

# 머신 러닝

2023 Fall

강미선

- 딥러닝

- 신경망에 층을 많이 두어(깊게 만들어) 성능을 높이는 기술
- 현재 영상 인식, 음성 인식, 언어 번역 등의 최첨단 인공지능 제품을 딥러닝으로 제작함

- 딥러닝 프로그래밍에 사용되는 패키지

- 텐서플로와 파이토치



## 5.1 딥러닝의 등장

- 1980년대의 깊은 신경망

- 구조적으로는 쉬운 개념

- 다층 퍼셉트론에 은닉층을 많이 두면 깊은 신경망

- 하지만 학습이 잘 안됨

- 그레이디언트 소멸 문제

- 작은 데이터셋 문제(추정할 매개변수가 많아지는데 데이터는 적어 과잉 적합 발생)

- 과다한 계산 시간(비싼 슈퍼컴퓨터)

## 5.1.1 딥러닝의 기술 혁신

- 딥러닝은 새로 창안된 이론이나 원리는 빈약
  - 신경망의 구조와 동작, 학습 알고리즘의 기본 원리는 거의 그대로
- 딥러닝의 기술 혁신 요인
  - 값싼 GPU 등장
    - 10~100배의 속도 향상으로 학습 시간 단축
  - 데이터셋 커짐
    - 인터넷을 통한 데이터 수집과 레이블링(예, 1400만장을 담은 ImageNet)
  - 학습 알고리즘의 발전
    - ReLU 활성화 함수
    - 규제 기법(가중치 감소, 드롭아웃, 조기 멈춤, 데이터 증대, 앙상블 등)
    - 다양한 손실 함수와 옵티마이저 개발

## 5.1.1 딥러닝의 기술 혁신

- 딥러닝으로 인한 인공지능의 획기적 발전
  - 2010년대에 딥러닝의 성공 사례 발표(예, AlexNet)
  - 고전적인 기계 학습을 사용하던 연구 그룹이 딥러닝으로 전환
  - 낮은 성능 때문에 대학 실험실에 머물던 프로토타입 시스템에 획기적인 성능 향상
  - 뛰어난 인공지능 제품이 시장에 속속 등장하여 '인공지능 붐' 조성
  - 딥러닝은 인공지능을 구현하는 핵심 기술로 자리잡음

## 5.1.1 딥러닝의 기술 혁신

### • 학술적인 측면의 혁신 사례

- 컨볼루션 신경망이 딥러닝의 가능성을 엿
  - 작은 크기의 컨볼루션 마스크를 사용하여 우수한 특징을 추출
  - 1990년대 르쿤은 필기 숫자에서 획기적 성능 향상(수표 자동인식 시스템)
- AlexNet은 컨볼루션 신경망으로 자연 영상 인식이 가능하다는 사실을 보여줌
  - 2012년 ILSVRC 대회에서 15.3% 오류율이라는 당시 좋은 성능으로 우승
  - 이후 컴퓨터 비전 연구는 고전적인 기계학습에서 딥러닝으로 대전환



그림 5-2 자연 영상의 심한 변화 예(ImageNet의 고양이 부류 사진)

- 음성 인식에서 혁신
  - 힌튼 교수는 딥러닝을 적용하여 오류율을 단숨에 20%만큼 줄임
  - “10년 걸릴 일을 단번에 이루었다. ... 기술 혁신이 한꺼번에 일어났다”자평

## 5.1.1 딥러닝의 기술 혁신

### NOTE ImageNet과 ILSVRC 대회

자연 영상을 분류하는 문제는 ImageNet이라는 데이터베이스가 만들어진 이후에 다시 주목받기 시작했다. ImageNet은 WordNet의 계층적 단어 분류 체계에 따라 부류를 정하고 약 2만 부류에 대해 각각 500~1,000 장의 영상을 인터넷에서 수집해 구축하였다[Deng2009]. ILSVRC는 ImageNet에서 1,000개의 부류를 뽑아 분류 문제를 푸는 대회이다[Russakovsky2015]. 총 120만 장의 훈련 집합, 5만 장의 검증 집합, 15만 장의 테스트 집합이 주어진다. [그림 5-2]는 1,000개의 부류 중 'cat' 부류의 예제 영상인데, 같은 부류 안에서 변화가 아주 심하다는 것을 확인할 수 있다. 만일 신경망이 cat 부류에 속하는 영상을 보고 'cat(0.9), dog(0.1), ...'이라고 출력하면 1순위로 맞힌 것으로 간주한다. 괄호 속 숫자는 해당 부류에 속할 확률이다. 그리고 만약 'dog(0.5), bear(0.3), swing(0.1), cat(0.09), abacus(0.02), Great white shark(0.01), ...'이라고 출력하면 정답 부류가 5순위 안에 있으므로 5순위로 맞힌 것으로 간주한다. ILSVRC는 1순위 오류율과 5순위 오류율 두 가지로 성능을 측정한다. 2012년에 AlexNet은 5순위 오류율 15.3%를 달성했다. 그리고 2015년에는 마이크로소프트 팀에서 지름길 연결이라는 아이디어를 구현한 ResNet이 3.5%의 5순위 오류율로 우승했다.

## 5.1.2 딥러닝 소프트웨어

### • 대표적인 딥러닝 소프트웨어

- 현재는 텐서플로와 파이토치가 대세
- 대략 텐서플로는 기업, 파이토치는 대학 연구자들이 많이 사용

표 5-1 딥러닝 소프트웨어

이름	개발 그룹	최초 공개일	작성 언어	인터페이스 언어	전이학습 지원	철저한 관리
씨아노(Theano)	몬트리올 대학교	2007년	파이썬	파이썬	○	X
카페(Caffe)	UC버클리	2013년	C++	파이썬, 매트랩, C++	○	X
텐서플로 (TensorFlow)	구글 브레인	2015년	C++, 파이썬, CUDA	파이썬, C++, 자바, 자바스크립트, R, Julia, Swift, Go	○	○
케라스(Keras)	프랑소와 솔레 (François Chollet)	2015년	파이썬	파이썬, R	○	○
파이토치(PyTorch)	페이스북	2016년	C++, 파이썬, CUDA	파이썬, C++	○	○

**TIP** 딥러닝 라이브러리를 보다 폭넓게 살펴보려면 위키피디아에서 'comparison of deep-learning software'를 검색한다.



## 5.1.2 딥러닝 소프트웨어

- 구글 트렌드를 통한 텐서플로와 파이토치의 영향력 비교
  - 이 책은 텐서플로를 채택하여 프로그래밍 실습

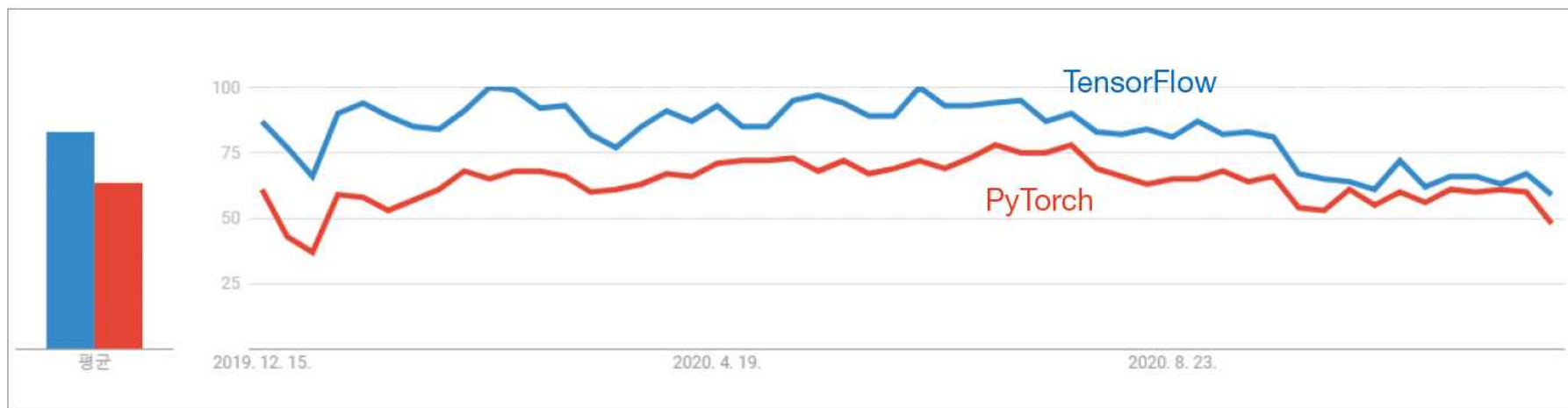


그림 5-3 구글 트렌드를 통해 비교한 텐서플로와 파이토치의 영향력

## 5.2 텐서플로 개념 익히기

- 이 절이 다루는 내용
  - 텐서플로의 호환성 확인
  - 텐서 이해하기

## 5.2.1 텐서플로와 넘파이의 호환

- 텐서플로의 동작을 확인하는 [프로그램 5-1]
  - 03행은 버전 확인
  - 04행은 tf가 제공하는 random 클래스의 uniform 함수로 난수 생성
    - [0,1] 사이의 난수를 2\*3 행렬에 생성

프로그램 5-1

텐서플로 버전과 동작 확인

```
01 import tensorflow as tf
02
03 print(tf.__version__)
04 a=tf.random.uniform([2,3],0,1)
05 print(a)
06 print(type(a))
```

2.0.0

```
tf.Tensor(
[[0.11333311 0.3000914 0.27562833]
 [0.20253515 0.5314199 0.4504068 ]], shape=(2, 3), dtype=float32)
```

tensorflow.python.framework.ops.EagerTensor

## 5.2.1 텐서플로와 넘파이의 호환

- 텐서플로와 넘파이의 호환을 확인하는 [프로그램 5-2]
  - 03행과 04행은 각각 텐서플로와 넘파이로 2\*3 난수 행렬 생성
  - 07행은 텐서플로와 넘파이 배열을 덧셈

프로그램 5-2

tensorflow와 numpy의 호환

```
01 import tensorflow as tf
02 import numpy as np
03
04 t=tf.random.uniform([2,3],0,1)
05 n=np.random.uniform(0,1,[2,3])
06 print("tensorflow로 생성한 텐서:\n",t,"\n")
07 print("numpy로 생성한 ndarray:\n",n,"\n")
08
09 res=t+n      # 텐서 t와 ndarray n의 덧셈
10 print("덧셈 결과:\n",res)
```

## 5.2.1 텐서플로와 넘파이의 호환

tensorflow로 생성한 텐서:

```
tf.Tensor(  
[[0.6962328  0.66963243 0.37720442]  
 [0.3201455  0.18887758 0.31701887]], shape=(2, 3), dtype=float32)
```

Numpy로 생성한 ndarray:

```
[[0.118294  0.98357681 0.23846388]  
 [0.49663294 0.15434053 0.1276853 ]]
```

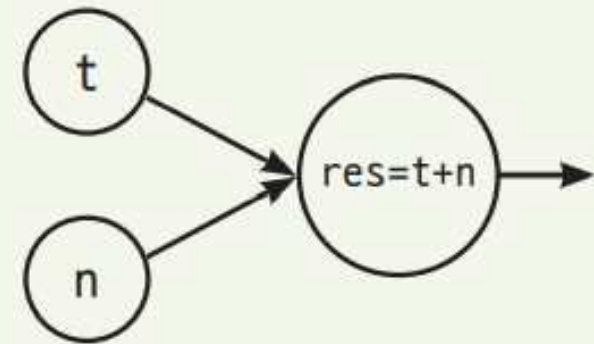
덧셈 결과:

```
tf.Tensor(  
[[0.8145268  1.6532092  0.6156683 ]  
 [0.8167784  0.34321812 0.44470417]], shape=(2, 3), dtype=float32)
```

## 5.2.1 텐서플로와 넘파이의 호환

### NOTE 텐서플로 버전 2.0의 큰 변화

텐서플로 버전 1에서는 tensorflow 객체가 numpy와 호환되지 않는 등 tensorflow로 만든 객체에 제약이 많았다. 또한 [프로그램 5-2]에서 06행을 실행하면 데이터 내용이 출력되지 않았다. 그 이유는 텐서플로가 사용하는 계산 그래프(computation graph) 때문이다. 오른쪽 그림은 [프로그램 5-2]의 계산 절차를 표현한 계산 그래프다.



버전 1에서는 계산 그래프를 만드는 단계와 실제 계산을 실행하는 단계를 엄격하게 구분한다. 따라서 두 단계를 모두 수행한 후에야 데이터 내용을 확인할 수 있다. 버전 1은 계산 그래프를 만들면서 동시에 실행할 수 있는 이거 모드(eager mode)를 제공하는데, 이거 모드를 쓰려면 프로그램에 특수한 코드를 삽입해야 하는 불편이 따른다. 텐서플로 버전 2에서는 이거 모드를 반대로 적용한다. 즉 이거 모드가 기본이고, 계산 그래프를 만드는 단계와 실행하는 단계를 구분하려면 특수한 코드를 삽입해야 한다. 두 단계를 구분하면 속도가 빨라지는 장점이 있다.

## 5.2.2 텐서 이해하기

- 딥러닝에서 텐서

- 다차원 배열을 텐서라 부름
  - 데이터를 텐서로 표현
  - 신경망의 가중치(매개변수)를 텐서로 표현
- 넘파이는 ndarray 클래스, 텐서플로는 Tensor 클래스로 표현. 둘은 호환됨

**TIP** 특징 벡터의 차원과 텐서의 차원을 구별해야 한다. 예를 들어 iris 데이터의 경우 샘플 하나를 (꽃잎의 길이, 꽃잎의 너비, 꽃받침의 길이, 꽃받침의 너비)의 특징 벡터로 표현한다. 이 특징 벡터는 요소가 4개이므로 4차원이라고 말한다. 하지만 이 특징 벡터는 한 방향으로 길쭉한 [그림 5-4(b)]의 1차원 모양 텐서다. 엄밀하게는 1차원 구조의 텐서(tensor with 1-dimensional shape)라고 해야 하지만 줄여서 1차원 텐서라 부른다.



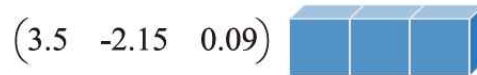
## 5.2.2 텐서 이해하기

### • 0~4차원 구조의 텐서의 예

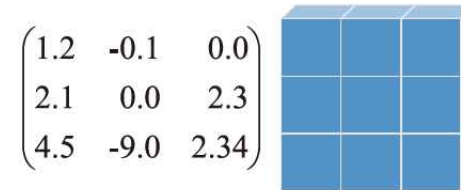
- 1차원: iris 샘플 하나
- 2차원: iris 샘플 여러 개, 명암 영상 한 장
- 3차원: 명암 영상 여러 장, 컬러 영상 한 장
- 4차원: 컬러 영상 여러 장, 컬러 동영상 하나
- 5차원: 컬러 동영상 여러 개



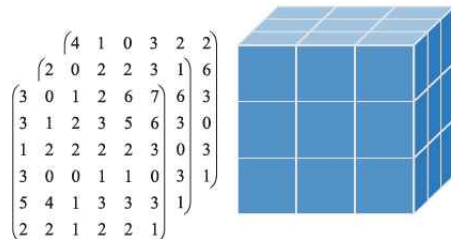
(a) 0차원 텐서(스칼라)



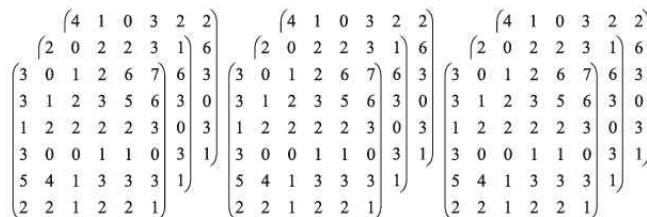
(b) 1차원 텐서(벡터)



(c) 2차원 텐서(행렬)



(d) 3차원 텐서



(e) 4차원 텐서

그림 5-4 텐서의 구조



## 5.2.2 텐서 이해하기

### • 텐서플로가 제공하는 데이터셋의 텐서 구조

- [프로그램 5-3]은 MNIST, cifar10, Boston housing, Reuters 데이터셋의 텐서 구조 확인

프로그램 5-3

텐서플로가 제공하는 데이터셋의 텐서 구조 확인하기

```
01 import tensorflow as tf
02 import tensorflow.keras.datasets as ds
03
04 # MNIST 읽고 텐서 모양 출력
05 (x_train, y_train), (x_test, y_test) = ds.mnist.load_data()
06 yy_train = tf.one_hot(y_train, 10, dtype=tf.int8) # 원한 코드로 변환
07 print("MNIST: ", x_train.shape, y_train.shape, yy_train.shape)
08
09 # CIFAR-10 읽고 텐서 모양 출력
10 (x_train, y_train), (x_test, y_test) = ds.cifar10.load_data()
11 yy_train = tf.one_hot(y_train, 10, dtype=tf.int8)
12 print("CIFAR-10: ", x_train.shape, y_train.shape, yy_train.shape)
13
14 # Boston Housing 읽고 텐서 모양 출력
15 (x_train, y_train), (x_test, y_test) = ds.boston_housing.load_data()
16 print("Boston Housing: ", x_train.shape, y_train.shape)
17
18 # Reuters 읽고 텐서 모양 출력
19 (x_train, y_train), (x_test, y_test) = ds.reuters.load_data()
20 print("Reuters: ", x_train.shape, y_train.shape)
```

레이블 정보를 원한 코드로 변환

[5,0,4,...]처럼 표현

[[6],[9],[9],...]처럼 표현

MNIST: (60000, 28, 28) (60000,) (60000, 10)  
CIFAR-10: (50000, 32, 32, 3) (50000, 1) (50000, 1, 10)  
Boston Housing: (404, 13) (404,)  
Reuters: (8982,) (8982,)

13개 특징으로 표현되는 404개 샘플

## 5.2.2 텐서 이해하기

- 텐서플로가 제공하는 데이터셋의 텐서 구조

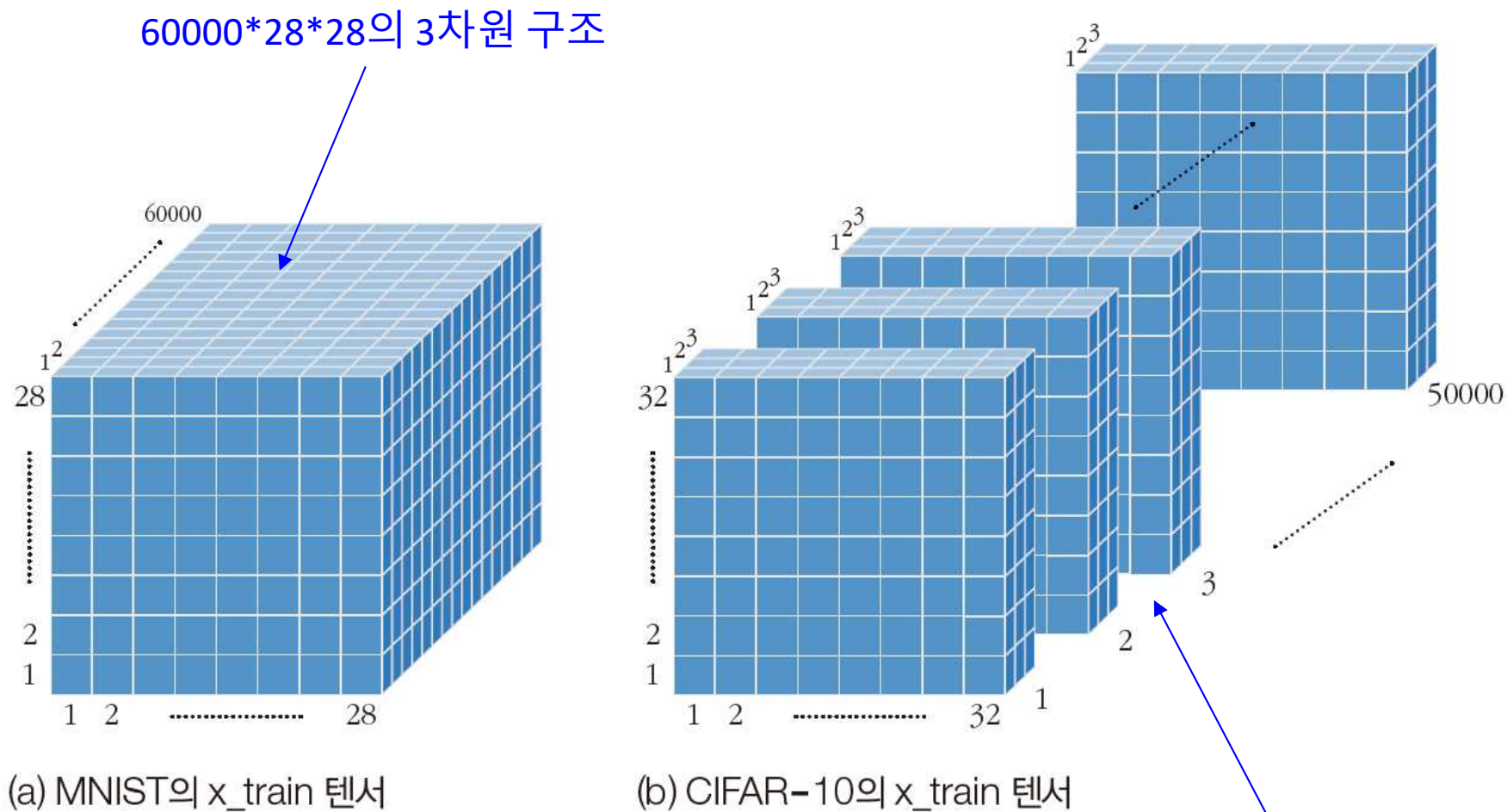


그림 5-5 데이터셋의 텐서 구조

50000\*32\*32\*3의 4차원 구조

## 5.3.1 sklearn의 표현력 한계

- [프로그램 4-3]의 골격과 표현력의 한계

```
from sklearn.neural_network import MLPClassifier

digit=datasets.load_digits()
x_train,x_test,y_train,y_test=train_test_split(digit.data,digit.
                                                target,train_size=0.6)

mlp=MLPClassifier(hidden_layer_sizes=(100),learning_rate_init=0.001, batch_
                  size=32,max_iter=300,solver='sgd',verbose=True)
mlp.fit(x_train,y_train)
```

- 딥러닝은 sklearn의 ②행으로 표현 불가능(딥러닝은 서로 다른 기능의 층을 쌓는 방식)
- 텐서플로나 파이토치는 딥러닝의 복잡도를 지원할 수 있게 완전히 새로 설계함

## 5.3.2 텐서플로로 퍼셉트론 프로그래밍

- 학습된 퍼셉트론의 동작을 확인하는 [프로그램 5-4]
  - 08~09행의 Variable 함수는 그레이디언트를 구하고 가중치를 갱신하는 연산을 지원

프로그램 5-4    텐서플로 프로그래밍: [예제 4-1]의 퍼셉트론 동작

```
01 import tensorflow as tf
02
03 # OR 데이터 구축
04 x=[[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]]
05 y=[[-1],[1],[1],[1]]
06
07 # [그림 4-3(b)]의 퍼셉트론
08 w=tf.Variable([[1.0],[1.0]])
09 b=tf.Variable(-0.5)
10
11 # 식 4.3의 퍼셉트론 동작
12 s=tf.add(tf.matmul(x,w),b)
13 o=tf.sign(s)
14
15 print(o)
```

```
tf.Tensor(
[[-1.]
 [ 1.]
 [ 1.]
 [ 1.]], shape=(4, 1), dtype=float32)
```

## 5.3.2 텐서플로로 퍼셉트론 프로그래밍

- 퍼셉트론을 학습하는 [프로그램 5-5]

프로그램 5-5

텐서플로 프로그래밍: 퍼셉트론 학습

```
01 import tensorflow as tf
02
03 # OR 데이터 구축
04 x=[[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]]
05 y=[[-1],[1],[1],[1]]
06
07 # 가중치 초기화
08 w=tf.Variable(tf.random.uniform([2,1],-0.5,0.5))
09 b=tf.Variable(tf.zeros([1]))
10
11 # 옵티마이저
12 opt=tf.keras.optimizers.SGD(learning_rate=0.1)
13
14 # 전방 계산(식 (4.3))
15 def forward():
16     s=tf.add(tf.matmul(x,w),b)
17     o=tf.tanh(s)
18     return o
19
```



## 5.3.2 텐서플로로 퍼셉트론 프로그래밍

```
20 # 손실 함수 정의
21 def loss():
22     o=forward()
23     return tf.reduce_mean((y-o)**2)
24
25 # 500세대까지 학습(100세대마다 학습 정보 출력)
26 for i in range(500):
27     opt.minimize(loss, var_list=[w,b])
28     if(i%100==0): print('loss at epoch',i,'=',loss().numpy())
29
30 # 학습된 퍼셉트론으로 OR 데이터를 예측
31 o=forward()
32 print(o)
```

```
loss at epoch 0 = 0.8947841
loss at epoch 100 = 0.09448623
loss at epoch 200 = 0.04298807
loss at epoch 300 = 0.026879318
loss at epoch 400 = 0.019305129
```

```
tf.Tensor(
[[-0.81562793]
 [ 0.8859462 ]
 [ 0.88595307]
 [ 0.9992574 ]], shape=(4, 1), dtype=float32)
```

← 학습 과정을 모니터링

← 학습을 마친 모델로 예측 수행  
(네 개 샘플 모두 옳게 분류)

## 5.3.3 케라스 프로그래밍

- [프로그램 5-5]의 문제점

- 신경망의 동작을 직접 코딩해야 함
- 케라스는 이런 부담을 덜기 위해 탄생

- 프로그래밍의 추상화

- 컴퓨터 프로그래밍은 추상화를 높이는 방향으로 발전해 옴 (디테일을 숨김)
- 텐서플로 자체가 아주 높은 추상화 수준이지만 추가로 추상화할 여지 있음
- 케라스는 이 여지를 활용한 라이브러리
  - `model.add(Dense(노드 개수, 활성화 함수,...))` 방식의 코딩
- `keras.io` 공식 사이트에 있는 케라스의 철학

Being able to go from idea to result with the least possible delay is key to doing good research.

아이디어를 될 수 있는 대로 빨리 결과로 연결하는 능력은 훌륭한 연구의 핵심이다.

Keras is an API designed for human beings, not machines.

케라스는 기계가 아닌 사람을 위해 설계된 API이다.

## 5.3.3 케라스 프로그래밍

### • 케라스로 퍼셉트론 프로그래밍 [프로그램 5-6]

- 01~03행의 tensorflow.keras: tensorflow의 하위 클래스로 keras(텐서플로 버전 2부터 케라스가 텐서플로에 편입됨)
- keras 클래스의 중요한 세 가지 하위 클래스

자주 등장하니  
꼭 기억

- models 클래스: Sequential과 functional API 모델 제작 방식 제공
- layers 클래스: 다양한 종류의 층 제공
- optimizers 클래스: 다양한 종류의 옵티마이저 제공

프로그램 5-6

케라스 프로그래밍: 퍼셉트론 학습

```
01 from tensorflow.keras.models import Sequential
02 from tensorflow.keras.layers import Dense
03 from tensorflow.keras.optimizers import SGD
04
```

Sequential은 층을 한 줄로 쌓는데 사용

완전 연결 층

SGD 옵티마이저



## 5.3.3 케라스 프로그래밍

### • 전형적인 절차

- 데이터 구축 → 신경망 구조 설계 → 학습 → 예측

```
05 # OR 데이터 구축
06 x=[[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]]
07 y=[[-1],[1],[1],[1]]
08
09 n_input=2
10 n_output=1
11
12 perceptron=Sequential()
13 perceptron.add(Dense(units=n_output,activation='tanh',
14                       input_shape=(n_input,),kernel_initializer='random_uniform',
15                       bias_initializer='zeros'))
16
17
18 perceptron.compile(loss='mse',optimizer=SGD
19                   (learning_rate=0.1),metrics=['mse'])
20 perceptron.fit(x,y,epochs=500,verbose=2)
21
22 res=perceptron.predict(x)
23 print(res)
```

Sequential 클래스로 객체를 생성

add 함수로 Dense (완전연결) 층을 쌓음

신경망 구조 설계

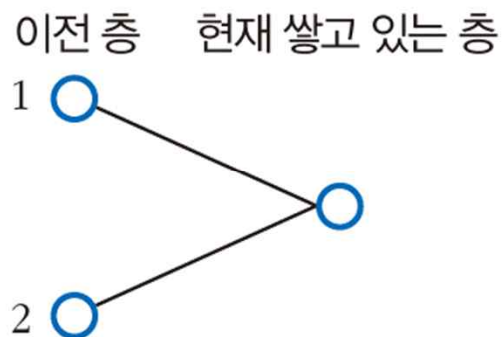
신경망 학습

학습된 신경망으로 예측

## 5.3.3 케라스 프로그래밍

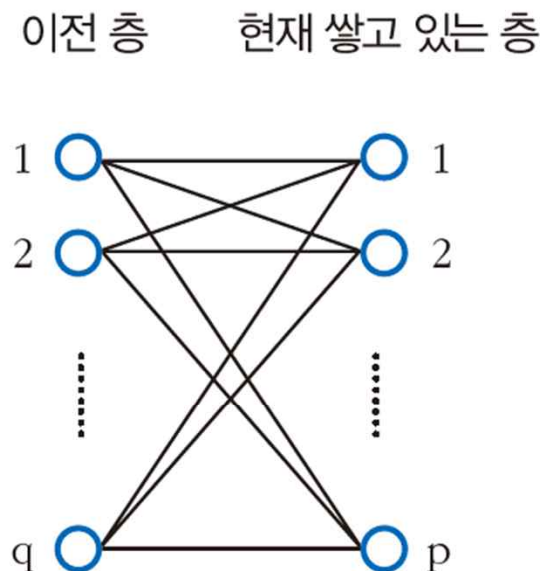
- Dense로 완전연결층을 쌓는 방식

- [그림 5-6]은 units와 input\_shape 매개변수에 대한 설명



`Dense(units=1, ..., input_shape=(2,))`

(a) [프로그램 5-6]의 13행으로 쌓은 층



`Dense(units=p, ..., input_shape=(q,))`

(b) p개 노드를 가진 층을 q개 노드를 가진 층 뒤에 쌓음

**그림 5-6** Dense 클래스로 완전연결층을 쌓음

## 5.3.3 케라스 프로그래밍

### • 실행 결과

```
Epoch 1/200  
4/4 - 0s - loss: 1.4654  
Epoch 2/200  
4/4 - 0s - loss: 1.1761  
...  
Epoch 199/200  
4/4 - 0s - loss: 0.0148  
Epoch 200/200  
4/4 - 0s - loss: 0.0147
```

← 16행의 fit 함수의 학습 과정

```
[[ -0.8179741]  
 [ 0.886851 ]  
 [ 0.8878835]  
 [ 0.9992872]]
```

← 18행의 predict 함수의 예측 결과

## 5.3.3 케라스 프로그래밍

### NOTE 케라스의 지위 업 또는 다운?

텐서플로 버전 1에서는 케라스를 별도의 라이브러리로 취급했다. 텐서플로를 설치한 다음에 별도로 케라스를 설치해야 사용할 수 있었고, 프로그래밍할 때도 다음과 같이 별도의 라이브러리로 취급해야 했다.

```
from keras.models import Sequential
```

텐서플로 버전 2에서는 케라스가 텐서플로에 편입되었다. 따라서 텐서플로를 설치하면 케라스가 따라오기 때문에 별도로 설치할 필요가 없다. [프로그램 5-6]의 01행에 있는 `from tensorflow.keras.models import Sequential`처럼 케라스가 텐서플로의 하위 클래스가 되었다. 둘이 한 몸이 된 것이다. 이 책은 앞으로 [프로그램 5-6]을 케라스로 프로그래밍했다고 하지 않고 텐서플로로 프로그래밍했다고 말할 것이다. 케라스는 지위가 약해진 것일까? 강해진 것일까?

- 앞으로 케라스로 프로그래밍

- 케라스가 텐서플로에 편입되었으므로  
텐서플로로 프로그래밍한다고 말할 것임

- 이 절은

- MNIST와 fashion MNIST 데이터를 다층  
퍼셉트론으로 인식하는 프로그래밍 실습

## 5.4.1 MNIST 인식

- 다층 퍼셉트론으로 MNIST 인식하는 [프로그램 5-7(a)]
  - 퍼셉트론을 코딩한 [프로그램 5-6]과 디자인 패턴 공유. 복잡도만 다르지 핵심은 같음
- 단계 1: 데이터 준비

프로그램 5-7(a)    텐서플로 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import mnist
04
05 from tensorflow.keras.models import Sequential
06 from tensorflow.keras.layers import Dense
07 from tensorflow.keras.optimizers import Adam
08
09 # MNIST 읽어 와서 신경망에 입력할 형태로 변환
10 (x_train, y_train), (x_test, y_test) = mnist.load_data()
11 x_train = x_train.reshape(60000, 784)
12 x_test = x_test.reshape(10000, 784)
13 x_train = x_train.astype(np.float32)/255.0
14 x_test = x_test.astype(np.float32)/255.0
15 y_train = tf.keras.utils.to_categorical(y_train, 10)
16 y_test = tf.keras.utils.to_categorical(y_test, 10)
17
```

11~12행: reshape 함수로 2차원 구조의 텐서를 1차원 구조로 변환

13~14행: float32 데이터형으로 변환하고 [0,255] 범위를 [0,1] 범위로 정규화

15~16행: 레이블을 원핫 코드로 변환

# 텐서 모양 변환

# ndarray로 변환

# 원핫 코드로 변환



## 5.4.1 MNIST 인식

### • 단계 2: 신경망 구조 설계

- 18~20행: 신경망의 입력층, 은닉층, 출력층의 노드 개수 설정
- 22행: Sequential 모델을 생성하여 mlp 객체에 저장
- 23행: 은닉층을 추가(input\_shape은 입력층, units은 현재 쌓고 있는 은닉층으로 설정)
- 24행: 출력층을 추가(input\_shape은 생략 가능, units은 현재 쌓고 있는 출력층으로 설정)

```
18 n_input=784
19 n_hidden=1024
20 n_output=10
21
22 mlp=Sequential()
23 mlp.add(Dense(units=n_hidden,activation='tanh',input_shape=(n_input,),
24               kernel_initializer='random_uniform',bias_initializer='zeros'))
25 mlp.add(Dense(units=n_output,activation='tanh',kernel_
26               initializer='random_uniform',bias_initializer='zeros'))
```

신경망  
구조 설계

## 5.4.1 MNIST 인식

### • 단계 3: 신경망 학습

- 26행: compile 함수로 학습을 준비함(loss 매개변수는 손실 함수, optimizers는 옵티마이저 설정)
- 27행: fit 함수는 실제 학습을 수행(batch\_size는 미니배치 크기, epochs는 최대 세대수, validation\_data는 학습 도중에 사용할 검증 집합 설정)

### • 단계 4: 예측

- 29행: evaluate 함수로 정확률 측정

```
26 mlp.compile(loss='mean_squared_error',optimizer=Adam(learning_
   rate=0.001),metrics=['accuracy'])
27 hist=mlp.fit(x_train,y_train,batch_size=128,epochs=30,validation
   _data=(x_test,y_test),verbose=2)
28
29 res=mlp.evaluate(x_test,y_test,verbose=0)
30 print("정확률은",res[1]*100)
```

손실 함수로 MSE 사용      옵티마이저로 Adam 사용

신경망 학습

학습된 신경망으로 예측

학습 도중에 발생한 정보를 hist 객체에 저장해 둬(시각화에 활용)



## 5.4.1 MNIST 인식

- 실행 결과

- 테스트 집합에 대해 97.65% 정확률

```
Train on 60000 samples, validate on 10000 samples
```

```
Epoch 1/30
```

```
60000/60000 - 2s - loss: 0.0427 - accuracy: 0.8492 - val_loss: 0.0272 - val_accuracy: 0.9173
```

```
Epoch 2/30
```

```
60000/60000 - 2s - loss: 0.0223 - accuracy: 0.9305 - val_loss: 0.0184 - val_accuracy: 0.9432
```

```
...
```

```
Epoch 30/30
```

```
60000/60000 - 2s - loss: 0.0049 - accuracy: 0.9919 - val_loss: 0.0074 - val_accuracy: 0.9765
```

```
정확률은 97.64999747276306
```

## 5.4.2 학습 곡선 시각화

- 학습 곡선을 시각화 하는 [프로그램 5-7(b)]
  - hist 객체가 가진 정보를 이용하여 학습 곡선을 그림

프로그램 5-7(b)

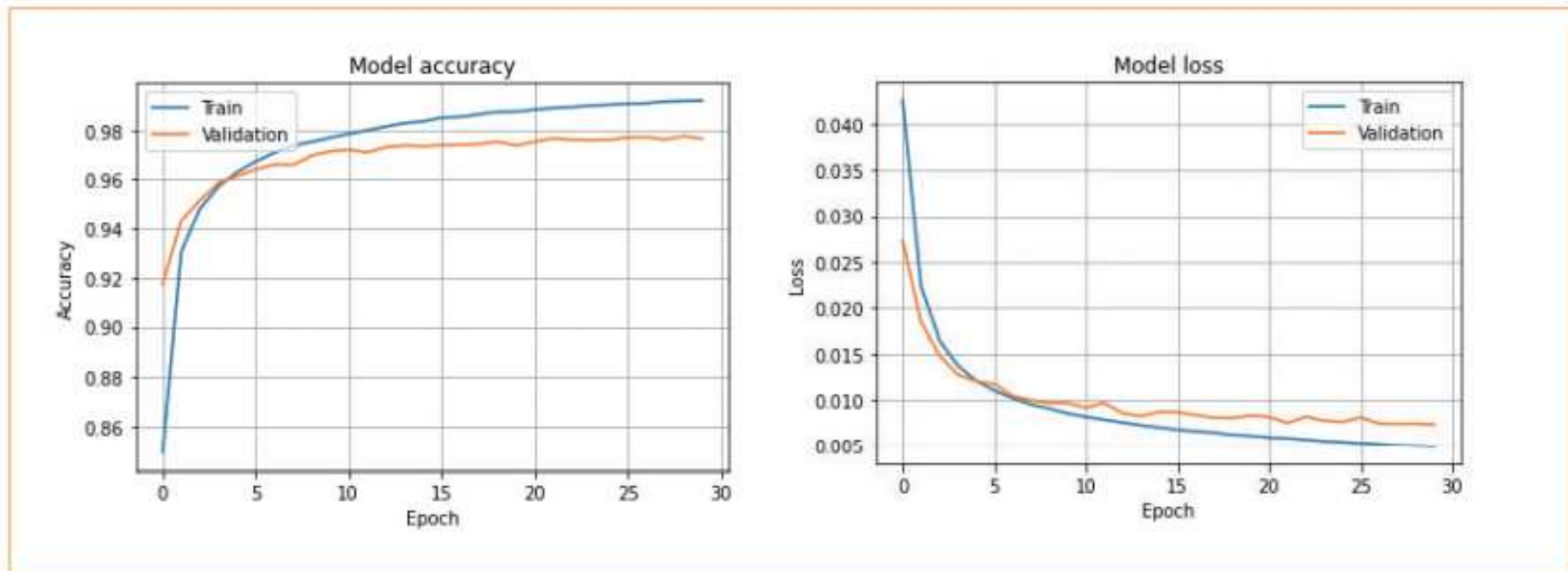
텐서플로 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
31 import matplotlib.pyplot as plt
32
33 # 정확률 곡선
34 plt.plot(hist.history['accuracy'])
35 plt.plot(hist.history['val_accuracy'])
36 plt.title('Model accuracy')
37 plt.ylabel('Accuracy')
38 plt.xlabel('Epoch')
39 plt.legend(['Train', 'Validation'], loc='upper left')
40 plt.grid()
41 plt.show()
42
43 # 손실 함수 곡선
44 plt.plot(hist.history['loss'])
45 plt.plot(hist.history['val_loss'])
46 plt.title('Model loss')
47 plt.ylabel('Loss')
48 plt.xlabel('Epoch')
49 plt.legend(['Train', 'Validation'], loc='upper right')
50 plt.grid()
51 plt.show()
```

## 5.4.2 학습 곡선 시각화

```
import os
```

```
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```



### NOTE matplotlib을 이용한 시각화

파이썬에서 matplotlib 라이브러리는 시각화에 가장 널리 쓰인다. 인공지능은 학습 과정이나 예측 결과를 시각화하는 데 matplotlib을 자주 사용한다. matplotlib 사용이 처음이라면 부록 B를 공부해 기초를 먼저 다진다. matplotlib의 공식 사이트에서 제공하는 튜토리얼 문서를 공부하는 것도 효과적인 방법이다. [표 2-1]에서 제시한 <https://matplotlib.org/users>에 접속해 [Tutorials] 메뉴를 선택한다. 튜토리얼은 Introductory, Intermediate, Advanced로 나뉘어 있으니 최소한 Introductory 코스를 숙지하고 넘어간다. [3.3.2절]

## 5.4.3 fashion MNIST 인식

- fashion MNIST 데이터셋

- MNIST와 비슷
- 내용이 패션 관련 그림이고 레이블이 {T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot}인 점만 다름



그림 5-7 fashion MNIST 데이터셋

## 5.4.3 fashion MNIST 인식

- fashion MNIST를 인식하는 [프로그램 5-8]
  - MNIST를 인식하는 [프로그램 5-7]에서 데이터 준비하는 곳만 달라짐(음영 표시한 부분만 달라짐)

프로그램 5-8

텐서플로 프로그래밍: 다층 퍼셉트론으로 fashion MNIST 인식

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import fashion_mnist
04
05 from tensorflow.keras.models import Sequential
06 from tensorflow.keras.layers import Dense
07 from tensorflow.keras.optimizers import Adam
08
09 # fashion MNIST 데이터셋을 읽어와 신경망에 입력할 형태로 변환
10 (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
11 ~ } # [프로그램 5-7]과 같음
51
```



## 5.4.3 fashion MNIST 인식

Train on 60000 samples, validate on 10000 samples

Epoch 1/30

60000/60000 - 5s - loss: 0.0693 - accuracy: 0.6463 - val\_loss: 0.0332 - val\_accuracy: 0.8166

Epoch 2/30

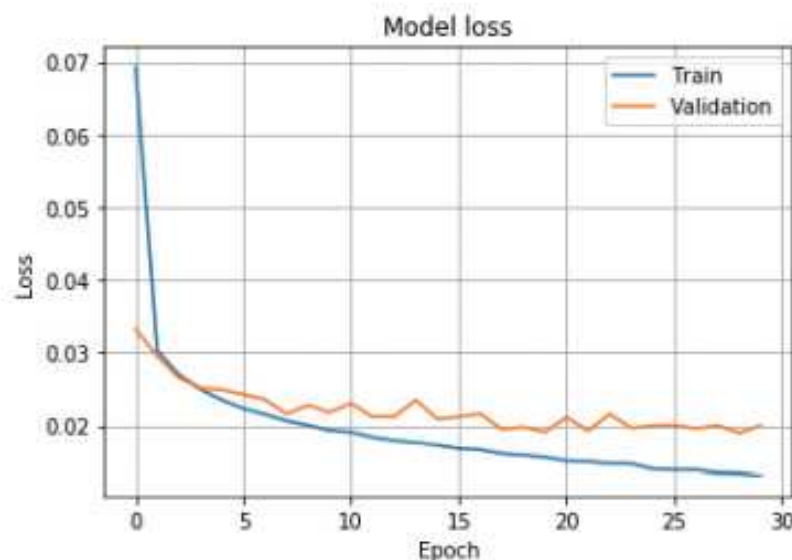
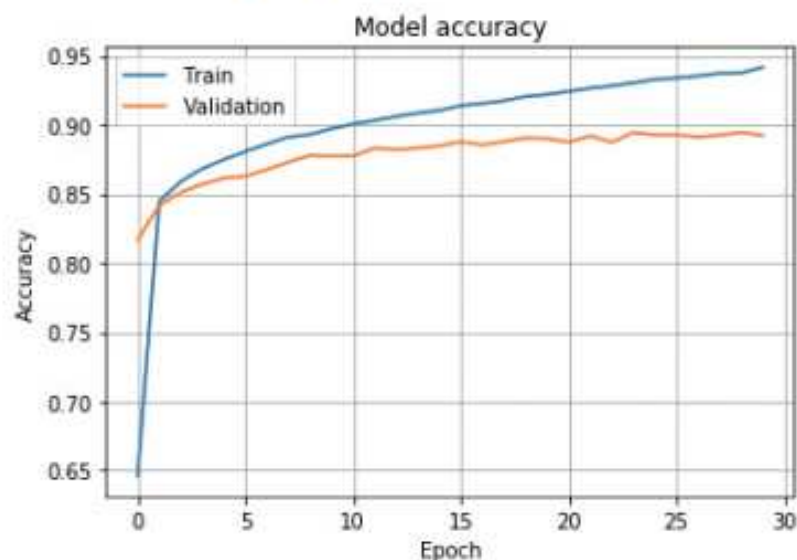
60000/60000 - 4s - loss: 0.0303 - accuracy: 0.8447 - val\_loss: 0.0295 - val\_accuracy: 0.8417

...

Epoch 30/30

60000/60000 - 4s - loss: 0.0131 - accuracy: 0.9416 - val\_loss: 0.0200 - val\_accuracy: 0.8925

정확률은 89.24999833106995



## 5.5 깊은 다층 퍼셉트론

- 다층 퍼셉트론에 은닉층을 더 많이 추가하면 깊은 다층 퍼셉트론
  - 깊은 다층 퍼셉트론은 가장 쉽게 생각할 수 있는 딥러닝 모델

## 5.5.1 구조와 동작

### • 깊은 다층 퍼셉트론 DMLP(deep MLP)의 구조

- $L-1$ 개의 은닉층이 있는  $L$ 층 신경망. 입력층에  $d+1$ 개의 노드, 출력층에  $c$ 개의 노드.  $i$ 번째 은닉층에  $n_i$ 개의 노드( $n_i$ 는 하이퍼매개변수)
- 인접한 층은 완전 연결, 즉 FC(fully-connected) 구조. 아주 많은 가중치: 예)  $n_i=500$ 이고  $L=5$ 라면, MNIST데이터에서  $(784+1)*500+(500+1)*500*3+(500+1)*10=1,149,010$ 개의 가중치

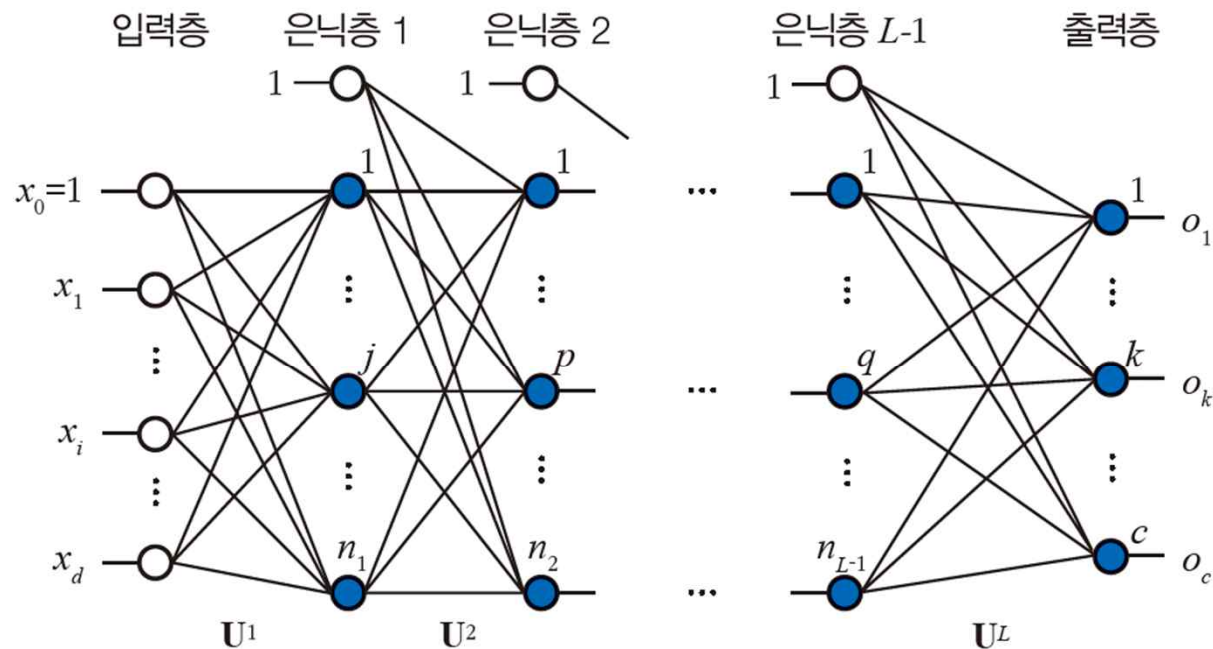


그림 5-8 깊은 다층 퍼셉트론의 구조



## 5.5.1 구조와 동작

- 깊은 다층 퍼셉트론의 동작

- 식 (5.1)은  $l-1$ 번째 층과  $l$ 번째 층을 연결하는 가중치 행렬

- $u_{ji}^l$  은  $l-1$ 번째 층의  $i$ 번째 노드와  $l$ 번째 층의  $j$ 번째 노드를 연결하는 가중치

$$\mathbf{U}^l = \begin{pmatrix} u_{10}^l & u_{11}^l & \cdots & u_{1n_{l-1}}^l \\ u_{20}^l & u_{21}^l & \cdots & u_{2n_{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ u_{n_l 0}^l & u_{n_l 1}^l & \cdots & u_{n_l n_{l-1}}^l \end{pmatrix}, \quad l=1,2,\cdots,L \quad (5.1)$$

- 입력층으로 들어오는 특징 벡터

$$\mathbf{z}^0 = (z_0^0, z_1^0, \cdots, z_{n_0}^0) = (1, x_1, x_2, \cdots, x_d) \quad (5.2)$$

## 5.5.1 구조와 동작

- 깊은 다층 퍼셉트론의 동작

- $l$ 번째 층의  $j$ 번째 노드가 수행하는 연산

$l$ 번째 은닉층의  $j$ 번째 노드의 연산:

$$z_j^l = \tau_l(s_j^l)$$

이때  $s_j^l = \mathbf{u}_j^l \mathbf{z}^{l-1}$ 이고  $\mathbf{u}_j^l = (u_{j0}^l, u_{j1}^l, \dots, u_{jn_{l-1}}^l)$ ,  $\mathbf{z}^{l-1} = (1, z_1^{l-1}, z_2^{l-1}, \dots, z_{n_{l-1}}^{l-1})^T$  (5.3)

- $l$ 번째 층의 연산을 행렬 표현으로 쓰면

$$l\text{번째 층의 연산: } \mathbf{z}^l = \tau_l(\mathbf{U}^l \mathbf{z}^{l-1}), \quad l=1, 2, \dots, L \quad (5.4)$$

- 훈련 집합 전체에 대한 연산

$$\mathbf{O} = \tau_L(\dots \tau_2(\mathbf{U}^2 \tau_1(\mathbf{U}^1 \mathbf{X}^T))) \quad (5.5)$$

- 1,2,3,...,L-1층의 활성화 함수는 주로 ReLU, L층(출력층)은 softmax 사용

## 5.5.2 오류 역전파 알고리즘

### • 다층 퍼셉트론의 학습 알고리즘을 조금 확장

- 식 (5.6)은 손실 함수

$$J(\mathbf{U}^1, \mathbf{U}^2, \dots, \mathbf{U}^L) = \frac{1}{|M|} \sum_{\mathbf{x} \in M} \|\mathbf{y} - \mathbf{o}\|^2$$

$$= \frac{1}{|M|} \sum_{\mathbf{x} \in M} \|\mathbf{y} - \tau_L(\dots \tau_2(\mathbf{U}^2 \tau_1(\mathbf{U}^1 \mathbf{x}^T))\|^2 \quad (5.6)$$

- 식 (5.7)은 가중치 갱신 규칙

$$\mathbf{U}^l = \mathbf{U}^l + \rho(-\nabla \mathbf{U}^l), \quad l = L, L-1, \dots, 1 \quad (5.7)$$

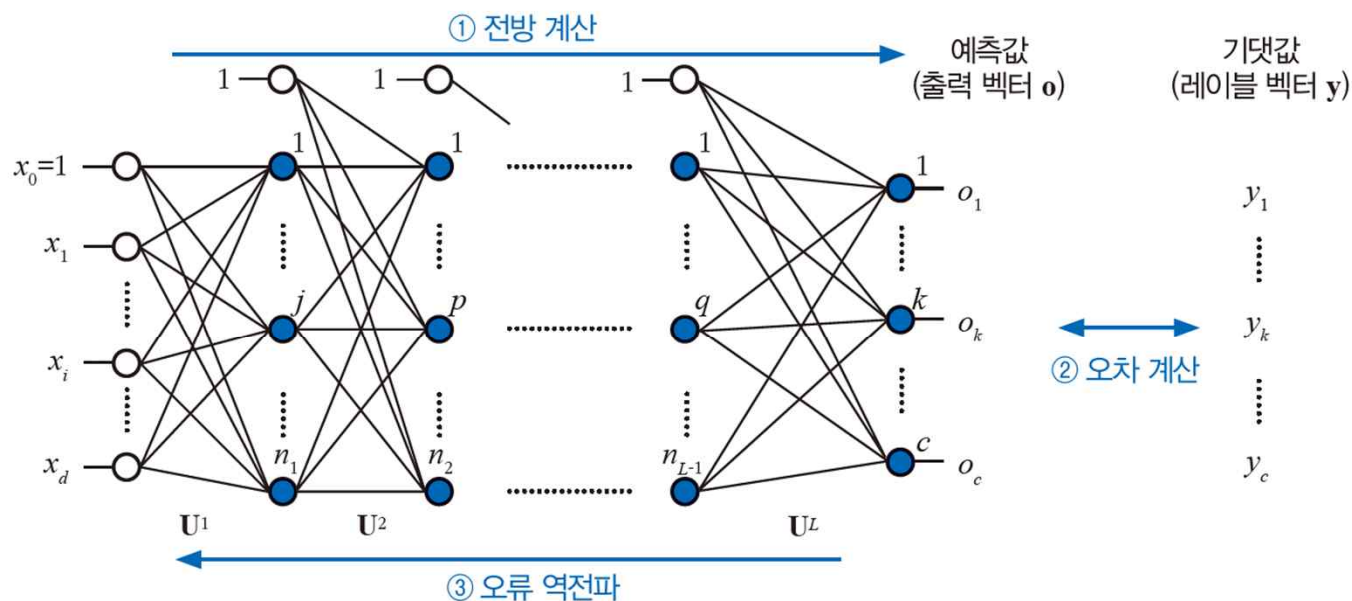


그림 5-9 깊은 다층 퍼셉트론이 사용하는 오류 역전파 알고리즘

## 5.5.3 깊은 다층 퍼셉트론 프로그래밍

- [프로그램 5-9]는 깊은 다층 퍼셉트론으로 MNIST 인식
  - 다층 퍼셉트론을 구현한 [프로그램 5-7]과 유사함
  - 단지 은닉층 1개가 4개로 확장된 차이(음영 부분만 달라짐)

프로그램 5-9

깊은 다층 퍼셉트론으로 MNIST 인식

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import mnist
04 from tensorflow.keras.models import Sequential
05 from tensorflow.keras.layers import Dense
06 from tensorflow.keras.optimizers import Adam
07
08 # MNIST 읽어 와서 신경망에 입력할 형태로 변환
09 (x_train, y_train), (x_test, y_test) = mnist.load_data()
10 x_train = x_train.reshape(60000,784)          # 텐서 모양 변환
11 x_test = x_test.reshape(10000,784)
12 x_train=x_train.astype(np.float32)/255.0      # ndarray로 변환
13 x_test=x_test.astype(np.float32)/255.0
14 y_train=tf.keras.utils.to_categorical(y_train,10) # 원핫 코드로 변환
15 y_test=tf.keras.utils.to_categorical(y_test,10)
16
```

## 5.5.3 깊은 다층 퍼셉트론 프로그래밍

```
17 # 신경망 구조 설정
18 n_input=784
19 n_hidden1=1024
20 n_hidden2=512
21 n_hidden3=512
22 n_hidden4=512
23 n_output=10
24
25 # 신경망 구조 설계
26 mlp=Sequential()
27 mlp.add(Dense(units=n_hidden1,activation='tanh',input_shape=(n_input,),kernel_
    initializer='random_uniform',bias_initializer='zeros'))
28 mlp.add(Dense(units=n_hidden2,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
29 mlp.add(Dense(units=n_hidden3,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
30 mlp.add(Dense(units=n_hidden4,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
31 mlp.add(Dense(units=n_output,activation='tanh',kernel_initializer='random_
    uniform',bias_initializer='zeros'))
32
33 # 신경망 학습
34 mlp.compile(loss='mean_squared_error',optimizer=Adam(learning_rate=0.001),met
    rics=['accuracy'])
35 hist=mlp.fit(x_train,y_train,batch_size=128,epochs=30,validation_data=(x_
    test,y_test),verbose=2)
36
```



## 5.5.3 깊은 다층 퍼셉트론 프로그래밍

```
37 # 신경망의 정확률 측정
38 res=mlp.evaluate(x_test,y_test,verbose=0)
39 print("정확률은",res[1]*100)
40
41 import matplotlib.pyplot as plt
42
43 # 정확률 곡선
44 plt.plot(hist.history['accuracy'])
45 plt.plot(hist.history['val_accuracy'])
46 plt.title('Model accuracy')
47 plt.ylabel('Accuracy')
48 plt.xlabel('Epoch')
49 plt.legend(['Train','Validation'], loc='upper left')
50 plt.grid()
51 plt.show()
52
53 # 손실 함수 곡선
54 plt.plot(hist.history['loss'])
55 plt.plot(hist.history['val_loss'])
56 plt.title('Model loss')
57 plt.ylabel('Loss')
58 plt.xlabel('Epoch')
59 plt.legend(['Train','Validation'], loc='upper right')
60 plt.grid()
61 plt.show()
```

## 5.5.3 깊은 다층 퍼셉트론 프로그래밍

Train on 60000 samples, validate on 10000 samples

Epoch 1/30

60000/60000 - 5s - loss: 0.0260 - accuracy: 0.8971 - val\_loss: 0.0132 - val\_accuracy: 0.9471

Epoch 2/30

60000/60000 - 5s - loss: 0.0101 - accuracy: 0.9543 - val\_loss: 0.0078 - val\_accuracy: 0.9614

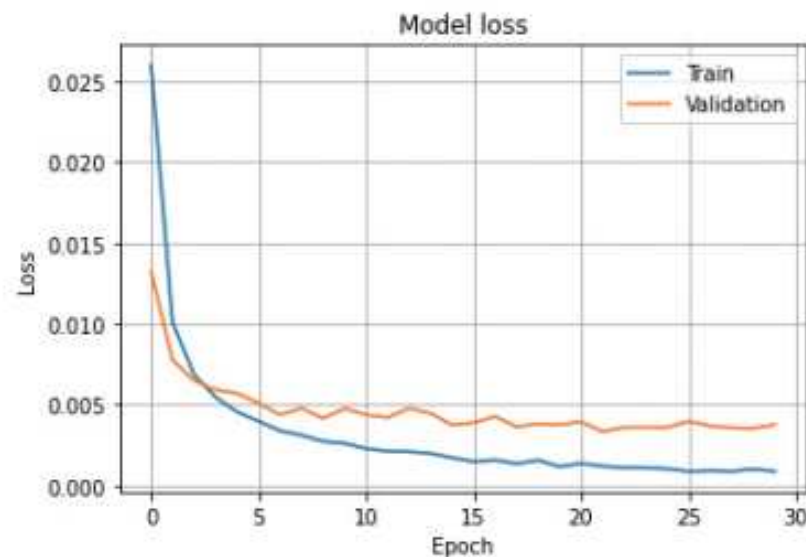
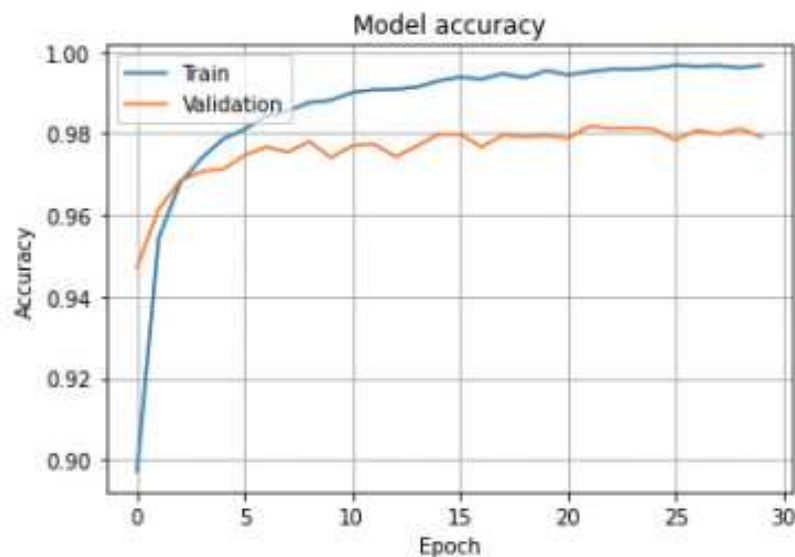
Epoch 29/30

60000/60000 - 5s - loss: 0.0010 - accuracy: 0.9961 - val\_loss: 0.0035 - val\_accuracy: 0.9812

Epoch 30/30

60000/60000 - 5s - loss: 8.8359e-04 - accuracy: 0.9967 - val\_loss: 0.0038 - val\_accuracy: 0.9791

정확률은 97.9099988937378 ← [프로그램 5-7] 다층 퍼셉트론의 97.65%에 비해 0.26% 향상





## 5.5.4 가중치 초기화 방법

- [프로그램 5-9]의 27~31행

- `kernel_initializer='random_uniform'`으로 설정했으므로 균일 분포에서 난수 생성하여 가중치를 초기화함

- Dense 함수의 API

- `kernel_initializer`의 기본값은 '`glorot_uniform`'

[Dense 함수의 API]

```
tensorflow.keras.layers.Dense(units, activation=None, use_bias=True,  
kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_  
regularizer=None, bias_regularizer=None, activity_regularizer=None,  
kernel_constraint=None, bias_constraint=None)
```

- `glorot_uniform`은 [Glorot2010]에서 유래하는데, 텐서플로는 좋은 성능이 입증되었다고 판단하여 기본값으로 제공함(보통 균일 분포보다 우수한 성능을 제공한다고 알려짐)

## 5.5.4 가중치 초기화 방법

- 이런 사실에 따라 앞으로는 생략하여 `glorot_uniform`을 사용
  - 성능 향상 효과
  - 파이썬 코드가 간결해지는 효과

```
27 mlp.add(Dense(units=n_hidden1,activation='tanh',input_shape=(n_input,)))
28 mlp.add(Dense(units=n_hidden2,activation='tanh'))
29 mlp.add(Dense(units=n_hidden3,activation='tanh'))
30 mlp.add(Dense(units=n_hidden4,activation='tanh'))
31 mlp.add(Dense(units=n_output,activation='tanh'))
```

## 5.6 딥러닝의 학습 전략

- 깊은 다층 퍼셉트론의 학습 알고리즘인 식 (5.7)은 수학적으로 아주 깔끔
  - 코딩도 깔끔
- 하지만 층이 깊어지면 현실적인 문제 발생
  - 이 장에서는 대표적인 두가지 문제 제시하고 해결 전략 설명
    - 그레이디언트 소멸 문제
    - 과잉 적합 문제

## 5.6.1 그레이디언트 소멸 문제와 해결책

- 미분의 연쇄 법칙 chain rule에 따르면,
  - l번째 층의 그레이디언트는 오른쪽에 있는 l+1번째 층의 그레이디언트에 자신 층에서 발생한 그레이디언트를 곱하여 구함
  - 따라서 그레이디언트가 0.001처럼 작은 경우 왼쪽으로 진행하면서 점점 작아짐
  - 왼쪽으로 갈수록 가중치 갱신이 느려져서 전체 신경망의 학습이 매우 느린 현상이 발생
- 병렬 처리로 해결
  - GPU 사용 또는 colab에서 tpu 설정

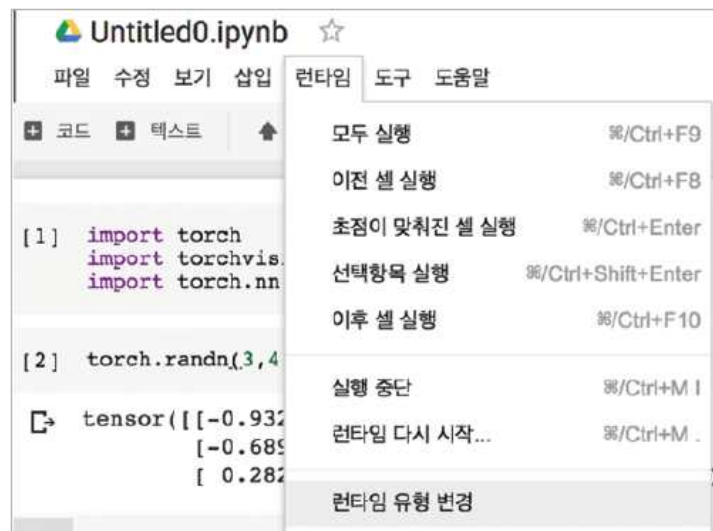
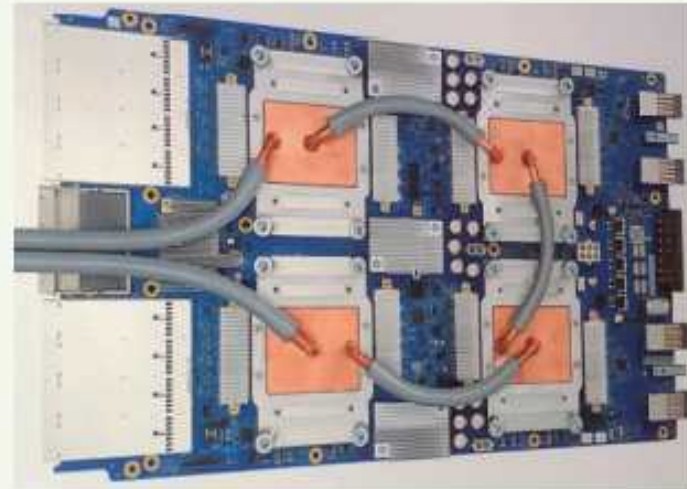


그림 5-10 코랩에서 GPU 또는 TPU 사용하기

## 5.6.1 그레이디언트 소멸 문제와 해결책

### NOTE TPU

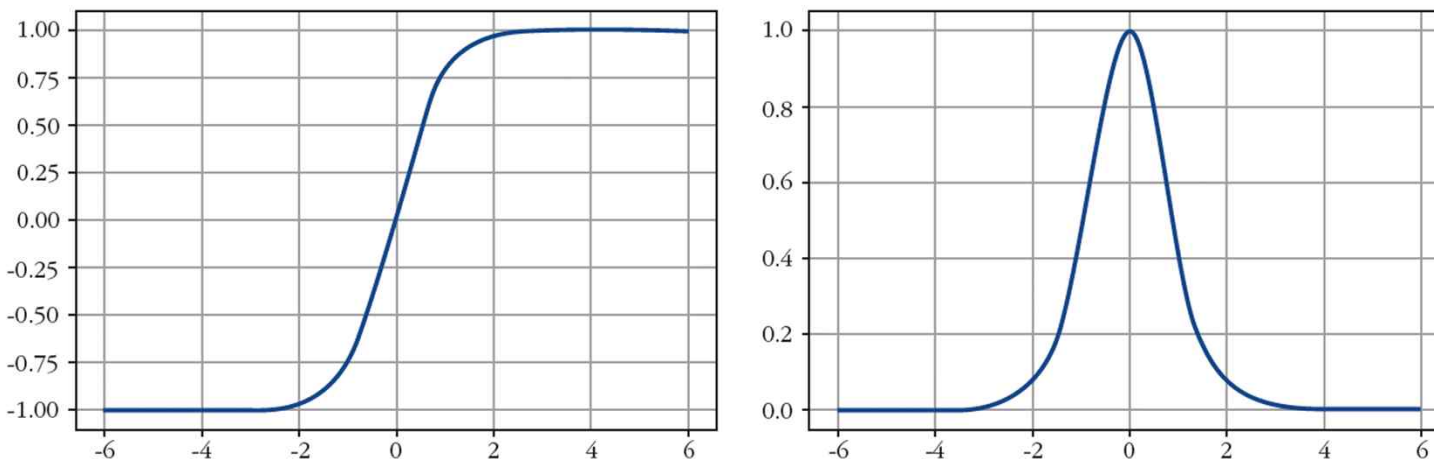
오른쪽 사진은 TPU(Tensor Processing Unit)이다. TPU는 구글이 신경망 학습을 빠르게 할 목적으로 개발한 기계 학습 전용 병렬 처리 기계이다. GPU는 원래 그래픽 가속기로 개발되었다는 점에서 차이가 있다. TPU는 그래픽 처리 측면에서는 GPU보다 열등하지만 기계 학습 측면에서는 GPU보다 뛰어나다. TPU는 텐서플로 소프트웨어에 맞추어 개발되었기 때문에 텐서플로로 개발하는 사람에겐 희소식이다. 구글에서는 알파고를 TPU로 학습했다고 밝혔다. 아직 정식 시판은 되지 않았으나, <https://coral.ai>에서 코랄 보드라는 초기 제품을 10~20만 원에 구입할 수 있다.



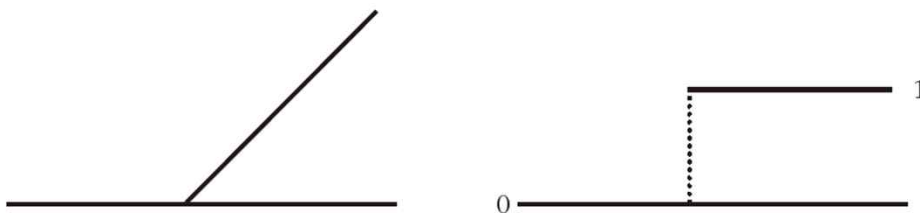
## 5.6.1 그레이디언트 소멸 문제와 해결책

### • ReLU 함수를 사용하여 해결

- $\tanh(s)$  시그모이드 함수의 문제점
  - $s$ 가 클 때 그레이디언트가 0에 가까워짐( $s=8$ 이면 그레이디언트 값은 0.0000004501)
- ReLU는  $s$ 가 음수일 때 그레이디언트는 0, 양수일 때 1



(a)  $\tanh$  시그모이드와 그레이디언트



(b) ReLU와 그레이디언트

그림 5-11  $\tanh$  시그모이드와 ReLU의 그레이디언트 특성 비교



## 5.6.2 과잉 적합과 과잉 적합 회피 전략

- [그림 5-12]는 과소 적합과 과잉 적합을 설명
  - $x$ 는 특징이고  $y$ 는 레이블인 회귀 문제로 설명
  - 모델로 1차 다항식을 사용하면 과소 적합 under fitting (데이터에 비해 모델 용량이 작은 상황)
  - 용량이 가장 큰 12차 다항식은 훈련 집합에 대해 가장 적은 오류

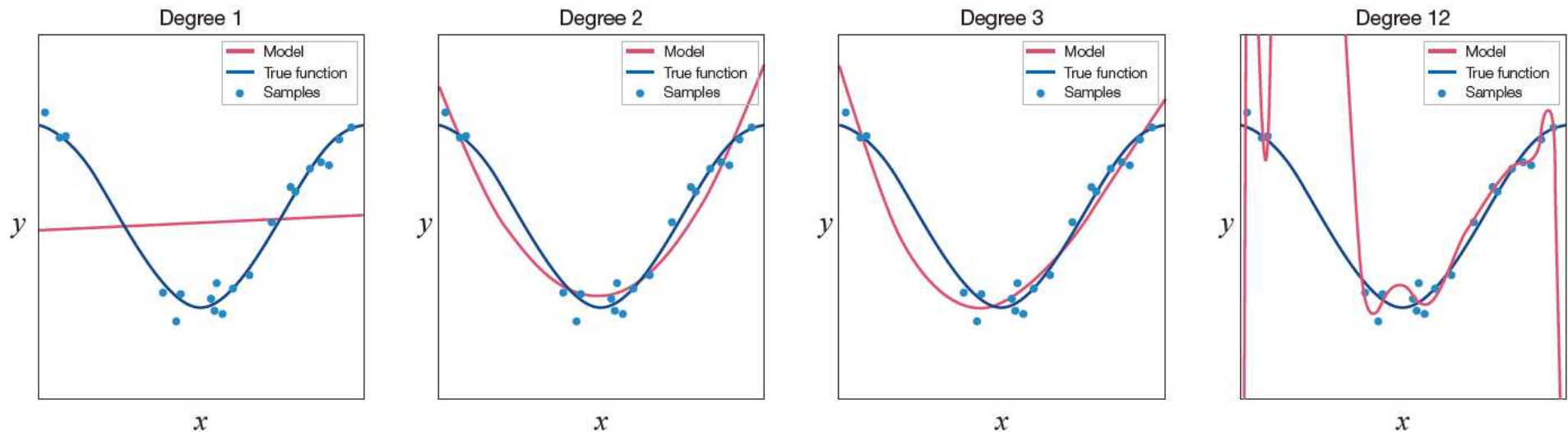


그림 5-12 과소 적합과 과잉 적합

## 5.6.2 과잉 적합과 과잉 적합 회피 전략

- 12차 다항식 모델은 일반화 능력이 떨어짐([그림 5-13])
  - 예를 들어,  $x_0$ 에서 부정확한 예측
  - 데이터의 복잡도에 비해 너무 큰 용량의 모델을 사용한 탓 ← 과잉 적합 <sub>over fitting</sub> 현상

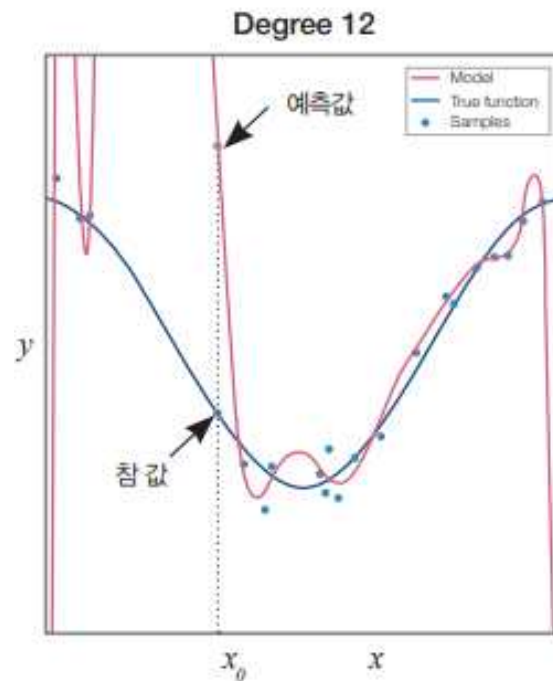


그림 5-13 과잉 적합에 따른 부정확한 예측

## 5.6.2 과잉 적합과 과잉 적합 회피 전략

- 딥러닝의 과잉 적합 회피 전략

- 데이터 양을 늘림. 데이터 양을 늘릴 수 없는 상황에서는 훈련 샘플을 변형하여 인위적으로 늘리는 데이터 증대 data augmentation 사용

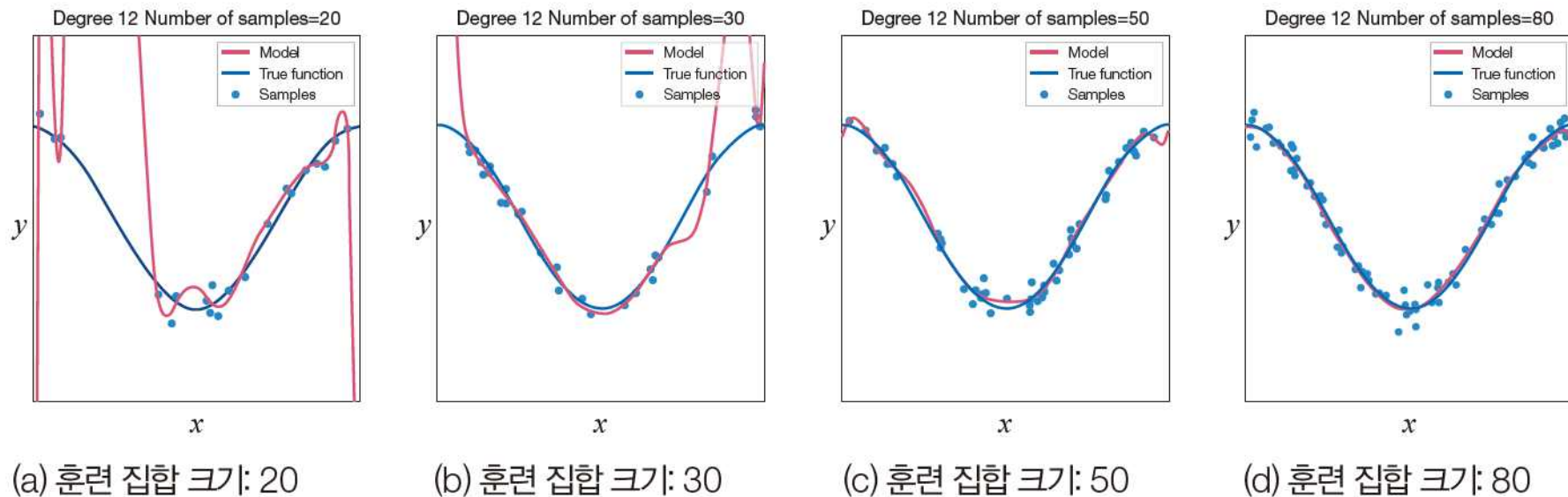


그림 5-14 데이터 양을 늘려 과잉 적합을 해소

**TIP** 규제 기법은 6장에서 설명한다.

- 규제 기법 적용

- 데이터 증대, 가중치 감소, 드롭아웃, 앙상블 등

## 5.6.2 과잉 적합과 과잉 적합 회피 전략

### NOTE 분류와 회귀 문제

분류(classification) 문제는 레이블이 이산인 경우이다. MNIST 숫자 인식, fashion MNIST 인식, iris 인식 등은 모두 분류 문제이다. 레이블이 연속인 경우를 분류와 구별해 회귀(regression) 문제라 부른다. 비트코인 가격을 예측하는 문제는 가격이 연속된 값을 가지므로 회귀이다.

예전 통계학자들은 회귀 문제를 푸는 일반화 선형 모델(generalized linear model)을 개발했는데, 이 모델은 나중에 분류 문제를 풀 수 있는 로지스틱 회귀(logistic regression) 모델로 개선되었다. 기계 학습 연구자는 주로 분류 문제를 푸는 알고리즘을 개발한다. 3장에서 공부한 SVM, 지금 공부하고 있는 신경망은 모두 분류 문제를 푸는 용도로 개발되었다. 회귀를 푸는 일반화 선형 모델을 분류 문제를 푸는 로지스틱 회귀 모델로 개조해 사용하듯이, 분류 문제를 푸는 SVM을 회귀 문제를 푸는 SVM으로 개조할 수 있다.

sklearn 라이브러리에서는 분류 문제를 푸는 SVM을 SVC, 회귀 문제를 푸는 SVM을 SVR이라는 함수로 구분해 제공한다.

## 5.7 딥러닝이 사용하는 손실 함수

- 시험 점수의 역할

- 점수가 낮은 학생에게 F학점 또는 낙방과 같은 벌점을 부여하면 자신을 성찰하고 더 열심히 공부할 동기 부여
- 점수가 낮거나 높거나 비슷한 벌점을 받으면 공정성이 깨지고 공부 의욕을 꺾음

- 신경망 학습도 비슷

- 신경망 가중치가 학생, 손실 함수가 시험 점수에 해당

## 5.7.1 평균제곱오차

- 샘플 하나의 오류

- 레이블  $y$ 와 신경망이 예측한 값  $\mathbf{o}$ 의 차이

$$e = \| \mathbf{y} - \mathbf{o} \|^2 \quad (5.8)$$

- 평균제곱오차 MSE(mean-squared error)

- 통계학에서 오랫동안 사용해온 식을 기계 학습이 빌려다 쓰는 셈

$$\begin{aligned} J(\mathbf{U}^1, \mathbf{U}^2, \dots, \mathbf{U}^L) &= \frac{1}{|M|} \sum_{\mathbf{x} \in M} \| \mathbf{y} - \mathbf{o} \|^2 \\ &= \frac{1}{|M|} \sum_{\mathbf{x} \in M} \| \mathbf{y} - \tau_L \left( \dots \tau_2 \left( \mathbf{U}^2 \tau_1 \left( \mathbf{U}^1 \mathbf{x}^T \right) \right) \right) \|^2 \end{aligned} \quad (5.6)$$

- 평균제곱오차의 문제점

- 교정에 사용하는 값, 즉 그레이디언트가 벌점에 해당. 오차  $e$ 가 더 큰데 그레이디언트가 더 작은 상황이 발생(공부를 못하는 학생이 더 높은 점수를 받는 상황에 비유)
- 학습이 느려지거나 학습이 안되는 상황을 초래할 가능성



## 5.7.2 교차 엔트로피

- 엔트로피<sub>entropy</sub>

- 확률 분포의 무작위성(불확실성)을 측정하는 함수 (식 (5.9))
  - 공정한 주사위의 엔트로피는 찌그러진 주사위보다 높음(예, 1의 면적이 더 넓은 찌그러진 주사위는 불확실성이 낮아짐)
  - 예를 들어, 공정한 주사위의  $-\left(\frac{1}{6}\log\frac{1}{6} + \dots + \frac{1}{6}\log\frac{1}{6}\right) = 1.7918$

$$H(x) = -\sum_{i=1,k} P(e_i) \log P(e_i) \quad (5.9)$$

- 교차 엔트로피<sub>cross entropy</sub>

- 두 확률 분포 P와 Q가 다른 정도를 측정하는 함수(식 (5.10))

$$H(P, Q) = -\sum_{i=1,k} P(e_i) \log Q(e_i) \quad (5.10)$$

- 예를 들어,  $-\left(\frac{1}{6}\log\frac{1}{6} + \dots + \frac{1}{6}\log\frac{1}{6}\right) = 1.7918$ 
  - 공정한 주사위 P와 Q의 교차 엔트로피
  - 공정한 주사위 P와 찌그러진 주사위 Q(1이 1/2, 나머지는 1/10 확률)의 교차 엔트로피  $-\left(\frac{1}{6}\log\frac{1}{2} + \frac{1}{6}\log\frac{1}{10} + \dots + \frac{1}{6}\log\frac{1}{10}\right) = 2.0343$

- 교차 엔트로피 손실 함수(식 (5.11))

- 교차 엔트로피는 평균제곱오차의 불공정성 문제를 해결해 줌
- 딥러닝은 주로 교차 엔트로피를 사용

교차 엔트로피 손실 함수:  $e = -\sum_{i=1,c} y_i \log o_i$  (5.11)

**[예제 5-1] 교차 엔트로피의 합리성 확인**

숫자 인식에서 부류 3에 속하는 샘플의 경우  $y=(0,0,0,1,0,\dots,0)$ 이다. 신경망의 출력이  $o=(0.1,0,0,0.9,0,\dots,0)$ 이라 하자. 부류 3에 해당하는 곳이 0.9로서 가장 크므로 신경망이 샘플을 맞힌 경우이다. 식 (5.11)을 계산하면  $e=-(0 \times \log(0.1)+0 \times \log(0)+0 \times \log(0)+1 \times \log(0.9)+\dots+0 \times \log(0))=0.1054$ 가 된다.

이제 신경망이  $o=(0,0.9,0,0.1,0,\dots,0)$ 을 출력했다고 가정하자. 부류 1에 해당하는 곳이 가장 큰 값을 갖기 때문에 신경망이 틀린 경우이다. 이 경우  $e=-(0 \times \log(0)+0 \times \log(0.9)+0 \times \log(0)+1 \times \log(0.1)+\dots+0 \times \log(0))=2.3026$ 이 된다. 후자의 틀린 경우에서 손실 함수 값이 훨씬 큰 사실을 확인할 수 있다.

## 5.7.3 손실 함수의 성능 비교 실험

- 텐서플로는 30여종의 손실 함수 제공
  - <http://keras.io/losses>
- 손실 함수 지정하는 세 가지 코딩 방식

```
model.compile(loss='categorical_crossentropy', optimizer=..., metrics=...)
```

```
import tensorflow as tf
model.compile(loss=tf.keras.losses.categorical_crossentropy, optimizer=..., metrics=...)
```

```
import tensorflow.keras.losses as ls
model.compile(loss=ls.categorical_crossentropy, optimizer=..., metrics=...)
```

### Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary\_crossentropy function
- categorical\_crossentropy function
- sparse\_categorical\_crossentropy function
- poisson function
- KLDivergence class
- kl\_divergence function

### Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean\_squared\_error function
- mean\_absolute\_error function
- mean\_absolute\_percentage\_error function
- mean\_squared\_logarithmic\_error function
- cosine\_similarity function
- Huber class
- huber function
- LogCosh class
- log\_cosh function

### Hinge losses for "maximum-margin" classification

- Hinge class
- SquaredHinge class
- CategoricalHinge class
- hinge function
- squared\_hinge function
- categorical\_hinge function

## 5.7.3 손실 함수의 성능 비교 실험

- 평균제곱오차와 교차 엔트로피를 비교하는 [프로그램 5-10]
  - 공정한 비교를 위해 하이퍼 매개변수는 동일하게 설정

프로그램 5-10

손실 함수의 성능 비교: 평균제곱오차와 교차 엔트로피

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import mnist
04 from tensorflow.keras.models import Sequential
05 from tensorflow.keras.layers import Dense
06 from tensorflow.keras.optimizers import Adam
07
08 # MNIST 읽어 와서 신경망에 입력할 형태로 변환
09 (x_train, y_train), (x_test, y_test) = mnist.load_data()
10 x_train = x_train.reshape(60000, 784)
11 x_test = x_test.reshape(10000, 784)
12 x_train = x_train.astype(np.float32) / 255.0
13 x_test = x_test.astype(np.float32) / 255.0
14 y_train = tf.keras.utils.to_categorical(y_train, 10)
15 y_test = tf.keras.utils.to_categorical(y_test, 10)
16
17 # 신경망 구조 설정
18 n_input = 784
19 n_hidden1 = 1024
20 n_hidden2 = 512
21 n_hidden3 = 512
22 n_hidden4 = 512
23 n_output = 10
24
```



## 5.7.3 손실 함수의 성능 비교 실험

```
25 # 평균제곱오차를 사용한 모델
26 dmlp_mse=Sequential()
27 dmlp_mse.add(Dense(units=n_hidden1,activation='tanh',input_shape=(n_input,)))
28 dmlp_mse.add(Dense(units=n_hidden2,activation='tanh'))
29 dmlp_mse.add(Dense(units=n_hidden3,activation='tanh'))
30 dmlp_mse.add(Dense(units=n_hidden4,activation='tanh'))
31 dmlp_mse.add(Dense(units=n_output,activation='softmax'))
32 dmlp_mse.compile(loss='mean_squared_error',optimizer=Adam(learning_rate=0.000
1),metrics=['accuracy'])
33 hist_mse=dmlp_mse.fit(x_train,y_train,batch_size=128,epochs=30,validation_
data=(x_test,y_test),verbose=2)

34
35 # 교차 엔트로피를 사용한 모델
36 dmlp_ce=Sequential()
37 dmlp_ce.add(Dense(units=n_hidden1,activation='tanh',input_shape=(n_input,)))
38 dmlp_ce.add(Dense(units=n_hidden2,activation='tanh'))
39 dmlp_ce.add(Dense(units=n_hidden3,activation='tanh'))
40 dmlp_ce.add(Dense(units=n_hidden4,activation='tanh'))
41 dmlp_ce.add(Dense(units=n_output,activation='softmax'))
42 dmlp_ce.compile(loss='categorical_crossentropy',optimizer=Adam(learning_rate
=0.0001),metrics=['accuracy'])
43 hist_ce=dmlp_ce.fit(x_train,y_train,batch_size=128,epochs=30,validation_
data=(x_test,y_test),verbose=2)

44
```

26~33행: 평균제곱오차를 사용하는  
모델 dmlp\_mse를 생성하고 학습

36~43행: 교차 엔트로피를 사용하는  
모델 dmlp\_ce를 생성하고 학습

## 5.7.3 손실 함수의 성능 비교 실험

```
45 # 두 모델의 정확률 비교
46 res_mse=dmlp_mse.evaluate(x_test,y_test,verbose=0)
47 print("평균제곱오차의 정확률은",res_mse[1]*100)
48 res_ce=dmlp_ce.evaluate(x_test,y_test,verbose=0)
49 print("교차 엔트로피의 정확률은",res_ce[1]*100)
50
51 # 하나의 그래프에서 두 모델을 비교
52 import matplotlib.pyplot as plt
53 plt.plot(hist_mse.history['accuracy'])
54 plt.plot(hist_mse.history['val_accuracy'])
55 plt.plot(hist_ce.history['accuracy'])
56 plt.plot(hist_ce.history['val_accuracy'])
57 plt.title('Model accuracy comparison between MSE and cross entropy')
58 plt.ylabel('Accuracy')
59 plt.xlabel('Epoch')
60 plt.legend(['Train_mse','Validation_mse','Train_ce','Validation_ce'], loc='best')
61 plt.grid()
62 plt.show()
```



## 5.7.3 손실 함수의 성능 비교 실험

...

Epoch 29/30

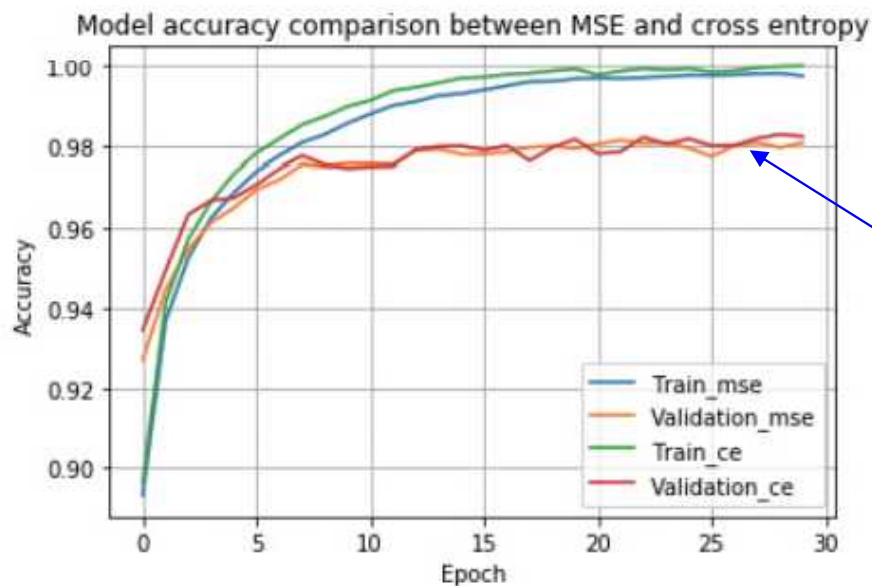
60000/60000 - 11s - loss: 0.0010 - accuracy: 0.9998 - val\_loss: 0.0680 - val\_accuracy: 0.9829

Epoch 30/30

60000/60000 - 10s - loss: 3.7642e-04 - accuracy: 1.0000 - val\_loss: 0.0682 - val\_accuracy: 0.9824

평균제곱오차의 정확률은 98.089998960495

교차 엔트로피의 정확률은 98.24000000953674



교차 엔트로피가 0.15% 우수

교차 엔트로피가 미세하게 우세한 경향

**TIP** 5.10절에서는 교차 검증을 사용해 성능 비교의 신뢰성을 높이는 방법을 공부한다.

## 5.8 딥러닝이 사용하는 옵티마이저

- 손실 함수의 최저점을 찾아주는 옵티마이저
  - 표준에 해당하는 SGD 옵티마이저([그림 4-5])를 개선하는 두 가지 아이디어
    - 모멘텀 momentum
    - 적응적 학습률 adaptive learning rate

## 5.8.1 모멘텀을 적용한 옵티마이저

- 물리에서 모멘텀

- 이전 운동량을 현재에 반영(관성과 관련)
- 옵티마이저에 적용하면 뚜렷한 성능 향상

- 모멘텀의 원리

- 모멘텀에서는 이전 방향 정보  $v$ 를 같이 고려( $\alpha$ 는  $[0,1]$ 사이에서 조절)
  - $\alpha=0$ 는 고전적 SGD,  $\alpha$ 가 1에 가까울수록 이전 정보에 큰 가중치 부여
  - 보통  $\alpha=0.5, 0.9$ 를 사용

$$\left. \begin{array}{l} \text{고전적 SGD: } \mathbf{w} = \mathbf{w} - \rho \frac{\partial J}{\partial \mathbf{w}} \rightarrow \text{모멘텀을 적용한 SGD: } \mathbf{v} = \alpha \mathbf{v} - \rho \frac{\partial J}{\partial \mathbf{w}} \\ \mathbf{w} = \mathbf{w} + \mathbf{v} \end{array} \right\} \quad (5.12)$$

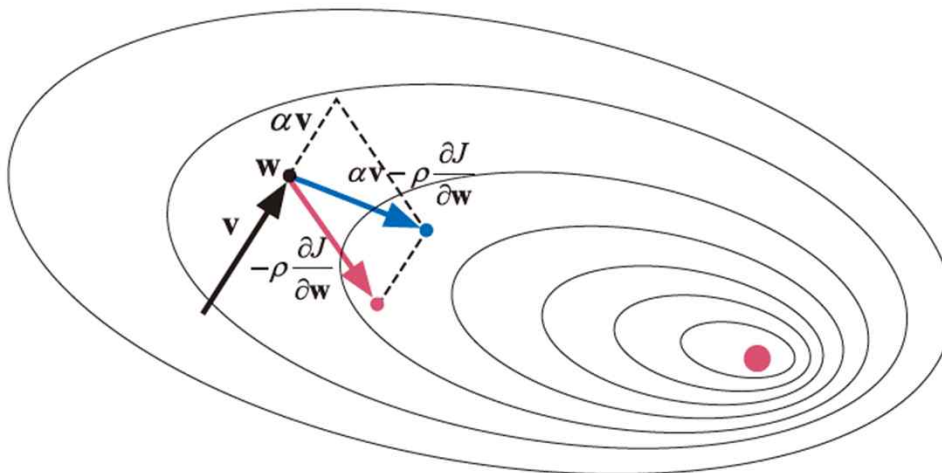


그림 5-15 모멘텀의 원리와 효과

## 5.8.1 모멘텀을 적용한 옵티마이저

- 네스테로프 모멘텀

- 현재 점  $w$ 에서 미분하는 대신, 이전 정보  $\alpha v$ 를 이용하여 다음에 이동할 곳  $\hat{w}$ 을 예측하고 그곳에서 그레디언트를 계산

- 모멘텀 효과를 시각화하는 사이트

**TIP** <https://distill.pub/2017/momentum>에 접속하면 모멘텀 계수  $\alpha$ 를 변경해봄으로써 최저점 탐색 과정과 결과가 어떻게 달라지는지 애니메이션으로 확인할 수 있다.

### Why Momentum Really Works



## 5.8.1 모멘텀을 적용한 옵티마이저

- 텐서플로에서 모멘텀 적용

- 기본값은 모멘텀 적용 않고 네스테로프 적용 안함

[SGD 옵티마이저의 API]

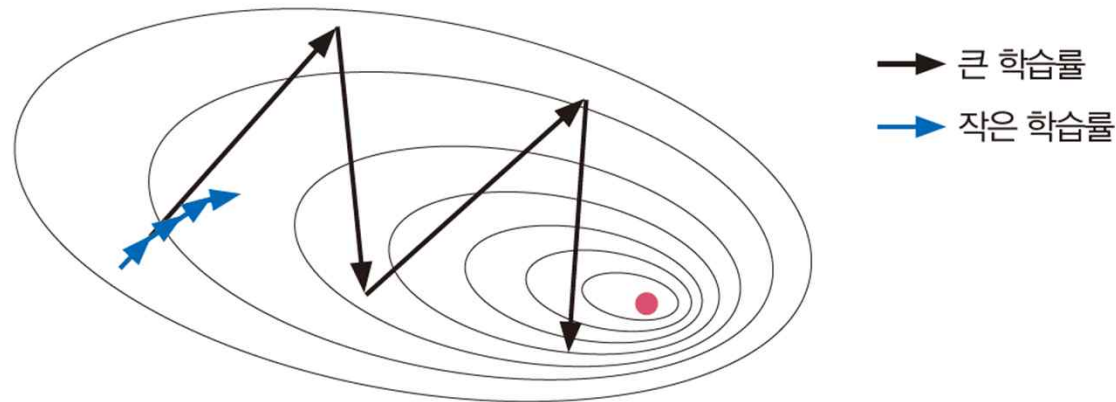
```
tensorflow.keras.optimizers.SGD(learning_rate=0.01,momentum=0.0,nesterov=False,name='SGD',**kwargs)
```

- 만일 학습률 0.0001, 모멘텀 0.9, 네스테로프 적용하려면
  - tensorflow.keras. optimizer.SGD(learning\_rate=0.0001, momentum=0.9, nesterov=True)로 호출

## 5.8.2 적응적 학습률을 적용한 옵티마이저

### • 식 (5.12)의 학습률

- 그레이디언트는 최저점의 방향은 알려주지만 이동량에 대한 정보는 없기 때문에 작은 학습률을 곱해 조금씩 보수적으로 이동
- 학습률이 너무 작으면 학습에 많은 시간 소요. 너무 크면 진동 가능성



### • 적응적 학습률

그림 5-16 학습률에 따른 수렴 특성

- 상황에 맞게 학습률을 조절하는 방법

- Adagrad: 이전 그레이디언트를 누적한 정보를 이용하여 학습률을 적응적으로 설정하는 기법
- RMSprop: 이전 그레이디언트를 누적할 때 오래된 것의 영향을 줄이는 정책을 사용하여 AdaGrad를 개선한 기법
- Adam: RMSProp에 식 (5.12)의 모멘텀을 적용하여 RMSprop을 개선한 기법



## 5.8.2 적응적 학습률을 적용한 옵티마이저

### • 옵티마이저의 API

[AdaGrad, RMSprop, Adam 옵티마이저의 API]

```
tensorflow.keras.optimizers.Adagrad(learning_rate=0.001, initial_
    accumulator_value=0.1, epsilon=1e-07, name='Adagrad', **kwargs)
tensorflow.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9, momentum=0.0,
    epsilon=1e-07, centered=False, name='RMSprop', **kwargs)
tensorflow.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
    epsilon=1e-07, amsgrad=False, name='Adam', **kwargs)
```

오래된 그레디언트의 영향을 줄임  
(값이 작을수록 영향력이 줄어듦)

모멘텀 관련

RMS의 rho

## 5.8.2 적응적 학습률을 적용한 옵티마이저

### • 옵티마이저의 수렴 특성

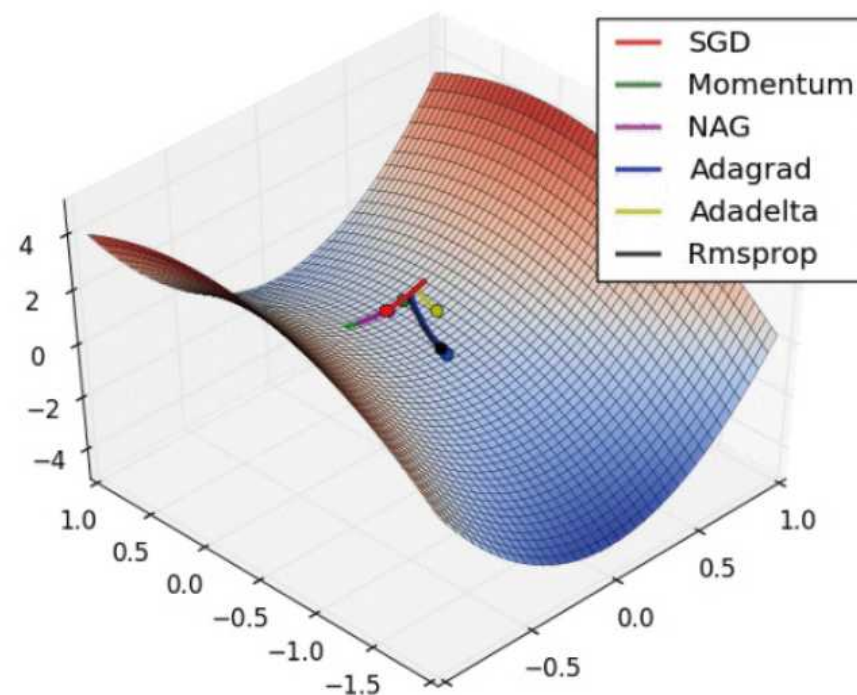
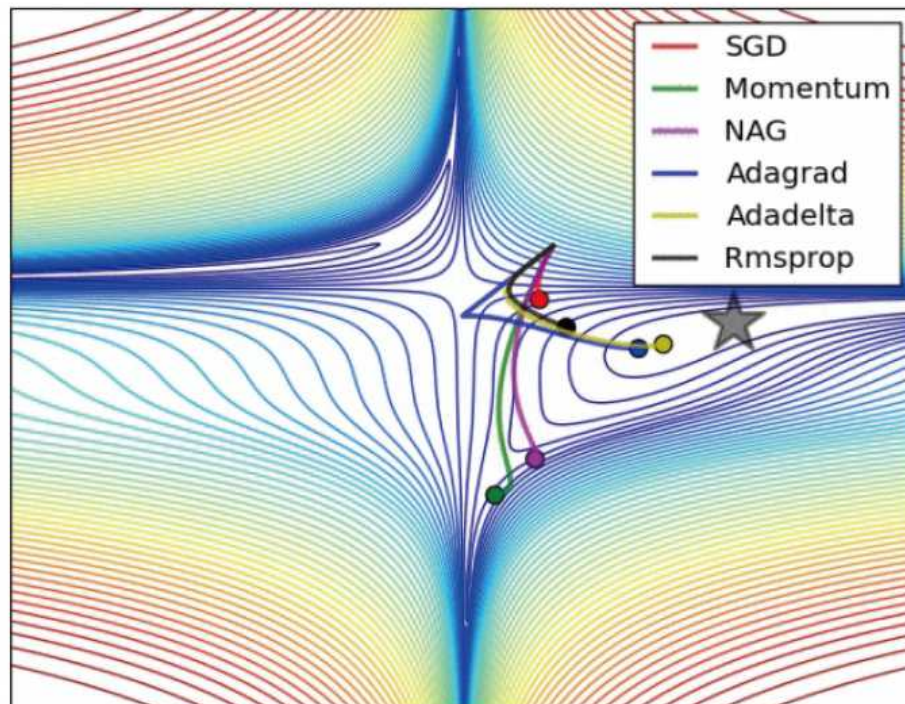


그림 5-17 옵티마이저에 따른 수렴 특성(출처: <https://cs231n.github.io/neural-networks-3>)

## 5.8.3 옵티마이저의 성능 비교 실험

- [프로그램 5-11]은 네 가지 옵티마이저의 성능을 비교
  - 손실 함수를 비교하는 [프로그램 5-10]을 약간 개조하면 됨
  - 같은 코드를 네 번 반복하므로 함수를 사용하여 프로그램 품질 높임
  - 공정한 비교를 위해 모든 옵티마이저는 기본값 사용, batch\_size와 epochs는 같은 값 사용

프로그램 5-11

옵티마이저의 성능 비교: SGD, Adam, Adagrad, RMSprop

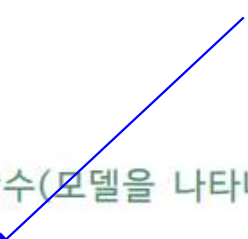
```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import fashion_mnist
04 from tensorflow.keras.models import Sequential
05 from tensorflow.keras.layers import Dense
06 from tensorflow.keras.optimizers import SGD, Adam, Adagrad, RMSprop
07
08 # fashion MNIST 읽어 와서 신경망에 입력할 형태로 변환
09 (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
10 x_train = x_train.reshape(60000, 784)
11 x_test = x_test.reshape(10000, 784)
12 x_train = x_train.astype(np.float32) / 255.0
13 x_test = x_test.astype(np.float32) / 255.0
14 y_train = tf.keras.utils.to_categorical(y_train, 10)
15 y_test = tf.keras.utils.to_categorical(y_test, 10)
16
```



## 5.8.3 옵티마이저의 성능 비교 실험

```
17 # 신경망 구조 설정
18 n_input=784
19 n_hidden1=1024
20 n_hidden2=512
21 n_hidden3=512
22 n_hidden4=512
23 n_output=10
24
25 # 하이퍼 매개변수 설정
26 batch_size=256
27 n_epoch=50
28
29 # 모델을 설계해주는 함수(모델을 나타내는 객체 model을 반환)
30 def build_model():
31     model=Sequential()
32     model.add(Dense(units=n_hidden1,activation='relu',input_shape=(n_input,)))
33     model.add(Dense(units=n_hidden2,activation='relu'))
34     model.add(Dense(units=n_hidden3,activation='relu'))
35     model.add(Dense(units=n_hidden4,activation='relu'))
36     model.add(Dense(units=n_output,activation='softmax'))
37     return model
38
```

같은 코드를 네 번 반복하므로 함수를 사용하여 프로그램 품질 높임



## 5.8.3 옵티마이저의 성능 비교 실험

```
39 # SGD 옵티마이저를 사용하는 모델
40 dmlp_sgd=build_model()
41 dmlp_sgd.compile(loss='categorical_crossentropy',optimizer=SGD(),metrics=['ac
    curacy'])
42 hist_sgd=dmlp_sgd.fit(x_train,y_train,batch_size=batch_siz,epochs=n_
    epoch,validation_data=(x_test,y_test),verbose=2)
43
44 # Adam 옵티마이저를 사용하는 모델
45 dmlp_adam=build_model()
46 dmlp_adam.compile(loss='categorical_crossentropy',optimizer=Adam(),metrics=['
    accuracy'])
47 hist_adam=dmlp_adam.fit(x_train,y_train,batch_size=batch_siz,epochs=n_
    epoch,validation_data=(x_test,y_test),verbose=2)
48
49 # Adagrad 옵티마이저를 사용하는 모델
50 dmlp_adagrad=build_model()
51 dmlp_adagrad.compile(loss='categorical_crossentropy',optimizer=Adagrad(),met
    rics=['accuracy'])
52 hist_adagrad=dmlp_adagrad.fit(x_train,y_train,batch_size=batch_siz,epochs=n_
    epoch,validation_data=(x_test,y_test),verbose=2)
53
54 # RMSprop 옵티마이저를 사용하는 모델
55 dmlp_rmsprop=build_model()
56 dmlp_rmsprop.compile(loss='categorical_crossentropy',optimizer=RMSprop(),met
    rics=['accuracy'])
57 hist_rmsprop=dmlp_rmsprop.fit(x_train,y_train,batch_size=batch_siz,epochs=n_
    epoch,validation_data=(x_test,y_test),verbose=2)
58
```

## 5.8.3 옵티마이저의 성능 비교 실험

```
59 # 네 모델의 정확률을 출력
60 print("SGD 정확률은",dmlp_sgd.evaluate(x_test,y_test,verbose=0)[1]*100)
61 print("Adam 정확률은",dmlp_adam.evaluate(x_test,y_test,verbose=0)[1]*100)
62 print("Adagrad 정확률은",dmlp_adagrad.evaluate(x_test,y_test,verbose=0)[1]*100)
63 print("RMSprop 정확률은",dmlp_rmsprop.evaluate(x_test,y_test,verbose=0)[1]*100)
64
65 import matplotlib.pyplot as plt
66
67 # 네 모델의 정확률을 하나의 그래프에서 비교
68 plt.plot(hist_sgd.history['accuracy'],'r')
69 plt.plot(hist_sgd.history['val_accuracy'],'r--')
70 plt.plot(hist_adam.history['accuracy'],'g')
71 plt.plot(hist_adam.history['val_accuracy'],'g--')
72 plt.plot(hist_adagrad.history['accuracy'],'b')
73 plt.plot(hist_adagrad.history['val_accuracy'],'b--')
74 plt.plot(hist_rmsprop.history['accuracy'],'m')
75 plt.plot(hist_rmsprop.history['val_accuracy'],'m--')
76 plt.title('Model accuracy comparison between optimizers')
77 plt.ylim((0.6,1.0))
78 plt.ylabel('Accuracy')
79 plt.xlabel('Epoch')
80 plt.legend(['Train_sgd','Val_sgd','Train_adam','Val_adam','Train_adagrad','Val_
    adagrad','Train_rmsprop','Val_rmsprop'], loc='best')
81 plt.grid()
82 plt.show()
```



## 5.8.3 옵티마이저의 성능 비교 실험

...

Epoch 49/50

60000/60000 - 11s - loss: 0.1559 - accuracy: 0.9458 - val\_loss: 1.0185 - val\_accuracy: 0.8867

Epoch 50/50

60000/60000 - 11s - loss: 0.1546 - accuracy: 0.9474 - val\_loss: 0.9143 - val\_accuracy: 0.8993

SGD 정확률은 87.08000183105469

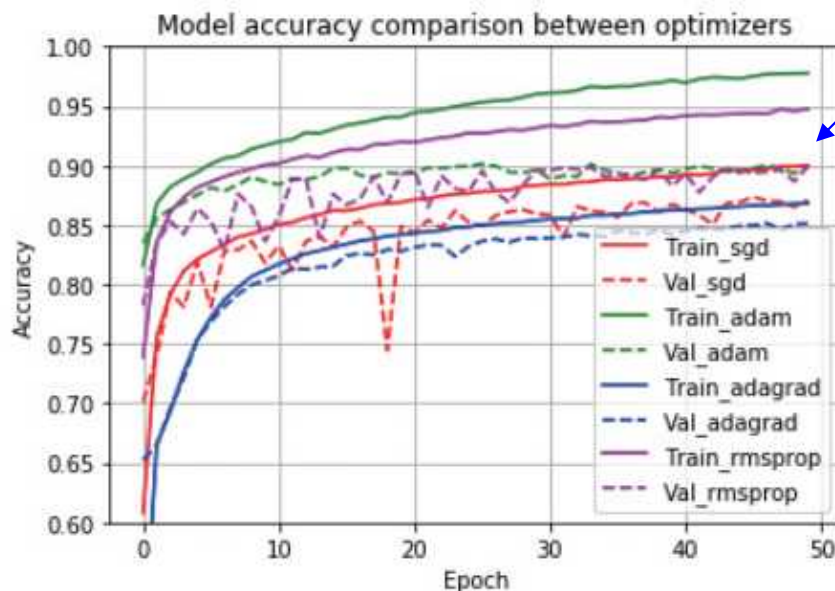
Adam 정확률은 89.85999822616577

Adagrad 정확률은 85.11000275611877

RMSprop 정확률은 89.92999792098999

Adam과 RMSprop이 선두  
RMSprop이 0.07% 우수

RMSprop은 훈련 집합과 검증 집합  
차이가 작아 일반화 능력이 우수



- 분석 결과는 fashion MNIST에 국한됨.
- 다시 실행하면 결과가 달라짐 (교차 검증을 사용하면 결과에 대한 신뢰도 높아짐)

## 5.9 좋은 프로그래밍 스킬

### • 프로그래밍 스킬의 중요성

- 프로그래밍을 못하면 아무런 인공지능 프로그램도 만들 수 없음
- 프로그래밍이 미숙하면 좋은 아이디어보다 디버깅에 에너지 소진
- 프로그래밍은 인공지능에서 충분조건은 아니지만 핵심 필요조건

### 1. 모듈화하라.

- [프로그램 5-11]의 build\_model 함수가 좋은 사례
- 26~27행의 batch\_size와 n\_epoch 상수(상수로 정의해 놓고 여러 군데에서 활용)

### 2. 언어의 좋은 특성을 최대한 활용하라.

- [프로그램 5-11]의 60행 사례(②의 두 행을 간결하게 ①의 한 행으로 코딩)

```
print("SGD 정확률은", dmlp_sgd.evaluate(x_test, y_test, verbose=0)[1]*100)
```

①

```
res_sgd=dmlp_mse.evaluate(x_test, y_test, verbose=0)  
print("평균제곱오차의 정확률은", res_sgd[1]*100)
```

②

## 5.9 좋은 프로그래밍 스킬

### 3. 점증적으로 코딩하라.

- 한번에 한가지 기능을 추가하고 옳게 작동하는지 확인하는 일을 반복
- [프로그램 5-11]의 그래프 그리는 68~82행

### 4. 디자인 패턴을 몸에 배게 하라.

- 다른 프로그램과 공유하는 디자인 패턴에 대한 눈썰미

### 5. 도구에 한없이 익숙해져라.

- 통합개발환경인 스파이더 사용법에 익숙
- 라이브러리 사용에 익숙

### 6. 기초에 충실하라.

- 파이썬의 기초 자료구조인 리스트, 튜플, 딕셔너리
- 중요한 라이브러리인 numpy
- 기계 학습의 기초 이론 등

## 5.10 교차 검증을 이용한 하이퍼 매개변수 최적화

- [프로그램 5-11]의 성능 측정에 대한 우려
  - 성능 그래프를 보면 전반적으로 Adam이 우세한데 마지막 세대에서 RMSprop이 운이 좋게 우승
  - 교차 검증은 우연을 배제하는데 효과적(3.6.3항)

## 5.10.1 교차 검증을 이용한 옵티마이저 선택

- [프로그램 5-12]는 교차 검증으로 성능 측정의 신뢰도 높임
  - 텐서플로는 교차 검증을 지원하는 함수가 없어 직접 작성해야 함(42~51행의 cross\_validation 함수)
  - k개로 분할하는 일은 sklearn의 KFold 함수 이용

프로그램 5-12

교차 검증을 이용한 옵티마이저의 성능 비교: SGD, Adam, Adagrad, RMSprop

```
01 import numpy as np
02 import tensorflow as tf
03 from tensorflow.keras.datasets import fashion_mnist
04 from tensorflow.keras.models import Sequential
05 from tensorflow.keras.layers import Dense
06 from tensorflow.keras.optimizers import SGD, Adam, Adagrad, RMSprop
07 from sklearn.model_selection import KFold
08
09 # fashion MNIST를 읽고 신경망에 입력할 형태로 변환
10 (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
11 x_train = x_train.reshape(60000, 784)
12 x_test = x_test.reshape(10000, 784)
13 x_train = x_train.astype(np.float32)/255.0
14 x_test = x_test.astype(np.float32)/255.0
15 y_train = tf.keras.utils.to_categorical(y_train, 10)
16 y_test = tf.keras.utils.to_categorical(y_test, 10)
17
```

## 5.10.1 교차 검증을 이용한 옵티마이저 선택

```
18 # 신경망 구조 설정
19 n_input=784
20 n_hidden1=1024
21 n_hidden2=512
22 n_hidden3=512
23 n_hidden4=512
24 n_output=10
25
26 # 하이퍼 매개변수 설정
27 batch_size=256
28 n_epoch=20
29 k=5 # 5-겹
30
31 # 모델을 설계해주는 함수(모델을 나타내는 객체 model을 반환)
32 def build_model():
33     model=Sequential()
34     model.add(Dense(units=n_hidden1,activation='relu',input_shape=(n_input,)))
35     model.add(Dense(units=n_hidden2,activation='relu'))
36     model.add(Dense(units=n_hidden3,activation='relu'))
37     model.add(Dense(units=n_hidden4,activation='relu'))
38     model.add(Dense(units=n_output,activation='softmax'))
39     return model
40
```



## 5.10.1 교차 검증을 이용한 옵티마이저 선택

```
41 # 교차 검증을 해주는 함수(서로 다른 옵티마이저(opt)에 대해)
42 def cross_validation(opt):
43     accuracy=[]
44     for train_index, val_index in KFold(k).split(x_train):
45         xtrain, xval = x_train[train_index], x_train[val_index]
46         ytrain, yval = y_train[train_index], y_train[val_index]
47         dmlp = build_model()
48         dmlp.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['
            accuracy'])
49         dmlp.fit(xtrain, ytrain, batch_size=batch_size, epochs=n_epoch, verbose=0)
50         accuracy.append(dmlp.evaluate(xval, yval, verbose=0)[1])
51     return accuracy
52
53 # 옵티마이저 4개에 대해 교차 검증을 실행
54 acc_sgd = cross_validation(SGD())
55 acc_adam = cross_validation(Adam())
56 acc_adagrad = cross_validation(Adagrad())
57 acc_rmsprop = cross_validation(RMSprop())
58
59 # 옵티마이저 4개의 정확률을 비교
60 print("SGD:", np.array(acc_sgd).mean())
61 print("Adam:", np.array(acc_adam).mean())
62 print("Adagrad:", np.array(acc_adagrad).mean())
63 print("RMSprop:", np.array(acc_rmsprop).mean())
```

Kfold 함수를 이용하여  
훈련과 검증 집합으로 분할

옵티마이저를 매개변수로 넘겨  
옵티마이저 각각을 교차 검증함

## 5.10.1 교차 검증을 이용한 옵티마이저 선택

```
64
65 import matplotlib.pyplot as plt
66
67 # 네 옵티마이저의 정확률을 박스플롯으로 비교
68 plt.boxplot([acc_sgd, acc_adam, acc_adagrad, acc_rmsprop], labels=["SGD", "Adam", "Ad
    agrad", "RMSprop"])
69 plt.grid()
```

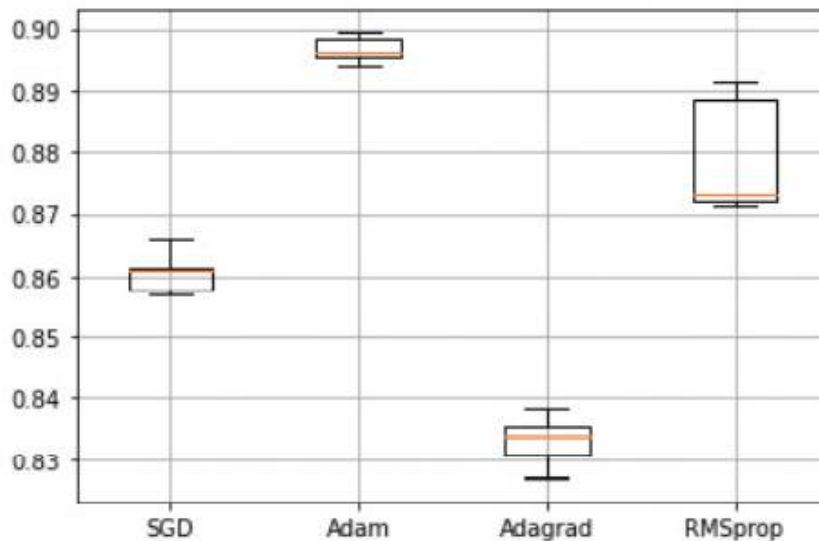
SGD: 0.86066663

Adam: 0.8967167

Adagrad: 0.83313334

RMSprop: 0.8793

Adam이 RMSprop보다 1.74% 우수  
(교차 검증으로 얻었기 때문에 높은 신뢰도)



박스 플롯은 아웃라이어, 최저, 최대,  
평균, 1/4, 3/4을 표시

## 5.10.2 과도한 계산 시간과 해결책

- 교차 검증은 많은 시간 소요

- [프로그램 5-12] 계산 시간 분석

- 44~50행의 for문은 k번 반복. 49행의 fit 함수는 가장 많은 시간 소요. fit가 소요하는 시간을 t라하면  $k \cdot t$ 만큼 지나야 옵티마이저 하나 처리
    - 옵티마이저가 4개이므로  $4kt$  시간 소요( $t=5$ 분,  $k=5$ 라면  $4 \cdot 5 \cdot 5 = 100$ 분 소요)
    - $k=10$ 으로 늘리고 n\_epoch을 20에서 100으로 늘리면 1000분(약 16.6시간) 소요

- 실제에서는

- 데이터 크기가 MNIST에 비해 수십~수백 배
  - 더 많은 하이퍼 매개변수를 동시에 최적화
  - 예를 들어, 옵티마이저 4개, 학습률 7개, 미니배치 크기 6개라면 총 168개의 조합

- 해결책

- GPU 사용
  - 욕심을 버림(경험을 통해 조합의 수를 축소)