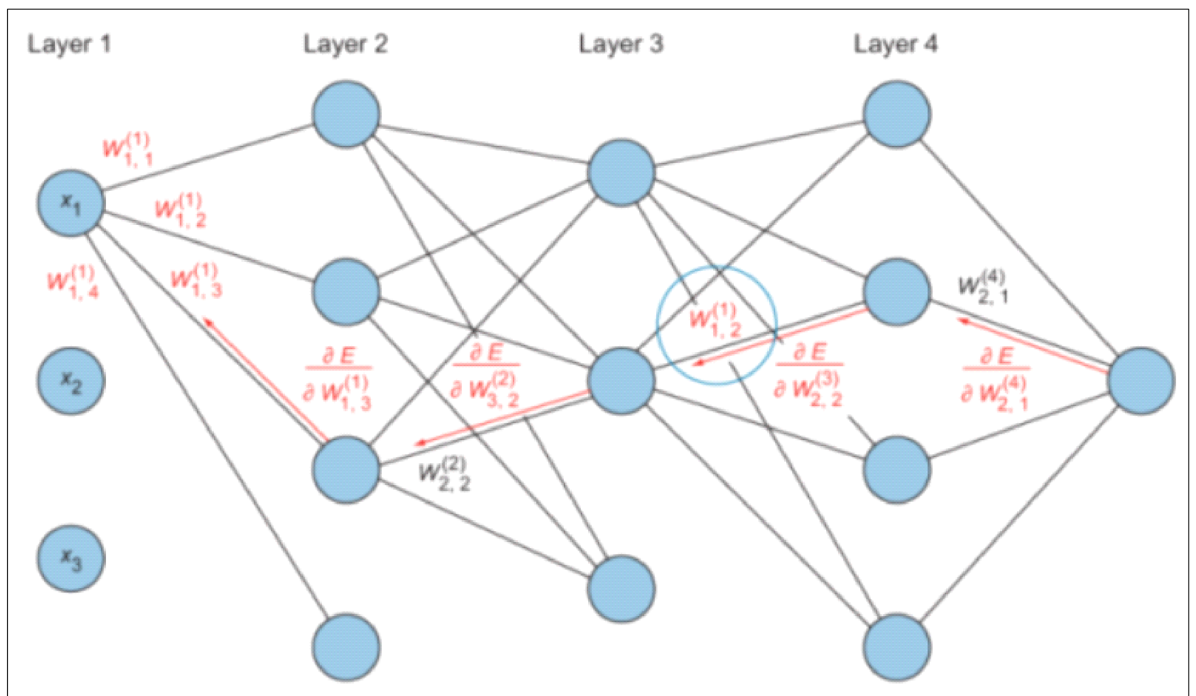


3) 다층 퍼셉트론처럼 가중치 수가 늘어나면 역전파 계산 과정이 복잡해짐

4) 오차를 입력층까지 전파하는 것이 역전파의 목표임을 기억하며, 오차에 미치는 입력(x)의 영향인 $\frac{dE}{dx}$를 계산하면 다음과 같음



5) 역전파 계산은 연쇄 법칙을 이용해서 경사를 신경망 전체에 전달함



- 연쇄 법칙을 적용해서 첫 번째 입력의 세 번째 가중치 $W^{(1)}_{1,3}$에 대한 오차의 경사를 계산하면 다음과 같음

$$\frac{dE}{dw^{(1)}_{1,3}} = \frac{dE}{dw^{(4)}_{2,1}} * \frac{dw^{(4)}_{2,1}}{dw^{(3)}_{2,2}} * \frac{dw^{(3)}_{2,2}}{dw^{(2)}_{3,2}} * \frac{dw^{(2)}_{3,2}}{dw^{(1)}_{1,3}}$$

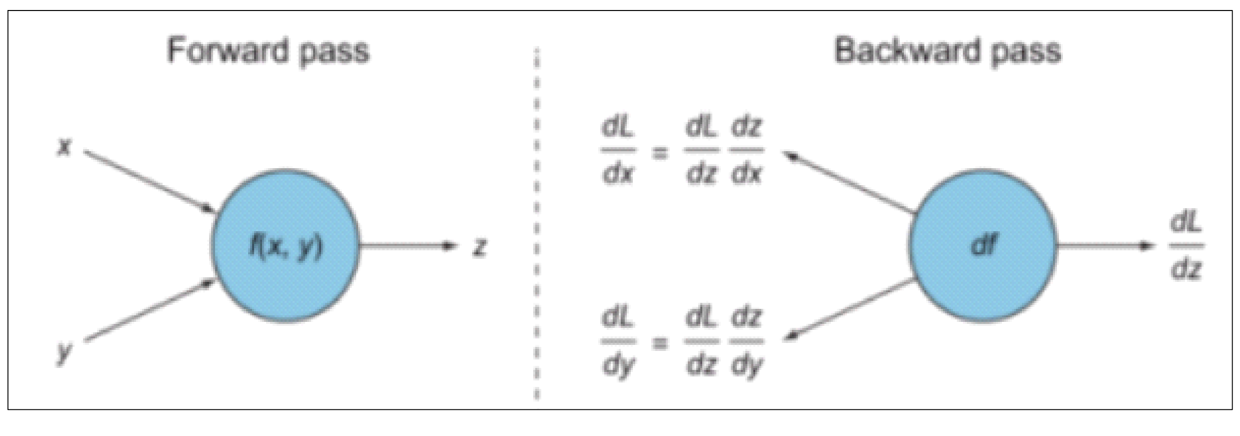
- 언뜻 보면 수식이 매우 복잡해보이나 각 에지의 편미분을 입력 방향으로 거슬러 올라가며 곱하는 것에 지나지 않음

- 위 식을 다음과 같이 풀어 쓸 수 있음

$W^{(1)}_{1,3}$ 에 전달되는 오차 = 오차가 에지 4에 미치는 영향 × 오차가 에지 3에 미치는 영향 × 오차가 에지 2에 미치는 영향 × 오차가 목표 에지에 미치는 영향

● 1 에포크 = 순방향 계산 + 역방향 계산

- 순방향 계산에서는 예측값이 출력됨
- 역방향 계산에서는 오차의 미분을 역방향으로 전달하며 가중치를 수정함



● 기억합시다!

- 신경망은 순방향 계산, 오차 계산, 가중치 최적화의 과정을 반복하며 학습함
- 파라미터는 가중치 등 학습 과정에서 신경망에 의해 수정되는 변수를 말함. 파라미터에 대한 수정은 학습을 통해 자동으로 이루어짐
- 하이퍼파라미터는 신경망의 층수, 활성화 함수, 손실 함수, 최적화 알고리즘, 학습률 등 사람이 직접 조정해야 하는 변수를 말하며, 학습을 시작하기 전에 조정을 마쳐야 함

17 합성곱 신경망

● 신경망이 학습하고 파라미터를 최적화하는 기본 원리는 CNN(합성곱 신경망)과 MLP(다층 퍼셉트론) 모두 같음

- 신경망 구조: MLP와 CNN은 뉴런층이 겹겹이 쌓인 구조로 구성됨. CNN의 구조는 조금 차이(합성곱층과 전결합층의 차이)가 있음
- 가중치와 편향: MLP와 CNN의 예측은 동일한 방식으로 작동함. 두 가지 모두 무작위 값으로 초기화되는 가중치와 편향을 가짐

※ MLP의 가중치와 CNN의 가중치의 가장 큰 차이점은 전자가 벡터 형태인데 비해 후자는 합성곱 필터 또는 커널 형태라는 점임

- 하이퍼파라미터: MLP와 마찬가지로 CNN을 설계할 때도 오차 함수, 활성화 함수, 최적화 알고리즘 등을 결정해야 함

※ CNN에서도 앞 장에서 설명한 하이퍼파라미터가 그대로 적용되며, CNN에서만 사용되는 하이퍼파라미터는 추가로 설명할 예정임

- 학습: MLP와 CNN은 학습 방식에 큰 차이가 없음

※ 순방향 계산을 하고, 손실 함수를 계산하며, 마지막으로 경사 하강법을 이용해서 파라미터를 최적화함. 이 과정에서 오차가 각각의 가중치까지 역전파되며 손실 함숫값이 작아지는 방향으로 파라미터가 수정됨

1. 다층 퍼셉트론을 활용한 이미지 분류

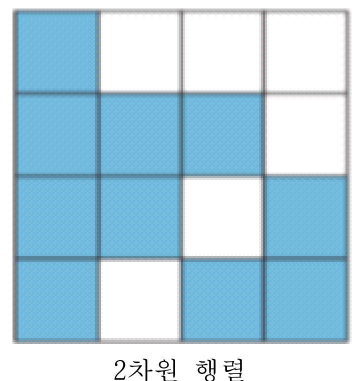
가. 입력층

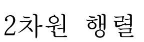
1) 28×28 크기의 이미지를 컴퓨터가 보면 28×28 크기의 행렬이 됨

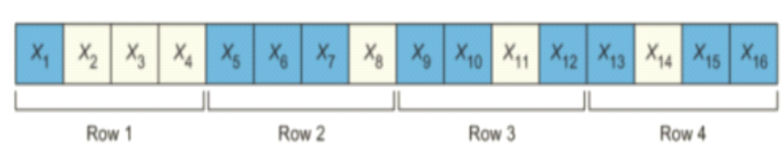
· 컴퓨터는 모든 이미지를 숫자로 기억함
· 28×28 크기의 행렬의 요솟값은 0부터 255의 범위를 가짐
· 0은 검은색, 255는 흰색, 그 사이의 값은 회색조를 나타냄

2) MLP는 모양이 (1,n)인 1차원 벡터만 입력받을 수 있음. 즉, 모양이 (x, y)인 2차원 이미지 행렬은 입력할 수 없음 → 행렬을 입력층에 입력하려면 모양이 (1,n)인 벡터로 변환해야 하는데 이 과정을 이미지 벡터 변환(Image flattening)이라고 함

3) 28×28 = 784 픽셀 → 신경망에 입력하려면 (28×28) 행렬을 모양이 (1×784)인 벡터로 변환해야 함 → 결국 우리가 사용할 신경망의 입력층이 갖게 될 노드는 784개(x_1, x_2, …, x_{784}) 가 됨




2차원 행렬



변환된 1차원 벡터

4) 픽셀값 0이 검정색, 255거 흰색이라고 하면 이 입력 벡터는 다음과 같음
   - Input = [0, 255, 255, 255, 0, 0, 0, 255, 0, 0, 255, 0, 0, 255, 0, 0]

5) 케라스 라이브러리를 활용하면 다음의 코드로 이미지 행렬을 벡터로 변환할 수 있음

```
from keras.models import Sequential
from keras.layers import Flatten

model = Sequential()
model.add( Flatten(input_shape = (28,28) ))
```

나. 은닉층

1) 신경망에서는 하나 이상의 은닉층을 원하는 만큼 가질 수 있으며, 각 층은 하나 이상의 뉴런으로 구성됨

2) 은닉층에서는 대부분의 경우 ReLU 활성화 함수가 가장 성능이 좋음

3) 예제에서는 일단 노드 512개를 가진 은닉층을 두 층 만들고, 각 층마다 ReLU 함수를 추가하도록 함

※ 여기서 2개의 층은 전결합층(=밀집층, dense layer)이라고도 함

```
from keras.layers import Dense

model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))
```

다. 출력층

1) 분류 문제이고, 클래스 수가 중복되지 않는 경우라면 출력층에 소프트맥스를 사용하는 것이 가장 좋음

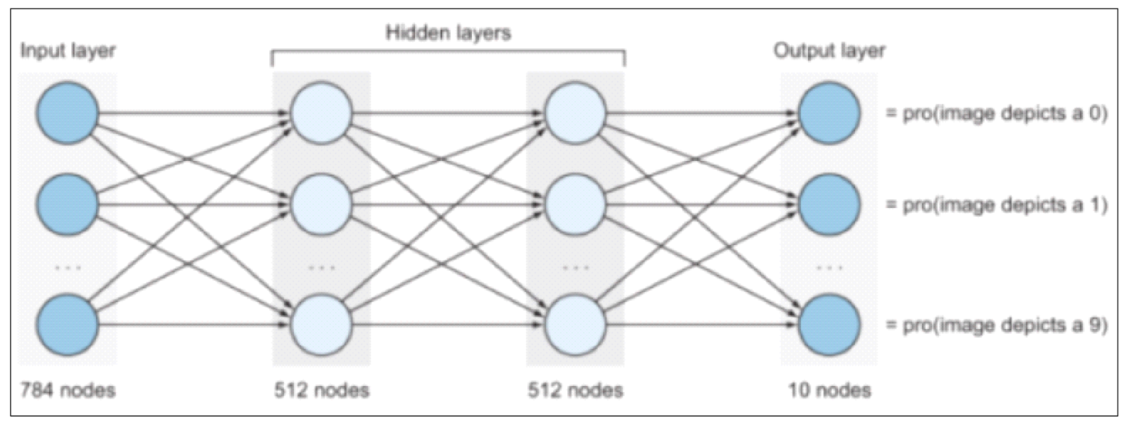
1) 분류 문제의 출력층 노드의 수는 분류 대상 클래스의 수와 같음

2) 이 문제에서는 0부터 9까지의 숫자 10개가 분류 대상이므로 노드 10개를 갖는 Dense 층을 추가하면 됨

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

라. 모델 완성하기

1) 추가한 층을 모두 합쳐 다음과 같은 그림의 신경망을 완성함



2) 이를 케라스로 구현한 모델의 전체 코드는 다음과 같음

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()

model.add( Flatten(input_shape = (28,28) ))
```

```
model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))

model.add(Dense(10, activation = 'softmax'))
model.summary()
```

3) 코드를 실행하면 모델의 구조를 아래와 같이 간략히 정리해서 출력함

```
_________________________________________________________________
Layer (type)          Output Shape          Param #
=================================================================
flatten (Flatten)       (None, 784)          0

dense (Dense)           (None, 512)          401920

dense_1 (Dense)         (None, 512)          262656

dense_2 (Dense)         (None, 10)           5130

=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

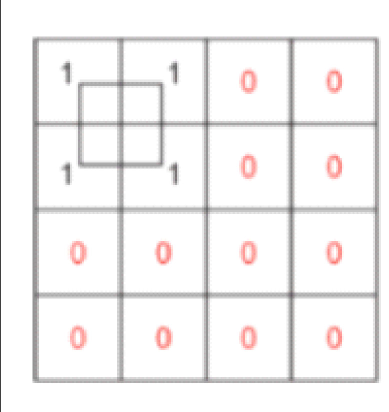
● Param # 필드는 무엇을 나타내는 것일까?
- 해당 층의 파라미터(가중치)의 수를 의미함
- Flatten 층과 다음 층을 잇는 파라미터 수는 0임(이미지 행렬을 벡터로 변환하는 역할만 하기 때문임)
- dense 층과 dense_1 층을 잇는 파라미터의 수는 (784×512)+512 = 401,920개
- dense_1 층과 dense_2 층을 잇는 파라미터의 수는 (512×512)+512 = 262,656개
- dense_2 층과 출력층을 잇는 파라미터의 수는 (512×10)+10 = 5,130개
- 최종적으로, 이 모두를 합한 신경망 전체 파라미터의 수는 669,706개

마. MLP로 이미지를 다룰 때의 단점

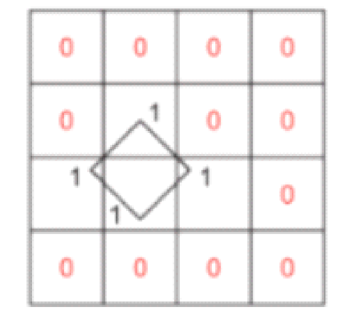
1) 공간적 특징의 손실

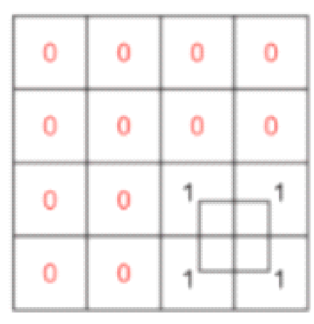
- 2차원 이미지를 1차원 벡터로 변형하면 이미지 내 공간적 특징이 손실됨

- 즉, 이미지에 포함된 2차원 정보를 모두 폐기한 것과 같음
- 2차원 이미지를 이렇게 다루면 신경망이 서로 가까이 위치한 픽셀 간의 관계를 알 수 없으므로 정보의 손실이 생기는 것임



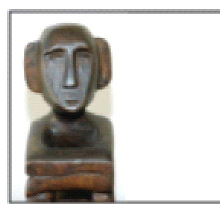
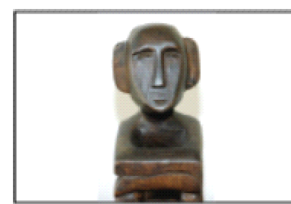
- ● 1차원으로 변환한 입력 벡터
  = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- ● 이럴 경우, 이 신경망은 x_1, x_2, x_5, x_6 이 발화된 경우에만 정사각형을 인식할 것임
- ● 즉, 정사각형의 모양을 특징으로 학습하지 않았기 때문에 이 신경망이 정사각형을 학습하려면 이미지 내 모든 자리에 위치한 정사각형의 이미지가 필요함
  → 이렇게 되면 그냥 사람이 직접 하는 게 편함





- ● MLP는 이런 이미지에 정사각형이 포함되어 있는지 알지 못함

Translation Invariance

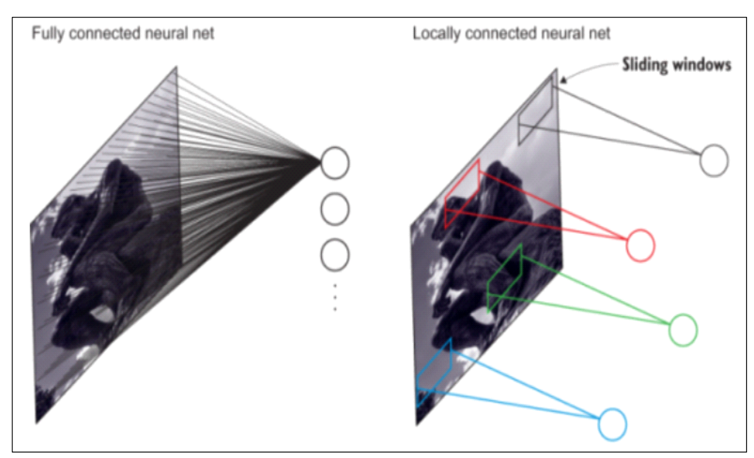



- 신경망이 석상의 특징을 나타내는 모든 형태를 이미지 내 위치와 상관없이 학습해야 이 그림들이 모두 같은 석상임을 비로소 알 수 있게 되는데 이때, 이미지의 공간적 특징이 필요함

2) 전결합층

- 전결합(Fully-Connected)은 이전 층의 모든 노드가 다음 층의 모든 노드와 연결된다는 의미임
- 만약 1,000×1,000 크기의 이미지를 다룬다면 첫 번째 은닉층의 각 노드마다 백 만개의 파라미터를 갖게 되어 계산 복잡도가 너무 커짐
- 합성곱 신경망(CNN)은 지역적으로 연결된 구조의 층을 갖는데, 합성곱

신경망의 노드는 이전 층의 노드 중 일부하고만 연결되며, 전결합층에 비해 파라미터 수가 훨씬 적음



- ● 합성곱 신경망의 의의
  - 공간 정보 손실을 최소화
  - 계산 복잡도를 낮춤
- ● CNN은 2차원 이미지 행렬을 그대로 입력받을 수 있어서 픽셀값에 숨어 있는 패턴을 이해할 수 있음

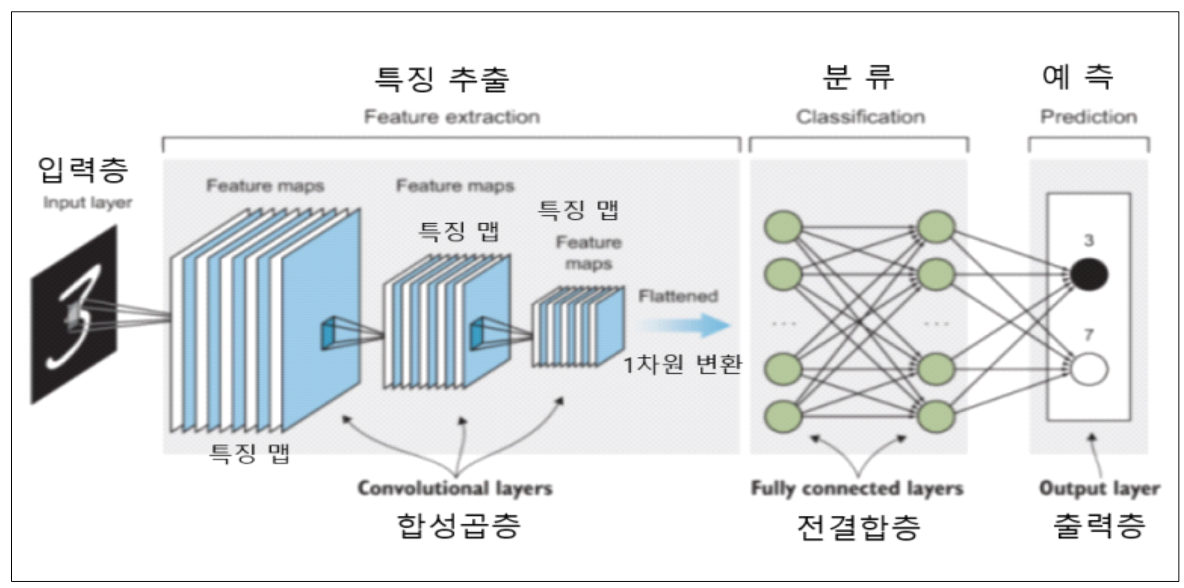
2. 합성곱 신경망 구조

가. 전체 학습 과정

1) 일반적인 신경망의 학습 과정과 크게 다르지 않음

2) 다만, 차이점은 합성곱 신경망은 전결합층 대신 합성곱층을 사용해서 특징을 학습한다는 것뿐임

3) CNN의 추상적인 구조: 입력층, 합성곱층, 전결합층, 출력층

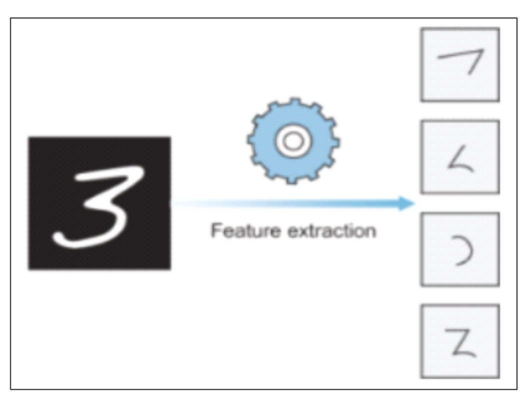


- 입력층은 말 그대로 이미지를 입력하는 층임
- 합성곱층에서는 특징의 추출 즉, 특징의 학습이 이루어짐
- 전결합층에서는 분류를 담당함
- 마지막 출력층은 분류 결과를 출력함

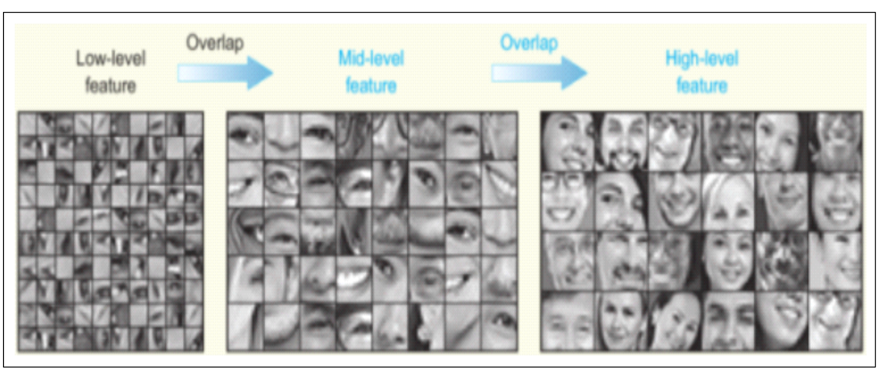
● 신경망의 구조와 전체적인 작동 과정
- 입력층에 손글씨 숫자 3이 들어옴
- 이 이미지가 합성곱층에 들어오며 특징이 학습됨
- 학습하여 발견된 패턴은 특징 맵으로 추출되는데 점점 특징 추출의 단계가 진행될수록 이 특징맵 이미지의 크기는 줄어들고, 특징 맵의 수(층의 깊이)는 계속 추가됨
- 학습의 결정체인 특징 벡터는 1차원으로 변환되어 전결합층에 입력됨
- 전결합층에서는 이 특징 벡터를 기반으로 이미지를 분류함
- 분류 결과에 해당하는 출력층의 노드가 발화됨(여기에서는 출력층의 노드 3과 7 둘 뿐이고, 이 중 3이 발화됨)

나. 특징 추출 과정 들여다보기

1) 특징 추출 단계는 큰 이미지를 여러 개의 작은 특징 맵으로 나눈 뒤 이를 쌓아 벡터로 만드는 과정임
2) 예를 들어 숫자 3이 쓰인 이미지가 입력되었다면(깊이=1) 이 이미지는 숫자 3을 나타내는 다양한 특징의 위치가 담긴 여러 개의 이미지로 조각조각 나눠짐(깊이=4)



● 이미지가 합성곱층을 차례대로 통과하면 이미지의 크기는 작아지고, 특징의 가짓수가 늘어나므로 깊이도 점점 깊어짐



입력 이미지
+ 은닉층1 → 패턴
+ 은닉층2 → 패턴의 패턴
+ 은닉층3 → 패턴의 패턴의 패턴

다. 분류 과정 들여다보기

1) 전결합층(MLP)을 추가해서 특징 벡터를 입력받음

2) 전결합층은 '첫 번째 특징을 보면 모서리 같은 부분이 있어서 3, 7 또는 3일 것이다. 두 번째 특징을 보면 곡선이 있는 것으로 보아하니 7은 확실히 아니다' 와 같은 식으로 결국 숫자 3이라는 결론에 이르게 됨

3. 합성곱 신경망의 기본 요소

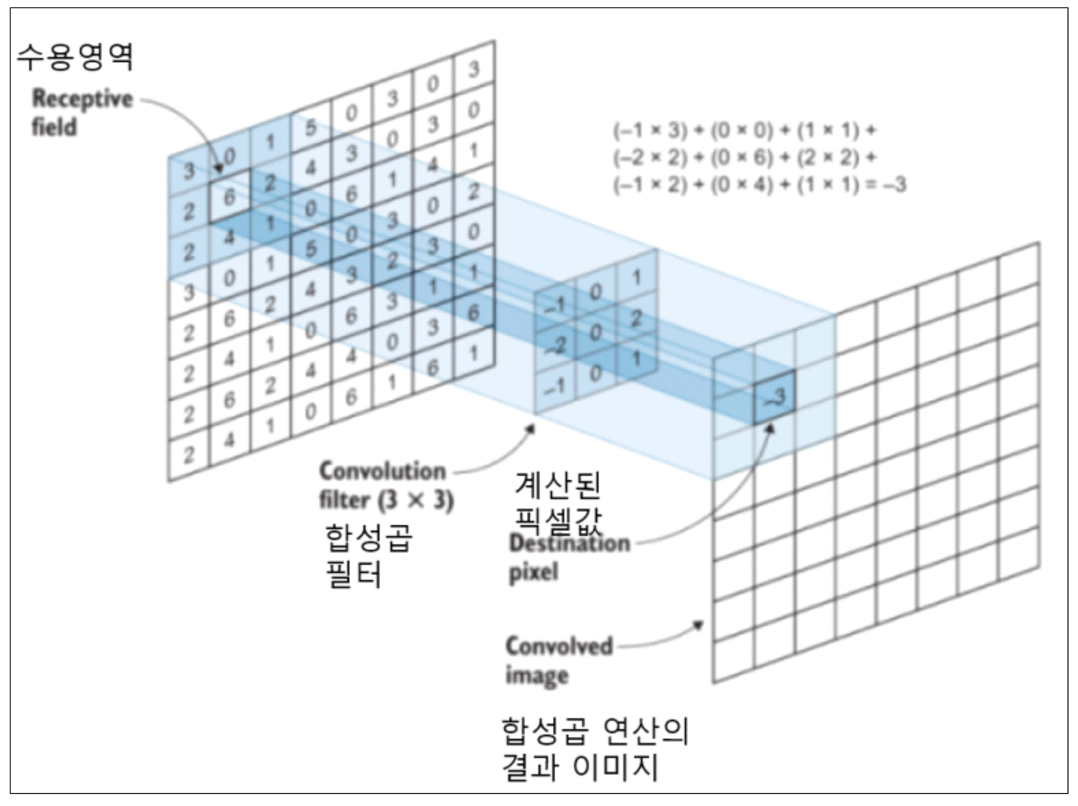
가. 합성곱층

1) 합성곱 신경망의 핵심 구성 요소로, 이미지 데이터의 특징을 추출함

2) 합성곱이란?

- 수학에서는 두 함수를 인수로 해서 새로운 함수를 만들어내는 연산을 뜻함
- 합성곱 신경망에서의 두 인수는 입력 이미지와 합성곱 필터를 의미하며 이 둘의 계산을 통해 새로운 특징 맵을 만들어 냄

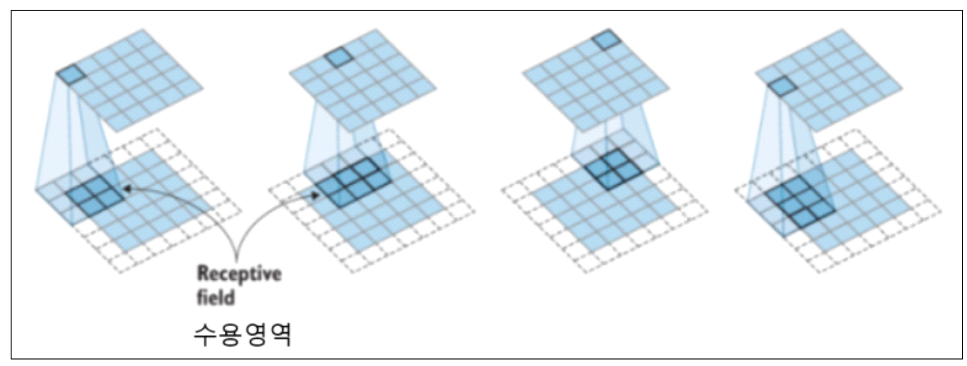
3) 합성곱층이 이미지를 처리하는 과정



- 위 그림과 같이 가운데 위치한 3*3 크기의 행렬이 합성곱 필터이며, 커널(Kernel)이라고 부르기도 함
- 커널은 입력 이미지 위를 픽셀 단위로 움직여다니며 연산을 수행함
- 각 위치에서 연산된 픽셀값을 모아 '합성곱 연산'을 거친 새로운 이미지

즉, 특징 맵을 만들고 이를 다음 층으로 전달함

- 이때 합성곱 연산이 한 번 수행되는 입력 이미지 상의 범위를 수용 영역이라고 부름



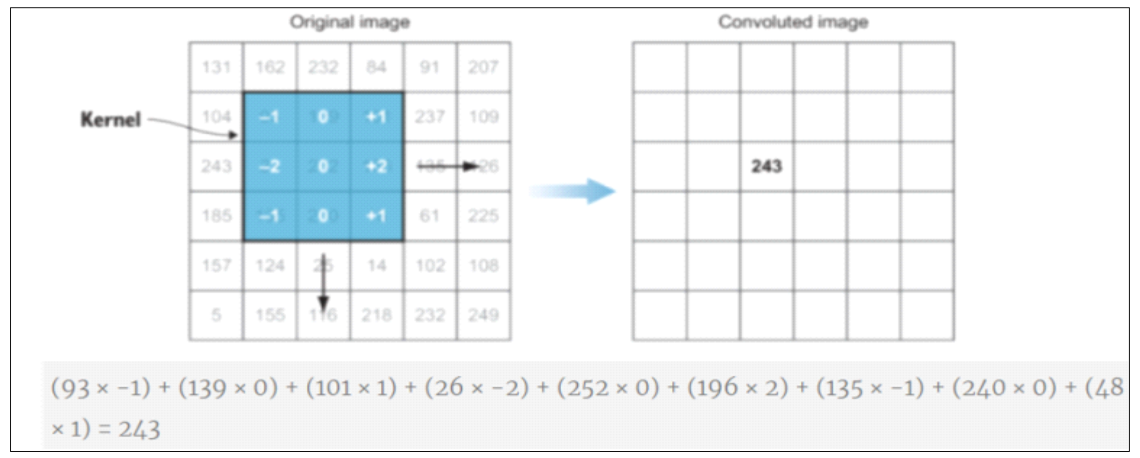
- 합성곱 신경망에서는 커널 즉, 합성곱 행렬이 바로 **가중치**가 됨
- 합성곱 행렬은 처음에는 무작위 값으로 초기화되며, 시간이 지나면서 신경망에 의해 학습되는 값임

4) 합성곱 연산

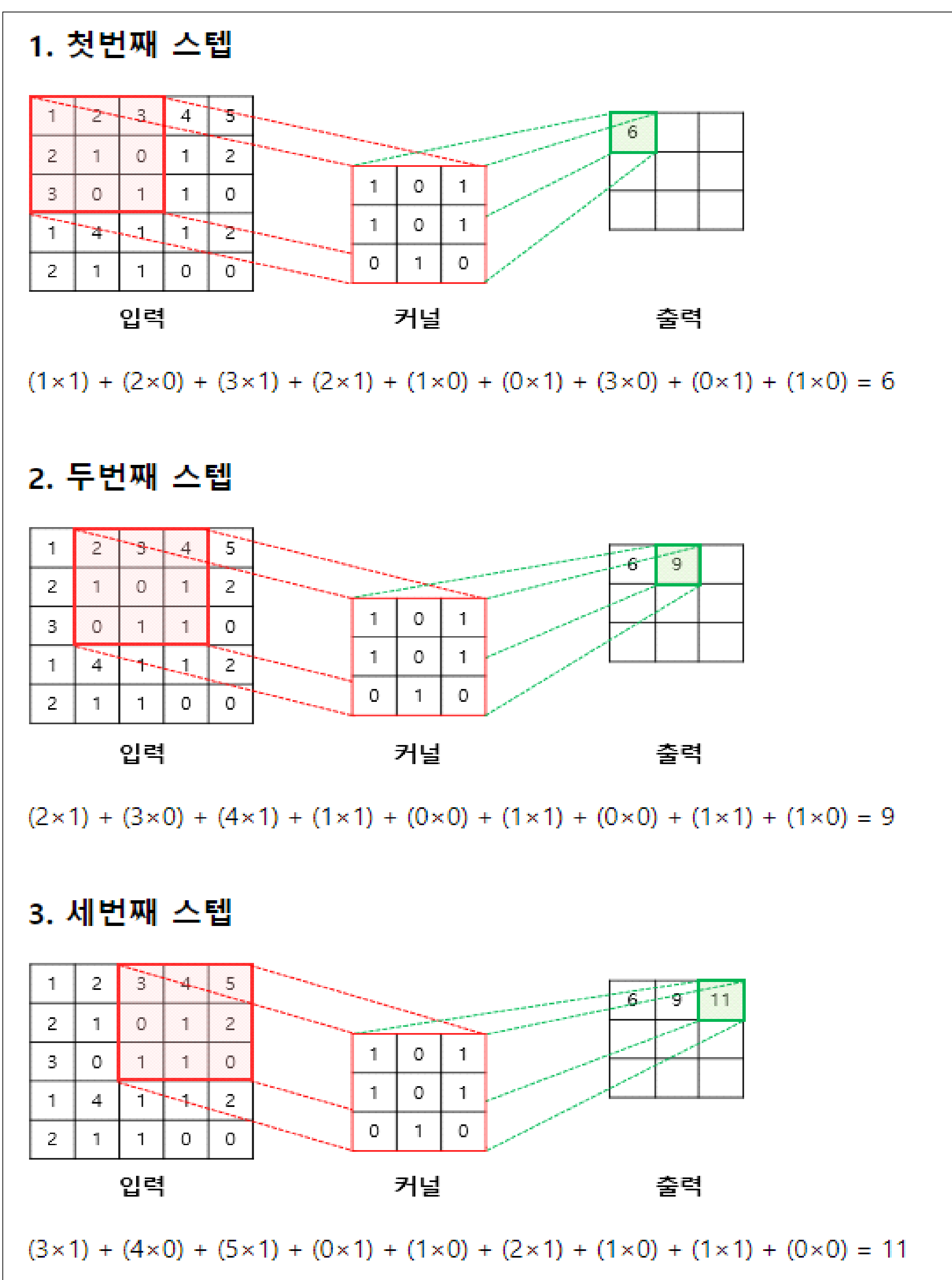
- 다층 퍼셉트론의 가중합(Weighted sum)과 비슷한 계산이 이루어짐

$\sum_i w_i x_i + b$ (w: 가중치, b: 바이어스)	· 입력값은 수용 영역 안에 들어와 있는 각각의 입력 픽셀값 · 가중치는 합성곱 필터의 픽셀값
다층 퍼셉트론의 가중합	합성곱 연산

- 아래 그림과 같이 수용영역과 합성곱 필터가 겹치는 픽셀값을 각각 곱해주고, 그 결과를 합함
- 그 결과값을 필터 중심에 해당하는 픽셀 위치의 값으로 넣어 특징 맵(=활성화 맵)을 완성해나감

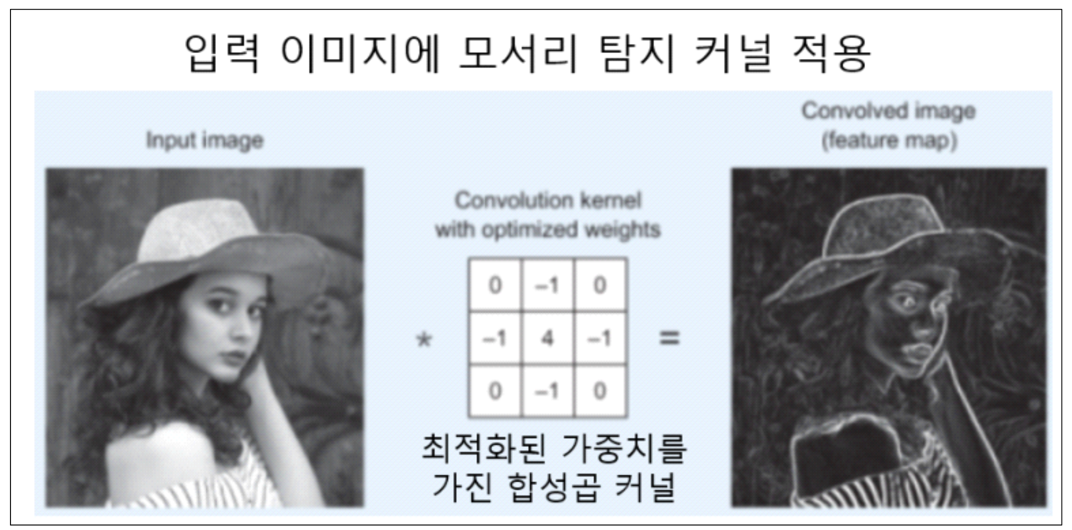


- 합성곱 필터를 활용한 합성곱 연산의 과정은 다음과 같음

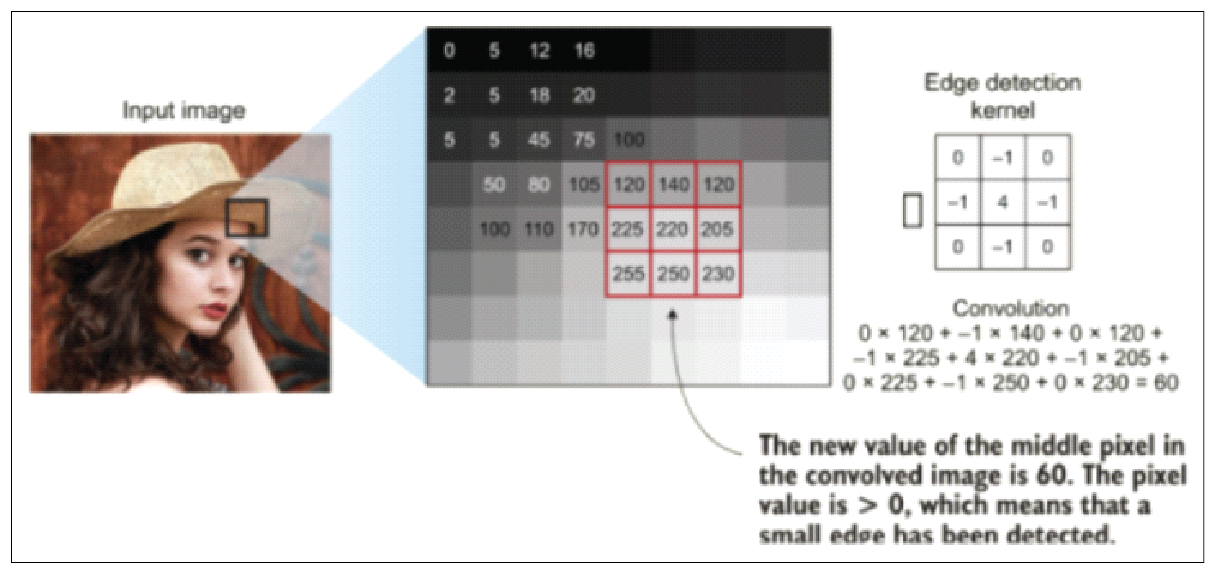


- 합성곱 필터(커널)는 입력 이미지 위를 이동하면서 입력 이미지 전체를 커버하는데, 이 과정에서 입력 이미지가 여러 번 중첩되므로, 수용 범위가 넓어진다는 장점이 있음

- 커널이 한번 움직일 때마다 픽셀 단위로 가중합이 계산되어 필터 중심에 해당하는 픽셀의 새로운 값이 결정됨 → 이런 식으로 특징 맵이 만들어지게 됨



모서리 탐지 커널을 적용한 결과 이미지(특징 맵)



입력 이미지에 모서리 탐지 커널 적용 연산

- 합성곱 층에는 하나 혹은 그 이상의 합성곱 필터가 존재함
- 사실, 필터의 수만큼 합성 곱 연산으로 생성된 이미지인 특징 맵이 출력되기 때문에, 합성곱 필터의 수가 많을수록 생성되는 특징 맵의 개수도 많아져, 출력이 깊어지는 것임
- 필터 수가 많아지면 특징 맵이 증가하고, 픽셀 즉, 신경망의 노드가 증가하게 되는 효과를 일으켜 더 복잡한 패턴을 탐지할 수 있게 됨
- 물론 반대급부로 계산 복잡도가 증가하는 것을 감수하여야 함

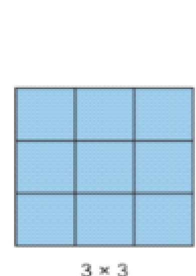
- 합성곱층을 구현하는 코드는 다음과 같음

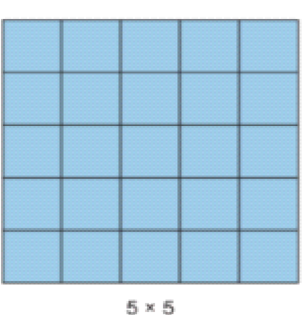
```
from keras.layers import Conv2D

model.add(Conv2D(filters=16, kernel_size=2, strides='1', padding='same',
          activation='relu'))
```

- 합성곱층에 존재하는 5개의 인자(하이퍼파라미터)에 대해 알아보자

● filters: 합성곱 필터 수, 출력의 깊이를 결정함
- 예: 3×3 크기의 커널이 하나 있다면 합성곱층의 뉴런은 9개가 됨

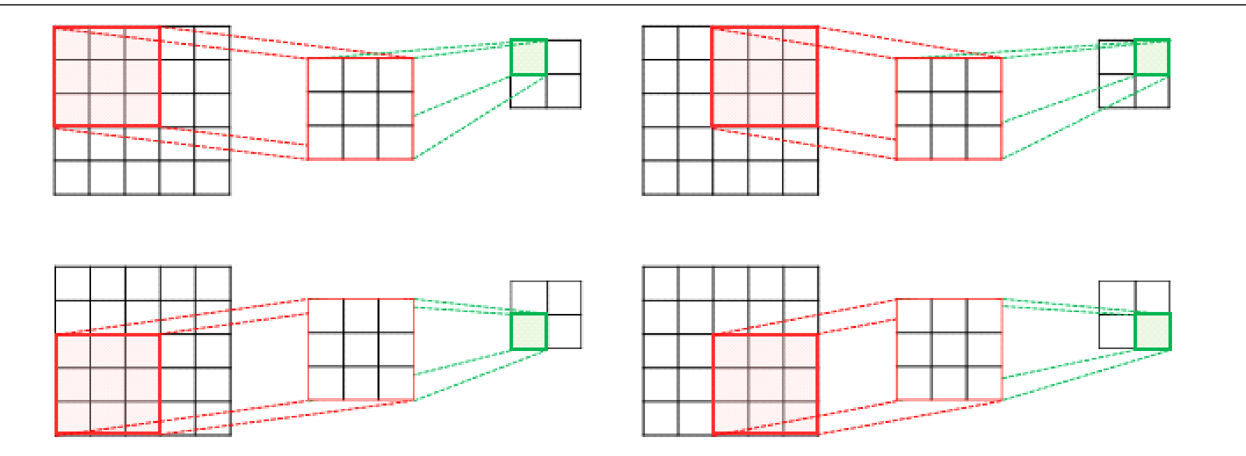
● kernel_size: 합성곱 필터(커널) 행렬의 크기를 의미함
- 보통 커널 크기(kernel_size)는 합성곱 필터의 크기×높이를 뜻함
- 통상적으로 2×2, 3×3, 5×5 이렇게 주로 사용함


3 × 3


5 × 5

- 직관적으로 봤을 때, 커널의 크기가 작을수록 전체 이미지를 여러 부분 중첩하며 수용 범위를 넓혀 특징을 추출할 수 있기 때문에 디테일한 부분까지 잡아낼 수 있으며, 반대로 커널의 크기가 클수록 놓치는 이미지의 세세한 부분이 많을 것임(그래서 최대 5×5보다 큰 필터를 잘 사용하지 않음)

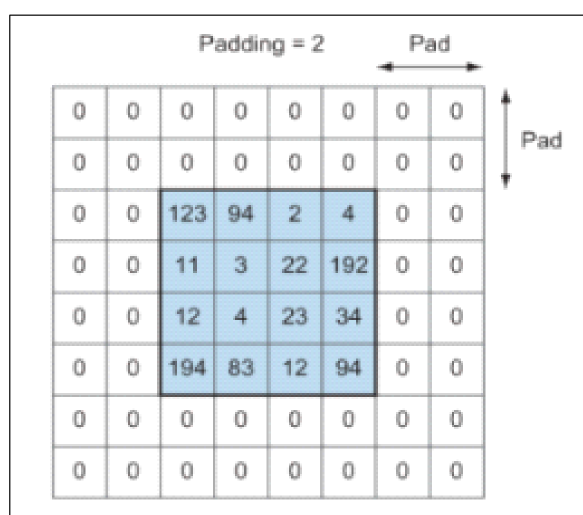
● strides: 스트라이드
- 커널(=필터)이 입력 이미지를 한 번에 이동하는 픽셀의 수를 의미함
- 만약 필터가 입력 이미지를 한 번에 한 픽셀씩 이동하면 스트라이드 값은 1이 되며, 한 번에 두 픽셀(=두 칸)을 건너 뛴다면 스트라이드 값은 2가 됨
- 스트라이드(strides)는 3 이상을 잘 적용하지 않음
- 왜 그럴까? 여러 픽셀을 건너뛰게 되면 출력의 크기가 작아지기 때문임
- 다음의 예제는 스트라이드가 2일 경우에 5×5 이미지에 합성곱 연산을 수행하는 3×3 커널의 움직임을 보여주며, 최종적으로 2×2의 특성 맵을 얻음



- 위의 그림과 같이 5×5 이미지에 3×3의 커널로 합성곱 연산을 하였을 때, 스트라이드가 1일 경우에는 3×3의 특성 맵을 얻게 됨
- 합성곱 연산의 결과로 얻은 특성 맵은 입력보다 크기가 작아진다는 특징이 있음

● padding: 패딩

- 만약, 합성곱 층을 여러개 쌓았다면 최종적으로 얻은 특성 맵은 초기 입력보다 매우 작아진 상태가 되버림
- 합성곱 연산 이후에도 특성 맵의 크기가 입력의 크기와 동일하게 유지되도록 하고 싶다면 패딩(padding)을 사용하면 됨
- 패딩(Padding)은 우리가 입는 패딩 점퍼를 생각하면 쉬움
- 흔히 제로 패딩(Zero-padding)을 많이 하는데 이미지 둘레에 패딩 점퍼처럼 픽셀값이 0인 픽셀들을 추가로 감싸 덧붙이는 것임



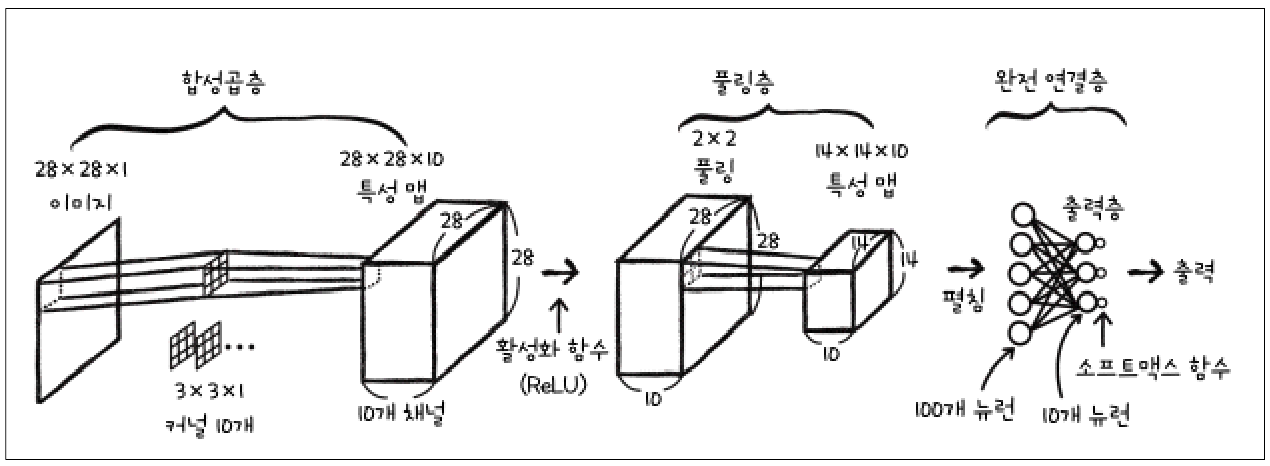
· 패딩의 주된 목적은 합성곱 연산을 거친 출력 이미지의 크기를 입력 이미지의 크기와 같게 유지하는 것임

· 패딩을 활용해서 이미지 크기를 유지하며 합성곱층을 여러 개 사용하여 층수가 깊은 합성곱 신경망을 구성할 수 있게 됨

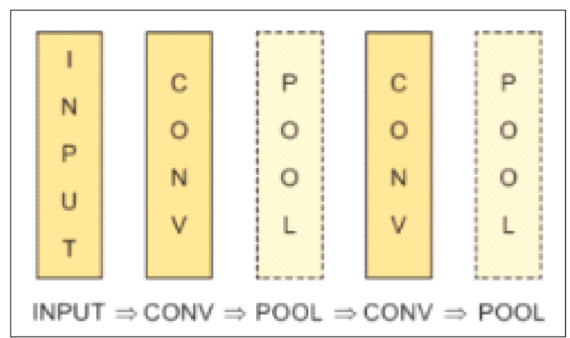
※ 스트라이드와 패딩의 목적
- 첫 번째는 이미지의 중요 세부 사항을 놓치지 않고 다음 층으로 전달하기 위함(스트라이드가 1이고, 패딩을 사용하면 이미지 크기를 같게 유지할 수 있음)
- 두 번째는 이미지의 공간적 정보 계산의 부하를 적절한 수준으로 감소시키기 위함(제로 패딩을 통해 출력의 크기를 유지하면서도 값을 0으로 내어 적절히 무시할 수 있게 함)

● activation: 활성화 함수이며 은닉층에서는 주로 ReLU를 사용함

나. 풀링층과 서브샘플링

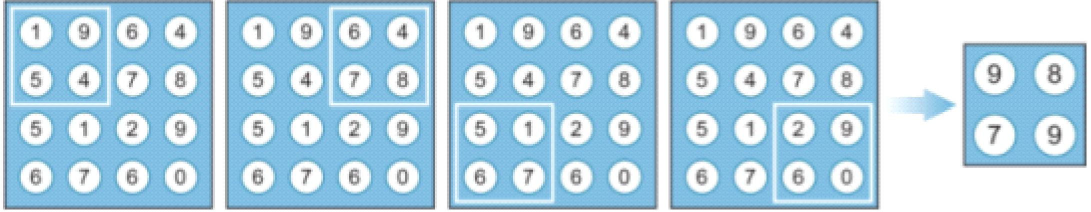


1) 위의 그림을 살펴보면, 합성곱 신경망에서 합성곱층을 통과하면 입력층의 28×28×1 이미지가 28×28×10의 특징 맵으로 바뀜
2) 즉, 깊이가 깊어지게 되어(마치 부피가 증가하는 것처럼) 최적화해야 할 파라미터의 수가 급격히 늘어나게 됨
3) 파라미터의 수가 증가하면 학습 시간이 오래 걸리게 되는데, 이를 극복하게 해주는 것이 풀링층임
4) 풀링(Pooling) 또는 서브샘플링(Subsampling)이라 불리는 이 층은 신경망의 가로와 세로의 크기를 줄여 다음 층으로 전달하는 파라미터의 수를 감소시키는 역할을 함
5) 즉, 합성곱에서 만든 특징 맵(28×28×10)을 다운 샘플링(14×14×10)해서 계산 복잡도를 낮춰줌
6) 보통 CNN에서는 대개 합성곱층마다 하나씩 합성곱층 뒤에 배치하는 경우가 많음



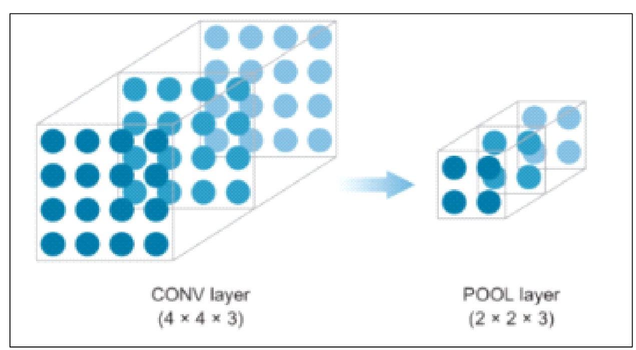
7) 최대 풀링과 평균 풀링

- 합성곱층에서는 합성곱 연산이 이루어지듯, 풀링층에서도 풀링 연산이 이루어짐
- 풀링 연산에는 최대 풀링과 평균 풀링이 있음
- 합성곱 커널처럼 최대 풀링에도 커널이 존재하는데, 이 최대 풀링 커널에도 윈도우 크기와 스트라이드가 있음. 그런데 행렬에 가중치가 없다는 점에서 합성곱 커널과의 차이점이 있음
- 즉, 최대 풀링층에서는 최대 풀링 커널이 합성곱 층에서 받은 특징맵 위를 사전에 지정한 스트라이드 값 간격으로 돌아다니는데, 커널 윈도우 내에 들어온 픽셀값의 최대값을 찾아 이를 출력 이미지의 픽셀값으로 삼는 것임

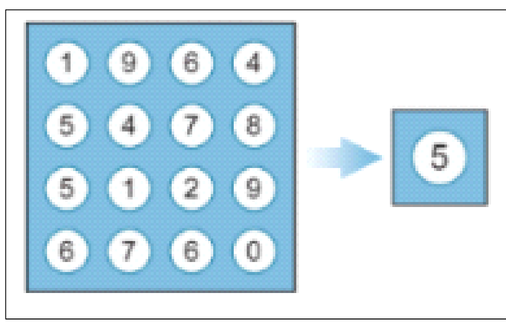


스트라이드가 2인 2×2 크기의 풀링 필터가 4×4의 특징 맵을 2×2로 축소시킨 예

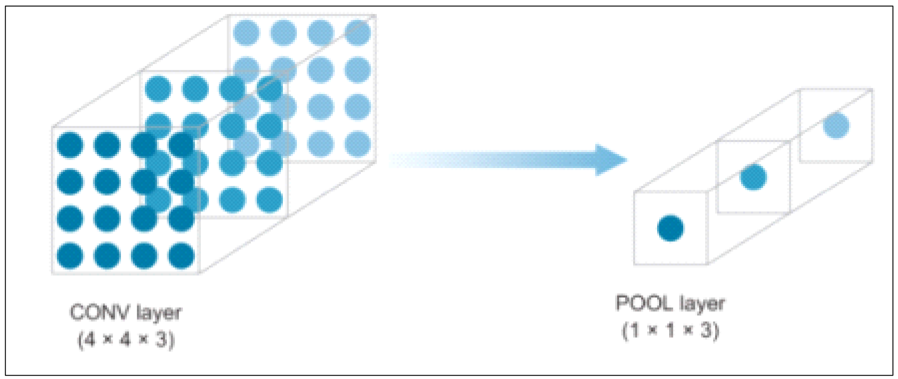
- 커널 윈도우 내에 들어온 픽셀값의 최대값인 9, 8, 7, 9를 그 다음 출력으로 삼은 것을 확인할 수 있음



- 그림에서처럼 4×4가 2×2로 줄어들고, 깊이의 개수는 그대로 3인 것을 볼 수 있음
- 정리하면 합성곱 층의 특징 맵이 3개라면 이어지는 풀링층도 마찬가지로 (크기는 더 작은) 3개의 특징 맵을 출력함
- 평균 풀링은 커널 윈도우 내에 들어온 픽셀값의 최대값이 아니라 평균값을 구하여 이를 출력 이미지의 픽셀값으로 삼는 것임
- '전역 평균 풀링'은 특징 맵 크기를 극단적으로 줄이는 방법으로, 윈도우 커널 크기와 스트라이드를 설정하지 않고 전체 특징 맵 픽셀값의 평균을 구하여 출력함



- 전역 평균 풀링을 활용하면 3차원 배열을 1차원의 벡터로 변환할 수 있음



● 풀링층을 사용하는 목적
- CNN에서는 합성곱 층에서 수많은 합성곱 필터를 사용하게 되는데, 합성곱 연산의 결과 출력되는 특징 맵의 깊이가 깊어지고, 파라미터의 수가 기하급수적으로 증가하게 됨
- 이때, 풀링층을 사용하면 중요한 특징을 간직하면서도 이미지의 크기를 줄여 다음 층에 전달할 수 있게 됨(물론 세세한 모든 특징을 다 전달할 수는 없지만 중요한 특징은 기억하여 보냄)
- 이미지를 압축하면 해상도가 떨어지지만, 중요한 특징을 유지하면서 용량을

줄여주는 효과가 있음. 바로 풀링층은 합성곱 층에서 '이미지 압축 작업'의 역할을 수행함

이미지의 중요한 특징을 유지하면서 해상도를 떨어뜨리는 풀링층

- 그러나 최근에는 풀링층을 배제하고, 합성곱 층만을 활용해 CNN을 구성하는 방법이 많이 제안되고 있음. 왜 그런 것일까?
- 아무래도 풀링층은 이미지 압축 작업이자 뿌옇게 만드는(Blur 처리) 것에 비유될 수 있음. 그러다보니 중요한 특징은 보관한다고 하더라도, 이미지 정보의 손실은 피할 수 없기 때문임
- 앞으로는 합성곱 신경망에서 적은 수의 풀링층을 사용하거나 아예 사용하지 않을지도 모르지만, 현재 기준에서는 합성곱 층 사이에 이미지 다운 샘플링(차원 축소)을 위한 풀링층 배치가 아직 일반적임

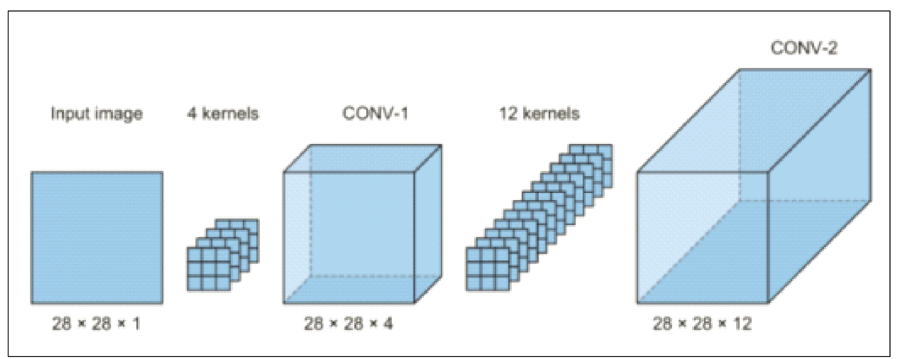
- 풀링층의 코드 구현은 다음과 같음(맥스 풀링의 예)

```
from keras.layers import MaxPooling2D

model.add(MaxPooling2D(pool_size=(2, 2), strides = 2))
```

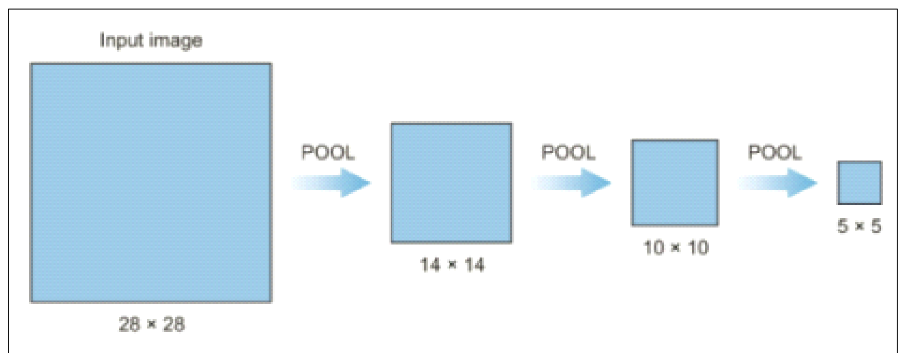
- MaxPooling2D 내의 인자에 커널 윈도우(=pool_size)의 크기를 지정해주고, 스트라이드의 간격을 설정해주는 것이 전부이며, 합성곱층과 풀링층을 번갈아가며 배치하면서 신경망을 구성함

● 합성곱층과 풀링층의 입력과 출력을 정리해볼까요?
- 입력 이미지가 합성곱 층을 통과하면 크기(가로×세로)는 그대로이지만, 이미지의 깊이가 늘어남
- 말로는 잘 이해가 되지 않으니, 그림으로 정리해보도록 하자
- 28×28 크기의 입력 이미지를 필터의 개수 4, 스트라이드 및 패딩이 1로 설정된 합성곱층(Conv_1)을 통과하면 출력 이미지는 28×28 크기로 동일

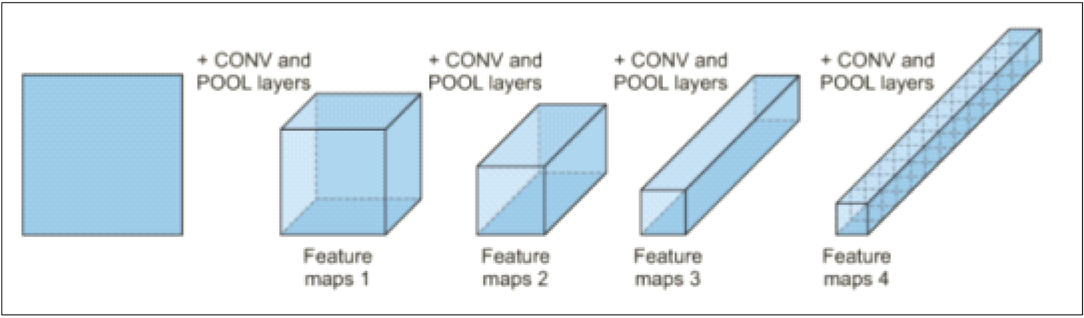
하나, 이미지의 깊이는 4(28×28×4)가 됨
- 하이퍼파라미터는 동일하게 유지한 채, 이 출력이 12개의 커널(=필터)를 만나 합성곱 연산을 거치면 다음과 같이 28×28×12가 됨
- 늘어난 깊이만큼 합성곱 연산을 통해 수용 영역은 넓어지게 됨
- 즉, CNN은 합성곱 층에서 늘어난 깊이와 수용 영역만큼 특징 학습이 이루어짐



- 풀링층을 통과하게 되면 깊이는 그대로 유지가 되지만, 가로와 세로의 크기는 줄어들게 됨(차원 축소 = 파라미터 수 감소 = 계산 복잡도 감소 = 이미지 압축 = Blur 효과)



- CNN은 합성곱층과 풀링층이 번갈아가며 배치된 구조이므로 다음의 그림과 같이 표현될 수 있음



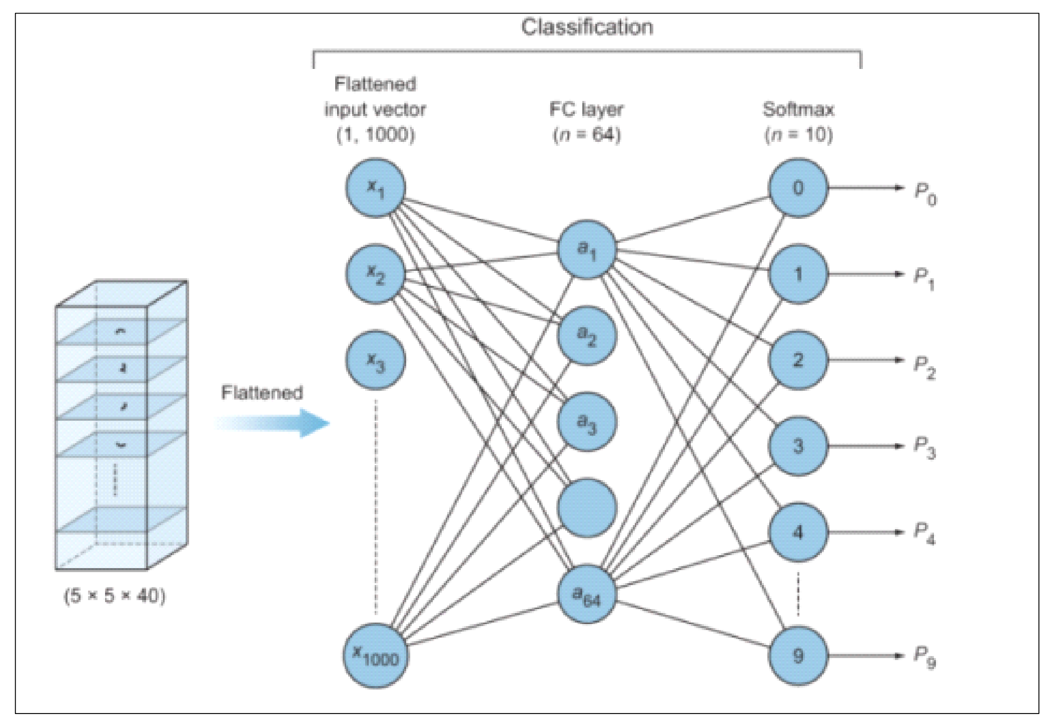
- 깊이는 늘어나고, 이미지의 크기는 계속해서 줄어듦

- 마치 길고 깊게 이어진 빨대의 모습이 될 때까지 이 작업은 계속 진행됨
- 보통 빨대의 형태는 5×5×40 정도가 되며, 이 정도의 사이즈가 되면 이제 분류를 담당하는 전결합층으로 1차원 벡터로 변환하여 넘기게 됨
- 1차원 벡터로 변환하면 5×5×40=1,000이 되며, 이 1,000개를 일렬로 늘어뜨려 전결합층에 전달함 → 이 1,000개가 유닛의 개수가 되며 전결합층의 입력값이 되는 것임

다. 전결합층

1) 분류를 위해서 특별한 구조가 필요한 것은 아니며, 분류는 이전에 살펴보았던 일반적인 신경망 구조인 MLP(전결합층)으로도 충분함

2) 전결합층을 사용하는 이유는?
   - MLP는 분류에 매우 효과적임
   - 합성곱층을 사용한 이유는 이미지를 2차원 그대로 다룰 수 있는 합성곱층과 달리 MLP로 이미지에서 특징을 추출하면 특징을 많이 잃어버리기 때문이었음(이미지를 1차원 벡터로 변환해야 하므로)
   - 하지만 이미 특징을 추출했으므로 이 특징을 1차원으로 변환하면 MLP를 사용해서 이미지를 분류할 수 있음
   - 전결합층에 대한 설명은 이전 장에서 이미 자세히 설명함



분류를 위한 전결합층(MLP)

● 용어를 정리해볼까요?
- MLP는 각 층의 모든 노드가 인접한 모든 층의 모든 노드와 연결됨
- 다층 퍼셉트론(MLP) = 전결합층 = 밀집층 = 피드포워드 신경망
- 위의 용어들 모두 일반적인 신경망 구조를 나타내는 용어로 간주해도 무방함

4. 합성곱 신경망을 활용한 이미지 분류

가. 모델 구조 구성하기

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), strides=1, padding='same',
            activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(64, activation='relu'))

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

model.summary()
```

1) input_shape 인수는 첫 번째 합성곱층에만 지정하며, 그 다음부터는 입력 모양을 지정해 줄 필요가 없음
2) 모든 합성곱층과 풀링층의 출력은 모양이(None, height, width, channels)인 3차원 텐서임. height, width는 출력 이미지의 가로와 세로 크기이며, channels는 출력 이미지의 깊이, 다시 말해 특징 맵의 수를 의미함

3) 튜플의 첫 번째 요소인 None은 이 층에서 처리된 이미지 수이므로 이 값이 None이면 batch_size를 자유롭게 정할 수 있음
4) 출력된 모델의 개요에서 Output Shape 항목을 보면 신경망의 층이 거듭될수록 출력되는 이미지의 크기가 작아지는 것을 알 수 있음
5) 전체 신경망의 파라미터 수는 220,234개임. 먼저 구성했던 MLP 신경망의 파라미터 수가 669,706개였던 것에 비하면 거의 $\frac{1}{3}$로 크게 줄어듦

```
_________________________________________________________________
Layer (type)              Output Shape           Param #
=================================================================
conv2d (Conv2D)           (None, 28, 28, 32)     320

max_pooling2d (MaxPooling2D  (None, 14, 14, 32)   0
)

conv2d_1 (Conv2D)         (None, 14, 14, 64)     18496

max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)     0
2D)

flatten (Flatten)         (None, 3136)           0

dense (Dense)             (None, 64)             200768

dense_1 (Dense)           (None, 10)             650

=================================================================
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
```

나. 파라미터(가중치) 수
1) 파라미터(Parameter)는 가중치를 가리키는 다른 이름이며, 신경망이 학습하는 대상임
2) 파라미터 수를 계산하는 방법
- 파라미터 수 = 필터 수 × 커널크기 × 이전 층 출력의 깊이 + 필터 수(편향)

```
model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
```

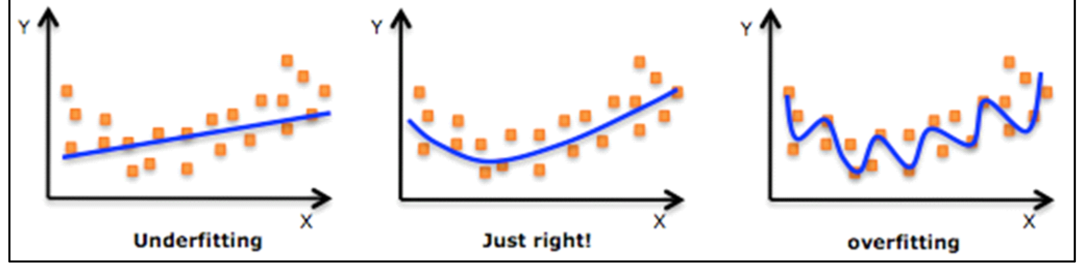
- 파라미터 수 = 64 × 3 × 3 × 32 + 64 = 18,496개
-> Param # 항목을 더해보면 신경망의 전체 파라미터 수는 220,234개

3) 학습 가능한 파라미터와 불가능한 파라미터
- 출력된 모델 개요의 아랫부분을 보면 학습 가능한 파라미터와 학습 불가능한 파라미터 수를 볼 수 있음
- 학습 가능한 파라미터는 학습 과정을 통해 신경망이 최적화해야 하는 파라미터임(이번 예제는 모든 파라미터가 학습 가능한 파라미터임)

5. 과적합 방지를 위한 드롭아웃층 추가하기

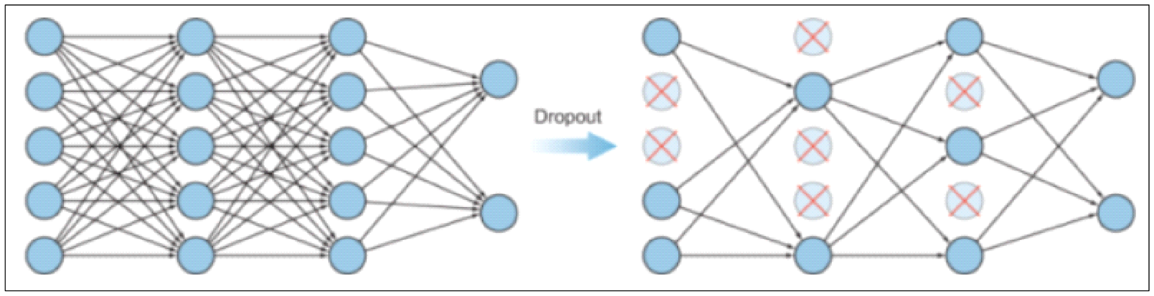
가. 과적합이란?

1) 머신러닝에서 모델의 성능이 잘 나오지 않는 이유는 보통 과적합이나 과소적합이기 때문임
2) 과소적합(Underfitting)은 이름 그대로 모델이 학습 데이터에 부합하지 못하는 현상임(예: 퍼셉트론 하나로 비선형 데이터셋을 분류하는 경우)
3) 과적합(Overfitting)은 모델이 학습 데이터에 지나치게 부합하는 현상임 (예: 표현력이 매우 좋은 신경망이 학습 데이터에는 잘 부합하나, 처음 보는 데이터는 잘 예측하지 못해 일반화 성능이 떨어지는 경우)



나. 드롭아웃층이란?

1) 과적합을 방지하기 위한 수단 중 가장 널리 쓰임
2) 드롭아웃을 적용하면 층을 구성하는 뉴런(노드)의 일정 비율을 비활성화함
3) 이 비율은 신경망의 하이퍼파라미터로 지정됨
4) **비활성화**라는 말은 이들 뉴런이 순방향 또는 역전파 계산에 참여하지 않는다는 뜻임



다. 드롭아웃층은 왜 필요한가?

1) 뉴런은 학습 과정을 거치며 상호 의존 관계를 구축하는데, 이 관계는 각 뉴런의 영향력을 결정하여 과적합으로 이어지는 원인이 됨 → 드롭아웃을 적용하여 노드의 일정 비율을 비활성화한다면 다른 노드는 비활성화된 노드가 가진 특징 없이 패턴을 학습해야 함 → 모든 특징이 이런 식으로 비활성화될 수 있으므로 가중치가 특징 간에 고르게 분산되는 효과가 발생하여 더 잘 학습된 뉴런으로 이어짐
2) 뉴런 간에 발생하는 상호 의존 관계도 완화시킬 수 있음 → 여러 개의 약 분류기가 따로 학습되었기 때문에 데이터의 서로 다른 측면을 학습했고, 일으키는 실수도 다름(다중 시점의 획득) → 이들을 한데 묶으면 과적합을 적게 일으키는 더 강력한 분류기를 만들 수 있음(일종의 앙상블 학습 기법)
3) 간혹 신경망의 특정 가중치가 매우 커져 학습 과정 전체를 장악하는 일이 발생함 → 이로 인해 신경망의 다른 부분은 학습이 제대로 되지 않음 → 드롭아웃은 이런 뉴런을 비활성화해서 다른 뉴런이 학습할 기회를 제공함

유연함은 완벽을 덜어내는 데 있다. 유연함을 연습할수록 강건해진다

라. 드롭아웃층은 어디에 끼워 넣어야 할까?

1) 드롭아웃층은 추출된 특징의 1차원 벡터 변환이 끝난 다음부터 마지막 출력층 사이에 배치하는 것이 일반적임
2) 왜 그럴까? 드롭아웃은 경험적으로 합성곱 신경망의 전결합층에 적용하는 것이 효과가 좋다고 알려져 있음

3) CNN의 구조: CONV → POOL → CONV → POOL → 1차원 벡터 변환 → DO(드롭아웃) → FC →DO(드롭아웃) → FC

4) 케라스를 사용해 앞서 정의한 모델에 드롭아웃층을 추가하면 다음과 같음

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), strides=1, padding='same',
          activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dropout(rate=0.3))

model.add(Dense(64, activation='relu'))

model.add(Dropout(rate=0.5))

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

model.summary()
```

- 드롭아웃층을 정의할 때 rate을 인수로 받음
- rate은 비활성화할 노드의 비율을 의미함(CNN의 하이퍼파라미터임)
- 예: 0.3이면 각 에포크마다 이 층의 뉴런 중 무작위로 선택된 30%가 비활성화됨 / 보통 0.3에서 0.5 사이의 값으로 설정함
- 무작위로 선택되므로 특정 노드가 더 자주 비활성화될 수 있지만, 여러 번 반복하다보면 모든 뉴런이 거의 비슷한 비율로 비활성화됨

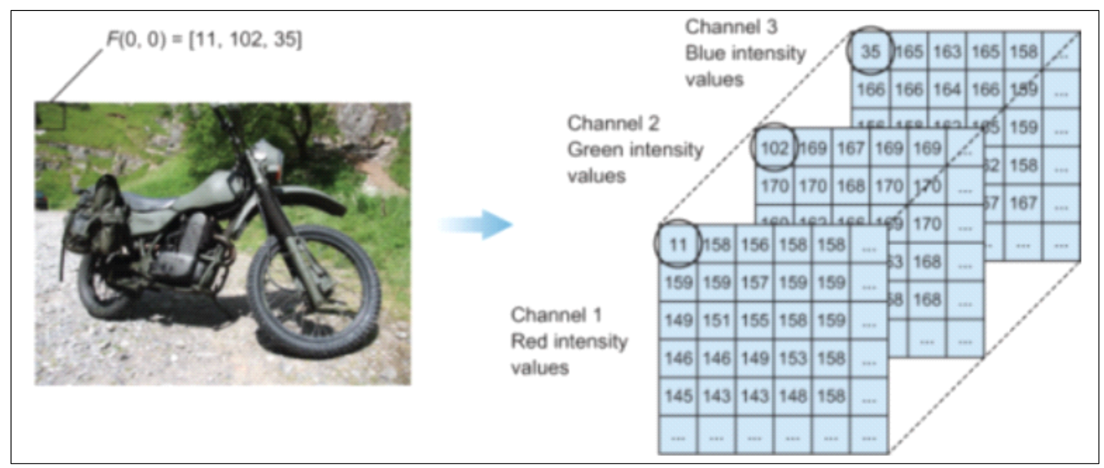
6. 컬러 이미지 분류 문제 프로젝트

가. 컴퓨터는 회색조 이미지를 픽셀값의 2차원 행렬로 다룸(픽셀값은 원색의 강도로 표현됨)

사람이 보는 것(왼쪽)과 컴퓨터가 보는 것(오른쪽)

나. 컴퓨터가 본 컬러 이미지는 너비, 높이, 깊이를 가진 3차원 행렬의 형태임

- RGB 이미지라면 깊이는 채널별로 하나씩 3임
- 예: 28×28 크기의 컬러 이미지를 컴퓨터는 28×28×3 크기의 행렬로 다룸
- 즉, 2차원 행렬 3개(각각 빨간색, 녹색, 파란색에 해당)가 세로로 쌓인 것으로 생각하면 이해하기 쉬움
- 각 행렬의 요솟값은 해당 픽셀의 채널 색의 강도를 의미하며, 이 세 행렬이 겹쳐져야 완전한 컬러 이미지가 됨



컬러 이미지(왼쪽)와 RGB 채널(오른쪽)

● 기억합시다!

- 일반화하기 쉽도록 이미지를 모양이 높이×너비×깊이인 3차원 배열로 다룸
- 회색조 이미지는 깊이가 1이며, 컬러 이미지는 깊이가 3임

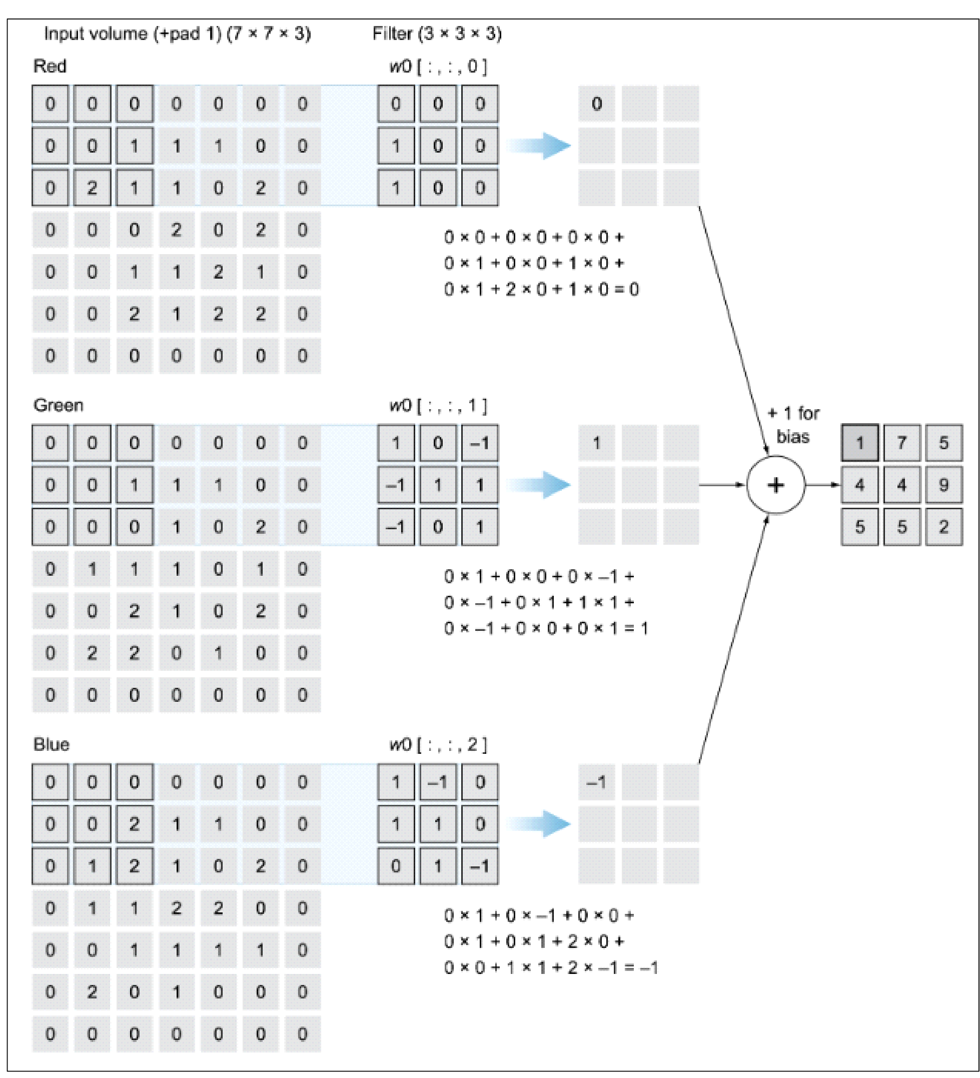
다. 컬러 이미지를 대상으로 합성곱 연산하기

1) 컬러 이미지의 합성곱은 회색조 이미지와 마찬가지로 합성곱 커널을 입력 이미지 위로 이동시키며 특징 맵을 계산하면 됨

2) 그러나 컬러 이미지에 합성곱 커널을 이동시키며 특징 맵을 계산하려면 커널도 3차원이어야 함

● 컬러 이미지의 합성곱 연산 과정

- 색상 채널별로 별도의 필터를 가짐
- 각 필터는 해당하는 채널 이미지 위를 이동함. 겹쳐진 픽셀끼리 픽셀값을 곱하고 곱한 값을 모두 합해 전과 마찬가지로 픽셀값을 계산함
- 세 채널의 픽셀값을 모아 특징 맵의 픽셀값을 계산함. 이때 편향 값 1을 더하는 것을 잊으면 안 됨. 그다음 필터를 스트라이드 값만큼 이동시키고, 특징 맵의 모든 픽셀값을 계산할 때까지 이 과정을 반복함

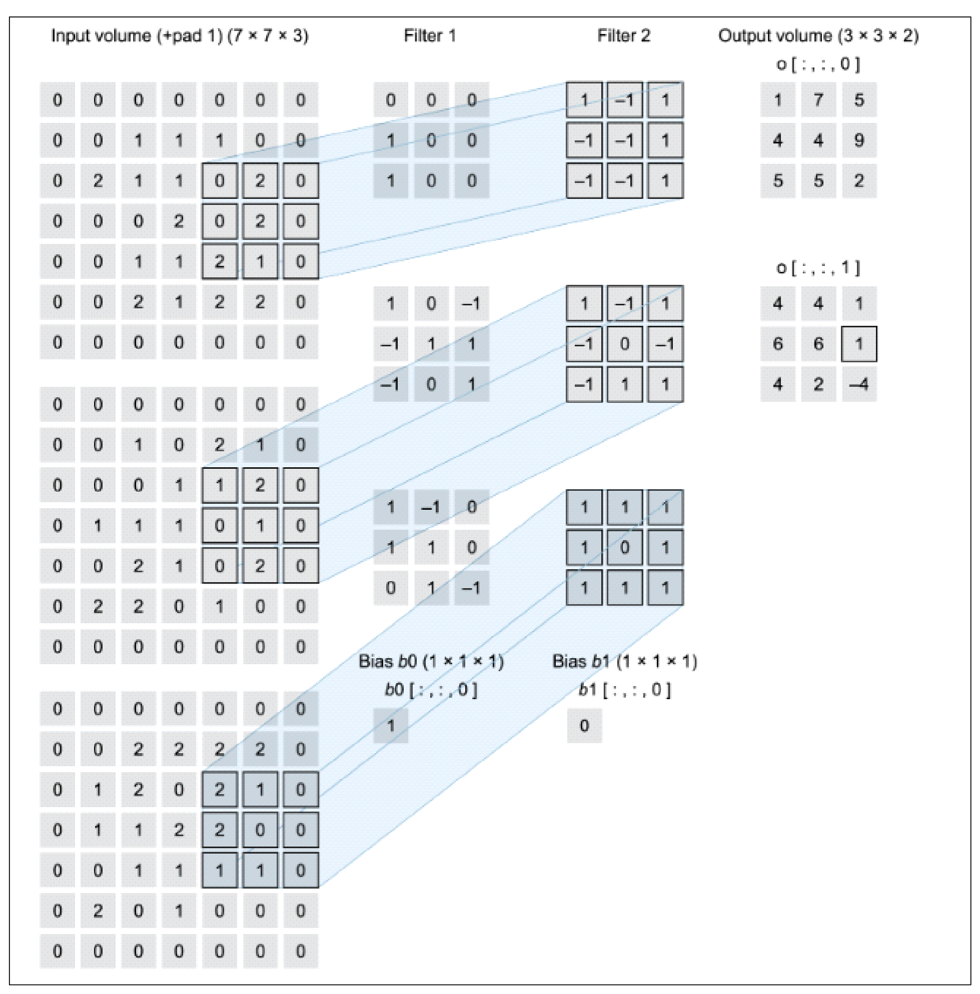


라. 계산 복잡도의 변화

1) 회색조 이미지에 3×3 크기의 필터를 적용하면 필터 하나마다 9개의 파라미터(가중치)가 생김
2) 여기에 컬러 이미지의 경우 필터 자체도 3차원 행렬임 → 다라서 파라미터 수도 3×3×3 = 27개가 됨
3) 파라미터의 수가 증가한 만큼 컬러 이미지는 계산 복잡도가 증가하게 됨

● 무조건 컬러 이미지를 사용하는 것이 좋나요? - 그렇지 않음 - 색상이 중요한 경우의 일이라면 컬러 이미지를 사용하는 것이 좋음 - 그러나 빛의 밝기(강도)만으로 대상의 모양과 특징을 정의할 수 있는 문제일 경우, 계산 복잡도가 낮은 회색조 이미지만으로도 충분함

4) 합성곱 필터 하나를 추가하면 출력되는 특징 맵은 회색조 이미지와 마찬가지로 깊이가 2임



마. 컬러 이미지 분류 문제 실습

1) CNN을 학습해서 CIFAR-10 데이터셋의 이미지를 분류해보자

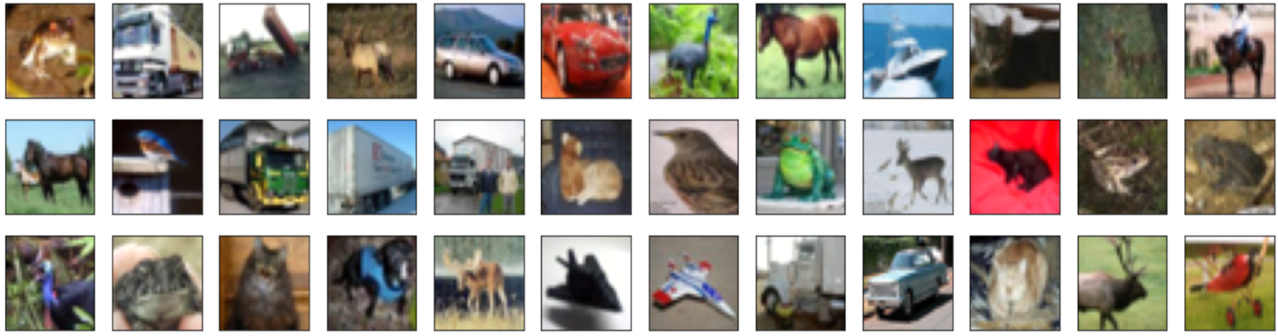
● CIFAR-10 데이터셋이란?
- 컴퓨터 비전 분야에서 널리 알려진 사물 인식 문제 데이터셋
- 32×32 크기의 컬러 이미지 6만장으로 구성됨
- 클래스당 6천 장씩 10가지 클래스로 분류됨

2) 1단계: 데이터셋 읽어 들이기

- 데이터셋을 읽어 들이고, 학습 데이터와 테스트 데이터로 분할함

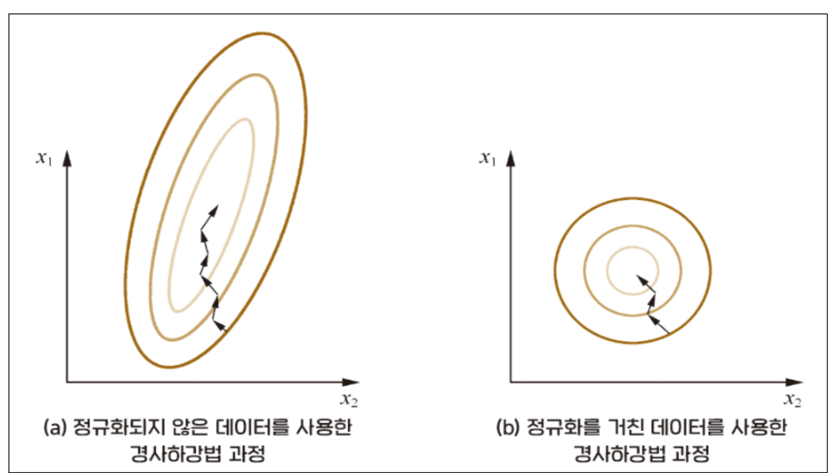
```
import keras
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```

- 실행 결과, 3행 12열 형태의 이미지 데이터가 화면에 출력됨



3) 2단계: 이미지 전처리

- 모델 학습을 시작하기 전에 약간의 데이터 클린징과 전처리를 거침
- 경사 하강법을 사용할 때는 모든 특징의 배율을 비슷하게 맞추는 것이 좋은데, 그렇지 않으면 학습 시간이 길어짐
- 모든 특징을 같은 배율로 정규화하면 둥근 사발 모양의 그래프가 만들어짐(오른쪽). 반면 정규화되지 않아 배율이 서로 다른 특징은 타원형 사발 모양의 그래프를 갖기 쉬움(왼쪽)



- 다음의 코드로 이미지 픽셀값을 정규화(픽셀의 구간을 [0,255]에서 [0,1]로 바꿈)할 수 있음

```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

- CIFAR-10은 이미지 유형에 따라 airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck 등 10개 레이블이 부여되어 있음
- 이 텍스트 레이블 역시 컴퓨터가 다룰 수 있는 숫자 형태로 변환해야 함
- 주로 범주형 변수를 숫자로 변환하는 방법인 ‘원-핫 인코딩(one-hot encoding)’이라는 방법을 사용함

color
red
green
blue
red

one-hot encoding →

color_red	color_green	color_blue
1	0	0
0	1	0
0	0	1
1	0	0

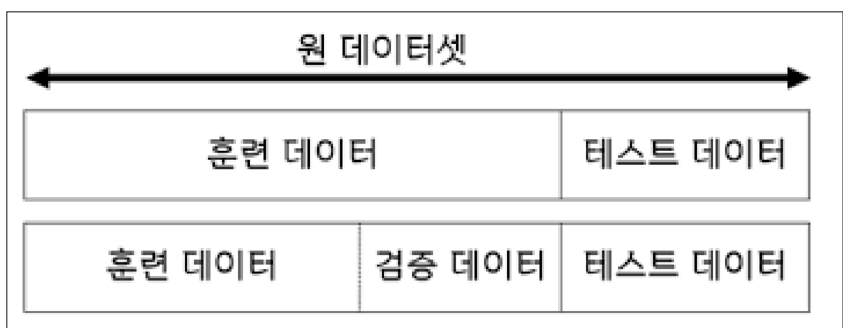
원-핫 인코딩의 적용

- 케라스에서 데이터를 원-핫 인코딩해보자

```
from keras.utils import np_utils

num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

- 학습 데이터를 훈련 데이터와 테스트 데이터로 분할하고, 훈련 데이터를 다시 훈련 데이터와 검증 데이터로 분할해야 함

● 훈련 데이터: 모델을 학습하는 데 사용하는 데이터
● 검증 데이터: 하이퍼파라미터를 튜닝할 때 훈련 데이터에 치우치지 않도록 하기 위한 별도의 데이터
● 테스트 데이터: 모델의 성능을 최종 판단하기 위해 사용하는 데이터



- 케라스에서 데이터 분할을 구현해보자

```
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

print('x_train shape:', x_train.shape)

print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

- 실행 결과, 다음과 같이 출력됨

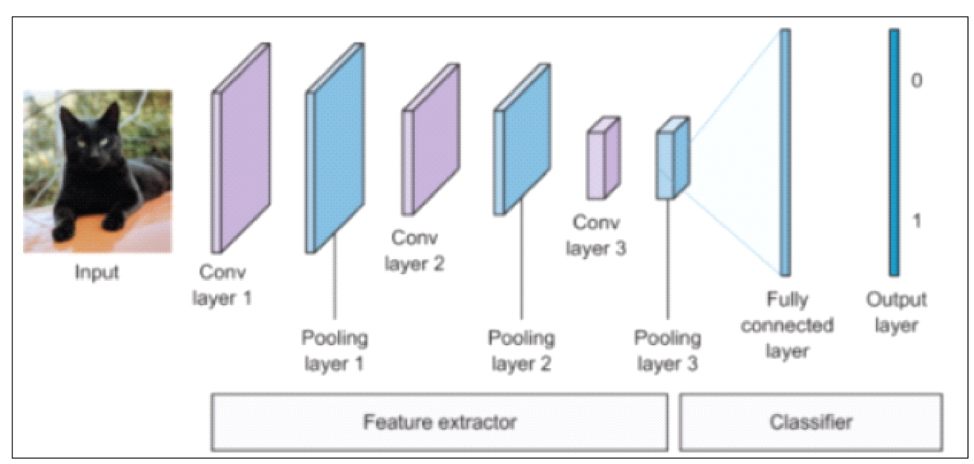
```
x_train shape: (45000, 32, 32, 3)
45000 train samples
10000 test samples
5000 validation samples
```

4) 3단계: 모델 구조 정의하기

- 처음 CNN을 설계하면서 하이퍼파라미터 설정에만 매달리는 것은 바람직하지 못함
- 다른 사람이 만들어놓은 결과물을 참고하는 것이 신경망을 구성하고 하이퍼파라미터를 설정하는 감을 잡는데 유용함

● 다음의 내용을 꼭 숙지합시다!
- CNN을 구성하는 주요 층의 동작 원리(합성곱층, 풀링층, 전결합층, 드롭아웃층)와 필요한 이유
- 하이퍼파라미터의 의미(필터 수, 커널 크기, 스트라이드, 패딩 등)
- 원하는 신경망 구조를 케라스를 사용해 구현하는 방법, 이 교재의 내용을 그대로 코드로 모방할 수 있는 수준

- AlexNet을 토대로 합성곱층과 풀링층 각각 3개씩, 전결합층은 2개로 간략화한 신경망을 구성해보자



- 모든 은닉층의 활성화 함수는 ReLU를 사용함
- 마지막 전결합층은 노드 10개에 활성화 함수는 소프트맥스 함수를 사용해서 분류 대상 클래스의 확률(모든 노드의 출력값을 합하면 1이 됨)을 출력함

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
    activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=32, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=64, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Flatten())

model.add(Dropout(0.3))

model.add(Dense(500, activation='relu'))

model.add(Dropout(0.4))

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

model.summary()
```

- 실행 결과, 출력되는 모델 구조의 개요에서 층을 거칠 때마다 특징 맵의 크기가 달라지는 것을 확인할 수 있음
- 실행 결과 출력된 모델 구조의 개요는 다음과 같음

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 32, 32, 16)        208

 max_pooling2d_5 (MaxPooling  (None, 16, 16, 16)       0
 2D)

 conv2d_6 (Conv2D)           (None, 16, 16, 32)        2080

 max_pooling2d_6 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_7 (Conv2D)           (None, 8, 8, 64)          8256

 max_pooling2d_7 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 1024)              0

 dropout_4 (Dropout)         (None, 1024)              0

 dense_4 (Dense)             (None, 500)               512500

 dropout_5 (Dropout)         (None, 500)               0

 dense_5 (Dense)             (None, 10)                5010

=================================================================
Total params: 528,054
Trainable params: 528,054
Non-trainable params: 0
_________________________________________________________________
```

5) 4단계: 모델 컴파일하기

- 모델의 학습을 시작하기 전에 마지막으로 손실 함수, 최적화 알고리즘, 학습 과정 모니터링에 사용할 평가 지표 등 세 가지 하이퍼파라미터를 결정해야 함

● 손실 함수: 학습 데이터를 대상으로 신경망의 성능을 파악하는 측정 수단
● 최적화 알고리즘: 손실값이 최소가 되도록 파라미터(가중치와 편향)를 최적화하기 위해 사용할 알고리즘
● 평가 지표: 학습 및 테스트 과정에서 사용할 모델의 평가 지표

- 다음과 같이 모델을 컴파일해 줌

```
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
    metrics=['accuracy'])
```

6) 5단계: 모델 학습하기
- .fit() 메서드를 호출하면 학습이 진행됨
- 이는 학습 데이터에 모델을 부합시킨다는 의미임

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
    save_best_only=True)

hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
    validation_data=(x_valid, y_valid), callbacks=[checkpointer],
    verbose=2, shuffle=True)
```

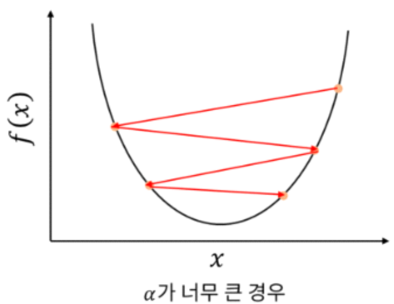
- 100번의 에포크는 시간이 많이 걸리므로 13만큼만 에포크를 설정한 후, 실행 결과는 다음과 같음(아랫부분 생략)

```
Epoch 1/13

Epoch 1: val_loss improved from inf to 1.24128, saving model to model.weights.best.hdf5
1407/1407 - 54s - loss: 1.3625 - accuracy: 0.5094 - val_loss: 1.2413 - val_accuracy: 0.5466 - 54s/epoch - 38ms/step
Epoch 2/13

Epoch 2: val_loss improved from 1.24128 to 1.03798, saving model to model.weights.best.hdf5
1407/1407 - 53s - loss: 1.2006 - accuracy: 0.5717 - val_loss: 1.0380 - val_accuracy: 0.6368 - 53s/epoch - 37ms/step
Epoch 3/13

Epoch 3: val_loss did not improve from 1.03798
1407/1407 - 55s - loss: 1.1166 - accuracy: 0.6081 - val_loss: 1.0558 - val_accuracy: 0.6314 - 55s/epoch - 39ms/step
Epoch 4/13

Epoch 4: val_loss did not improve from 1.03798
1407/1407 - 52s - loss: 1.0642 - accuracy: 0.6267 - val_loss: 1.1533 - val_accuracy: 0.6022 - 52s/epoch - 37ms/step
Epoch 5/13

Epoch 5: val_loss improved from 1.03798 to 0.98312, saving model to model.weights.best.hdf5
1407/1407 - 50s - loss: 1.0246 - accuracy: 0.6431 - val_loss: 0.9831 - val_accuracy: 0.6550 - 50s/epoch - 35ms/step
Epoch 6/13
```

● loss와 acc : 각각 학습 데이터에 대한 오차와 정확도를 의미함
● 한 에포크가 끝날 때마다 val_loss는 감소하고, val_acc는 증가하고 있다면 에포크마다 학습이 제대로 진행되고 있다 이해하면 됨
● 위 코드에서는 검증 데이터에 대한 손실값이 개선된 에포크마다 가중치를 저장함 → 즉, 가장 성능이 좋았던 가중치는 해당 에포크가 끝나면 저장함

● 주의 깊게 관찰해야 하는 현상
- 다음의 그림의 오른쪽과 같이 val_loss가 진동한다면 학습률을 감소시킨 후, 학습을 계속하며 모니터링하는 것이 좋음

α가 너무 작은 경우



α가 너무 큰 경우

- val_loss가 감소하지 않는다면 모델이 데이터에 비해 너무 단순해서 과소적합을 일으켰을 가능성이 높음 → 은닉층을 추가하여 모델 복잡도를 높이는 것이 좋음
- loss는 감소하는데 val_loss가 정체될 경우, 훈련 데이터에 대한 과적합이 발생했다는 신호임 → 드롭아웃 등 과적합을 방지할 수단을 적용해야 함

7) 6단계: val_acc가 가장 좋았던 모델 사용하기
- 검증 데이터에 대한 정확도가 가장 좋았던 가중치를 읽어들임

```
model.load_weights('model.weights.best.hdf5')
```

8) 7단계: 모델 평가하기
- 마지막 단계에서는 모델을 평가하고 정확도를 계산하여 이미지 분류를 올바르게 예측하는 정도를 백분율로 계산함

```
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

- 코드 실행 결과, 다음과 같은 정확도를 얻을 수 있었음

```
Test accuracy: 0.6801000237464905
```

- 모델의 성능을 개선할 수 있을까? 하이퍼파라미터 튜닝을 통해 성능을 끌어 올릴 수 있음

18 하이퍼파라미터 튜닝

1. 성능 지표란
  가. 정확도
    1) 모델의 성능을 평가하는 가장 간단한 수단
    2) 모델의 예측이 정답과 일치한 비율로 정의함
    3) 정확도 = $\frac{\text{정답을 맞힌 횟수}}{\text{전체 표본 수}}$
    4) 어느 특정 질환이 1백만 명 중 1명 꼴로 발병한다면, 99.999%의 정확도를 갖지만 실제 질환을 가진 사람을 찾기 어려운 단점이 있음
  나. 혼동 행렬

		예측	
		양성 Positive	음성 Negative
실제	양성 Positive	진양성 True Positive	위음성 False Negative
	음성 Negative	위양성 False Positive	진음성 True Negative

    1) 진양성(TP): 모델이 양성이라고 정확하게 예측(질환이 있음)
    2) 진음성(TN): 모델이 음성이라고 정확하게 예측(질환이 없음)
    3) 위양성(FP): 실제로는 음성이지만 모델이 양성이라고 잘못 예측(1종 오류)
    4) 위음성(FN): 실제로는 양성이지만 모델이 음성이라고 잘못 예측(2종 오류)
  다. 정밀도와 재현율
    1) 정밀도(=특이성)는 모델이 질환이 없는 사람을 얼마나 잘못 진단했는지 알려줌
      - 정밀도 = $\frac{\text{진양성}}{\text{진양성} + \text{위양성}}$
    2) 재현율(=민감도)은 모델이 질환이 있는 사람을 얼마나 잘못 진단했는지 알려줌
      - 재현율 = $\frac{\text{진양성}}{\text{진양성} + \text{위음성}}$

※ 위양성이 더 나쁘다면 정밀도, 위음성이 더 나쁘다면 재현율을 사용함
※ 스팸메일 분류기 → 일반 메일을 잘못 분류해서는 안되므로 정밀도가 적합함

라. F-점수

1) 재현율(r)과 정밀도(p)를 합쳐 단일 지표인 F-점수로 변환이 가능함

- F-점수 = $\frac{2pr}{p+r}$

2) 질환 판정 모델은 재현율이 더 중요하지만, 위양성 건수가 많을 경우 의료 낭비를 초래할 수 있으므로 정밀도도 함께 살펴봐야 함

마. 모델의 평가 지표는 향후 시스템의 개선 방향을 결정하는 중요한 요소임

1) 지표를 명확하게 정의하지 않으면 머신러닝 시스템을 변경할 때 변경이 개선으로 이어질지 후퇴로 이어질지 분명하게 판단하기 어려움

2. 베이스라인 모델 설정하기

가. 베이스라인 모델 설정 시, 고려할 사항

1) 어떤 유형의 신경망을 사용해야 하는가?

2) 신경망의 층수는 얼마나 두어야 하는가?

3) 활성화 함수는 어떤 것을 사용해야 하는가?

4) 최적화 알고리즘은 어떤 것을 사용해야 하는가?

5) 드롭아웃, 배치 정규화 등의 규제화 기법을 사용해서 과적합을 방지해야 하는가?

나. 비슷한 분야의 문제를 해결하는데 사용된 우수한 모델의 답습이 효율적임

다. 모델을 처음부터 학습하지 않고, 다른 데이터 셋으로 이미 학습된 모델을 가져와 사용하는 전이학습도 고려할 만함

3. 학습 데이터 준비하기

가. 훈련 데이터, 검증 데이터, 테스트 데이터

1) 훈련 데이터는 실제 학습에 사용하고, 테스트 데이터로 모델의 성능을 평가함

2) 가장 중요한 원칙: **테스트 데이터를 학습에 사용해서는 안 된다**는 점

3) 검증 데이터는 무엇이고, 왜 필요할까? 학습 중 한 에포크가 끝나고 끝날 때마다 모델의 정확도와 오차를 확인해서 대략적인 모델의 성능을 체크하고 파라미터를 튜닝해야 함. 이때, 테스트 데이터를 활용하면 원칙을 깨게 되므로 훈련 데이터를 다시 분할한 별도의 검증 데이터를 활용해 파라미터를 튜닝함

```
Epoch 1: val_loss improved from inf to 1.24128, saving model to model.weights.best.hdf5
1407/1407 - 54s - loss: 1.3625 - accuracy: 0.5094 - val_loss: 1.2413 - val_accuracy: 0.5466 - 54s/epoch - 38ms/step
Epoch 2/13
```

나. 훈련 데이터, 검증 데이터, 테스트 데이터를 잘 분할하는 방법

1) 전통적으로 훈련 데이터와 테스트 데이터의 비중은 80:20 또는 70:30을 주로 사용함(검증 데이터를 추가한다면 60:20:20 또는 70:15:15의 비율을 많이 사용함)

2) 데이터셋의 규모가 크다면 검증 데이터와 테스트 데이터의 규모를 전체의 1%로도 충분할 수 있음(예: 데이터 수가 1백만 개라면?)

3) 이왕이면 학습에 충분한 양의 데이터를 사용하는 것이 좋음

※ 훈련 데이터, 검증 데이터, 테스트 데이터가 모두 같은 분포를 따라야 함 (예: 고해상도의 훈련 데이터로 학습한 후, 저해상도의 테스트 데이터로 성능 평가 시 만족스러운 성능을 얻지 못함)

다. 데이터 전처리

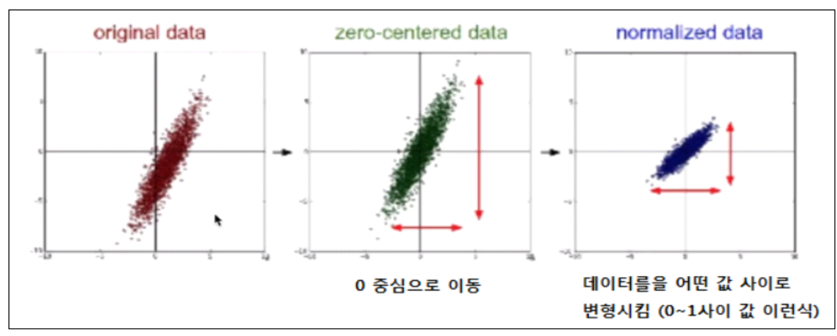
1) 개념: 학습을 시작하기 전, 모델에 데이터를 입력할 수 있도록 데이터를 깔끔하게 정리하는 과정을 뜻함

2) 회색조 이미지 변환 → 문제를 해결하는데 색상 정보가 필요하지 않거나 계산 복잡도를 줄여야 할 경우라면 고려해볼 만함

3) 이미지 크기 조절 → 신경망은 입력되는 모든 이미지의 크기가 같아야 함 (예: 32×32, 28×28, 64×64인 이미지가 있다면 이들 이미지의 크기를 모두 동일하게 조절해야 모델에 이미지 입력이 가능함)

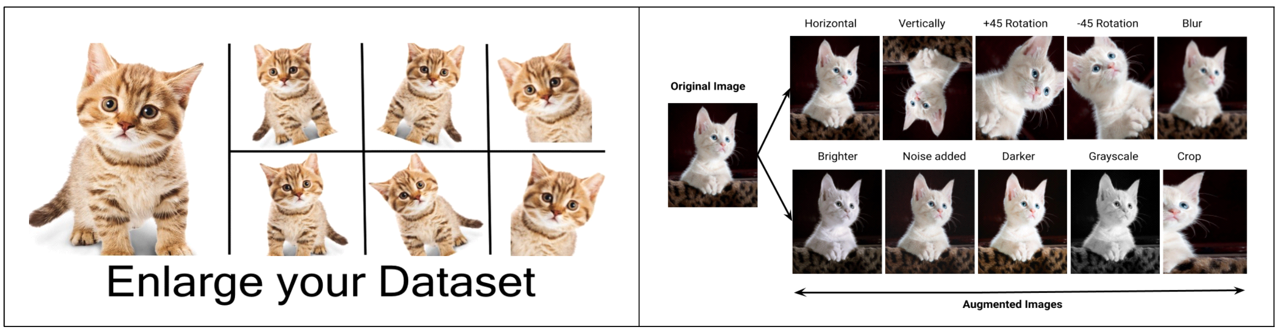
4) 데이터 정규화 → 데이터에 포함된 입력 특징(이미지의 경우 픽셀값)의 배율을 조정해서 비슷한 분포를 갖게 해야 함 → 모든 이미지의 픽셀값을 동일하게 조정하면 모델의 성능이 개선되거나 학습 시간이 단축됨

· 값은 작게: 대부분의 값을 [0, 1] 구간이 되게 함 · 값의 범위는 동일하게: 모든 이미지의 픽셀값 범위를 동일하게 함



모든 픽셀값의 평균과 표준 편차를 구한 후, 각 픽셀값에서 평균을 빼고 표준편차로 나누어 정규화함

5) 데이터 강화 → 데이터가 부족할 경우, 데이터를 뻥튀기하여 부풀림



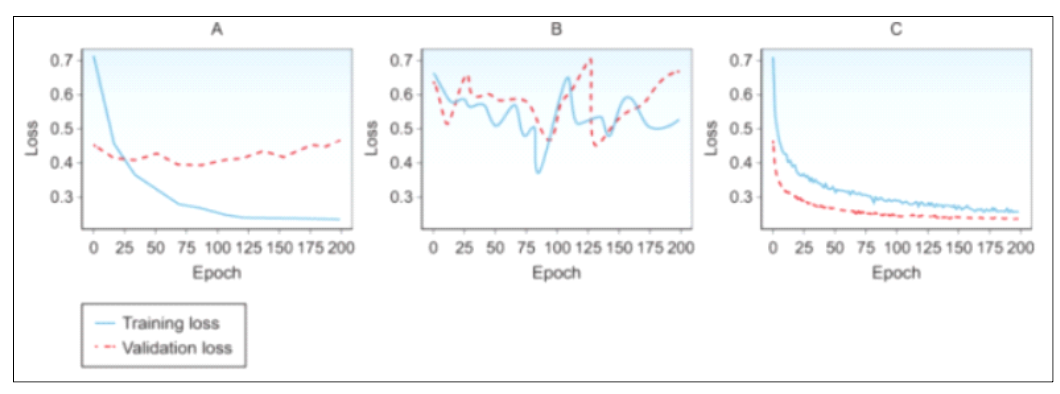
4. 모델 평가와 성능 지표 해석하기

가. 과적합의 징후

1) 훈련 데이터에 대한 성능은 높은데(train error 1%) 검증 데이터에 대한 성능(val error 10%)이 상대적으로 낮다면 과적합을 일으키고 있을 가능성이 높음

2) 훈련 데이터에 대한 성능이 낮다면 과소적합을 일으키고 있을 가능성이 높음 (예: train error 14%, val error 15%면 모델의 표현력이 낮아 데이터에 부합하지 못하는 것임 → 신경망에 은닉층을 추가하거나 학습 에포크 수를 늘리거나 다른 신경망 구조를 사용해야 함)

나. 학습 곡선 그리기



1) A 훈련 데이터에 대한 손실은 감소 중이나, 검증 데이터에 대한 손실은 감소하지 않아 일반화 성능이 나오지 않음(과적합) → 데이터를 더 많이 수집하거나 과적합 방지 조치를 취해줘야 함

2) B 훈련 데이터와 검증 데이터 모두에서 성능이 나오지 않음(과소적합) → 데이터를 더 많이 주기보다는 모델을 복잡하게 만들어야 함

3) C 훈련 데이터와 검증 데이터 모두 오차가 감소 중 → 학습이 잘 되어 좋은 성능을 낼 가능성이 높음

다. 실습: 신경망의 구성, 학습, 평가

1) 의존 모듈 임포트하기

```
from sklearn.datasets import make_blobs
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
```

2) 특징과 분류 클래스가 각각 2개인 실습용 소규모 데이터 셋을 생성함

```
X, y = make_blobs(n_samples=1000, centers=3, n_features=2,
      cluster_std=2, random_state=2)
```

3) 레이블에 원-핫 인코딩을 적용함

```
y = to_categorical(y)
```

4) 훈련 데이터와 테스트 데이터를 80:20으로 분류함

```
n_train = 800
train_X, test_X = X[:n_train, :], X[n_train:, :]
train_y, test_y = y[:n_train], y[n_train:]
print(train_X.shape, test_X.shape)
```

5) 층이 2개인 MLP 모델을 구성하고, 모델의 개요를 보여줌

```
model = Sequential()
model.add(Dense(25, input_dim=2, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
      metrics=['accuracy'])
model.summary()
```

```
Layer (type)            Output Shape            Param #
=================================================================
dense (Dense)           (None, 25)              75

dense_1 (Dense)         (None, 3)               78

=================================================================
Total params: 153
Trainable params: 153
Non-trainable params: 0
```

6) 1,000 에포크 동안 학습함

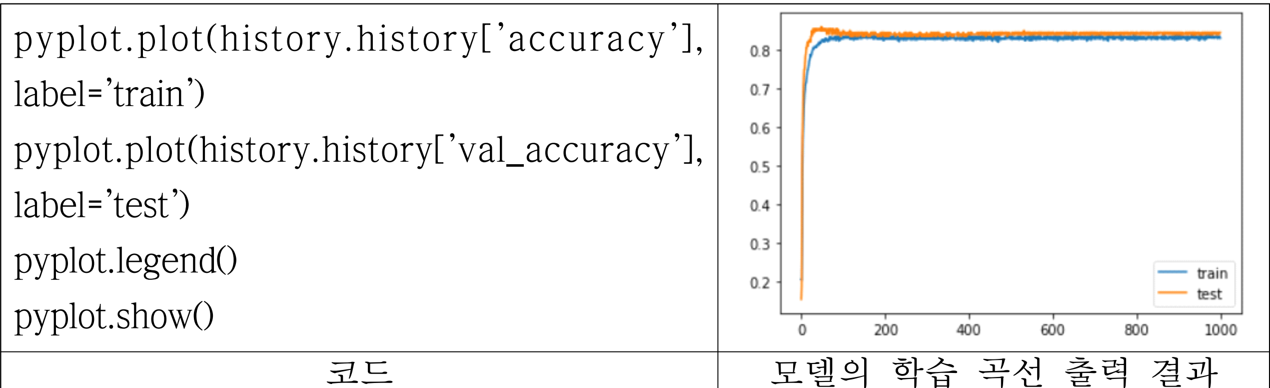
```
history = model.fit(train_X, train_y, validation_data=(test_X, test_y),
    epochs=1000, verbose=1)
```

7) 모델의 성능을 평가함

```
_, train_acc = model.evaluate(train_X, train_y)
_, test_acc = model.evaluate(test_X, test_y)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

```
25/25 [==============================] - 0s 2ms/step - loss: 0.3874 - accuracy: 0.8350
7/7 [==============================] - 0s 3ms/step - loss: 0.3575 - accuracy: 0.8450
Train: 0.835, Test: 0.845
```

8) 정확도를 기준으로 모델의 학습 곡선을 그림



- 학습 곡선 결과 훈련 데이터, 테스트 데이터 모두 비슷한 성능을 보이며 모델이 학습되었음 → 과대적합은 아니나, 84%는 좋은 성능이 아님 → 모델의 복잡도를 높여 과소적합을 방지해야 함

5. 하이퍼파라미터 튜닝과 신경망의 개선

가. 데이터 추가 수집 또는 하이퍼파라미터 튜닝

1) 성능 개선의 첫 단계로 데이터 추가 수집을 시작하는 것이 좋음 (그러나 수집 자체가 어렵거나 비용과 인력의 문제가 발생)

2) 훈련 데이터에 대한 성능이 낮다면 과소적합의 가능성이 있음 → 과소적합은 기존 데이터도 충분히 활용 못하는 상황이므로 데이터를 추가 수집할 필요가 없음 → 하이퍼파라미터를 조정하거나 기존 훈련 데이터를 보다 정교하게 전처리해야 함

3) 훈련 데이터에 대한 성능은 괜찮은데 테스트 데이터에 대한 성능이 떨어진다면 과적합 가능성이 있음 → 데이터 추가 수집이 유효한 경우임

● 낮은 성능의 원인이 데이터라면? → 데이터 전처리나 추가 수집이 필요함 ● 낮은 성능의 원인이 학습 알고리즘이라면? → 신경망의 하이퍼파라미터 조정이 필요함

나. 신경망의 하이퍼파라미터

1) 공짜 점심은 없음

- 만병통치약처럼 모든 상황에 유효한 값은 없음
- 어떤 데이터셋으로 어떤 문제를 해결하느냐에 따라 설정값이 달라짐

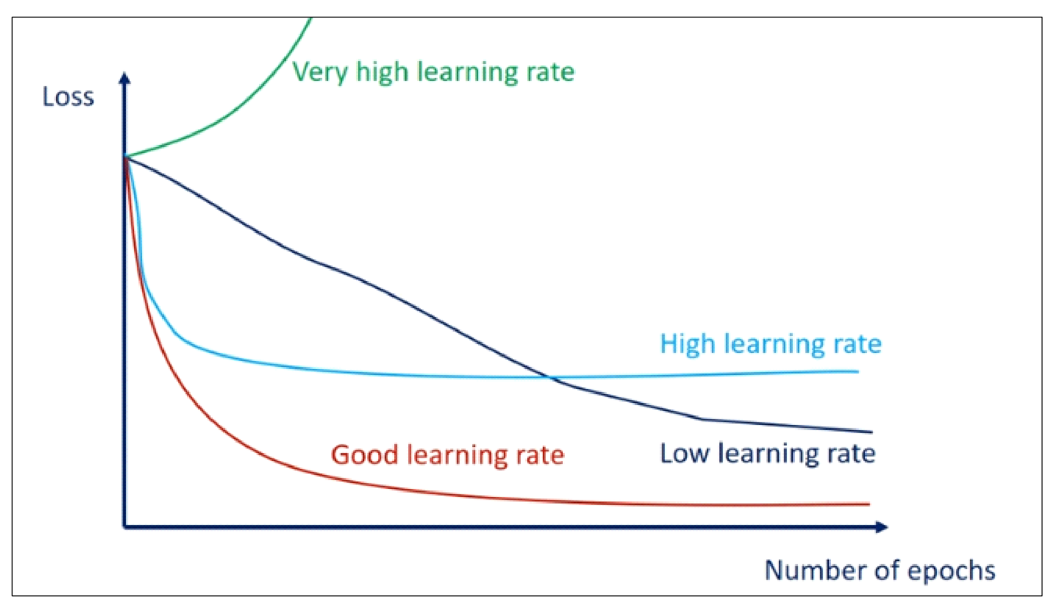
2) 3가지로 분류한 신경망의 하이퍼파라미터

신경망 구조	학습 및 최적화	규제화 및 과적합 방지 기법
· 은닉층 수(신경망의 깊이) · 각 층의 뉴런(층의 폭) · 활성화 함수의 종류	· 학습률, 학습률 감쇠 유형 · 미니배치 크기 · 최적화 알고리즘의 종류 · 에포크 수(조기 종료 포함)	· L2 규제화 · 드롭아웃층 · 데이터 강화

3) 학습률과 학습률 감쇠 유형

학습률	학습률 감쇠 유형	
작은 학습률	손실은 계속 감소하지만 수렴까지 시간이 훨씬 오래 걸림	

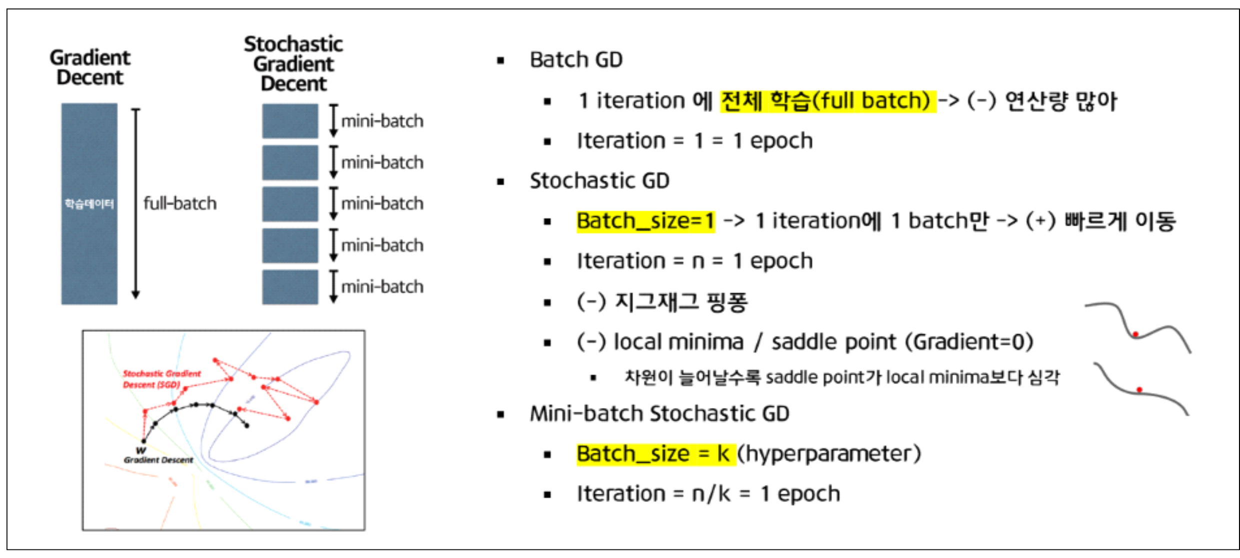
큰 학습률	학습 전과 비교하면 손실은 작지만 최소점과는 거리가 멂	
아주 큰 학습률	초기에는 손실이 감소하지만 가중치가 최소점과 멀어지면서 오히려 손실이 증가함 (발산)	
적당한 학습률	손실이 일정하게 감소하며 최솟값에 도달함	

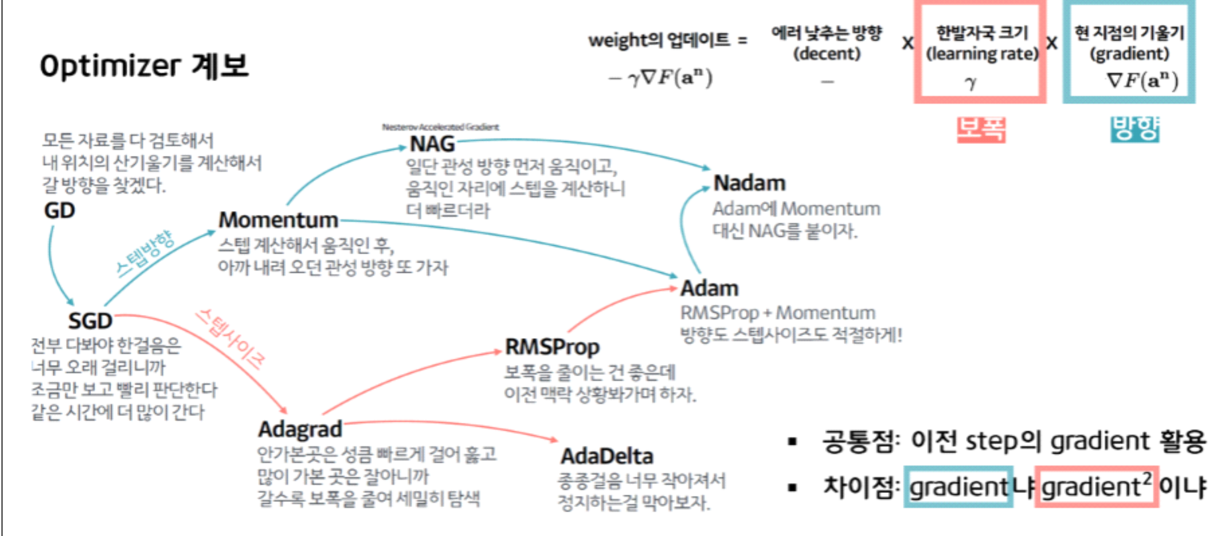


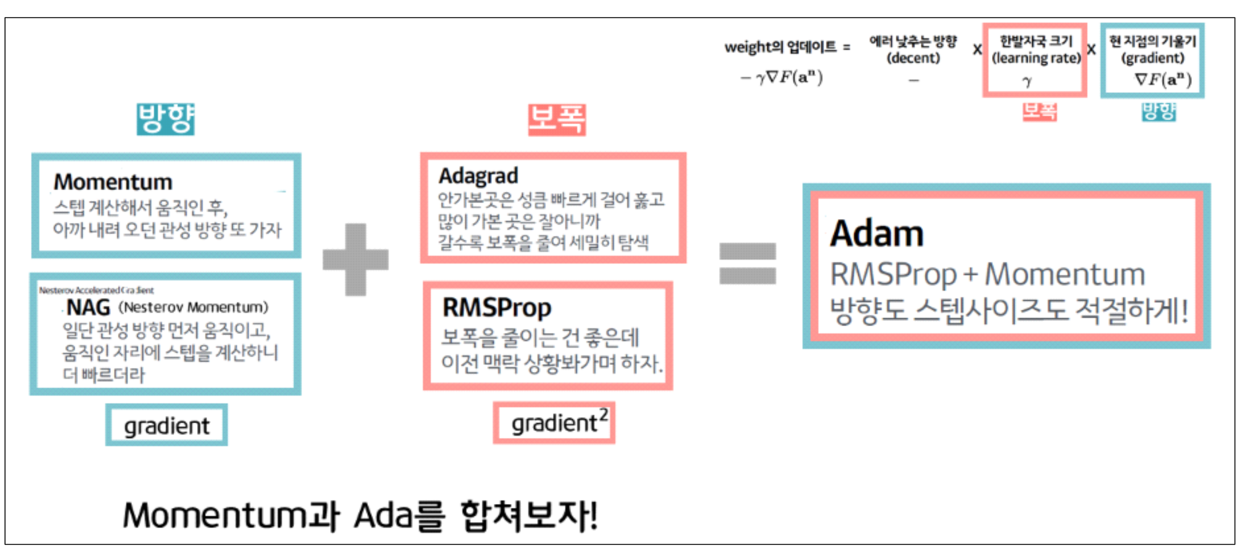
학습률에 따른 학습 양상의 차이

※ 최적의 학습률을 정하는 방법
· 파라미터가 수정될 때마다 val_loss가 감소한다면 정상임 → 학습 지속 · 학습이 끝나고 val_loss가 계속 감소 중이라면 학습률이 너무 작아 파라미터가 수렴하지 못한 상태임. 이런 경우에는 다음의 2가지 방법을 사용함 - 학습률은 그대로 두고 에포크 수를 늘려 학습을 다시 시작함 - 학습률을 조금 증가시키고 학습을 다시 시작함 · val_loss가 증감을 반복하며 진동한다면 학습률이 너무 큰 것임 → 학습률을 감소시킴

6. 최적화 알고리즘







7. 조기 종료

가. 과적합이 발생하기 전에 조기에 학습을 종료하는 알고리즘

나. 검증 오차(Val_loss)를 주시하다가 검증 오차가 증가하기 시작하면 학습을 중지하는 방식임

다. 조기 종료의 장점은 최대 에포크 수 하이퍼파라미터를 덜 신경 써도 된다는 점임 → 최대 에포크 수를 충분히 크게 설정한 후, 조기 종료 설정이 적절한 시점에 학습을 종료하게 하면 됨

라. 케라스의 조기 종료 함수는 다음과 같이 사용함

```
EarlyStopping(monitor='val_loss', min_delta=0, patience=20)
```

1) monitor: 학습 중 주시할 지표 설정

2) min_delta: 주시 중인 지표가 개선 중인지에 대한 기준이 되는 상승폭 지정 (기본값 0도 잘 동작함)

3) patience: 과적합이 발생했다고 판단하는 기준으로, 연속으로 주시 중인 지표가 개선되지 않는 에포크의 수를 의미함(좀 여유 있게 설정하며 10에포크 이상 지표에 개선이 없다면 학습을 중단해도 됨)

8. 과적합 방지를 위한 규제화 기법

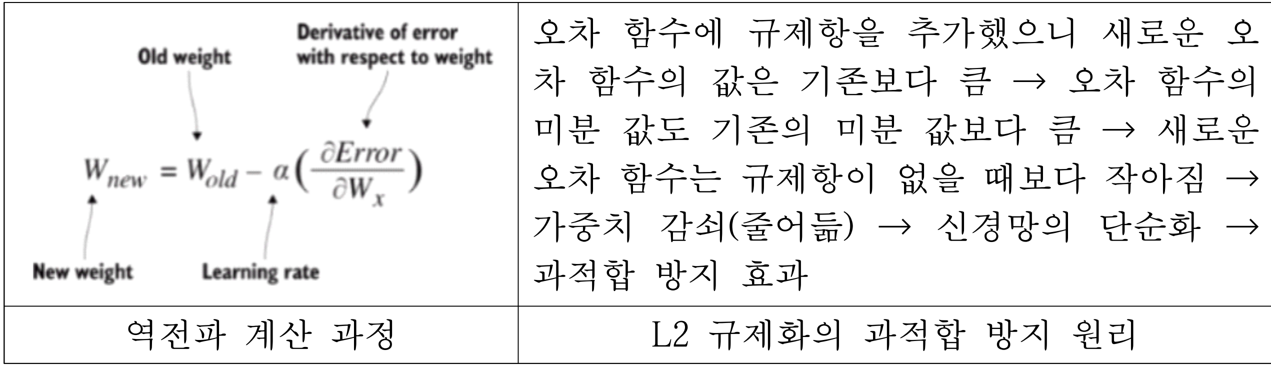
가. L2 규제화

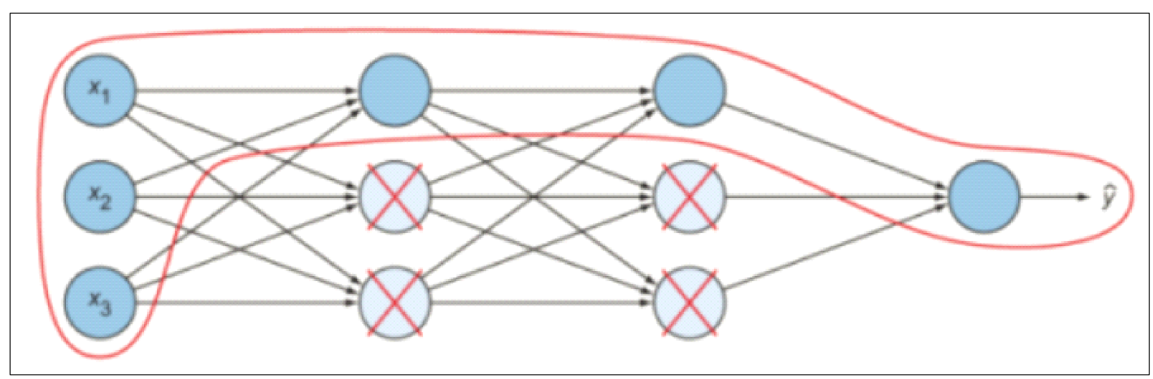
1) L2 규제화의 기본적인 아이디어는 오차 함수에 규제화항을 추가하는 것임

2) 이에 따라 은닉층 유닛의 가중치가 0에 가까워지고(0이 되지는 않음), 모델의 표현력을 감소시키는 데 도움을 줌

3) 규제화 파라미터 값이 크면 가중치가 매우 작아짐

· 오차함수 new = 오차 함수 old + 규제화항

	오차 함수에 규제항을 추가했으니 새로운 오차 함수의 값은 기존보다 큼 → 오차 함수의 미분 값도 기존의 미분 값보다 큼 → 새로운 오차 함수는 규제항이 없을 때보다 작아짐 → 가중치 감쇠(줄어듦) → 신경망의 단순화 → 과적합 방지 효과
역전파 계산 과정	L2 규제화의 과적합 방지 원리



신경망의 단순화와 과적합의 방지

4) 케라스의 L2 규제화는 다음과 같이 사용함

```
model.add(Dense(units=16, kernel_regularizer=regularizers.l2(lambda),
    activation='relu'))
```

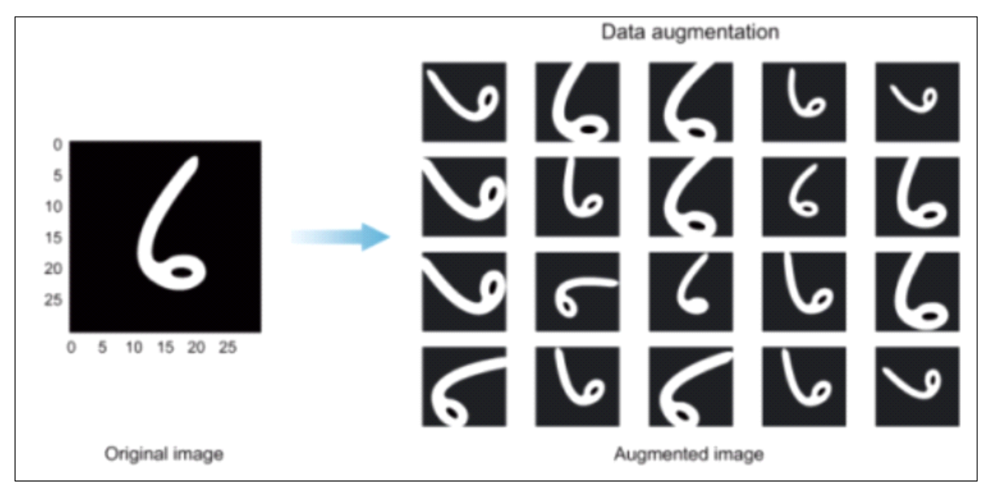
나. 드롭아웃과 L2 규제화

1) 공통점: 이 둘 모두 뉴런의 효율을 떨어뜨려 신경망의 복잡도를 감소시키는 방법임 → 과적합을 억제하는 효과가 있음

2) 차이점: 드롭아웃은 특정 뉴런의 영향력을 완전히 비활성화하는데 비해, L2 규제화는 가중치를 통해 뉴런의 영향력을 억제하는 방식을 사용함

다. 데이터 강화

1) 기존 데이터에 약간의 변형을 가해 훈련 데이터의 양을 늘려 과적합을 방지하는 기법



숫자 6의 손글씨 이미지를 대상으로 적용한 데이터 강화 기법

2) 모델이 특징 학습 중 대상의 원래 모습에 대한 의존도를 낮춰준다는 의미에서 일종의 규제화 기법으로 취급되기도 함

3) 케라스의 이미지 강화는 다음과 같이 사용함

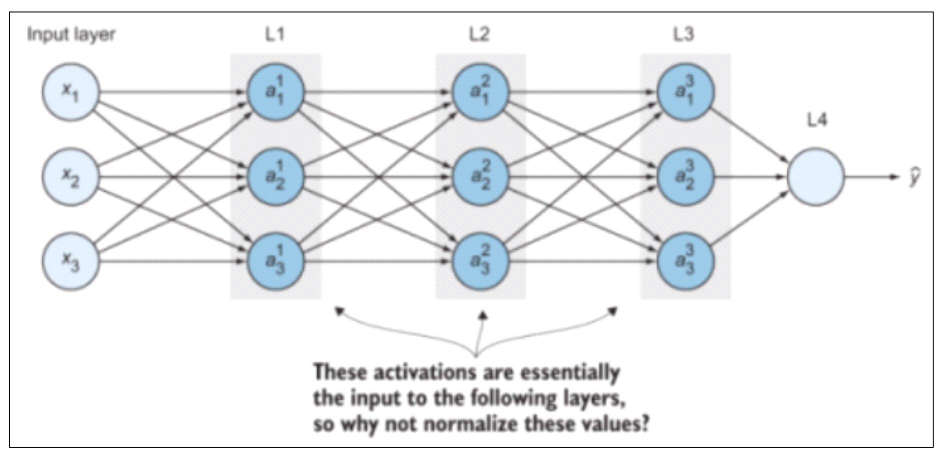
```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)

datagen.fit(training_set)
```

9. 배치 정규화

가. 앞서 다룬 정규화 기법은 입력층에 이미지를 입력하기 위한 학습 데이터의 전처리에 집중되어 있었음(입력층 뿐 아니라 은닉층도 정규화의 장점을 누리자)

나. 이미 추출된 특징을 정규화하면 은닉층도 마찬가지로 정규화의 도움을 받을 수 있음 → 추출된 특징은 변화가 심하므로 정규화를 통해 신경망의 학습 속도와 유연성을 더욱 개선할 수 있음

다. 이렇듯 은닉층에서 이미 추출된 특징을 정규화하는 기법을 배치 정규화라고 함



라. 배치 정규화는 은닉층 유닛을 비활성화하거나 유닛의 값을 변경시키지 않음 → 배치 정규화는 은닉층의 출력값 분포가 변화하는 것을 억제하여 이어지는 층의 학습이 좀 더 안정되도록 돕는 역할을 함

· 값 자체는 변화하지만 평균이 0으로 조정된 입력을 정규화하고, 연산 결과의 배율 및 위치를 조정하므로 값의 평균과 분산은 변화하지 않음

마. 케라스의 L2 규제화는 다음과 같이 사용함

1) 정규화된 결과를 다음 층에 전달할 수 있도록 은닉층 뒤에 배치 정규화층을 추가하는 형태로 구현함

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers.normalization import BatchNormalization

model = Sequential()

model.add(Dense(hidden_units, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.5))

model.add(Dense(units, activation='relu'))

model.add(BatchNormalization())

model.add(Dense(2, activation='softmax'))
```

10. 이미지 분류 정확도 개선 프로젝트

가. CIFAR-10 데이터셋의 분류 모델을 다시 사용함

나. 하이퍼파라미터 개선 기법을 적용해서 현재 약 68%인 정확도를 90%까지 개선해보자

1) 의존 라이브러리 임포트하기

```
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.utils import np_utils
from keras.layers import Dense, Activation, Flatten, Dropout, BatchNormalization, Conv2D, MaxPooling2D
from keras.callbacks import ModelCheckpoint
```

```
from keras import regularizers, optimizers

import numpy as np

from matplotlib import pyplot
```

2) 데이터 내려받기 및 준비

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```

```
print('x_train =', x_train.shape)
print('x_valid =', x_valid.shape)
print('x_test =', x_test.shape)
```

```
x_train = (45000, 32, 32, 3)
x_valid = (5000, 32, 32, 3)
x_test = (10000, 32, 32, 3)
```

- 데이터 정규화: 픽셀값의 평균을 배고, 표준편차로 나누는 방법

```
mean = np.mean(x_train,axis=(0,1,2,3))
std = np.std(x_train,axis=(0,1,2,3))
x_train = (x_train-mean)/(std+1e-7)
x_valid = (x_valid-mean)/(std+1e-7)
x_test = (x_test-mean)/(std+1e-7)
```

- 레이블에 원-핫 인코딩 적용하기

```
num_classes = 10
y_train = np_utils.to_categorical(y_train,num_classes)
y_valid = np_utils.to_categorical(y_valid,num_classes)
y_test = np_utils.to_categorical(y_test,num_classes)
```

- 데이터 강화: 이미지 회전, 이미지 평행 이동(가로, 세로), 가로 반전 적용

```
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False
    )
datagen.fit(x_train)
```

3) 모델 구조 정의하기

- 모델의 표현력을 늘리기 위해 층수를 더욱 늘려보자
- 지난 장의 3CONV + 2FC → 6CONV + 1FC로

```
base_hidden_units = 32
weight_decay = 1e-4
model = Sequential()

# CONV1
model.add(Conv2D(base_hidden_units, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay),
input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV2
model.add(Conv2D(base_hidden_units, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
```

```
model.add(Dropout(0.2))

# CONV3
model.add(Conv2D(base_hidden_units * 2, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV4
model.add(Conv2D(base_hidden_units * 2, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.3))

# CONV5
model.add(Conv2D(base_hidden_units * 4, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# CONV6
model.add(Conv2D(base_hidden_units * 4, kernel_size= 3, padding='same',
          kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# POOL + Dropout
model.add(MaxPooling2D(pool_size=(2,2)))
```

```
model.add(Dropout(0.4))

# FC7
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

model.summary()
```

4) 모델 학습하기

- batch_size의 값이 클수록 학습 속도가 빨라짐
- 초기에 설정한 epochs 후에도 오차가 계속 감소한다면 최대 에포크를 늘려주고, 학습 과정의 추이를 살펴야 함
- 구글 Colab의 런타임 → 런타임 유형 변경 → 하드웨어 가속기를 GPU로 설정(그럼에도 학습에 약 세 시간 걸림)

```
batch_size = 128
epochs = 125

checkpointer = ModelCheckpoint(filepath='model.100epochs.hdf5', verbose=1, save_best_only=True )
optimizer = keras.optimizers.Adam(lr=0.0001,decay=1e-6)

model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])

history = model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size), callbacks=[checkpointer], steps_per_epoch=x_train.shape[0] // batch_size, epochs=epochs, verbose=2, validation_data=(x_valid, y_valid))
```

- 다음과 같이 메시지가 출력되기 시작함

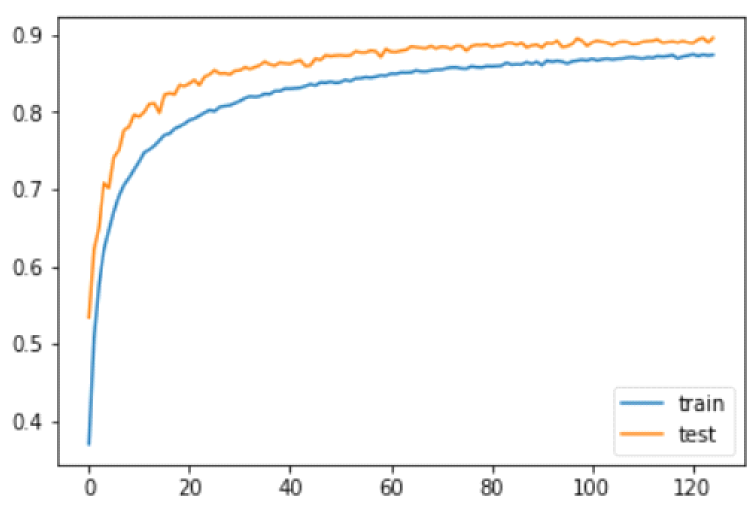
```
Epoch 1: val_loss improved from inf to 1.96815, saving model to model.100epochs.hdf5
351/351 - 32s - loss: 2.8793 - accuracy: 0.2627 - val_loss: 1.9681 - val_accuracy: 0.2830 - 32s/epoch - 91ms/step
Epoch 2/125

Epoch 2: val_loss improved from 1.96815 to 1.59385, saving model to model.100epochs.hdf5
351/351 - 22s - loss: 2.1096 - accuracy: 0.3579 - val_loss: 1.5939 - val_accuracy: 0.4606 - 22s/epoch - 63ms/step
Epoch 3/125
```

5) 모델 평가하기

```
scores = model.evaluate(x_test, y_test, batch_size=128, verbose=1)
print('\nTest result: %.3f loss: %.3f' % (scores[1]*100,scores[0]))
```

- Test result: 90.260 loss: 0.398
- 학습 곡선 그리기: 모델의 학습 과정과 과적합 및 과소적합 판단

```
pyplot.plot(history.history['accuracy'], label='train')
pyplot.plot(history.history['val_accuracy'], label='test')
pyplot.legend()
pyplot.show()
```



※ 추가적인 성능 개선의 여지를 노린다면?
- 최대 에포크 수 늘리기
- 층수 늘리기
- 학습률 낮추기
- 다른 CNN 구조 적용 → 인셉션 혹은 ResNet
- 전이학습 적용