딥러닝개론

Introduction to Deep Learning

* 동일교과지정: 무인항공시스템을 위한 딥러닝 개론

딥러닝과 파이토치

본 강의자료는 한빛아카데미 〈파이썬으로 만드는 인공지능〉을 내용을 기반으로 제작 및 보완되었음을 알립니다.

PREVIEW

- 이번장 에서는 앞장에서 다룬 얕은 신경망을 깊게 만든 딥러닝 모델을 공부함
- 영상 인식기, 음성 인식기, 언어 번역기와 같은 최첨단 인공지능 제품은 대부분 딥러닝 기술로 만듦
- 딥러닝을 프로그래밍 하는데 사용하는 소프트웨어로는 구글의 텐서플로우(tensorflow), 페이스북의 파이토치(pytorch), 몬트리올 대학교에서 만든 씨아노(Theano), UC 버클리에서 만든 카페(caffe) 등이 있음
- 가장 널리 쓰이는 것은 텐서플로우와 파이토치이며, 파이썬 라이브러리 형태로 제공됨

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

5.1 딥러닝의 등장

- 깊은 신경망에 대한 아이디어는 사실 다층 퍼셉트론(MLP)이 등장해 혁신을 이루던 1980년대에 이미 나왔었음. 다층 퍼셉트론에 단지 은닉층을 여러 개 추가하면 깊은 신경망이 되기 때문에 누구나 쉽게 생각할 수 있었음. 하지만 은닉층을 많이 추가해 신경망을 깊게 만들면 제대로 학습되지 않는 문제가 있었음

- 깊은 신경망 학습에 번번이 실패하는 이유
  - 1) 오류 역전파 알고리즘은 출력층에서 시작해 입력층 방향으로 진행하면서 그레디언트 계산하고 가중치를 갱신하는데, 여러 층을 거치면서 그레디언트 값이 점점 작아져 입력층에 가까워지면 변화가 거의 없는 그레디언트 소멸 문제gradient vanishing 가 발생함
  - 2) 게다가 훈련 집합의 크기는 작은 상태로 머물러 있는데 추정할 매개변수는 크게 늘어 과잉 적합over-fitting에 빠질 위험이 더욱 커짐
  - 3) 학습이 어려운 또 다른 이유는 과다한 계산 시간임. 예전에는 병렬 처리를 하려면 값비싼 슈퍼컴퓨터를 사용하는 수밖에 없었음

5.1.1 딥러닝의 기술 혁신

- 딥러닝의 기술 혁신 요인
  - 사실 딥러닝에 새로 창안된 이론이나 원리는 별로 없음. 신경망의 기본 구조와 동작, 학습 알고리즘의 기본 원리는 거의 그대로임

  - **저렴한 GPU 등장**
    - 성능이 뛰어나고 가격도 저렴한 GPU가 등장하면서 대학 실험실에서도 손쉽게 병렬 처리를 할 수 있게 되었고, 학습 시간이 10~100배 단축되어 성능 실험을 다채롭게 할 수 있게 되었음
    - 예들 들어, 딥러닝 모델 하나를 학습하는데 이틀이 걸린다면 200개의 서로 다른 하이퍼 매개변수 조합 중 최적을 선택하는 작업은 1년 이상 걸림. 하지만 GPU를 사용하면 일주일 이내에 마칠 수 있음

5.1.1 딥러닝의 기술 혁신

- 딥러닝의 기술 혁신 요인
  - **인터넷이 영향으로 학습 데이터가 크게 증가**
    - 예를 들어, ImageNet은 인터넷에서 수집한 1400만 장 정도의 자연 영상을 카테고리별로 분류해 제공함
    - 게다가, 데이터에 가우시안 잡음을 섞거나 영상을 조금 이동하거나 회전해 데이터를 인우적으로 수십~수백 배 증가시키는 기법이 개발되었음

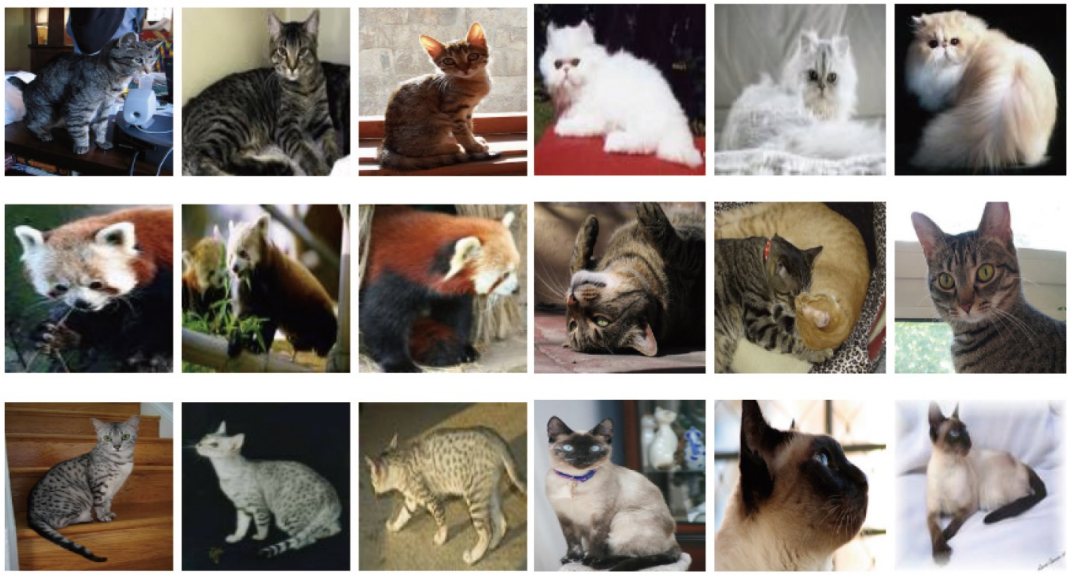


그림 5-2 자연 영상의 심한 변화 예(ImageNet의 고양이 부류 사진)

5.1.1 딥러닝의 기술 혁신

- 딥러닝의 기술 혁신 요인
  - **학습을 효과적으로 실행할 수 있는 다양한 알고리즘 개발**
    - 계산은 단순하고 성능은 더 좋은 활성 함수 ReLU를 발견함으로써 딥러닝의 성능이 향상되었고 그레디언트 소멸 문제가 크게 완화도었음
    - 학습에 효과적인 다양한 규제 기법regularization 이 개발되었음
      - 가중치를 작은 값으로 유지하는 가중치 축소weight decay 기법
      - 임의로 일정 비율의 노드를 선택해 불능으로 놓고 학습하는 드롭아웃dropout 기법
      - 조기 멈춤, 데이터 확대, 앙상블 등의 다양한 규제 기법이 개발되었음
      - 다양한 손실 함수와 옵티마이저 개발도 성능 향상에 크게 기여하였음

5.1.1 딥러닝의 기술 혁신

- 딥러닝의 기술 혁신 사례 (학술적인 측면)
  - **컨볼루션 신경망(CNN)이 딥러닝의 가능성을 열었음**
    - 컨볼루션 신경망은 특징 추출에 적합한 작은 크기의 컨볼루션 마스크를 사용하기 때문에 완전연결 구조인 다층 퍼셉트론 보다 매개변수가 훨씬 적지만 훨씬 우수한 특징을 추출함
    - 이런 이유로 컨볼루션 신경망에서 우수한 성능이 가장 먼저 입증되었음
    - 1990년대에 뉴욕 대학교의 리쿤 교수는 CNN으로 필기 숫자 인식에서 획기적인 성능 향상을 얻었고, 이 기술혁신으로 필기 숫자 인식이 수표 인식에서 실용화 수준에 도달하게 됨(1998)
    - 이후 CNN에서 개발된 여러 기법은 딥러닝 전반에 큰 영향을 미침

5.1.1 딥러닝의 기술 혁신

- 딥러닝의 기술 혁신 사례 (학술적인 측면)
  - **컨볼루션 신경망(CNN)이 딥러닝의 가능성을 열었음**
    - 르쿤이 필기 숫자에서 CNN의 가능성을 보였음에도 불구하고 자연 영상에 대한 시도는 소극적이 었음
    - 2012년 ILSVRC 에서 CNN을 사용한 AlexNet이 오류율 15.3%라는 경이로운 성능으로 우승함. 전년도에 고전적인 기계 학습을 사용한 팀이 오류율 25.8%로 우승한 사실을 감안하면 CNN의 가능성을 보여주기에 충분함
    - 이때부터 컴퓨터 비전 연구는 고전적인 기법에서 딥러닝으로 패러다임 전환이 이루어짐

  - **딥러닝은 음성인식 분야의 혁신을 가져옴**
    - 딥러닝 이전에는 음성 인식을 주로 통계적 기법과 HMM[Hidden Markov Model]을 이용해 구현하였음
    - 2009년경에 토론토 대학교의 힌튼 교수 연구팀은 음성 인식에 딥러닝을 적용하기 시작함
    - 힌튼 교수는 안드로이드 운영체제에 들어가는 음성 인식 소프트웨어의 단어 인식 오류율을 단숨에 25%나 줄였음

5.1.2 딥러닝 소프트웨어

- 여러 대학과 기업은 자체적으로 딥러닝 소프트웨어를 개발해 내부 연구 그룹끼리 공유하기 시작함
- 공유를 통해 딥러닝 소프트웨어는 점점 개선되어 큰 규모로 발전했고, 개발자들은 누구나 사용할 수 있도록 라이브러리 형태로 제작해 경쟁적으로 공개함
- 딥러닝 소프트웨어 자체는 대부분 C와 C++ 로 개발하지만, 소프트웨어를 불러 쓰는 인터페이스 언어로는 파이썬과 가은 스크립트 언어를 채택함

5.1.2 딥러닝 소프트웨어

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

5.1.2 딥러닝 소프트웨어

- 구글 트랜드를 통해 텐서플로우와 파이토치의 최근 영향력을 비교한 그래프를 살펴보면, 2020년 12월 기준으로 둘의 영향력은 막상막하로 보임

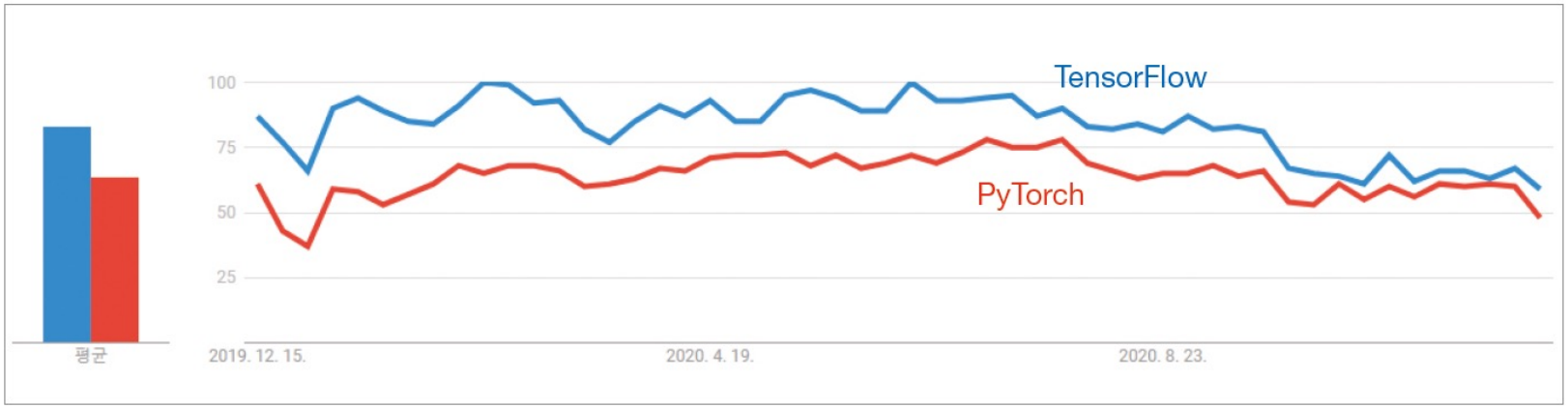
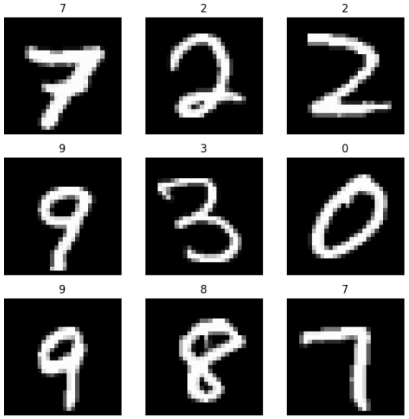


그림 5-3 구글 트렌드를 통해 비교한 텐서플로와 파이토치의 영향력

5.2 파이토치 개념 익히기

- 본 강의에서는 인공지능을 구현하기 위해 파이토치 라이브러리를 사용함
- 본 강의의 실험을 위해 파이토치를 자신의 컴퓨터에 직접 설치할 수 있으나, 본 강의에서는 환경 설정이 완료된 〈캐글 노트북〉을 사용할 예정임
- 즉, 개인 컴퓨터에 파이토치를 직접 설치하는 방법은 본 강의에서 다루지 않음

- 실습을 위해 사용할 코드 (5.3, 5.4)
  - https://www.kaggle.com/code/limguentaek/chapter-05-pytorch



5.2 파이토치 개념 익히기

5.2.1 파이토치와 넘파이이 호환

[프로그램 5-1] 파이토치 버전과 동작 확인

```
import torch

print(torch.__version__) # pytorch 버전 확인
a=torch.rand(2,3)        # (2 by 3의 크기를 가지는 Random 텐서 생성)
print(a)
print(type(a))
```

03행 torch version 확인

04행 0~1사이의 값을 가지는 난수를 원하는 크기에 해당하는 Tensor 반환

```
2.0.0
tensor([[0.5809, 0.6648, 0.1949],
        [0.0583, 0.1108, 0.4529]])
<class 'torch.Tensor'>
```

https://www.kaggle.com/code/limguentaek/chapter-05-pytorch

5.2 파이토치 개념 익히기

5.2.1 파이토치와 넘파이이 호환

[프로그램 5-2] pytorch와 numpy 의 호환

```
import torch
import numpy as np

t=torch.rand(2,3)
print("pytorch로 생성한 텐서 : \n {}".format(t))

n=np.random.uniform(low=0.0, high=1.0, size=(2,3))
print("\nnumpy로 생성한 ndarray : \n {}".format(n))

res=t+n
print("\n덧셈 결과 : \n {}".format(res))  # pytorch도 numpy와 호환되기 때문에 덧셈이 가능
```

04행 0~1사이의 값을 가지는 난수를 원하는 크기에 해당하는 Tensor 반환

07행 low~high사이의 랜덤 난수를 생성하여 원하는 사이즈의 ndarray 반환

```
pytorch로 생성한 텐서 :
 tensor([[0.1852, 0.6459, 0.8372],
        [0.9553, 0.1209, 0.7753]])

numpy로 생성한 ndarray :
 [[0.64764965 0.82919971 0.27473929]
 [0.02202838 0.02749204 0.27035916]]

덧셈 결과 :
 tensor([[0.8328, 1.4751, 1.1120],
        [0.9773, 0.1484, 1.0457]], dtype=torch.float64)
```

5.2 파이토치 개념 익히기

5.2.2 텐서 이해하기

[프로그램 5-3] pytorch가 제공하는 데이터셋의 텐서 구조 확인하기

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor

# MNIST 읽고 텐서 구조 확인 (학습 데이터)
MNIST_training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)
# MNIST 읽고 텐서 구조 확인 (평가 데이터)
MNIST_test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

print("MNIST 학습 이미지 데이터 텐서 모양 : {}".format(MNIST_training_data[0][0].shape))
print("MNIST 학습 라벨 데이터 : {}".format(MNIST_training_data[0][1]))

print("MNIST 평가 이미지 데이터 텐서 모양 : {}".format(MNIST_test_data[0][0].shape))
print("MNIST 평가 라벨 데이터 : {}".format(MNIST_test_data[0][1]))
```

06행 root : data를 다운 받을 root 경로

07행 train : Train data를 받을지 말지 (True/False)

08행 download : 직접 다운로드 하여 root 경로에 저장할지 말지 (True/False)

09행 transform : 데이터를 불러올 때 어떤 변환을 적용할 지

5.2 파이토치 개념 익히기

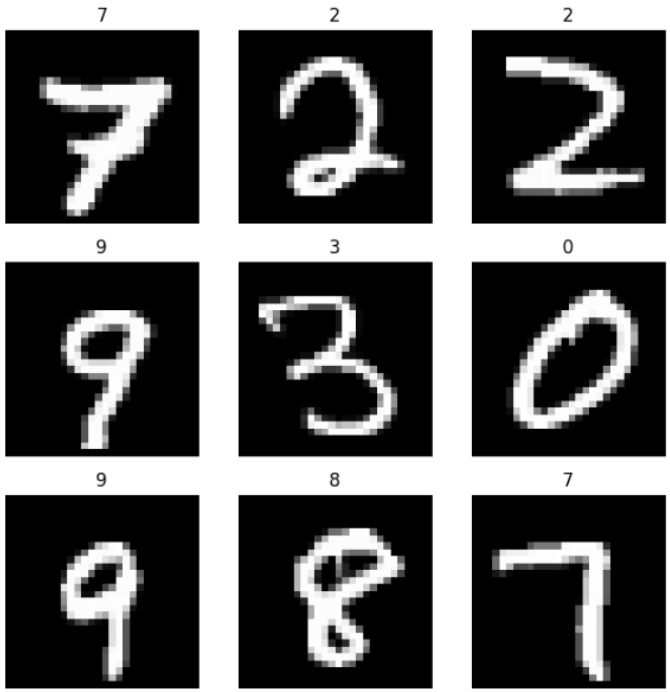
5.2.2 텐서 이해하기

[프로그램 5-3] pytorch가 제공하는 데이터셋의 텐서 구조 확인하기

```
import matplotlib.pyplot as plt

labels_map = {
    0: "0",
    1: "1",
    2: "2",
    3: "3",
    4: "4",
    5: "5",
    6: "6",
    7: "7",
    8: "8",
    9: "9",
}

figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(MNIST_training_data), size=(1,)).item()
    img, label = MNIST_training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



5.3 파이토치 프로그래밍 기초

5.3.1 sklearn의 표현력 한계

- 4장에서는 sklearn 라이브러리를 활용해 기계 학습을 구현하였음
- MLP를 떠올려보면 sklearn 은 딥러닝을 구현하기에는 표현력의 한계가 있음
- 딥러닝 모델은 MLP 처럼 단순하지 않으며, 컨볼루션 연산을 사용하는 “컨볼루션 신경망”으로 확장되고, 은닉 노드끼리 에지로 연결되는 “순환 신경망”으로 확장되며, 신경망 구조를 벗어난 “강화학습”으로 확장됨
- 〈텐서플로우〉나 〈파이토치〉 같은 라이브러리는 이런 복잡도를 지원할 수 있도록 바닥부터 새롭게 설계한 딥러닝 전용 소프트웨어 이며, 이들은 명령어로 층을 쌓아가는 방식으로 프로그래밍을 진행함
- 딥러닝 프로그램에도 디자인 패턴이 있고, 디자인 패턴을 따른 좋은 예제 코드가 많으니 이들을 잘 관찰하고 기억하면 어렵지 않게 딥러닝 프로그래밍에 익숙해질 수 있음

sklearn library ➔ pytorch library

5.3 파이토치 프로그래밍 기초

5.3.2 파이토치를 이용한 퍼셉트론 프로그래밍

[프로그램 5-4] 파이토치 프로그래밍 : 학습된 퍼셉트론 동작 (학습된 파라미터 이용하기)

```
import torch

x = torch.tensor([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]])
y = torch.tensor([[-1],[1],[1],[1]])

w = torch.autograd.Variable(torch.tensor([[1.0],[1.0]]),requires_grad=True)
b = torch.autograd.Variable(torch.tensor([-0.5]),requires_grad=True)

pred = torch.sign(x.matmul(w)+b)

print("OR 프로그래밍 정답 값 : \n {}".format(y))
print("OR 프로그래밍 예측 값 : \n {}".format(pred))
```

06행 gradient 추적을 허용하는 가중치 행렬 W 정의

07행 gradient 추적을 허용하는 편향 b 정의

09행 wx+b 형태로 계산한 다음 결과의 부호만을 반환

```
OR 프로그래밍 정답 값 :
 tensor([[-1],
        [ 1],
        [ 1],
        [ 1]])
OR 프로그래밍 예측 값 :
 tensor([[-1.],
        [ 1.],
        [ 1.],
        [ 1.]], grad_fn=<SignBackward0>)
```

5.3 파이토치 프로그래밍 기초

5.3.2 파이토치를 이용한 퍼셉트론 프로그래밍

[프로그램 5-5] 파이토치 프로그래밍 : 퍼셉트론 학습 (학습하여 파라미터 구하기)

```
01 import torch
02
03 def forward(w,b,x):
04     x = x.matmul(w)+b
05     x = torch.tanh(x)
06     return w,b, x
07
08 def loss_mse(pred,y):
09     loss = torch.mean((y-pred)**2)
10     return loss
11
12 if __name__ == '__main__':
13     x = torch.tensor([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]])
14     y = torch.tensor([[-1],[1],[1],[1]])
15
16     w = torch.autograd.Variable(torch.randn(2,1),requires_grad=True)
17     b = torch.autograd.Variable(torch.zeros(1),requires_grad=True)
18
19     optimizer = torch.optim.SGD([w,b], lr=0.1)
20
21     for iter in range(500):
22         w,b,pred = forward(w,b,x)
23         cost = loss_mse(pred,y)
24
25         optimizer.zero_grad()
26         cost.backward()
27         optimizer.step()
28
29     w,b,pred = forward(w,b,x)
30     print(torch.sign(pred))
```

```
tensor([[-1.],
        [ 1.],
        [ 1.],
        [ 1.]], grad_fn=<SignBackward0>)
```

02행 wx+b 형태로 계산하고 tanh 함수를 적용하여 순전파 연산 진행

07행 예측값과 정답값의 제곱 오차를 계산하는 MSE 손실함수

15행 gradient 추적을 허용하는 가중치,편향 정의

18행 확률적 경사하강법(SGD)에 해당하는 옵티마이저 선언(W,b를 최적화)

24행 optimizer.zero_grad() : 매 iteration 마다 gradient 초기화

25행 cost.backward() : 손실 함수로부터 역전파(backpropagation) 수행

26행 optimizer.step() : 역전파로부터 계산된 gradient를 토대로 W,b 업데이트

5.3 파이토치 프로그래밍 기초

5.3.2 파이토치를 이용한 퍼셉트론 프로그래밍

[프로그램 5-6] 파이토치 프로그래밍 : 퍼셉트론 학습 (추상화)

```
01 class Perceptron(torch.nn.Module):
02     def __init__(self,input_dim,output_dim):
03         super(Perceptron, self).__init__()
04
05         self.perceptron = torch.nn.Linear(in_features=input_dim,
06                                           out_features=output_dim,bias=True)
07         self.activation = torch.nn.Tanh()
08
09         self.model = torch.nn.Sequential(self.perceptron,self.activation)
10
11     def forward(self, x):
12         return self.model(x)
13
14 def loss_mse(pred,y):
15     loss = torch.mean((y-pred)**2)
16     return loss
17
18 if __name__ == '__main__':
19     x = torch.tensor([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]])
20     y = torch.tensor([[-1],[1],[1],[1]])
21
22     Perceptron = Perceptron(input_dim=2, output_dim=1)
23     optimizer = torch.optim.SGD(Perceptron.parameters(), lr=0.1)
24
25     for iter in range(500):
26         pred = Perceptron(x)
27         cost = loss_mse(pred,y)
28         optimizer.zero_grad()
29         cost.backward()
30         optimizer.step()
31
32     pred=Perceptron(x)
33     print(torch.sign(pred))
```

```
tensor([[-1.],
        [ 1.],
        [ 1.],
        [ 1.]], grad_fn=<SignBackward0>)
```

01행 torch.nn.Module을 상속 받는 Class 정의

05행 torch.nn.Linear를 이용하여 선형 모델(퍼셉트론)을 정의

07행 활성화 함수는 torch.nn.Tanh()를 사용

09행 torch.nn.Sequential을 사용하여 모듈을 순차적으로 정의

11행 순전파를 다음과 같이 정의 (torch.nn.Module을 상속 받았기 때문) Perceptron(x) 처럼 사용가능

28행 optimizer.zero_grad() : 매 iteration 마다 gradient 초기화

29행 cost.backward() : 손실 함수로부터 역전파(backpropagation) 수행

30행 optimizer.step() : 역전파로부터 계산된 gradient를 토대로 W,b 업데이트

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.1 MNIST 인식

[프로그램 5-7(a)] 파이토치 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
import torch
import tqdm

if __name__ == '__main__':

    history = dict()
    history['train_acc'] = list()
    history['test_acc'] = list()
    history['train_loss'] = list()
    history['test_loss'] = list()

    # MNIST 읽고 텐서 구조 확인 (학습 데이터)
    training_data = datasets.MNIST(
      root="data",
      train=True,
      download=True,
      transform=ToTensor()
    )
    # MNIST 읽고 텐서 구조 확인 (평가 데이터)
    test_data = datasets.MNIST(
      root="data",
      train=False,
      download=True,
      transform=ToTensor()
    )
```

➔ 학습 중간에 정확도와 손실을 기록할 딕셔너리 정의

➔ MNIST 학습 데이터 정의

➔ MNIST 평가 데이터 정의

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.1 MNIST 인식

[프로그램 5-7(a)] 파이토치 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
train_dataloader = torch.utils.data.DataLoader(training_data,batch_size=128)
test_dataloader = torch.utils.data.DataLoader(test_data,batch_size=128,shuffle=False)

MLP = MultiLayerPerceptron(input_dim=784, hidden_dim=1024, output_dim=10)
MLP = MLP.cuda()
optimizer = torch.optim.Adam(MLP.parameters(), lr=0.001)

for iter in tqdm.tqdm(range(30)):
    MLP, history = train_epoch(train_dataloader, optimizer, MLP, loss_mse, history)
    with torch.no_grad():
        MLP, history = test_epoch(test_dataloader, optimizer, MLP, loss_mse, history)
```

➔ 배치 단위 학습을 위해 MNIST 학습 데이터 로더 정의

➔ 배치 단위 학습을 위해 MNIST 평가 데이터 로더 정의

➔ 신경망 최적화를 위해 Adam 사용

```
import torch

class MultiLayerPerceptron(torch.nn.Module):
    def __init__(self,input_dim, hidden_dim, output_dim):
        super(MultiLayerPerceptron, self).__init__()

        self.input_layer = torch.nn.Linear(in_features=input_dim,
                                           out_features=hidden_dim,bias=True)
        self.hidden_layer = torch.nn.Linear(in_features=hidden_dim,
                                            out_features=output_dim,bias=True)
        self.activation = torch.nn.Tanh()

        self.model =  torch.nn.Sequential(self.input_layer,self.activation,
                                          self.hidden_layer, self.activation)

    def forward(self, x):
        return self.model(x)
```

➔ torch.nn.Module을 상속 받는 Class 정의

➔ torch.nn.Linear를 이용하여 선형 모델(퍼셉트론)을 정의

➔ 활성화 함수는 torch.nn.Tanh()를 사용

➔ torch.nn.Sequential을 사용하여 모듈을 순차적으로 정의

➔ 순전파를 다음과 같이 정의 (torch.nn.Module을 상속 받았기 때문) Perceptron(x) 처럼 사용가능

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.1 MNIST 인식

[프로그램 5-7(a)] 파이토치 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
def train_epoch(dataloader, optimizer, model, loss_fn, log_dict):
    acc=0
    loss=0
    model.train()
    for X, y in dataloader:
        X = X.view(-1,784).cuda()
        y = y.cuda()

        pred = model(X)
        cost = loss_fn(pred, y)

        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

        acc += torch.sum(torch.argmax(pred,dim=1)==y).item()
        loss += cost.item()

    log_dict['train_acc'].append(acc/len(dataloader.dataset))
    log_dict['train_loss'].append(loss)

    return model, log_dict
```

➔ model.train() : 모델을 학습 모드로 변경 (batchnorm이나 dropout 관련)

➔ 28x28 크기의 데이터(X)를 784x1 데이터로 reshape & GPU 메모리에 올려줌

➔ 정답 데이터(y)도 GPU 메모리에 올려줌

➔ optimizer.zero_grad() : 매 iteration 마다 gradient 초기화

➔ cost.backward() : 손실 함수로부터 역전파(backpropagation) 수행

➔ optimizer.step() : 역전파로부터 계산된 gradient를 토대로 W,b 업데이트

➔ 예측값과 정답값이 동일한 개수를 카운트하여 저장

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.1 MNIST 인식

[프로그램 5-7(a)] 파이토치 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
def test_epoch(dataloader, optimizer, model, loss_fn, log_dict):
    acc=0
    loss=0
    model.eval()
    for X, y in dataloader:
        X = X.view(-1,784).cuda()
        y = y.cuda()

        pred = model(X)
        cost = loss_fn(pred,y)

        acc += torch.sum(torch.argmax(pred,dim=1)==y).item()
        loss += cost.item()

    log_dict['test_acc'].append(acc/len(dataloader.dataset))
    log_dict['test_loss'].append(loss)

    return model, log_dict
```

➔ model.eval() : 모델을 평가 모드로 변경 (batchnorm이나 dropout 관련)

➔ 28x28 크기의 데이터를 784x1 데이터로 reshape & 동시에 GPU 메모리에 올림

➔ 정답 데이터도 GPU 메모리에 올림

14행 예측값과 정답값이 동일한 개수를 카운트하여 저장

5.4 파이토치 다층 퍼셉트론 프로그래밍

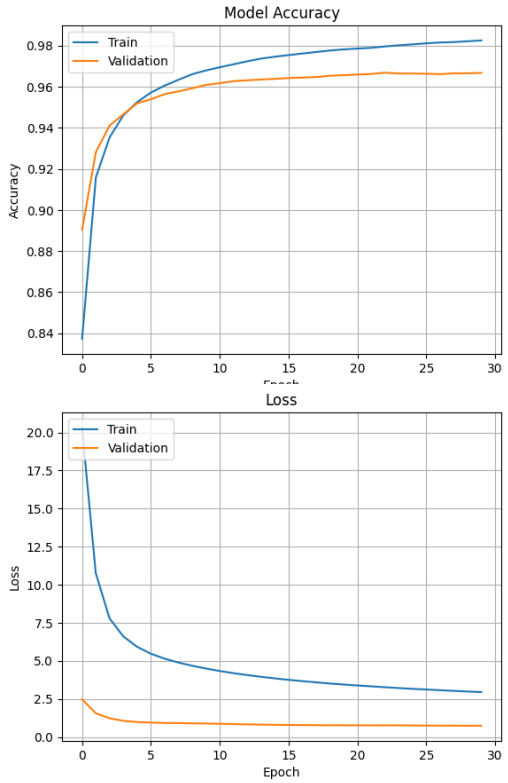
5.4.2 학습 곡선 시각화

[프로그램 5-7(b)] 파이토치 프로그래밍: 다층 퍼셉트론으로 MNIST 인식

```
import matplotlib.pyplot as plt

# 정확도 곡선
plt.plot(history['train_acc'])
plt.plot(history['test_acc'])
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(['Train','Validation'],loc='upper left')
plt.grid()
plt.show()

# 손실 곡선
plt.plot(history['train_loss'])
plt.plot(history['test_loss'])
plt.title("Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(['Train','Validation'],loc='upper left')
plt.grid()
plt.show()
```

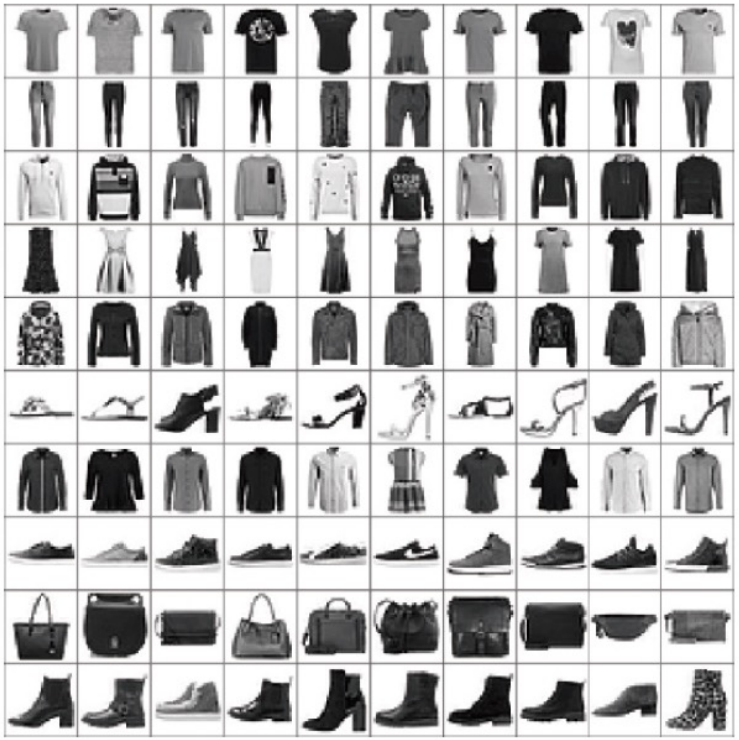


과적합 발생 여부 확인을 위한 학습 곡선 시각화

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.3 fashion MNIST 인식

- 파이토치는 필기 숫자 데이터인 MNIST와 아주 비슷한 fashion MNIST 를 제공함
- 샘플은 28x28 맵으로 표현되며 훈련 집합 60,000개 샘플, 테스트 집합 10,000개 샘플로 구성됨
- 라벨이 {0,1,2,3,4,5,6,7,8,9}에서 {T-shirt, top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot}로 바뀌고 샘플 영상이 패션 아이템이라는 점만 다름



5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.3 fashion MNIST 인식

[프로그램 5-8]은 [프로그램 5-7] 에서 데이터셋 준비하는 코드만 변경하면 됨

[프로그램 5-8] 파이토치 프로그래밍: 다층 퍼셉트론으로 Fashion MNIST 인식

```
import torch
import tqdm

if __name__ == '__main__':

    history = dict()
    history['train_acc'] = list()
    history['test_acc'] = list()
    history['train_loss'] = list()
    history['test_loss'] = list()

    training_data = datasets.FashionMNIST(
      root="data",
      train=True,
      download=True,
      transform=ToTensor()
    )

    test_data = datasets.FashionMNIST(
      root="data",
      train=False,
      download=True,
      transform=ToTensor()
    )
```

12행 FashionMNIST 학습 데이터 정의

19행 FashionMNIST 평가 데이터 정의

5.4 파이토치 다층 퍼셉트론 프로그래밍

5.4.3 fashion MNIST 인식

[프로그램 5-8] 파이토치 프로그래밍: 다층 퍼셉트론으로 Fashion MNIST 인식

```
train_dataloader = torch.utils.data.DataLoader(training_data,batch_size=128)
test_dataloader = torch.utils.data.DataLoader(test_data,batch_size=128,shuffle=False)

MLP = MultiLayerPerceptron(input_dim=784, hidden_dim=1024, output_dim=10)
MLP = MLP.cuda()
optimizer = torch.optim.Adam(MLP.parameters(), lr=0.001)

for iter in tqdm.tqdm(range(30)):
    MLP, history = train_epoch(train_dataloader, optimizer, MLP, loss_mse, history)
    with torch.no_grad():
        MLP, history = test_epoch(test_dataloader, optimizer, MLP, loss_mse, history)
```

[프로그램 5-7]과 같음

5.5 깊은 다층 퍼셉트론

- 〈다층 퍼셉트론〉에 은닉층을 더 많이 추가하면 〈깊은 다층 퍼셉트론〉 DMLP: Deep Multi-Layer Perceptron 이 되며, 깊은 다층 퍼셉트론은 은닉층만 추가하면 되기 때문에 가장 쉽게 생각할 수 있는 딥러닝 모델임

5.5 깊은 다층 퍼셉트론

5.5.1 구조와 동작

- [그림5-8]은 깊은 다층 퍼셉트론의 구조임
- 양 끝에 입력층과 출력층이 있으며 중간에 L-1개의 은닉층이 있어 총 L개 은닉층 수+출력층의 층이 있음
- 은닉층이 하나인 다층 퍼셉트론은 L=2인 특수한 경우임
- 입력층 특징 벡터의 요소 개수에 따라 d개 노드를 가지며 출력층은 부류 개수에 따라 c개 노드를 가짐
  - 예를 들어, MNIST 필기 숫자 데이터의 경우 화소를 특징으로 사용한다면 d=28x28=784이고, c=10임, i번째 은닉층의 개수는 프로그래머가 설정하는 하이퍼 매개변수이며 n_i로 표기함
- 깊은 다층 퍼셉트론은 인접한 두 층의 모든 노드 사이에 에지가 있는 완전 연결 fc: fully connected 구조임

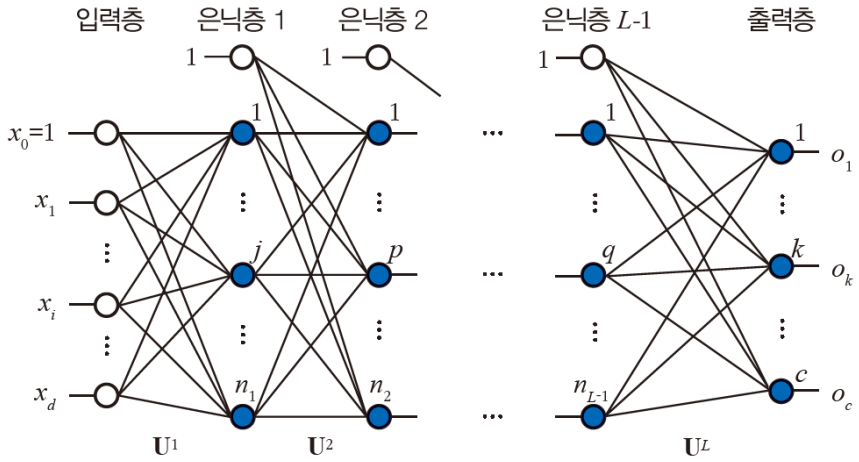


그림 5-8 깊은 다층 퍼셉트론의 구조

5.5 깊은 다층 퍼셉트론

5.5.1 구조와 동작

- 깊은 다층 퍼셉트론은 가중치가 많음
- 가중치가 많으면 1)계산 시간이 길어지고 2)과잉 적합이 발생할 가능성도 커짐
- 제시된 문제가 극복되어야 딥러닝(깊은 신경망 학습)이 성공할 수 있음
- 시간이 오래 걸리는 계산 문제는 GPU를 사용해 해결이 가능하고, 과잉 적합은 여러가지 규제 기법으로 해결 가능함

- MNIST 데이터셋 L=5, 은닉층 노드 개수가 500일 때의 총 가중치를 구해보자.
  - L=5이므로, 은닉층의 개수는 L-1 로 4개임
  - 입력층-은닉층1 : (784+1)*500 //+1은 바이어스
  - 은닉층1-은닉층2 : (500+1)*500*1
  - 은닉층2-은닉층3: (500+1)*500*1
  - 은닉층3-은닉층4: (500+1)*500*1
  - 은닉층4-출력층 : (500+1)*10 //10개의 클래스
  - 따라서 총 가중치의 수는 1,149,010개임

5.5 깊은 다층 퍼셉트론

5.5.2 오류 역전파 알고리즘

- [그림5-9]는 다층 퍼셉트론의 학습 절차를 설명
  - 특징 벡터를 입력층에 입력
  - 전방 계산 (forward)
  - 전방 계산으로 얻은 예측 값을 기대 값과 비교해 오차 계산
  - 오차에 따라 U^L, U^{L-1} ,…,U^1 순으로 가중치 갱신

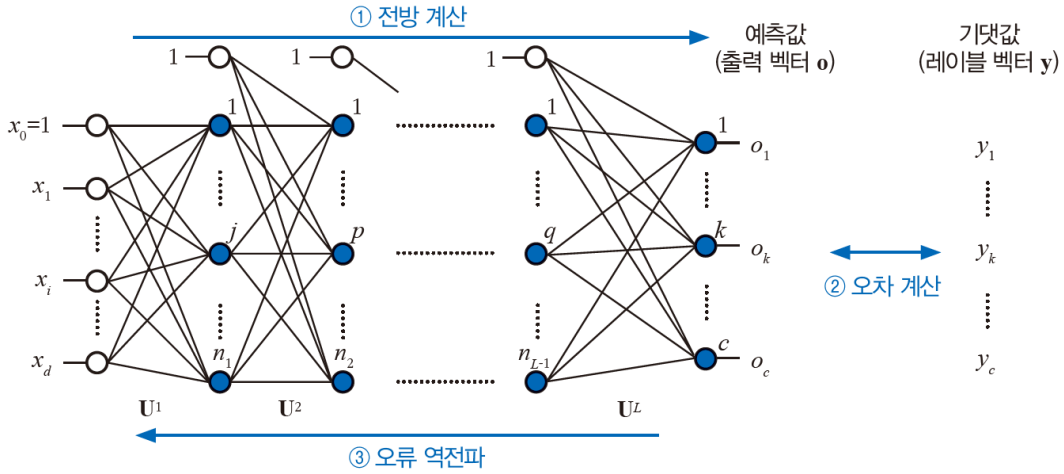


그림 5-9 깊은 다층 퍼셉트론이 사용하는 오류 역전파 알고리즘

5.5 깊은 다층 퍼셉트론

5.5.3 깊은 다층 퍼셉트론 프로그래밍

[프로그램 5-9] 깊은 다층 퍼셉트론으로 MNIST 인식 (프로그램 5-7 과 나머지 동일)

```
import torch

class DeepNeuralNetwork(torch.nn.Module):
    def __init__(self,input_dim, hidden_dim, output_dim):
        super(DeepNeuralNetwork, self).__init__()

        self.input_layer = torch.nn.Linear(in_features=input_dim,
                                           out_features=hidden_dim,bias=True)
        self.hidden_layer1 = torch.nn.Linear(in_features=hidden_dim,
                                             out_features=hidden_dim,bias=True)
        self.hidden_layer2 = torch.nn.Linear(in_features=hidden_dim,
                                             out_features=hidden_dim,bias=True)
        self.hidden_layer3 = torch.nn.Linear(in_features=hidden_dim,
                                             out_features=hidden_dim,bias=True)
        self.output_layer = torch.nn.Linear(in_features=hidden_dim,
                                            out_features=output_dim,bias=True)

        self.activation = torch.nn.Tanh()

        self.model =  torch.nn.Sequential(self.input_layer, self.activation,
                                          self.hidden_layer1, self.activation,
                                          self.hidden_layer2, self.activation,
                                          self.hidden_layer3, self.activation,
                                          self.output_layer, self.activation,)


    def forward(self, x):
        return self.model(x)
```

03행 torch.nn.Module을 상속 받는 Class 정의

07~16행 torch.nn.Linear를 이용하여 선형 모델(퍼셉트론)을 여러 개 정의

입력 : 784, 은닉층1 : 1024, 은닉층2 : 512, 은닉층3 : 512, 은닉층4 : 512, 출력층 : 10

09행 활성화 함수는 torch.nn.Tanh()를 사용

11행 torch.nn.Sequential을 사용하여 모듈을 순차적으로 정의

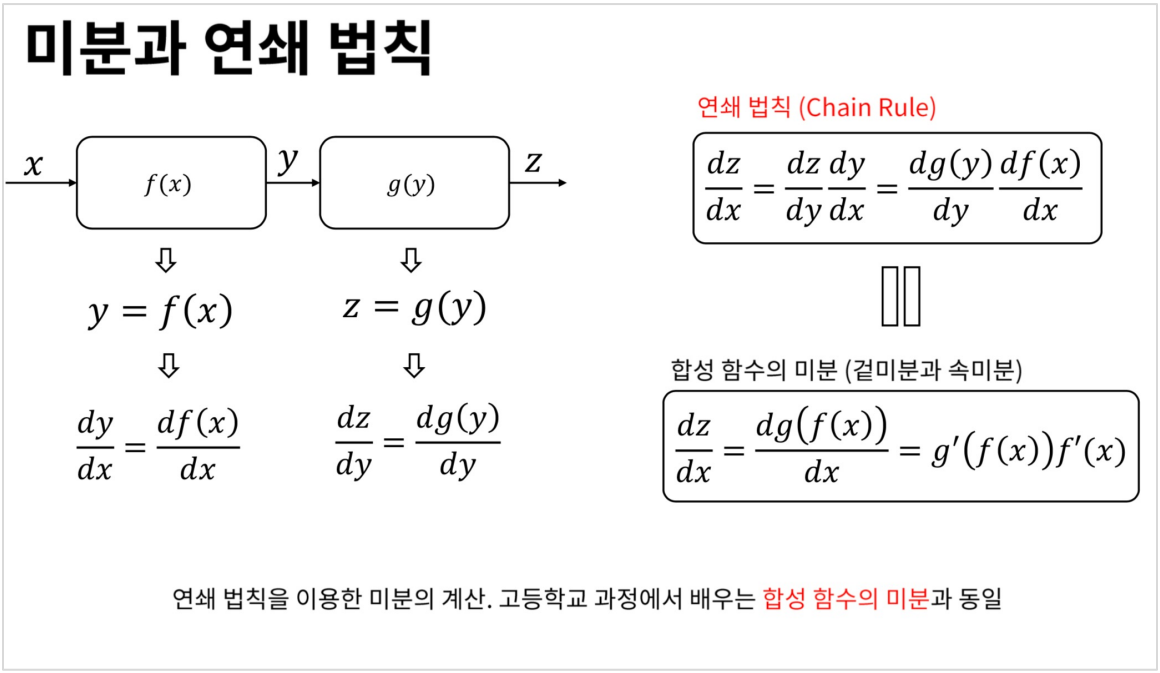
15행 순전파를 다음과 같이 정의 (torch.nn.Module을 상속 받았기 때문) Perceptron(x) 처럼 사용가능

5.6 딥러닝 학습 전략

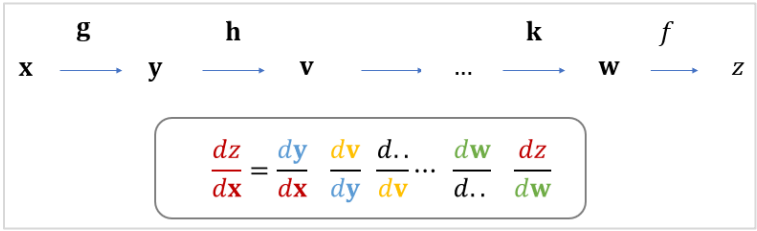
5.6.1 그레디언트 소멸 문제와 해결책

- 미분 이론의 연쇄 법칙(chain rule)에 따르면,

  [그림5-9]에서 l번째 층의 그레디언트는 오른쪽에 있는 l+1번째 층의 그레디언트에 자신의 층에서 발생한 그레디언트를 곱하여 구함



5.6 딥러닝 학습 전략



5.6.1 그레디언트 소멸 문제와 해결책

- 미분 이론의 연쇄 법칙(chain rule)에 따르면, [그림5-9]에서 l번째 층의 그레디언트 오른쪽에 있는 l+1번째 층의 그레디언트에 자신의 층에서 발생한 그레디언트를 곱하여 구함
- 그레디언트가 작으면 오른쪽에서 왼쪽으로 진행하면서 그레디언트가 점점 작아지는 현상이 발생함
- 그레디언트가 기하급수적으로 작아지는 현상을 그레디언트 소멸 gradient vanishing 이라 함
- 그레디언트 소멸 현상이 발생하면 오른쪽에 있는 층의 가중치는 갱신이 제대로 일어나지만 왼쪽으로 갈수록 갱신이 매우 느림. 결국 전체적으로 신경망 학습이 매우 느려져서 오랫동안 학습해도 수렴에 도달하지 못하는 문제가 발생함

- 예) 깊이가 10, 모든 층의 그레디언트 0.01 이라고 가정
  - 열 번 째 층의 그레디언트가 0.01인데,
  - 아홉 번째 층의 그레디언트는 0.01x0.01=0.0001
  - 여덟 번째 층의 그레디언트는 0.0001x0.01=0.000001
  - 일곱 번째 층의 그레디어느는 0.000001x0.01=0.00000001
  - ...

5.6 딥러닝 학습 전략

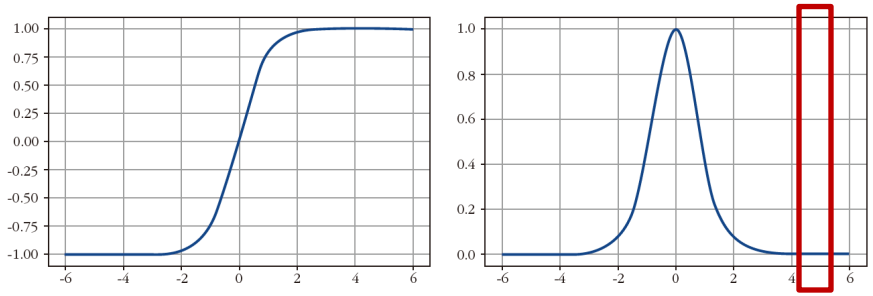
5.6.1.1 병렬 처리로 해결

- 그레디언트 소멸 문제에 대처하려면 여러 가지 방법을 결합해서 사용해야 함
- 가장 직접적인 방법은 더 빠른 컴퓨터를 사용하는 것
- 딥러닝은 병렬 처리를 하는 계산 가속기인 GPU를 사용
  - 개인이 쓰기에 적합한 GPU는 Nvidia GTX 등을 구매 가능
  - 캐글이나 코랩을 사용하는 사람들은 무료로 제공하는 GPU 또는 TPU를 사용할 수 있음

5.6 딥러닝 학습 전략

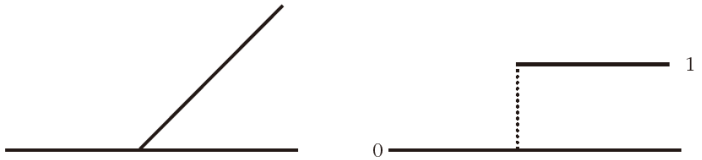
5.6.1.2 좋은 활성함수 사용으로 해결 : ReLU 함수 사용

- 그레디언트 소멸 문제는 시그모이드 함수에서 더욱 심함
  - (x축) S=5 이면 시그모이드의 그레디언트는 0에 가까움. 정확하게는 0.0000004501 임.



(a) tanh 시그모이드와 그레이디언트

- ReLU 함수는 시그모이드 함수의 문제점을 해소해 줌
  - ReLU는 s가 양수일 때 그레디언트가 1이고, 음수일 때 0임
  - 따라서 시그모이드에 비해 그레디언트 소멸이 발생할 가능성이 낮음



(b) ReLU와 그레이디언트

5.6 딥러닝 학습 전략

5.6.2 과잉 적합과 과잉 적합 회피 전략

- 과소 적합과 과잉 적합
  - 데이터에 비해 작은 용량의 모델을 사용해 오차가 많아지는 현상을 과소 적합underfitting이라고 함
  - 맨 왼쪽은 1차 방정식을 모델로 채택한 경우이며, 훈련데이터를 제대로 모델링하기에는 용량이 작다(가중치의 수가 작은 모델)는 사실을 쉽게 알 수 있음
  - 과소 적합을 해소하려면 모델의 용량을 크게 하면 됨
    - 아래의 그림은 용량이 큰 모델인 2차, 3차, 12차 방정식을 모델로 사용한 경우의 예시

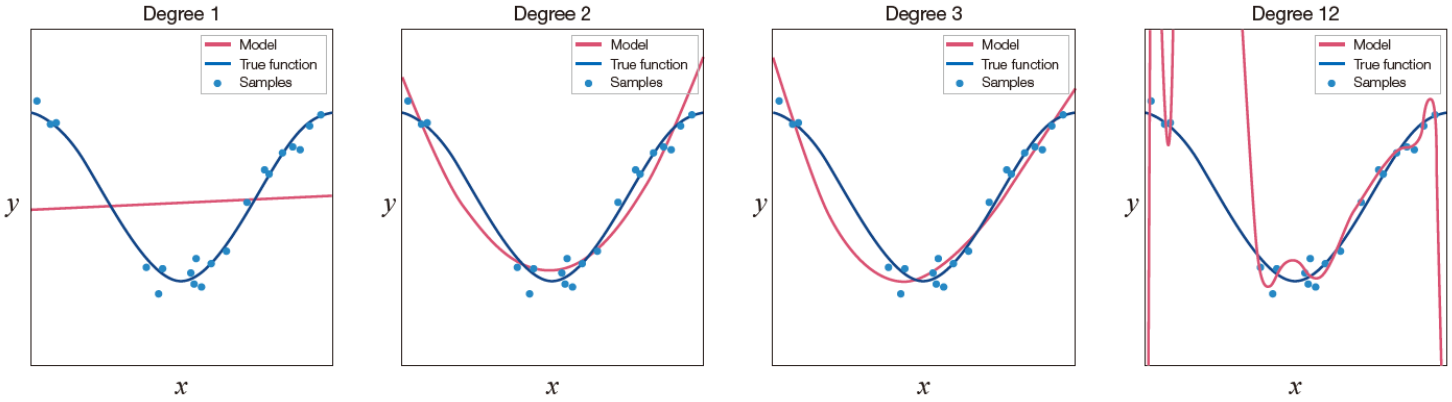


그림 5-12 과소 적합과 과잉 적합

5.6 딥러닝 학습 전략

5.6.2 과잉 적합과 과잉 적합 회피 전략

- 과소 적합과 과잉 적합
  - 〈너무 큰 모델〉이 데이터에 과도하게 적응하여 일반화 능력을 잃는 현상을 과잉 적합 overfitting이라 부름
  - 기계 학습의 최종 목적은 일반화 능력을 최대화 하는 것. 즉, 새로운 샘플에 대해 높은 정확률을 확보하는 것
  - 그런데, 빨간 점으로 표시된 예측 값이 파란 점으로 표시된 참값을 크게 벗어나는 결과를 보이고 있으며, 이는 모델의 용량이 데이터 복잡도에 비해 너무 커서 발생함
  - [그림 5-13]은 x0라는 새로운 샘플에 대해 12차 모델이 예측한 결과를 보여줌

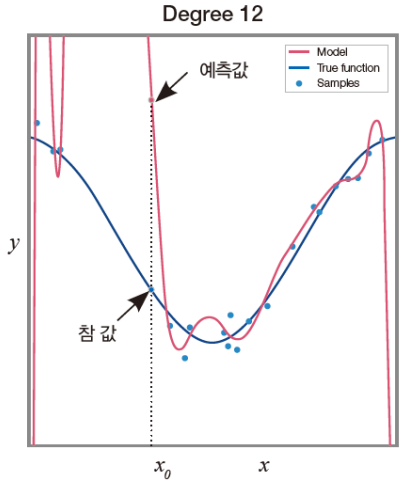
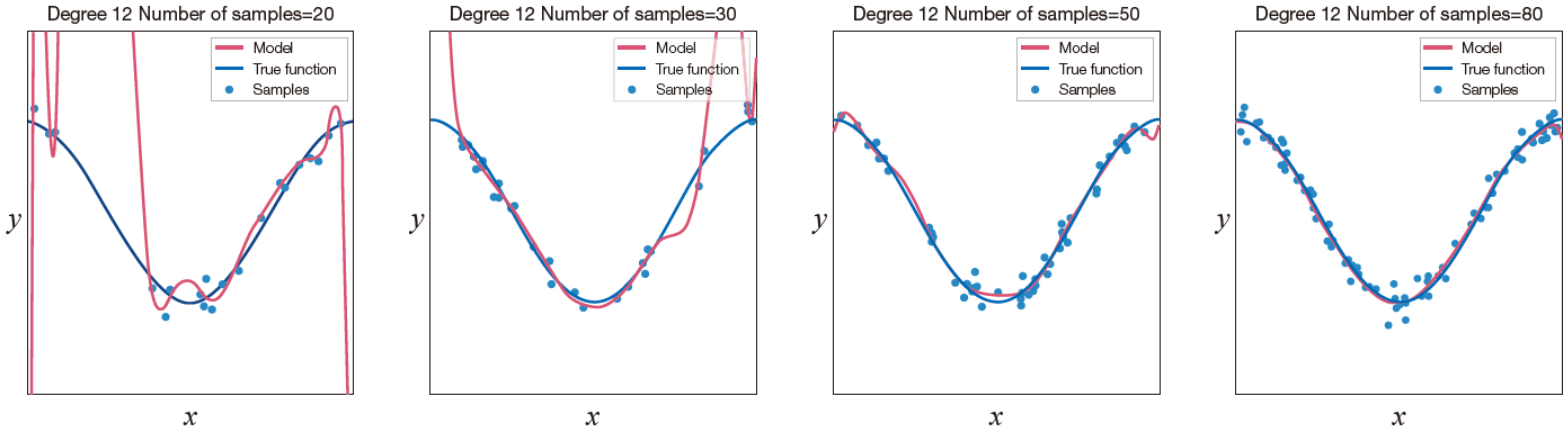


그림 5-13 과잉 적합에 따른 부정확한 예측

5.6 딥러닝 학습 전략

5.6.2 과잉 적합과 과잉 적합 회피 전략

- 과소 적합과 과잉 적합
  - 너무 큰 모델이 데이터에 과도하게 적응하여 일반화 능력을 잃는 현상을 과잉 적합 overfitting이라 부름
  - 과잉 적합을 해소하는 전략은 아주 많은데, 그중 가장 확실한 방법은 데이터의 양을 늘리는 것



(a) 훈련 집합 크기: 20 (b) 훈련 집합 크기: 30 (c) 훈련 집합 크기: 50 (d) 훈련 집합 크기: 80

그림 5-14 데이터 양을 늘려 과잉 적합을 해소

5.6 딥러닝 학습 전략

5.6.2 과잉 적합과 과잉 적합 회피 전략

- 과소 적합과 과잉 적합
  - 너무 큰 모델이 데이터에 과도하게 적응하여 일반화 능력을 잃는 현상을 과잉 적합 overfitting이라 부름

  - 딥러닝은 큰 용량의 모델을 사용하되 적절한 규제 기법을 적용해 과잉 적합을 피하는 전략을 구사함
  - 딥러닝에서 사용하는 규제 기법은 데이터의 양을 늘리는 것 외에 조기 멈춤, 가중치 감쇠, 드롭아웃, 앙상블 등이 있음 (6장에서 다룰 예정)

5.7 딥러닝이 사용하는 손실 함수

5.7.1 평균제곱오차

- 통계학에서 아주 오래 전부터 사용된 대표적인 손실 함수로서 신경망이 빌려 쓰는 것
  - 신경망은 여러 개의 샘플로 구성된 미니배치 단위로 손실 함수를 정의하며, 식(5.8)을 미니 배치 단위에 적용하고 평균을 취한 값이 평균제곱오차(MSE, mean squared error)에 해당 됨

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

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

- 평균제곱오차의 문제점
  - 오차 e가 더 큰데 그레이디언트가 더 작은 상황이 발생하기도 함
  - 이는 공부를 못하는 학생이 더 높은 점수를 받는 상황(학습 의욕 저하)과 비슷하며, 학습이 느려지거나 학습이 안되는 상황을 초래할 가능성이 높음

5.7 딥러닝이 사용하는 손실 함수

5.7.2 교차 엔트로피

- 교차 엔트로피는 평균제곱오차의 불공정성 문제를 해결해 줌

- 엔트로피는 확률 분포의 무작위성randomness, 즉 불확실성uncertainty을 측정하는 함수임
  - 엔트로피는 섀넌이 제안한 정보 이론에서 유래함
- 엔트로피 예시
  - 공정한 주사위의 확률 분포는 1,2,3,4,5,6이 나올 확률이 모두 1/6이므로 불확실성이 가장 높음
  - 찌그러진 주사위가 있어 1의 면적이 가장 넓다면 (예측 가능성이 높아지며) 불확실성이 낮아짐
  - ➔ **엔트로피로 표현하면, 공정한 주사위의 엔트로피가 가장 높고, 많이 찌그러질수록 엔트로피가 낮아짐**

- 엔트로피 식

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

5.7 딥러닝이 사용하는 손실 함수

5.7.2 교차 엔트로피

- 교차 엔트로피는 두 확률 분포가 다른 정도를 측정 (정답과 예측한 확률 분포의 다른 정도 측정에 활용)
- 교차 엔트로피 예시
  - 공정한 주사위 2개가 있을 때, 둘의 확률 분포가 같으므로 교차 엔트로피가 작음
  - 공정한 주사위와 찌그러진 주사위로 교차 엔트로피를 계산하면 큰 값이 됨
- 교차 엔트로피 식

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

공정한 주사위 P와 Q의 교차 엔트로피 (모두 1/6)

$$-\left(\frac{1}{6}\log\frac{1}{6}+\ldots+\frac{1}{6}\log\frac{1}{6}\right)=1.7918$$

공정한 주사위 P와 찌그러진 주사위 Q(1이 ½, 나머지는 1/10확률)의 교차 엔트로피

$$-\left(\frac{1}{6}\log\frac{1}{2}+\frac{1}{6}\log\frac{1}{10}+\cdots+\frac{1}{6}\log\frac{1}{10}\right)=2.0343$$

5.7 딥러닝이 사용하는 손실 함수

5.7.2 교차 엔트로피

- 교차 엔트로피는 평균제곱오차의 불공정성 문제를 해결해 줌
- 딥러닝은 손실 함수로 교차 엔트로피를 사용함
  - C : 부류의 개수
  - O: 신경망의 출력
  - Y: 레이블 (원핫인코딩)

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

[예제 5-1] 교차 엔트로피의 합리성 확인

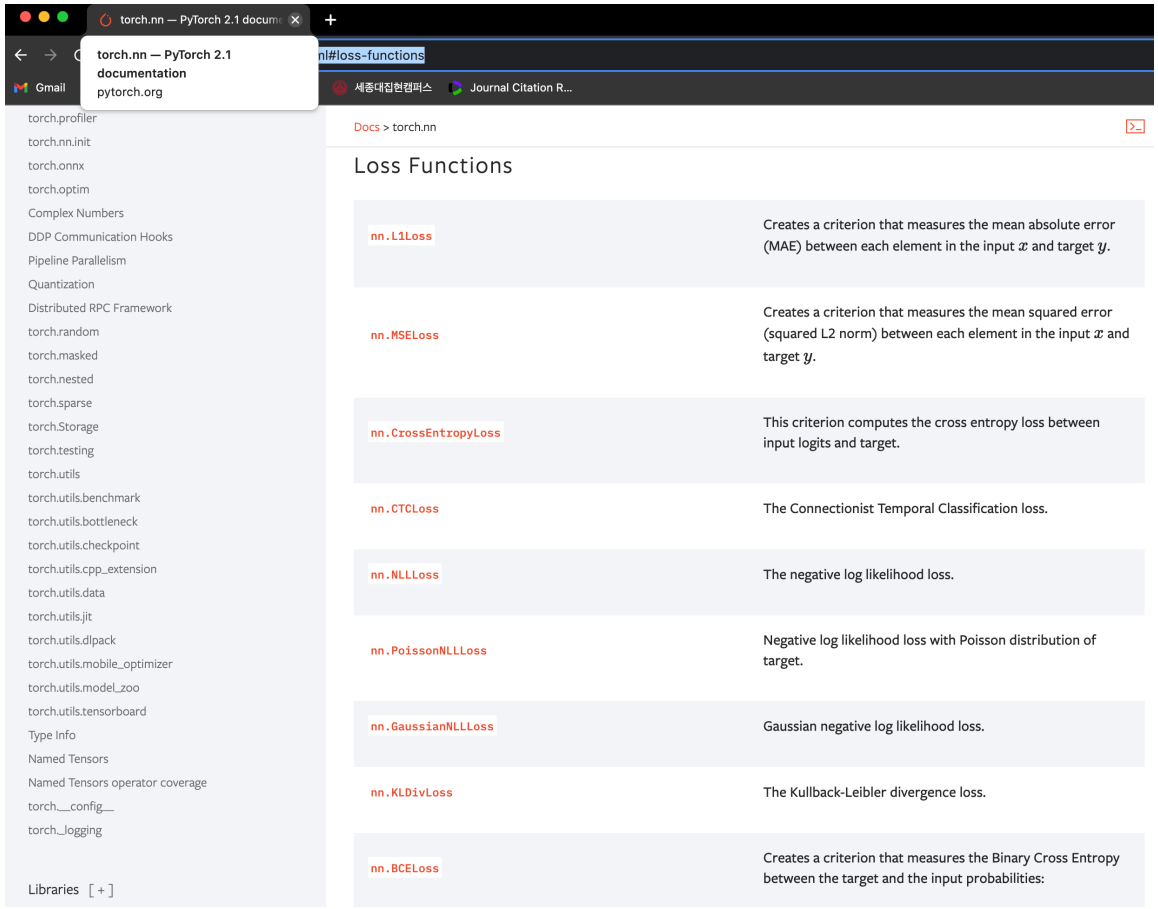
숫자 인식에서 부류 3에 속하는 샘플의 경우 $\mathbf{y}=(0,0,0,1,0,\cdots,0)$이다. 신경망의 출력이 $\mathbf{o}=(0.1,0,0,0.9,0,\cdots,0)$이라 하자. 부류 3에 해당하는 곳이 0.9로서 가장 크므로 신경망이 샘플을 맞힌 경우이다. 식 (5.11)을 계산하면 $e=-(0\times\log(0.1)+0\times\log(0)+0\times\log(0)+1\times\log(0.9)+\cdots+0\times\log(0))=0.1054$가 된다.

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

5.7 딥러닝이 사용하는 손실 함수

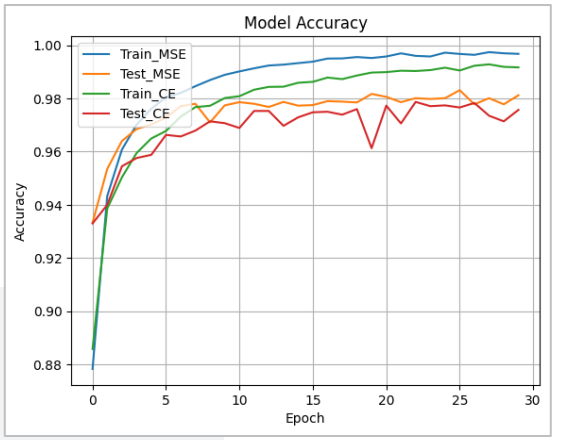
5.7.3 손실 함수의 성능 비교 실험 (MNIST)

- 파이토치에서 지원하는 손실함수는 아래와 같이 확인 가능
  - https://pytorch.org/docs/stable/nn.html#loss-functions



5.7 딥러닝이 사용하는 손실 함수

5.7.3 손실 함수의 성능 비교 실험 (MNIST)



[프로그램5-10] 손실 함수의 성능 비교: 평균제곱오차와 교차 엔트로피

```
history_mse = dict()
history_mse['train_acc'] = list()
history_mse['test_acc'] = list()
history_mse['train_loss'] = list()
history_mse['test_loss'] = list()

for iter in tqdm.tqdm(range(30)):
    DNN, history_mse = train_epoch(train_dataloader, optimizer, DNN, loss_mse, history_mse)
    with torch.no_grad():
        DNN, history_mse = test_epoch(test_dataloader, optimizer, DNN, loss_mse, history_mse)

DNN = DeepNeuralNetwork(input_dim=784, hidden_dim=512, output_dim=10)
DNN = DNN.cuda()
optimizer = torch.optim.Adam(DNN.parameters(), lr=0.001)

history_ce = dict()
history_ce['train_acc'] = list()
history_ce['test_acc'] = list()
history_ce['train_loss'] = list()
history_ce['test_loss'] = list()

for iter in tqdm.tqdm(range(30)):
    DNN, history_ce = train_epoch(train_dataloader, optimizer, DNN, torch.nn.CrossEntropyLoss(), history_ce)
    with torch.no_grad():
        DNN, history_ce = test_epoch(test_dataloader, optimizer, DNN, torch.nn.CrossEntropyLoss(), history_ce)
```

08행 loss_mse 사용하기

10행 loss_mse 사용하기

23행 CrossEntroyLoss 사용하기

25행 CrossEntroyLoss 사용하기

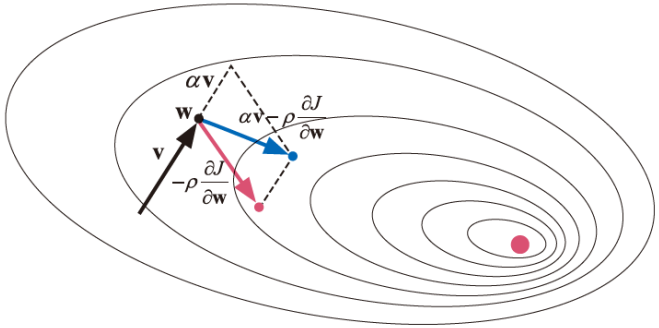
5.8 딥러닝이 사용하는 옵티마이저

- 신경망 학습은 손실 함수의 최저점을 찾아가는 과정
- 최저점을 찾는 문제는 전형적인 최적화 문제
- 신경망에서 SGD와 같은 최적화 알고리즘을 옵티마이저 optimizer 라고 부름
- 신경망 학습에 이용되는 데이터는 잡음과 변화가 아주 심하여 SGD 옵티마이저가 종종 한계를 들어내며, 이런 한계 극복을 위해 모멘텀 momentum과 적응적 학습률 adaptive learning rate 이라는 방법이 소개됨

5.8 딥러닝이 사용하는 옵티마이저

5.8.1 모멘텀을 적용한 옵티마이저

- 물리에서 모멘텀은 이전 운동량을 현재에 반영하는 아이디어를 뜻하며 관성과 관련이 깊음
- 모멘텀을 신경망에 적용하면 성능 향상이 뚜렷하게 나타남



5.8.1.1 모멘텀

- 검은색 실선 : 이전 미니배치에서 누적된 〈방향 벡터gradient〉 v
- 검은 점: 현재 가중치 벡터 w
- 빨간 실선 : 현재 가중치 w를 현재 미니배치에서 계산된 방향 벡터를 고려하여 빨간 점으로 이동

  ➔ 고전적인 SGD
- 파란 실선 : 현재 가중치 w를 현재 미니배치에서 계산된 방향 벡터와 이전 미니배치에서 누적된 방향 벡터를 함께 고려하여 파란 점으로 이동 ➔ 모멘텀을 적용한 SGD

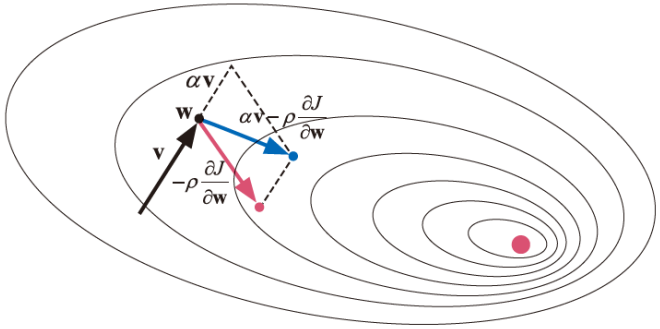
5.8 딥러닝이 사용하는 옵티마이저

5.8.1.1 모멘텀

고전적 SGD: $\mathbf{w} = \mathbf{w} - \rho \frac{\partial J}{\partial \mathbf{w}}$

모멘텀을 적용한 SGD: $\mathbf{v} = \alpha \mathbf{v} - \rho \frac{\partial J}{\partial \mathbf{w}}$

$\mathbf{w} = \mathbf{w} + \mathbf{v}$



- α=0는 고전적 SGD, 모멘텀을 적용한 SGD α는 [0,1]사이에서 조절
- α가 1에 가까울수록 이전 정보에 큰 가중치 부여
- 보통 α=0.5, 0.9를 사용

5.8 딥러닝이 사용하는 옵티마이저

5.8.2 적응적 학습률을 적용한 옵티마이저

- 고전적 SGD 옵티마이저에는 학습률learning rate 이라는 하이퍼 매개변수가 존재

$$\text{고전적 SGD: } \mathbf{w} = \mathbf{w} - \rho \frac{\partial J}{\partial \mathbf{w}}$$

- 그레디언트는 오류가 작아지는 방향을 지시하지만, 얼마나 이동해야 최저점에 도달하는지 정보 부재
- 따라서 SGD 옵티마이저는 작은 학습률을 사용해 조금씩 이동하는 보수적인 정책을 사용
- 일반적으로 SGD는 학습률을 0.01을 기본값으로 사용하는 편인데 더 작은 값을 사용하는 경우도 많음

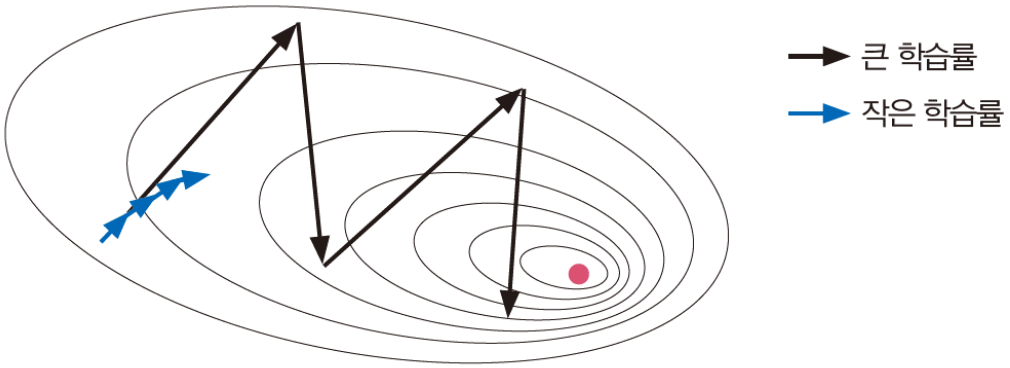


그림 5-16 학습률에 따른 수렴 특성

5.8 딥러닝이 사용하는 옵티마이저

5.8.2 적응적 학습률을 적용한 옵티마이저

- 고전적 SGD는 아래와 같이 고정된 학습률을 사용하지만 상황에 맞게 학습률을 조절하는 적응적 학습률이 존재함

$$\text{고전적 SGD: } \mathbf{w} = \mathbf{w} - \rho \frac{\partial J}{\partial \mathbf{w}}$$

- 적응적 학습률을 적용한 옵티마이저에는 Adagrad, RMSprop, Adam 등이 있음
  - Adagrad : 이전 그레디언트를 누적한 정보를 이용하여 학습률을 적응적으로 설정하는 기법
  - RMSprop : 이전 그레디언트를 누적할 때 오래된 것의 영향(rho)을 줄이는 정책을 사용하여 AdaGrad를 개선한 기법
  - Adam : RMSProp에 모멘텀(beta_1)을 적용하여 RMSprop(beta_2를 rho 대신 사용)을 개선한 기법

5.8 딥러닝이 사용하는 옵티마이저

5.8.3 옵티마이저 성능 비교 실험 (Fashion MNIST) 깊은 다층 퍼셉트론은 프로그램[5-10]과 동일

[프로그램5-11] 옵티마이저의 성능 비교: SGD, Adam, Adagrad, RMSprop

```
train_dataloader = torch.utils.data.DataLoader(training_data,batch_size=128,shuffle=True)
test_dataloader = torch.utils.data.DataLoader(test_data,batch_size=128,shuffle=False)

for optimizer_name in ['SGD', 'AdaGrad', 'RMSprop', 'Adam']:
    DNN = DeepNeuralNetwork(input_dim=784, hidden_dim=512, output_dim=10)
    DNN = DNN.cuda()
    if optimizer_name == 'SGD':
        optimizer = torch.optim.SGD(DNN.parameters(), lr=0.001)
    elif optimizer_name == 'AdaGrad':
        optimizer = torch.optim.Adagrad(DNN.parameters(), lr=0.001)
    elif optimizer_name == 'RMSprop':
        optimizer = torch.optim.RMSprop(DNN.parameters(), lr=0.001)
    elif optimizer_name == 'Adam':
        optimizer = torch.optim.Adam(DNN.parameters(), lr=0.001)

    history[optimizer_name]['train_acc'] = list()
    history[optimizer_name]['test_acc'] = list()
    history[optimizer_name]['train_loss'] = list()
    history[optimizer_name]['test_loss'] = list()

    for iter in tqdm.tqdm(range(30)):
        DNN, history[optimizer_name] = train_epoch(train_dataloader,
                                        optimizer, DNN, loss_mse, history[optimizer_name])
    with torch.no_grad():
        DNN, history[optimizer_name] = test_epoch(test_dataloader,
                                         optimizer, DNN, loss_mse, history[optimizer_name])
```

08행 SGD 옵티마이저 사용하기

10행 AdaGrad 옵티마이저 사용하기

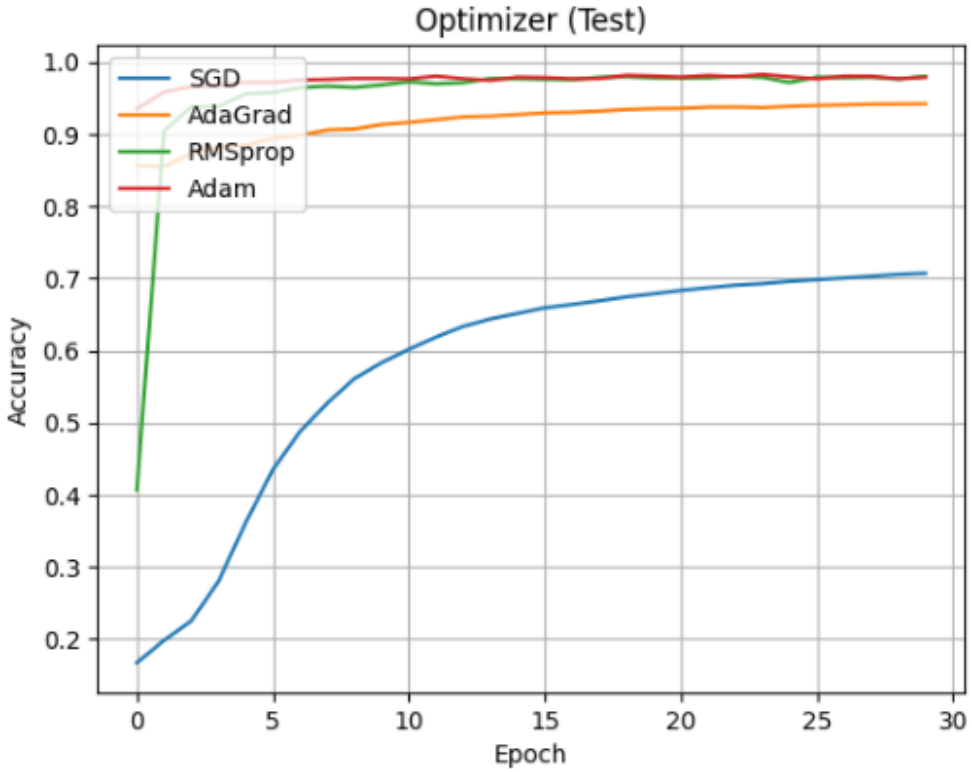
12행 RMSProp 옵티마이저 사용하기

14행 Adam 옵티마이저 사용하기

5.8 딥러닝이 사용하는 옵티마이저

5.8.3 옵티마이저 성능 비교 실험

[프로그램5-11] 옵티마이저의 성능 비교: SGD, Adam, Adagrad, RMSprop



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

- 모듈화 하라
  - 좋은 프로그램을 짜기 위해 가장 먼저 할 일은 함수를 만들어 반복되는 코드를 줄이는 것

- 언어의 좋은 특성을 최대한 활용하라
  - 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

- 점증적으로 코딩하라
  - 한번에 한가지 기능을 추가하고 옳게 작동하는지 확인하는 일을 반복
- 디자인 패턴이 몸에 배게 하라
  - 다른 프로그램과 공유하는 디자인 패턴에 대한 눈썰미
- 도구에 한없이 익숙해져라
  - 라이브러리 사용에 익숙
  - 통합개발환경 사용에 익숙
- 기초에 충실하라
  - 파이썬 기초 문법 익히기
  - 중요한 라이브러리 익히기
  - 기계학습의 기초 이론 등

5.10 교차 검증을 이용한 하이퍼 매개변수 최적화

- 높은 성능을 발휘하는 모델을 찾으려면 우연을 배제할 수 있
- 는 객관적인 성능 평가가 필수임
- 가장 효과적인 방법은 여러 번 반복하고 평균을 취하는 교차 검증임

5.10.1 교차 검증을 이용한 옵티마이저 선택

- 딥러닝 라이브러리 (파이토치, 텐서플로우) 에는 교차 검증을 지원하는 클래스가 없음
- 딥러닝은 주로 대용량 데이터셋으로 수행하므로 교차 검증 없이 측정한 성능도 어느 정도 신뢰할 수 있다는 생각 때문임
- 따라서 필요시 교차검증을 하는 함수를 직접 만들어야 함

5.10 교차 검증을 이용한 하이퍼 매개변수 최적화

5.10.1 교차 검증을 이용한 옵티 마이저 성능 선택 (학습/검증 데이터 속임수 해결을 위한 평가 방법)

[프로그램5-12] K-Fold 교차 검증을 활용한 옵티마이저의 성능 비교

```
118  # Dataset를 slicing하기 위해 Tensor로 변환
119  x_train = training_data.data
120  y_train = training_data.targets
121  x_train = x_train.type(torch.float32)
122
123  kfold_histroy = list()
124
125  for train_index, val_index in KFold(k).split(x_train):
126
127      history = dict()
128      history['SGD'] = dict()
129      history['AdaGrad'] = dict()
130      history['RMSprop'] = dict()
131      history['Adam'] = dict()
132
133      xtrain, xval = x_train[train_index], x_train[val_index]
134      ytrain, yval = y_train[train_index], y_train[val_index]
135
136      train_dataloader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(xtrain,ytrain),batch_size=128,shuffle=True)
137      val_dataloader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(xval,yval),batch_size=128,shuffle=False)
138
139      for optimizer_name in ['SGD', 'AdaGrad', 'RMSprop', 'Adam']:
140          DNN = DeepNeuralNetwork(input_dim=784, hidden_dim=512, output_dim=10)
141          DNN = DNN.cuda()
142          if optimizer_name == 'SGD':
143              optimizer = torch.optim.SGD(DNN.parameters(), lr=0.001)
144          elif optimizer_name == 'AdaGrad':
145              optimizer = torch.optim.Adagrad(DNN.parameters(), lr=0.001)
146          elif optimizer_name == 'RMSprop':
147              optimizer = torch.optim.RMSprop(DNN.parameters(), lr=0.001)
148          elif optimizer_name == 'Adam':
149              optimizer = torch.optim.Adam(DNN.parameters(), lr=0.001)
150
151          history[optimizer_name]['train_acc'] = list()
152          history[optimizer_name]['test_acc'] = list()
153          history[optimizer_name]['train_loss'] = list()
154          history[optimizer_name]['test_loss'] = list()
155
156          for iter in tqdm.tqdm(range(5)):
157              DNN, history[optimizer_name] = train_epoch(train_dataloader,
158                                                         optimizer, DNN, loss_mse, history[optimizer_name])
159              with torch.no_grad():
160                  DNN, history[optimizer_name] = test_epoch(val_dataloader,
161                                                            optimizer, DNN, loss_mse, history[optimizer_name])
162      kfold_histroy.append(history)
```

119-121행 K-Fold slicing을 위해 Tensor로 자료형 변경

125행 K-Fold 검증을 위해 k번 돌아가는 반복문 정의

133-137행 K-Fold 검증을 위해 train/val 데이터 정의

162행 K-Fold 검증을 위해 k별 validation 정확도 기록

5.10 교차 검증을 이용한 하이퍼 매개변수 최적화

5.10.1 교차 검증을 이용한 옵티 마이저 성능 선택

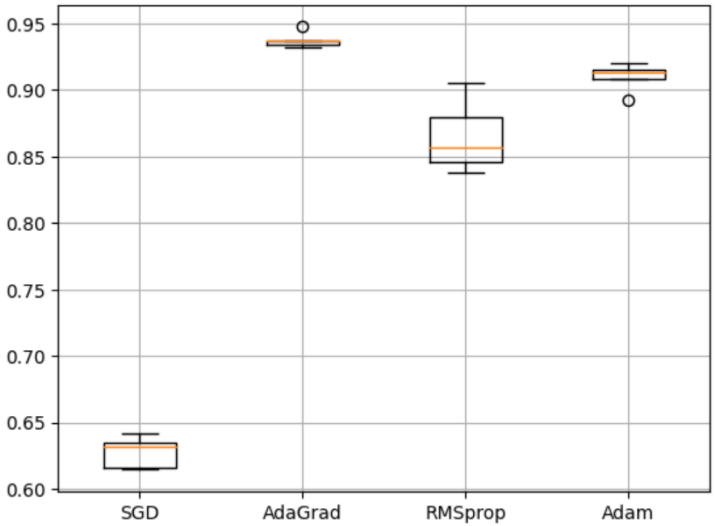
[프로그램5-12] K-Fold 교차 검증을 활용한 옵티마이저의 성능 비교

```
165    test_acc_dict = dict()
166    test_acc_dict['SGD'] = list()
167    test_acc_dict['AdaGrad'] = list()
168    test_acc_dict['RMSprop'] = list()
169    test_acc_dict['Adam'] = list()
170
171    for optimizer_name in ['SGD', 'AdaGrad', 'RMSprop', 'Adam']:
172        for idx in range(len(kfold_histroy)):
173            test_acc_dict[optimizer_name].append(kfold_histroy[idx][optimizer_name]['test_acc'][-1])
174
175
176    # 박스 플롯으로 정확률 표시
177    plt.grid()
178    plt.boxplot([test_acc_dict['SGD'], test_acc_dict['AdaGrad'], test_acc_dict['RMSprop'], test_acc_dict['Adam']], labels=['SGD', 'AdaGrad', 'RMSprop', 'Adam'])
```

165-169행 각각의 K 마다 마지막 epoch에 대한 정확도를 담아줄 list 정의

171-173행 옵티마이저 별 validation 정확도 저장

177행 boxplot을 활용해 K-Fold 교차 검증 시각화



5.10 교차 검증을 이용한 하이퍼 매개변수 최적화

5.10.2 과다한 계산 시간과 해결책

- 교차 검증은 시간이 많이 소요됨, 즉 성능 검증에 대한 신뢰도가 높아지는 대신 시간이 걸림
  - 학습을 한번 마치는데 t라는 시간이 걸린다면, kt 만큼 시간이 지나야 옵티마이저 하나의 성능 측정이 끝남
  - [프로그램5-12]는 옵티마이저 4개를 평가하므로 총 4kt 만큼의 시간이 걸림
  - [프로그램5-12]는 하이퍼 매개변수를 모두 고정시키고 최적화를 수행

- 딥러닝에서 만일 하이퍼 매개변수 최적화를 제대로 하려면 훨씬 많은 시간이 걸림
  - 데이터셋의 크기가 기존 토이 데이터셋보다 훨씬 큼
  - 딥러닝은 하이퍼 매개변수가 아주 많음
  - 예를들어, 옵티마이저{SGD, Adam, Adagrad, RMSprop}, 학습률{0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005, 0.00001}, 미니배치 크기{32, 64, 128,256,512,1024}의 조합을 최적화한다면 총 168개 조합에 대해서 학습을 진행해야 함

- 딥러닝은 GPU를 이용함으로써 과다한 계산 시간 문제를 해결 가능함