05. 딥러닝과 텐서플로

Preview

- 딥러닝
  - 신경망에 층을 많이 두어(깊게 만들어) 성능을 높이는 기술
  - 현재 영상 인식, 음성 인식, 언어 번역 등의 최첨단 인공지능 제품을 딥러닝으로 제작함
  - 딥러닝을 다룬 장
    - 5장. 깊은 다층 퍼셉트론
    - 6장. 컨볼루션 신경망
    - 8장. 순환 신경망
    - 9장. 강화 학습
    - 10장. 생성 모델
    - 12장. 설명 가능 인공지능
- 딥러닝 프로그래밍에 사용되는 패키지
  - 텐서플로와 파이토치

그림 5-1 딥러닝 프로그래밍에 널리 쓰이는 텐서플로, 파이토치, 케라스의 로고

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
  - 저렴한 GPU 등장
    - 10~100배의 속도 향상으로 학습 시간 단축
  - 데이터셋 커짐
    - 인터넷을 통한 데이터 수집과 레이블링(예, 1400만장을 담은 ImageNet)
  - 학습 알고리즘의 발전
    - ReLU 활성 함수
    - 규제 기법(가중치 감쇠, 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 엶
    - 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 경이로운 성능으로 우승 차지
    - 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. … 10개의 기술 혁신이 한꺼번에 일어났다"라고 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 500~1,000장의 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	O	X
카페(Caffe)	UC버클리	2013년	C++	파이썬, 매트랩, C++	O	X
텐서플로(TensorFlow)	구글 브레인	2015년	C++, 파이썬, CUDA	파이썬, C++, 자바, 자바스크립트, R, Julia, Swift, Go	O	O
케라스(Keras)	프랑소와 숄레(François Chollet)	2015년	파이썬	파이썬, R	O	O
파이토치(PyTorch)	페이스북	2016년	C++, 파이썬, CUDA	파이썬, C++	O	O

TIP 딥러닝 라이브러리를 보다 폭넓게 살펴보려면 위키피디어에서 '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

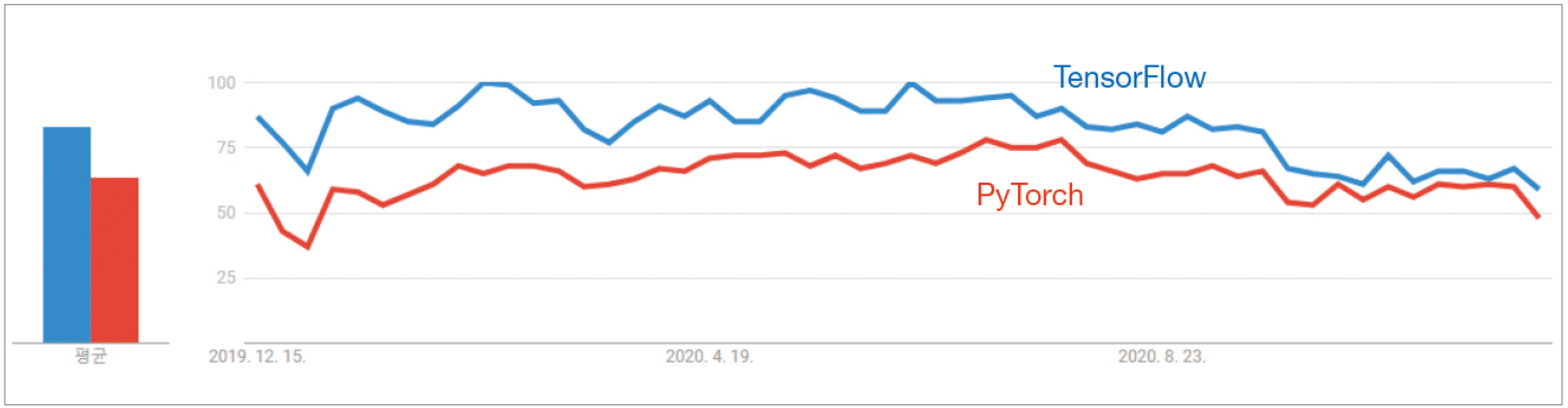


그림 5-3 구글 트렌드를 통해 비교한 텐서플로와 파이토치의 영향력

5.2.1 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)
```

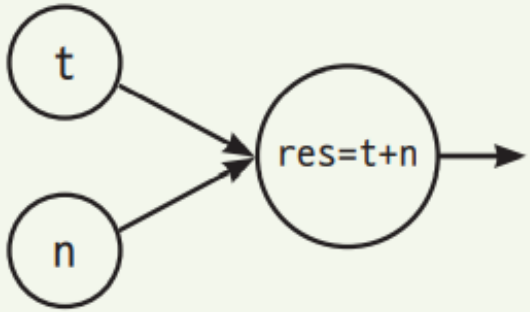
5.2.1 텐서플로와 넘파이의 호환

```
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

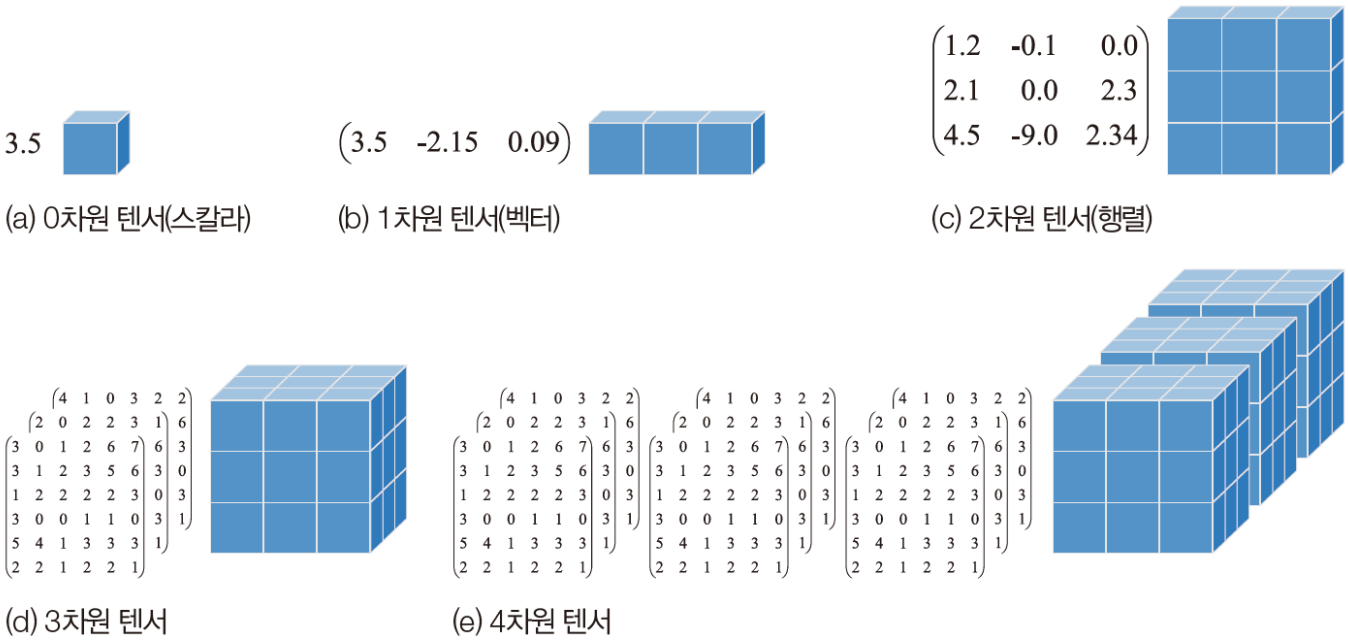
텐서플로 버전 1에서는 tensorflow 객체가 numpy와 호환되지 않는 등 tensorflow로 만든 객체에 제약이 많았다. 또한 [프로그램 5-2]에서 06행을 실행하면 데이터 내용이 출력되지 않았다. 그 이유는 텐서플로가 사용하는 계산 그래프(computation graph) 때문이다. 오른쪽 그림은 [프로그램 5-2]의 계산 절차를 표현한 계산 그래프다.



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

- 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)      # 원핫 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)
```

```
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,)
```

레이블 정보를 원핫 코드로 변환

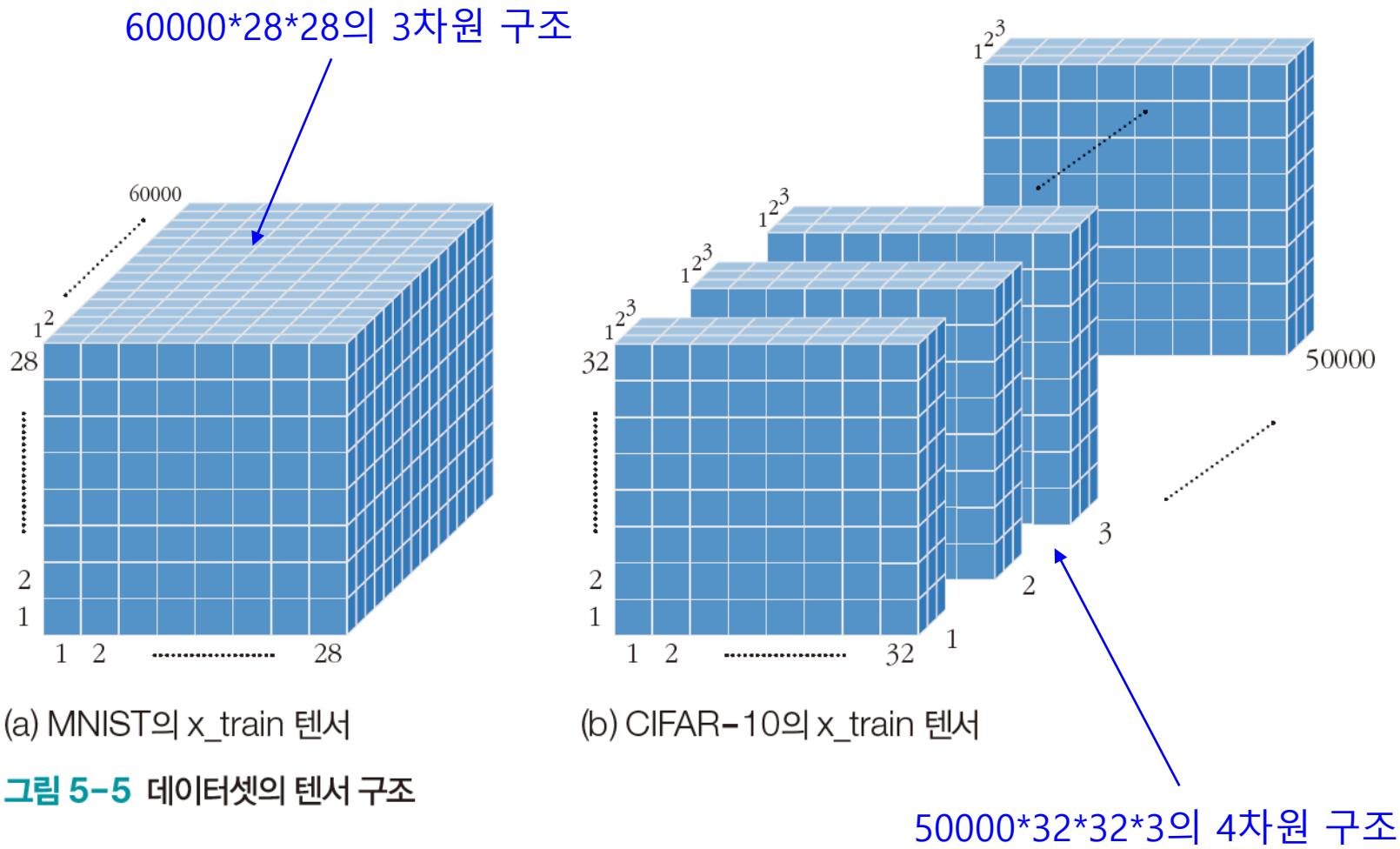
[5,0,4,...]처럼 표현

[[6],[9],[9],...]처럼 표현

13개 특징으로 표현되는 404개 샘플

5.2.2 텐서 이해하기

- 텐서플로가 제공하는 데이터셋의 텐서 구조



(a) MNIST의 x_train 텐서

(b) CIFAR-10의 x_train 텐서

그림 5-5 데이터셋의 텐서 구조

5.3.1 sklearn의 표현력 한계

- [프로그램 4-3]의 골격과 표현력의 한계

```
from sklearn.neural_network import MLPClassifier

digit=datasets.load_digits()                                    ❶
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)             ❷
mlp.fit(x_train,y_train)                                        ❸
```

  - 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
```

계단함수대신 사용

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, 활성 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

Sequential은 층을 한 줄로 쌓는데 사용

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
```

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',
    input_shape=(n_input,),kernel_initializer='random_uniform',
    bias_initializer='zeros'))
14
15  perceptron.compile(loss='mse',optimizer=SGD
    (learning_rate=0.1),metrics=['mse'])
16  perceptron.fit(x,y,epochs=500,verbose=2)
17
18  res=perceptron.predict(x)
19  print(res)
```

Sequential 클래스로 객체를 생성

add 함수로 Dense (완전연결) 층을 쌓음

신경망 구조 설계

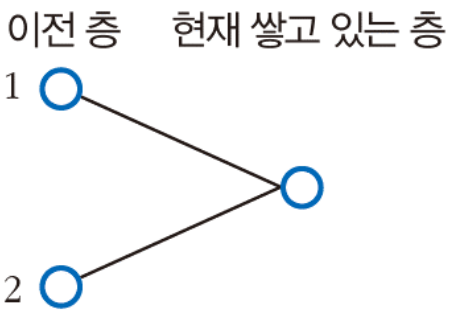
신경망 학습

학습된 신경망으로 예측

5.3.3 케라스 프로그래밍

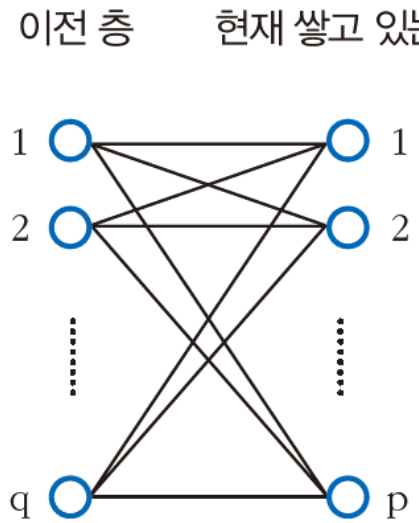
Dense로 완전연결층을 쌓는 방식

- [그림 5-6]은 units과 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
…                                  ← 16행의 fit 함수의 학습 과정
Epoch 199/200
4/4 - 0s - loss: 0.0148
Epoch 200/200
4/4 - 0s - loss: 0.0147

[[-0.8179741]                      ← 18행의 predict 함수의 예측 결과
 [ 0.886851 ]
 [ 0.8878835]
 [ 0.9992872]]
```

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

실습1
digits 데이터 셋으로 구현

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)                    #  텐서 모양 변환
12  x_test = x_test.reshape(10000,784)
13  x_train=x_train.astype(np.float32)/255.0                # ndarray로 변환
14  x_test=x_test.astype(np.float32)/255.0
15  y_train=tf.keras.utils.to_categorical(y_train,10)  # 원핫 코드로 변환
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

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,),
    kernel_initializer='random_uniform',bias_initializer='zeros'))
24  mlp.add(Dense(units=n_output,activation='tanh',kernel_
    initializer='random_uniform',bias_initializer='zeros'))
25
```

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

손실 함수로 MSE 사용

옵티마이저로 Adam 사용

```
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)
```

신경망 학습

학습된 신경망으로 예측

학습 도중에 발생한 정보를 hist 객체에 저장해 둠(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
```

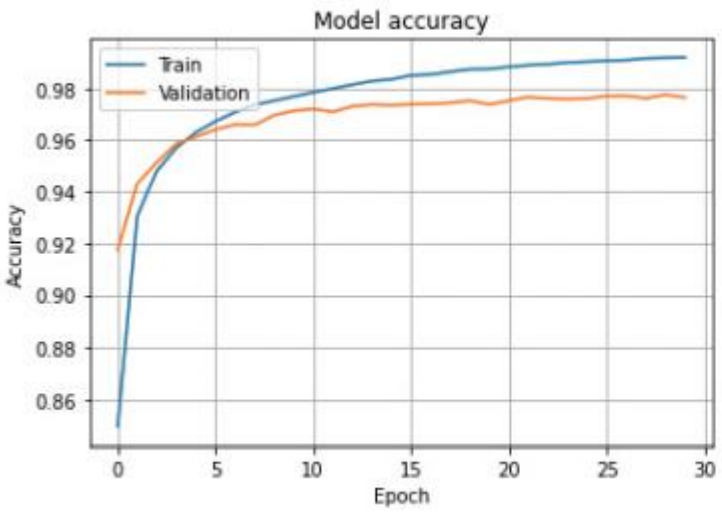
5.4.2 학습 곡선 시각화

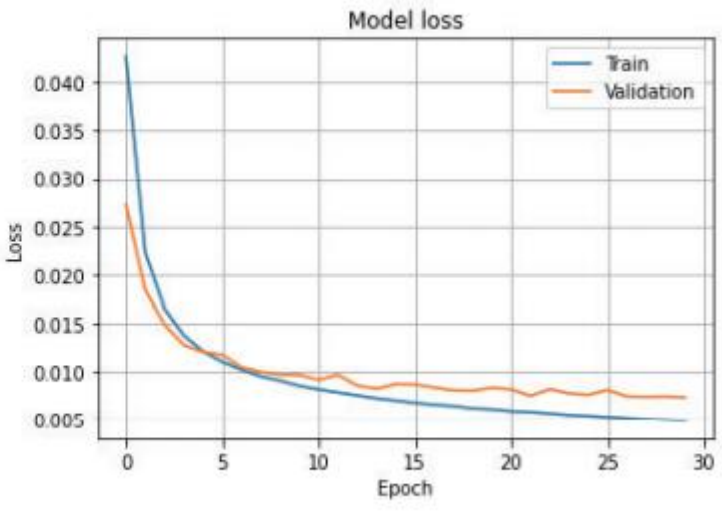
- 학습 곡선을 시각화 하는 [프로그램 5-7(b)]
  - hist 객체가 가진 정보를 이용하여 학습 곡선을 그림

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





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/users에 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

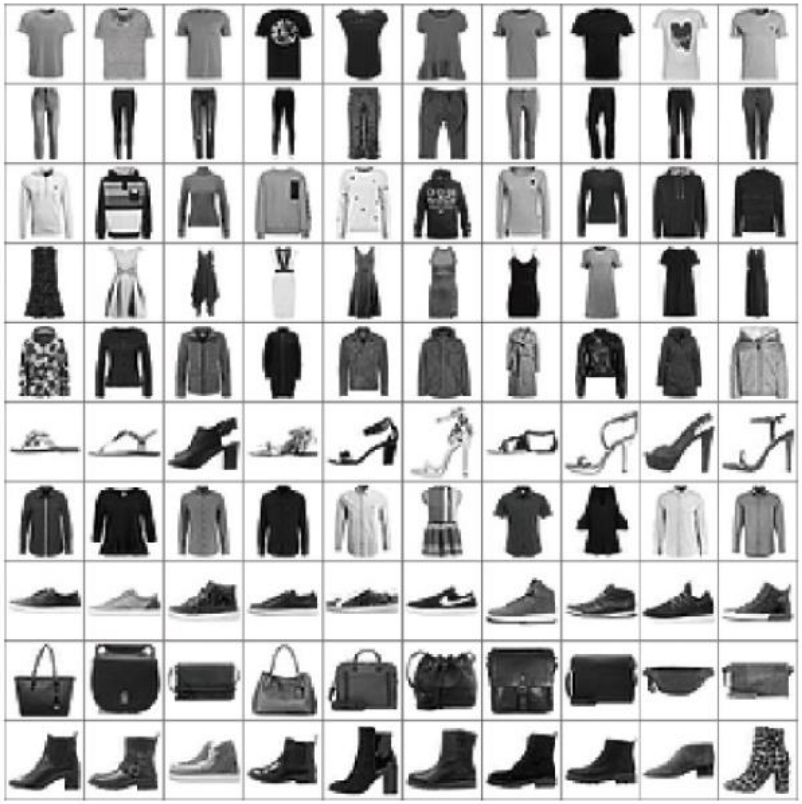


그림 5-7 fashion MNIST 데이터셋

5.4.3 fashion MNIST 인식

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

실습2
깊이를 5층 이상으로 구현

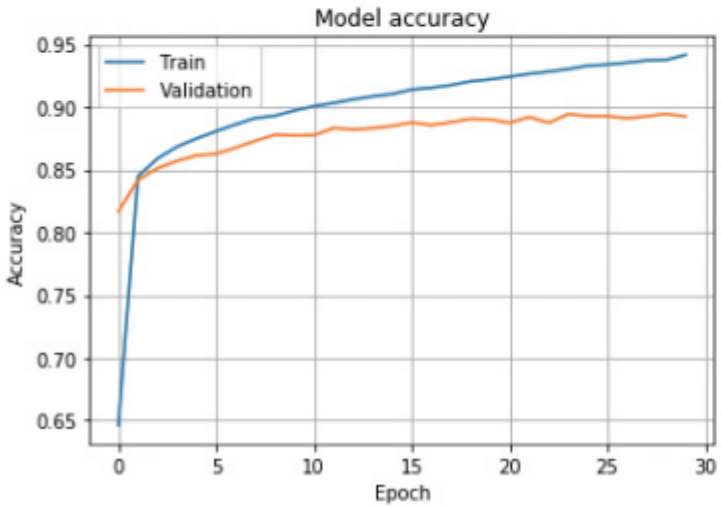
프로그램 5-8 텐서플로 프로그래밍: 다층 퍼셉트론으로 fashion MNIST 인식

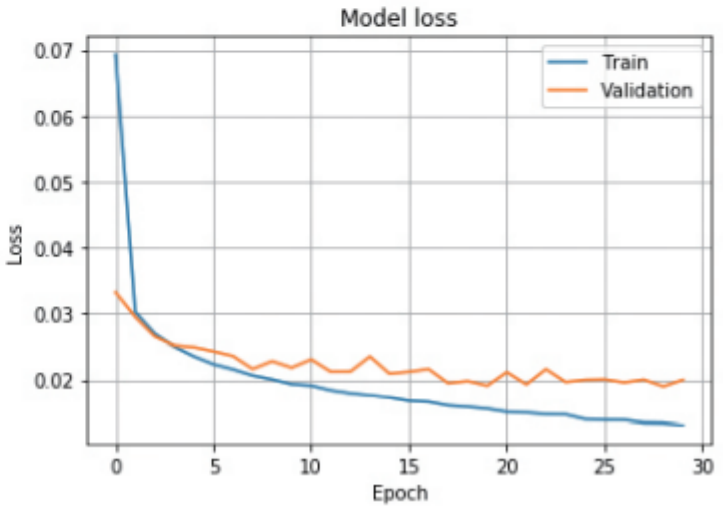
```
01  import numpy as np
02  import tensorflow as tf
03  from tensorflow.keras.datasets import fashion_mnist
04
05  from tensorflow.keras.models import Sequential
06  from tensorflow.keras.layers import Dense
07  from tensorflow.keras.optimizers import Adam
08
09  # fashion MNIST 데이터셋을 읽어와 신경망에 입력할 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$_{\text{DMLP(deep MLP)}}$의 구조
  - L-1개의 은닉층이 있는 L층 신경망. 입력층에 d+1개의 노드, 출력층에 c개의 노드. i번째 은닉층에 n_i개의 노드(n_i는 하이퍼 매개변수)
  - 인접한 층은 완전 연결, 즉 FC(fully-connected) 구조. 아주 많은 가중치: 예) n_i=500이고 L=5라면, MNIST데이터에서 (784+1)*500+(500+1)*500*3+(500+1)*10=1,149,010개의 가중치

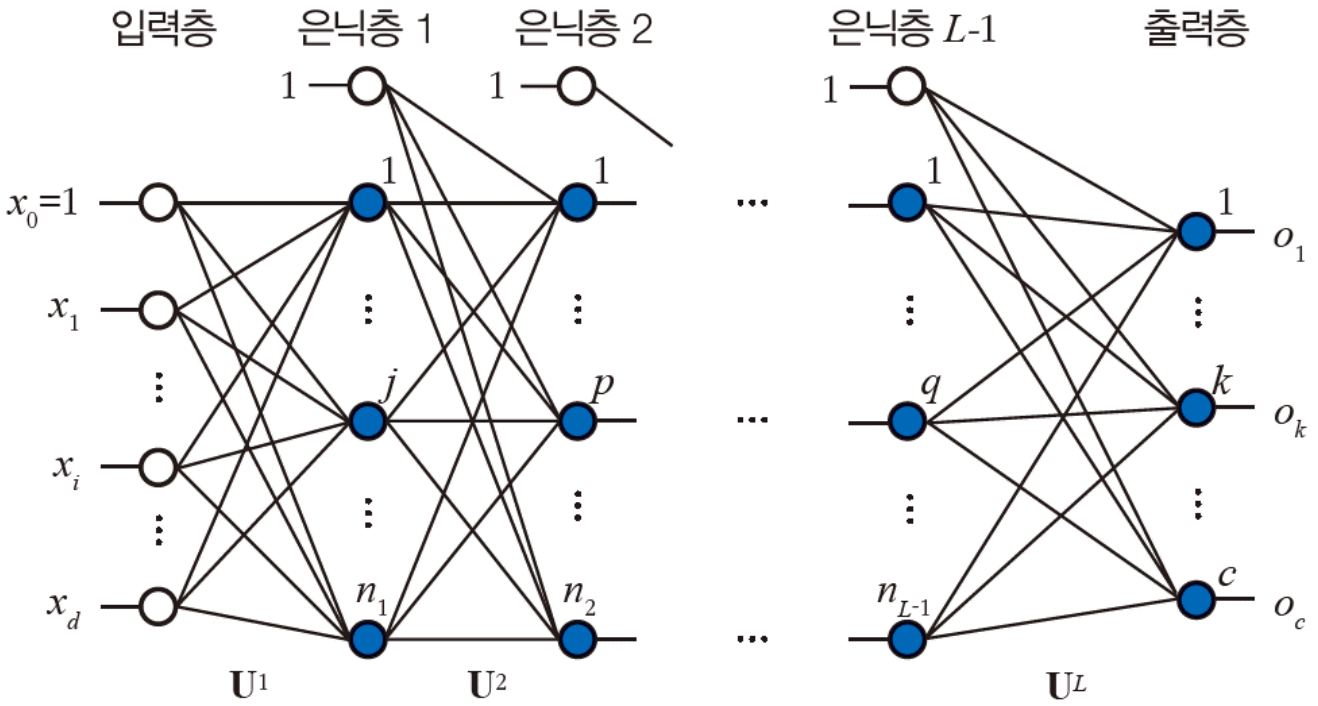


그림 5-8 깊은 다층 퍼셉트론의 구조

5.5.1 구조와 동작

깊은 다층 퍼셉트론의 동작

- 식 (5.1)은 l-1번째 층과 l번째 층을 연결하는 가중치 행렬
  - u_{ji}^{l}은 l-1번째 층의 i번째 노드와 l번째 층의 j번째 노드를 연결하는 가중치

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

- 입력층으로 들어오는 특징 벡터

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

5.5.1 구조와 동작

깊은 다층 퍼셉트론의 동작

- l번째 층의 j번째 노드가 수행하는 연산

l번째 은닉층의 j번째 노드의 연산:

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

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

- l번째 층의 연산을 행렬 표현으로 쓰면

l번째 층의 연산: $\mathbf{z}^l = \tau_l(\mathbf{U}^l \mathbf{z}^{l-1}),\ l = 1, 2, \cdots, L$ (5.4)

훈련 집합 전체에 대한 연산

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

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

5.5.2 오류 역전파 알고리즘

- 다층 퍼셉트론(4.8절)의 학습 알고리즘을 조금 확장
  - 식 (5.6)은 손실 함수

$$J\left(\mathbf{U}^1,\mathbf{U}^2,\cdots,\mathbf{U}^L\right)=\frac{1}{|M|}\sum_{\mathbf{x}\in M}\left\|\mathbf{y}-\mathbf{o}\right\|^2$$
$$=\frac{1}{|M|}\sum_{\mathbf{x}\in M}\left\|\mathbf{y}-\tau_L\left(\cdots\tau_2\left(\mathbf{U}^2\tau_1\left(\mathbf{U}^1\mathbf{x}^{\mathrm{T}}\right)\right)\right)\right\|^2 \qquad (5.6)$$

  - 식 (5.7)은 가중치 갱신 규칙

$$\mathbf{U}^l=\mathbf{U}^l+\rho\left(-\nabla\mathbf{U}^l\right),\ l=L,L-1,\cdots,1 \qquad (5.7)$$

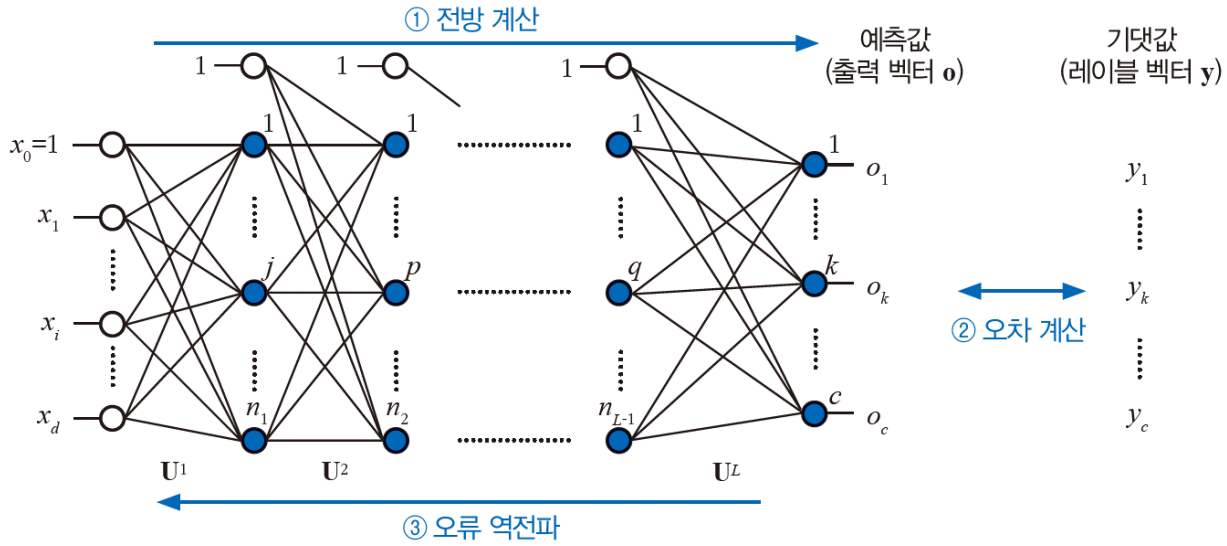


그림 5-9 깊은 다층 퍼셉트론이 사용하는 오류 역전파 알고리즘

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
```

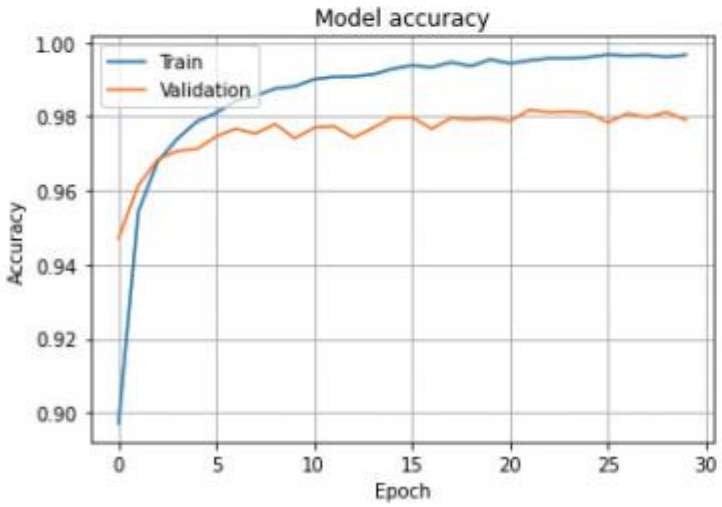
5.5.3 깊은 다층 퍼셉트론 프로그래밍

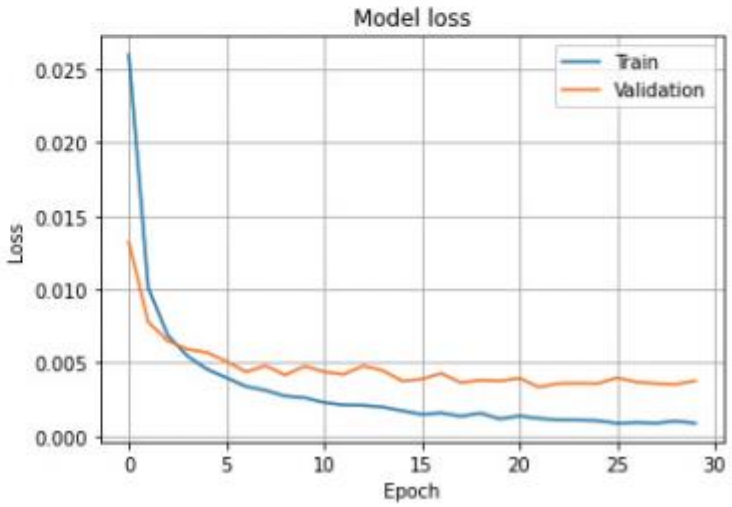
```
37  # 신경망의 정확률 측정
38  res=mlp.evaluate(x_test,y_test,verbose=0)
39  print("정확률은",res[1]*100)
40
41  import matplotlib.pyplot as plt
42
43  # 정확률 곡선
44  plt.plot(hist.history['accuracy'])
45  plt.plot(hist.history['val_accuracy'])
46  plt.title('Model accuracy')
47  plt.ylabel('Accuracy')
48  plt.xlabel('Epoch')
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_initializer의 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_uniform은 [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_uniform을 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'))
```

5.10.1 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_siz=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

Kfold 함수를 이용하여
훈련과 검증 집합으로 분할

옵티마이저를 매개변수로 넘겨
옵티마이저 각각을 교차 검증함

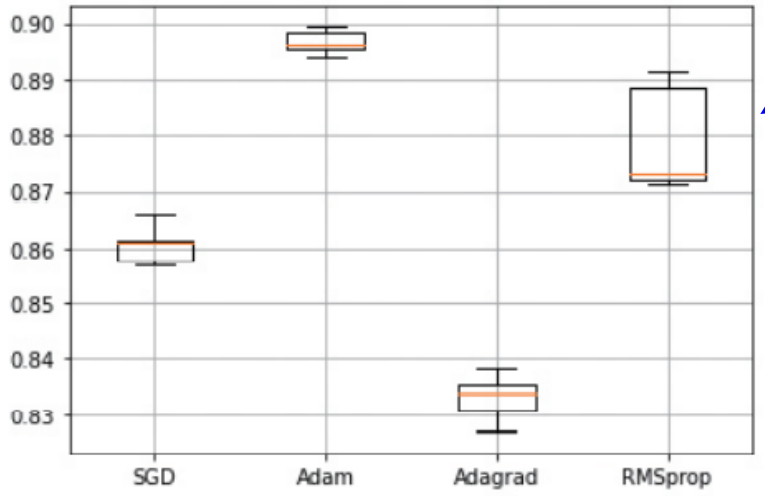
```
41  # 교차 검증을 해주는 함수(서로 다른 옵티마이저(opt)에 대해)
42  def cross_validation(opt):
43      accuracy=[]
44      for train_index,val_index in KFold(k).split(x_train):
45          xtrain,xval=x_train[train_index],x_train[val_index]
46          ytrain,yval=y_train[train_index],y_train[val_index]
47          dmlp=build_model()
48          dmlp.compile(loss='categorical_crossentropy',optimizer=opt,metrics=['
            accuracy'])
49          dmlp.fit(xtrain,ytrain,batch_size=batch_siz,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())
```

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 과다한 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*t만큼 지나야 옵티마이저 하나 처리
  - 옵티마이저가 4개이므로 4kt 시간 소요(t=5분,k=5라면 4*5*5=100분 소요)
  - k=10으로 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)