4. 신경회로망의 기초

실제 뉴런의 모양 및 구성요소는 우리가 구현하게 될 모델과는 많이 다르고, 우리는 길제 뉴런 모델을 간략화 하여 사용하게 된다 신경망을 구성하는 각 노드를 유닛(unit)이라고 부르고 각 유닛은 서로 연결되어 있다.

뉴런은 뇌를 구성하는 기본단위로, 사람의 뇌는 약 1,000억 개의 뉴런을 가지고 있다. 실제로 좁쌀만 한 크기의 작은 뇌 조각 안에 1만 개 이상의 뉴런이 있으며, 각 뉴런은 다른 뉴런과 평균 6,000개의 연결을 형성한다. 그 연결을 통해, 여러 뉴런으로부터 얻은 정보를 처리하고 그 결과를 다른 세포로 전송하게 된다. 한마디로 뇌는 거대한 생물학적 네트워크라고 볼 수 있으며, 사람은 이 신경망을 통해 감정을 느끼고 생각할 수 있게 된다. 인공 신경 회로망(ANN:Artificial Neural Networks)은 인간의 뇌를 구성하는 신경세포인 뉴런(Neuron)의 동작원리를 기초로 하여 구성되었다.

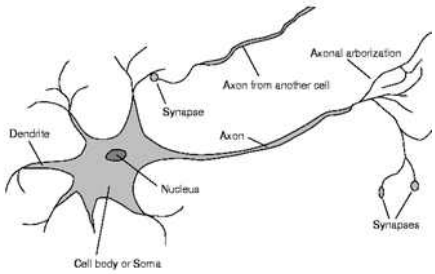


그림 4.1 실제 뉴런 및 정보전달

인공 신경망에 대한 최초의 발상은 1943년, 신경 생리학자 워렌 맥컬로치(Warren McCulloch)와 논리학자 월터 피츠(Walter Pitts)가 함께 제안한 'McCulloch-Pitts 뉴런' 모델이다. 이들은 인간의 두뇌를 이진 원소들의 집합으로 표현했는데, 이 발상을 기초로 하여 1957년, 코넬 항공 연구소(Cornell Aeronautical Laboratory)의 과학자 프랭크 로젠블래트(Frank Rosenblatt)가 '단층 퍼셉트론(SLP, Single-Layer Perceptron)' 모형을 제안한다.

*** 신경회로망의 역사**

- 1943년 : McCulloch & Pitts show that neurons can be combined to construct a **Turing machine** (using ANDs, Ors, & NOTs)
- 1958년 : Rosenblatt shows that perceptrons will converge if what they are trying to learn can be represented
- 1969년 : Minsky & Papert showed the limitations of perceptrons, killing research for a decade
- 1985년 : backpropagation algorithm revitalizes the field
- 2006 년 : Hinton proposed a method that the network which has deep structure can be trained.

가. 퍼셉트론(Perceptron)

단층 퍼셉트론은 다수의 신호(Input)을 받아서 하나의 신호(Output)를 출력한다. 이 때문에 그 동작은 뉴런과 굉장히 유사하며, 다수의 입력을 받았을 때, 퍼셉트론은 각 입력 신호의 세기에 따라 다른 가중치를 부여합니다. 그 결과를 고유한 방식으로 처리한 후, 입력 신호의 합이 일정 값을 초과한다면 그 결과를 다른 뉴런으로 전달한다.

그림 4-2는 입력이 2개인 퍼셉트론의 예를 보여주고 있다. 그림 4-2에서 x1, x2는 입력 신호, y는 출력 신호, w1과 w2는 가중치(weight), 그리고 원을 뉴런 또는 노드라고 한다. 입력 신호가 뉴런에 보내질 때 각각의 가중치가 곱해진다. 뉴런에서 보내온 신호의 총합이 정해진 한계를 넘어설 때에만 1을 출력하고, 이 때 한계를 임계값이라고 하며 θ라고 한다.

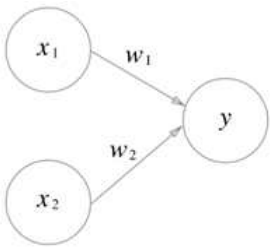


그림 4.2 입력이 2개인 퍼셉트론 모델의 예

$$y = \begin{cases} 0, & (w_1 x_1 + w_2 x_2 \le \theta) \\ 1, & (w_1 x_1 + w_2 x_2 > \theta) \end{cases} \tag{4-1}$$

아래는 θ를 -b(편향 : bias)로 치환한 것임. 즉, 입력 신호 × 가중치 + 편향 > 0 이면 1의 값을 가지게 된다.

$$y = \begin{cases} 0, & (b + w_1 x_1 + w_2 x_2 \le 0) \\ 1, & (b + w_1 x_1 + w_2 x_2 > 0) \end{cases} \tag{4-2}$$

나. 퍼셉트론과 논리회로

퍼셉트론으로 논리회로를 구현할 수 있다. 논리회로의 기본 게이트인 AND, NAND, OR 게이트를 구성하기 위해서는 아래와 같이 적절한 (w_1, w_2, b)를 설정하면 된다.

- AND : if (w_1, w_2, b) = (0.5, 0.5, -0.7)
  x_1 = 1, x_2 = 1일 때에만 $w_1 x_1 + w_2 x_2 + b > 0$를 만족
- NAND : if (w_1, w_2, b) = (-0.5, -0.5, 0.7), 조건식 만족
- OR : if (w_1, w_2, b) = (0.5, 0.5, -0.2), 조건식 만족

AND, NAND, OR 각각의 경우에 따라 달라지는 것은 매개변수의 값뿐이다. 즉, 똑같은 구조에서 매개변수만 바꾸는 것이라고 할 수 있다. 기계학습 문제는 이 매개변수의 값을 정하는 작업을 컴퓨터가 자동으로 하도록 하는 것이다. 기계학습에서 신경망이 적절한 매개변수 값을 정하는 작업을 **학습(training)**이라고 한다. 즉, AND, NAND, OR의 기능이 구현되도록 각각의 논리게이트에 해당하는 매개변수 값을 정해주어야 하는 것을 학습이라고 한다.

w는 입력신호(x)가 결과에 주는 영향력을 조절하는 변수이고, 편향은 뉴런이 얼마나 쉽게 활성화 (결과로 1을 출력) 하느냐를 조정하는 매개변수 이다. 예를 들어 편향의 값이 -10 이면 sum(입력 신호 * 가중치) 값이 10이 넘어야 활성화 된다는 의미이다.

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

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

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

그림 4.3 AND, NAND, OR 게이트 진리표

```
import numpy as np

def AND(x1, x2, w1 = 0.5, w2 = 0.5, b = -0.7):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    return 1 if np.sum(x*w) + b >=0 else 0

def NAND(x1, x2, w1 = -0.5, w2 = -0.5, b = 0.7):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    return 1 if np.sum(x*w) + b >=0 else 0

def OR(x1, x2, w1 = 0.5, w2 = 0.5, b = -0.2):
    x = np.array([x1, x2])
    w = np.array([w1, w2])
    return 1 if np.sum(x*w) + b >=0 else 0

result = AND(0, 0)
result = NAND(0, 0)
result = OR(0, 0)
print(result)
```

다. 단층 퍼셉트론의 한계(XOR 문제)

과거에는 지금과 같은 성능의 컴퓨터는 존재하지 않았기 때문에, 퍼셉트론을 하드웨어를 이용하여 구현하였다. 하드웨어만으로도 당시에 중요하게 생각했던 AND와 OR 문제를 해결할 수 있었고, 그 때문에 퍼셉트론을 제안했던 프랭크 로젠블라트(Frank Rosenblatt) 박사는 1958년, 뉴욕 타임스에 위와 같은 이야기를 게재하게 된다. 하지만, 이러한 퍼셉트론에 기대감에 찬물을 끼얹은 문제가 등장했으니, 그것은 바로 XOR 문제였다.


The New York Times

NEW NAVY DEVICE LEARNS BY DOING

July 8, 1958

"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence... Dr. Frank Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers"

"AND, NAND, OR 게이트 진리표우리가 개발한 인공지능은 스스로 학습해서 걷고 말하며, 보고 쓸 수 있게 될 것이다. 그리고 종국에는 자신의 존재를 인지할 것으로 기대한다."

그림 4.4 Frank Rosenblatt 박사의 뉴욕 타임즈 기사

AND와 OR 문제는 적절한 경계선만 찾으면 입력에 대한 정답을 알 수 있었다. 즉, 그럼 4-5에서 볼 수 있듯이, 출력 값 0 또는 1을 구분할 수 있는 직선을 적절히 찾으면 문제는 해결된다는 것이다. 하지만 XOR 문제는 어떤 경계선을 찾든, 문제를 완벽하게 해결할 수 없었다. 많은 사람이 문제에 도전했지만, 모두 실패로 끝이 나고 말았다. 실제로 1969년, MIT 대학의 인공지능 연구실 마빈 민스키(Marvin Minsky) 교수는 '퍼셉트론즈(Perceptrons)'라는 책을 통해 XOR 문제 해결이 불가능하다는 걸 수학적으로 증명하였고, 단층 퍼셉트론 모델(SLP)의 한계점을 지적하였다.

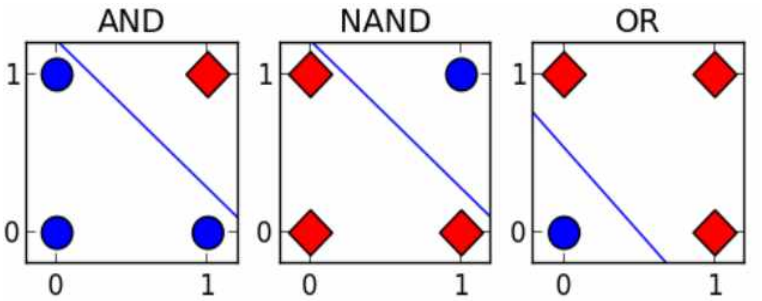


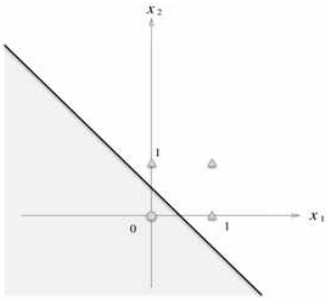
그림 4.5 선형직선으로 분리 가능한 AND, NAND, OR 문제

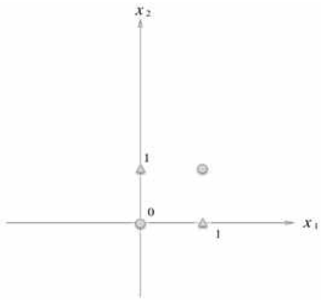
XOR 게이트는 배타적 논리합이라고 알려져 있다. 즉, 입력 2개중 하나만 1일 때 1을 출력한다. XOR 게이트는 AND, NAND, OR 과 같이 단일 퍼셉트론을 이용하여 구현할 수 없

다. 예를 들어 (w_1, w_2, b) = (1, 1, 0.5) 인 경우, 식 4-3과 같이 표현되며, 하나의 직선을 이용하여 구분할 수 없게 된다. 즉, 그림 4.6에서 볼 수 있듯이 XOR는 하나의 직선으로 영역을 둘로 나눌 수가 없다. 이 문제를 해결하기 위해서는, XOR의 경우 그림 4-7과 같이 비선형으로 영역을 나누어야 문제가 해결된다.

$$y = \begin{cases} 0, & (-0.5 + x_1 + x_2 \le 0) \\ 1, & (-0.5 + x_1 + x_2 > 0) \end{cases} \tag{4-3}$$

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





(a) XOR 진리표 (b) OR (c) XOR

그림 4.6 선형직선으로 분리가 불가능한 XOR

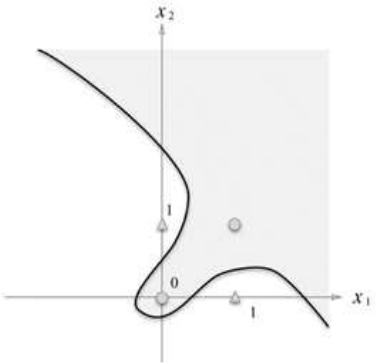


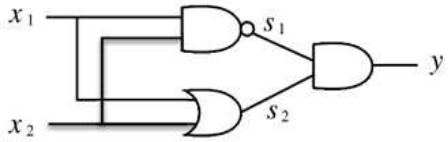
그림 4.7 XOR 문제를 선형이 아닌 비선형으로 영역을 나눈 경우의 예

라. XOR 문제 해결을 위한 다층 퍼셉트론(Multi-layer Perceptron)

XOR 게이트를 만드는 방법은 다양하다. 그 다양한 방법들 중에서 앞서 만든 AND, OR, NAND 게이트를 조합하여 만들어보자. 그림 4-8은 AND, OR, NAND 게이트를 조합하여 만든 XOR와 그에 대응하는 진리표를 보여주고 있다. 그림 4-8은 2층의 구조를 가지고 있다. 기존의 1층의 구조가 아닌, 게이트들의 조합으로 1층을 만들고, 그 출력을 다음 층에서 받아서 처리하는 구조이며, 이렇게 2개이상의 다층으로 이루어진 페셉트론을 다층 퍼셉트론(Multi-layer Perceptron)이라 한다. 그림 4-9는 이렇게 2개의 층으로 XOR를 두현한 다층 퍼셉트론의 예를 보여주고 있다. 퍼셉트론 모델에서 각층을 Layer라 하며, 일반적으로 제일 밑의 입력층을 Layer 0이라 하고 차례로 숫자가 증가하게 된다.

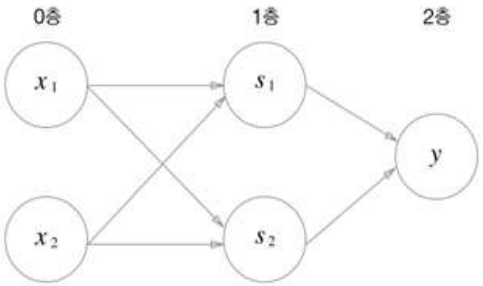
0층의 두 뉴런이 입력 신호를 받아 1층의 뉴런으로 신호를 보낸다. 1층의 뉴런이 2층의

뉴런으로 신호를 보내고, 2층의 뉴런은 이 신호를 바탕으로 y를 출력하게 된다. 즉, 다층 퍼셉트론은 단층 퍼셉트론으로는 구현하지 못하는 것을 층을 하나 늘리는 것으로 해결



x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

그림 4.8 AND, OR, NAND 게이트를 조합하여 만든 XOR




```
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    return AND(s1, s2)
```

그림 4.9 XOR를 구현하기 위한 다층 페셉트론 구조

마. 퍼셉트론에서 신경망으로

단순한 3층(3 layer)의 구조를 가진 신경망은 그림 4-10과 같이 표현할 수 있다. 차례대로 입력층(input layer), 은닉층(hidden layer) 그리고 출력층(output layer)이라고 한다. 또는 가중치를 가지는 은닉층과 출력층만을 고려해 2층(2 layer)이라고도 한다.

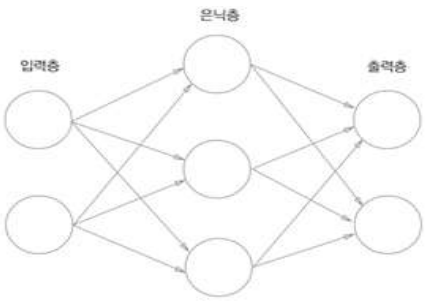


그림 4.10 3층 신경망의 예

단층 퍼셉트론의 구조에서는 입력 x_1과 x_2를 입력받아 가중치 w_1과 w_2를 각각 곱하고 그 결과를 더하여 y를 출력하는 구조이다. 이때 b는 편향(bias)을 나타내는 매개변수로, 뉴런이 얼마나 쉽게 활성화되느냐를 제어한다. 일반적으로 신경망에서는 그림 4-11과 같이 편향 b를 신경망 내에 포함하여 표현한다. 또한, 0을 넘으면 1을 출력하고, 그렇지 않으면 0을 출력하

는 동작을 새로운 함수 $h(x)$를 사용하여 표현한다. $h(x)$는 입력신호의 총 합을 출력신호로 변화하는 역할을 수행하는 함수로서 일반적으로 활성화 함수(activation function)라고 한다. 즉, 활성화 함수는 입력신호의 총 합이 활성화를 일으키는지를 정하는 역할을 한다. 즉, 가중치가 곱해진 입력 신호와 bias의 총합을 계산하고 이를 a라고 하면, a를 함수 h에 넣어 y를 출력하게 된다.

$$y = h(b + w_1 x_1 + w_2 x_2), \text{ where } \quad h(x) = \begin{cases} 0, (x \le 0) \\ 1, (x > 0) \end{cases} \tag{4-4}$$

$$y = h(a), \quad \text{where, } a = b + w_1 x_1 + w_2 x_2 \tag{4-5}$$

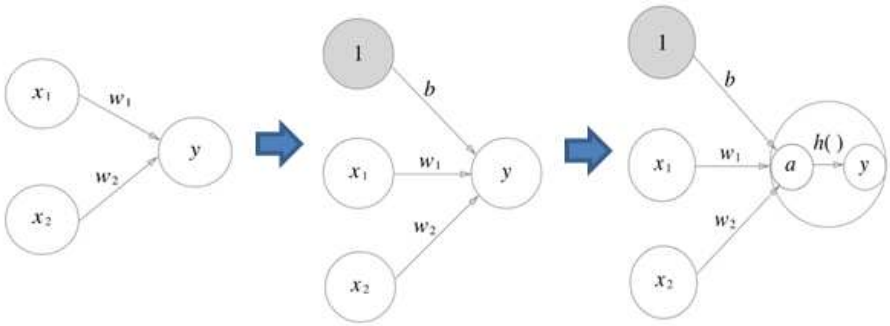
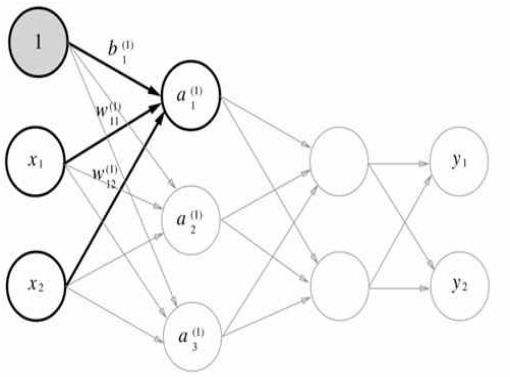


그림 4.11 바이어스와 활성화 함수를 포함한 퍼셉트론

입력층에서 첫 번째 은닉층인 1층으로 신호를 전달하는 과정은 그림 4-12와 같다. 그리고 이를 수식으로 표현하면 식(4-6)과 같다. 3층 신경망의 전체 동작과정은 그림 4.13과 같다.



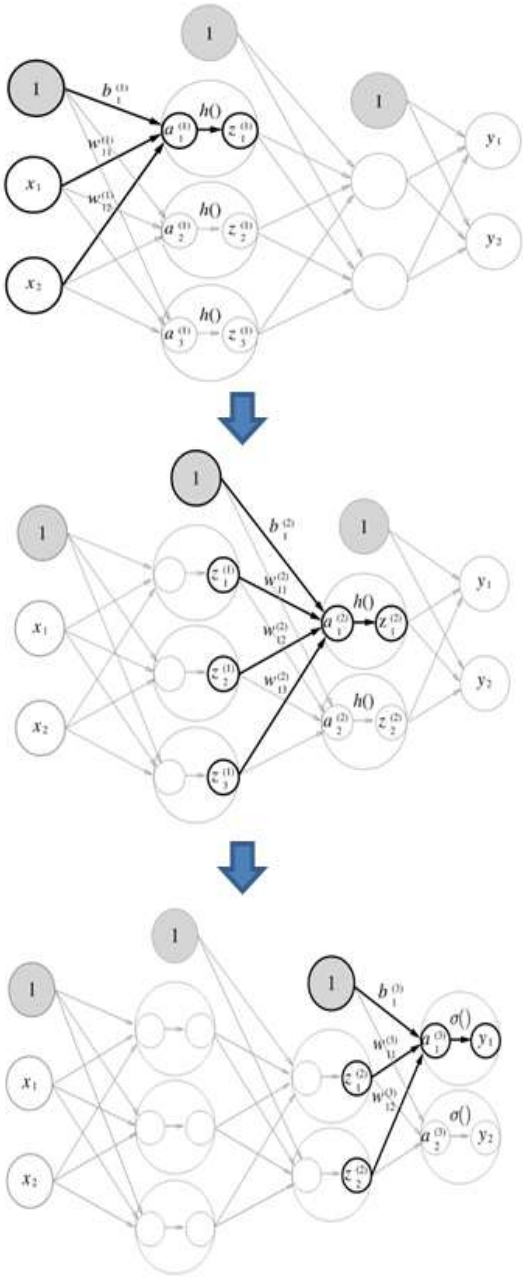

```
import numpy as np

x = np.array([1.0, 0.5]) # 1 x 2 벡터
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
# 2 x 3 매트릭스

B1 = np.array([0.1, 0.2, 0.3]) # 1 x 3 벡터

# np.dot(x, W1) → (1 x 2) · (2 x 3) => 1 x 3
A1 = np.dot(x, W1) + B1
print(A1)
```

그림 4.12 입력층에서 1층(첫 번째 은닉층)으로 신호 전달

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)} \quad A^{(1)} = XW^{(1)} + B^{(1)} \tag{4-6}$$

$$A^{(1)} = (a_1^{(1)} a_2^{(1)} a_3^{(1)}), X = (x_1 x_2), B^{(1)} = (b_1^{(1)} b_2^{(1)} b_3^{(1)})$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$



```
import numpy as np

x = np.array([1.0, 0.5]) # 1 x 2 벡터
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
# 2 x 3 매트릭스

B1 = np.array([0.1, 0.2, 0.3]) # 1 x 3 벡터

# np.dot(x, W1) → (1 x 2) · (2 x 3) => 1 x 3
A1 = np.dot(x, W1) + B1
Z1 = sigmoid(A1)
```

```
W2 = np.array([ [0.1, 0.4], [0.2, 0.5], [0.3, 0.6] ])
B2 = np.array([0.1, 0.2])
A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

```
W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])
A3 = np.dot(Z2, W3) + B3
Y = A3
```

그림 4.13 3층 신경망의 신호 전달

바. 활성화 함수(Activation Function)

식 (4-4)에서의 활성화 함수는 0보다 크면 1이고, 작으면 0인 계단함수(step function)라고 생각할 수 있다. 신경망을 구현하기 위한 활성화 함수에는 다양한 종류들이 있으며, 이 활성화 함수로 인해 신경망 기술이 정체되기도 하였으며, 비약적으로 발전하는 계기가 되기도 하였다. 아래 식(4-7)은 계단함수를, (4-8)은 신경망에서 실제 많이 사용되어온 시그모이드(sigmoid) 활성화 함수이다.

$$\text{Step Function} : \quad h(x) = \begin{cases} 0, (x \le 0) \\ 1, (x > 0) \end{cases} \tag{4-7}$$

$$\text{Sigmoid Function} : \ h(x) = \frac{1}{1+e^{-x}} \tag{4-8}$$

sigmoid 함수는 부드러운 곡선으로 표현되며, 계단함수는 0을 경계로 출력이 갑자기 바뀌게 된다. sigmoid 함수는 신경망 학습에서 아주 중요한 역할을 한다. 계단 함수가 0과 1중 하나의 값만 반환하는 반면 sigmoid 함수는 연속적인 값들을 반환하는 것을 알 수 있다. 둘 다 입력이 작을 때에는 출력이 0에 가까워지고, 입력이 커지면 출력이 1에 가까워 진다. 입력이 아무리 작거나 커도 출력의 범위는 [0, 1]이고, 둘 다 비선형 함수이다.

활성화 함수의 목적은 네트워크에 비선형성을 주기 위해서 존재하며, 비선형적으로 변화하는 변수를 모델링 할 수 있다. 또 다르게 말해보자면, 신경망에 비선형 활성화 함수가 없으면 아무리 많은 층이 있어도, 그 많은 선형 층들의 합은 또 다른 단 하나의 선형 함수와 같은 역할을 한다. 예를 들면 $f(x) = 3x$라는 선형함수를 3개 쌓았다고 할 때, 이 층을 통과한 결과는 그냥 $f(x) = 3 \times 3 \times 3x$ 선형함수 하나를 가진 단층 퍼셉트론을 통과시킨 것과 같은 결과를 낼 것이다. 즉, 층를 깊게 쌓더라도 활성화함수가 선형함수로만 구성되어 있다면 그 모델은 그저 단 하나의 Perceptron, 단층 퍼셉트론과 같은 결과를 낸다.

또한, 활성화 함수는 미분이 가능한 함수를 사용한다. Sigmoid는 미분값이 0이 아니고 연속적으로 변한다. 계단함수는 0점을 제외한 미분값이 0이 된다. 매개변수의 작은 변화가 주는 효과를 계단함수가 사라지게 한다(미분하면 0이 되므로). 이러한 이유로 실제 구현에서는 계단함수를 사용하지 않고 Sigmoid나 미분 가능한 활성화 함수를 사용하게 된다.

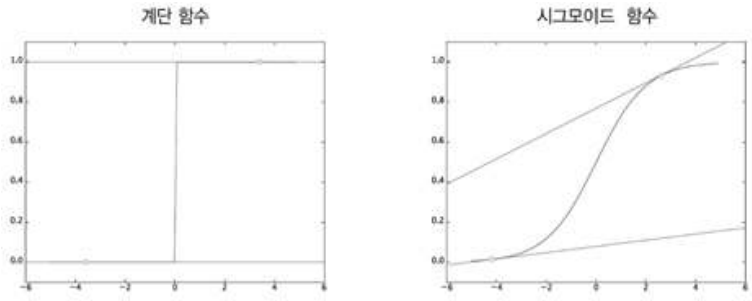
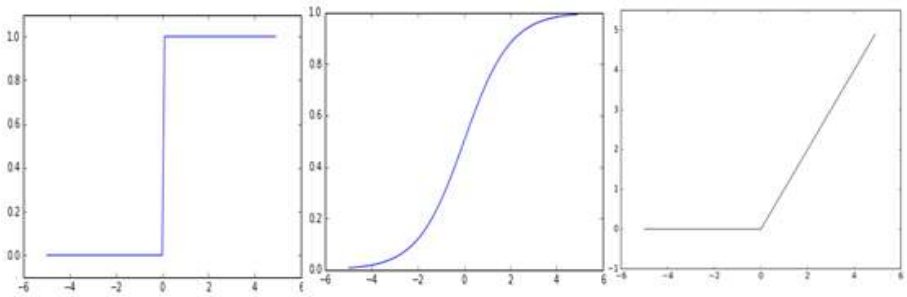


그림 4.14 활성화 함수의 미분가능 여부

sigmois 함수는 가장 널리 사용되어진 활성화 함수였다. 하지만, 딥러닝 기술이 보편화된 지금은 sigmoid 함수 대신에 ReLU 함수를 활성화 함수로 많이 사용한다. ReLU 함수는 입력이 0을 넘으면 그 입력 값 그대로 출력하고, 0 이하이면 0을 출력하는 함수이다.

$$\text{ReLU Function} : h(x) = \begin{cases} x\,, (x > 0) \\ 0\,, (x \leq 0) \end{cases} \tag{4-9}$$



(a) 계단 함수 (b)Sigmoid 함수 (c) ReLU 함수

그림 4.15 신경망에서 사용되는 활성화 함수

```
import numpy as np
import matplotlib.pyplot as plt

def step_function(x):
    return np.array(x > 0, dtype = np.int)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def ReLU(x):
    return np.maximum(0, x)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y), plt.ylim(-0.1, 1.1), plt.show()
```

신경망은 가중치와 활성화 함수의 ㅇ녀결로 이루어진 간단한 구조이다. 이러한 간단한 뉴런을 충분히 많이 연결해놓아서 복잡한 패턴을 스스로 학습 할 수 있게 하는 것이 신경망이다. 하지만 층이 깊어지고 넓어지게 되면 가중치 w와 바이어스 b의 조합이 기하급수적으로 늘어나게 되며, 학습에 매우 오랜 시간이 걸리게 된다. 3 또는 4층의 경우 오류역전파(backpropagation)이라는 학습방법으로 학습이 가능하다. 하지만, 네트워크 구조가 더 깊은 경우에는 이 방법으로는 학습이 불가능하게 된다. 이러한 어려움을 해결한 것이 딥러닝이다. 제프리 힌튼 교수는 자신이 제안한 제한된 볼쯔만 머신(RBM:Restricted Boltzman Machine)을 이용하여 깊은 신경망을 효율적으로 학습 할 수 있음을 증명하였다. 이후, 드롭아웃(drop-out), ReLU 함수 등이 개발됨으로서, 딥러닝은 급속한 발전을 이루게 된다. 그리고, 빅데이터와 GPU 성능의 발전으로 본격적인 딥러닝의 시대가 도래 하게 되었다.

5. Tensorflow를 이용한 기본 신경망 구현

가. 단층 신경망을 이용한 분류

4가지 종류의 클래스를 구분하는 간단한 분류기를 텐서플로우를 이용한 신경망을 통해 구현해보자. 키와 몸무게를 이용하여, 저체중, 정상체중, 경도비만, 비만의 단계를 추정하고자 한다. 일반적으로 BMI 지수를 이용하여 비만 단계를 측정한다. 18.5미만 저체중, 18.5-22.9 정상체중, 23-29.9 경도비만, 30이상 비만으로 본다. 입력으로 10명의 데이터를 선정하였다.

$$BMI = \frac{\text{몸무게}}{(0.1 \times \text{키})^2}$$

순번	키	체중	BMI	순번	키	체중	BMI
1	160	47	18.36(1)	6	172	60	20.28(2)
2	165	45	16.53(1)	7	165	65	23.88(3)
3	163	60	22.6(2)	8	175	80	26.12(3)
4	157	61	24.75(3)	9	180	70	21.6(2)
5	155	65	27.06(3)	10	178	120	37.87(4)

각 데이터에 해당하는 사람이 어떤 비만단계에 속하는지 나타내는 레이블(분류값)을 구성한다. 레이블 데이터는 원-핫 인코딩(One-hot Encoding)이라는 특별한 현태로 구성한다. 즉, 데이터가 가질 수 있는 값들을 일렬로 나열한 배열을 만들고, 해당되는 인덱스 값만 1로 하고, 나머지는 0으로 채운다.

- class 1(저체중) : [1, 0, 0, 0]
- class 2(정상체중) : [0, 1, 0, 0]
- class 3(경도비만) : [0, 0, 1, 0]
- class 4(비만) : [0, 0, 0, 1]

1) 학습에 필요한 데이터 정의

```
#[키, 몸무게]
x_data = np.array([[160, 47], [165, 45], [163, 60], [157, 61], [155, 65],
                   [172, 60], [165, 65], [175, 80], [180, 70], [178, 120]])
y_data = np.array([ [1, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0],
                    [0, 0, 1, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0],
                    [0, 1, 0, 0], [0, 0, 0, 1] ])
```

2) 신경망 모델 구성

```
#특징 X와 레이블 Y 설정 : X와 Y에 실제 데이터를 넣어서 학습 시킬 것이므로,
#X와 Y는 placeholder로 설정한다.
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)
```

3) 신경망을 결정하는 가중치와 바이어스 설정

```
# 신경망은 2차원으로 [입력층(특성), 출력층(레이블)] -> [2, 4] 으로 정합니다.
W = tf.Variable(tf.random_uniform([2, 4], -1., 1.))
# 편향을 각각 각 레이어의 아웃풋 갯수로 설정합니다.
# 편향은 아웃풋의 갯수, 즉 최종 결과값의 분류 갯수인 4로 설정합니다.
b = tf.Variable(tf.zeros([4]))
또는, b = tf.Variable(tf.random_uniform([4], -1., 1.))
```

4) 가중치와 바이어스를 적용하고 활성화함수를 통과 시킨다.

```
# 신경망에 가중치 W과 편향 b을 적용합니다
L = tf.add(tf.matmul(X, W), b)
# 계산한 결과 값에 활성화 함수인(sigmoid or ReLU)함수를 적용
L = tf.nn.relu(L)
또는 L = tf.nn.sigmois(L)
```

5) 출력값은 softmax 함수를 이용하여 정규화

```
# 마지막으로 softmax 함수를 이용하여 출력값을 사용하기 쉽게 만듭니다
# softmax 함수는 다음처럼 결과값을 전체합이 1인 확률로 만들어주는 함수입니다.
# 예) [8.04, 2.76, -6.52] -> [0.53 0.24 0.23]
model = tf.nn.softmax(L)
```

6) 신경망을 최적화 하기 위한 비용함수(Cost function) 설정

```
# 각 개별 결과에 대한 합을 구한 뒤 평균을 내는 방식을 사용합니다.
# 전체 합이 아닌, 개별 결과를 구한 뒤 평균을 내기 위해 axis 옵션을 사용
# axis 옵션이 없으면 -1.09 처럼 총합인 스칼라값으로 출력됩니다.
#          Y          model          Y * tf.log(model)    reduce_sum(axis=1)
# 예) [[1 0 0]  [[0.1 0.7 0.2]  ->   [[-1.0   0     0]  -> [-1.0, -0.09]
#      [0 1 0]]  [0.2 0.8 0.0]]       [ 0    -0.09 0]]
# 예측값과 실제값 사이의 확률 분포의 차이를 비용으로 계산 : Cross-Entropy
cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(model), axis=1))
```

7) 학습

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
train_op = optimizer.minimize(cost)
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
for step in range(100):
    sess.run(train_op, feed_dict={X: x_data, Y: y_data})
    if (step + 1) % 10 == 0:
        print(step + 1, sess.run(cost, feed_dict={X: x_data, Y: y_data}))
```

8) 학습결과 확인

```
#########
# 결과 확인
# 0: 저체중 1: 정상체중, 2: 경도비만, 3: 비만
# tf.argmax: 예측값과 실제값의 행렬에서 가장 큰 값을 가져옵니다.
# 예) [[0 1 0 0] [1 0 0 0]] -> [1 0]
#     [[0.2 0.6 0.1 0.1] [0.8 0.1 0.1 0.0]] -> [1 0]
prediction = tf.argmax(model, 1)
target = tf.argmax(Y, 1)
print('예측값:', sess.run(prediction, feed_dict={X: x_data}))
print('실제값:', sess.run(target, feed_dict={Y: y_data}))
```

9) 정확도 측정

```
is_correct = tf.equal(prediction, target)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도: %.2f' % sess.run(accuracy * 100, feed_dict={X: x_data, Y:
                                    y_data}))
```

위 예제를 실행시키면 결과가 다소 실망스럽게 나타날 것이다. 그 이유를 각자 생각해보고 의견을 교환해보자. 하지만 결과와 상관없이, 위의 구조에서 우리가 알 수 있는 사실은 다음과 같다. 우선 학습시키는데 필요한 데이터(x_data)와 정답에 해당하는 레이블(y_data)이 필요하다. 학습이 완료된 이후에는 학습된 모델을 대상으로 실제 적용시킬 테스트 데이터가 필요하게 된다(위의 예에서는 학습데이터(x_data)로 테스트를 진행하였다.). 이러한 작업을 데이터 기반 학습 알고리듬이라 하며, 지도학습(supervised learning) 이라고 한다.

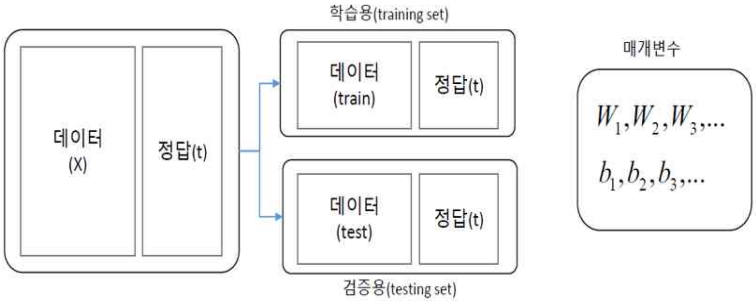
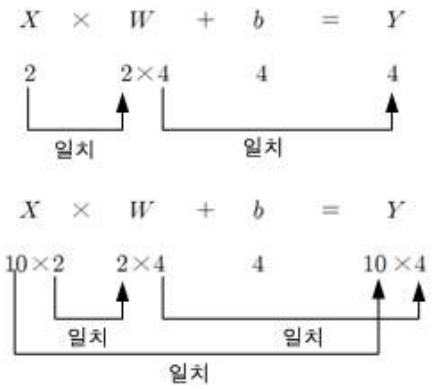
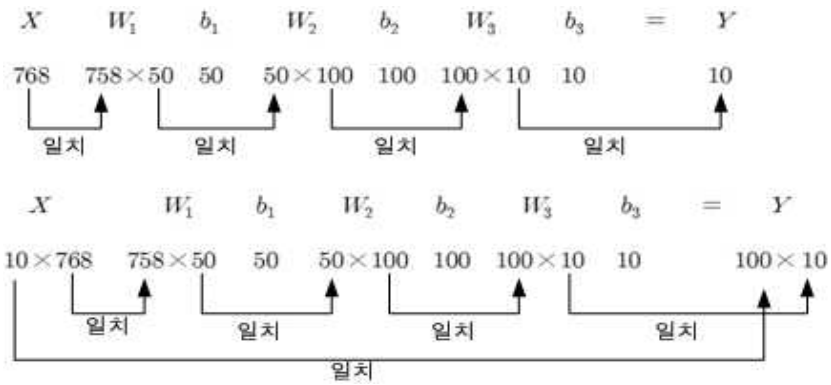


그림 5.1 지도학습(supervised learning) : 데이터기반 학습

또한, 가중치와 바이어스, 레이어 그리고 노드 수를 안다면, 각각 레이어의 입력과 출력에 따라 신경망을 설계 할 수 있다는 것이다. 아래 그림 5.2와 같이 가중치와 바이어스, 레이어 그리고 노드 수가 정해지면 행렬의 곱 규칙에 따라 쉽게 신경망을 구현 할 수 있다. 아래 그림 5-2에서는 레이어, 노드 수에 따른 규칙을 보여주고 있다.



(a) hidden layer 0개, 2-4-4 구조



(b) hidden layer 2개, 768-50-100-10 구조

그림 5.2 가중치와 바이어스, 레이어 그리고 노드 수에 따른 신경망 구성

< Example 5-1 > 4개의 클래스를 분류하는 단층 신경망

```
import tensorflow as tf
import numpy as np

x_data = np.array([[160, 47], [165, 45], [163, 60], [157, 61], [155, 65],
                   [172, 60], [165, 65], [175, 80], [180, 70], [178, 120]]) # [키, 몸무게]
# [저체중, 정상체중, 경도비만, 비만] : one-hot 형식의 데이터
y_data = np.array([
    [1, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0],
    [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 1] ])

X = tf.placeholder(tf.float32) # 신경망 모델 구성
Y = tf.placeholder(tf.float32)

# 신경망은 2차원으로 [입력층(특성), 출력층(레이블)] -> [2, 4] 으로 정합니다.
W = tf.Variable(tf.random_uniform([2, 4], -1., 1.))

# 편향을 각각 각 레이어의 아웃풋 갯수로 설정합니다.
# 편향은 아웃풋의 갯수, 즉 최종 결과값의 분류 갯수인 4로 설정합니다.
b = tf.Variable(tf.zeros([4]))

# 신경망에 가중치 W과 편향 b을 적용합니다
L = tf.add(tf.matmul(X, W), b)
L = tf.nn.relu(L)
# 마지막으로 softmax 함수를 이용하여 출력값을 사용하기 쉽게 만듭니다
model = tf.nn.softmax(L)

# 신경망을 최적화하기 위한 비용 함수
cost = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(model), axis=1))

optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)
train_op = optimizer.minimize(cost)

#신경망 모델 학습
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

for step in range(100):
    sess.run(train_op, feed_dict={X: x_data, Y: y_data})

    if (step + 1) % 100 == 0:
        print(step + 1, sess.run(cost, feed_dict={X: x_data, Y: y_data}))

## 결과 확인
prediction = tf.argmax(model, 1)
target = tf.argmax(Y, 1)
print('예측값:', sess.run(prediction, feed_dict={X: x_data}))
print('실제값:', sess.run(target, feed_dict={Y: y_data}))
is_correct = tf.equal(prediction, target)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도: %.2f' % sess.run(accuracy * 100, feed_dict={X: x_data, Y: y_data}))
```

< Example 5-2 > 4개의 클래스를 분류하는 다층 신경망 : 은닉층 1개

```
import tensorflow as tf
import numpy as np

x_data = np.array([[160, 47], [165, 45], [163, 60], [157, 61], [155, 65],
                   [172, 60], [165, 65], [175, 80], [180, 70], [178, 120]])
y_data = np.array([
    [1, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0],
    [0, 0, 1, 0], [0, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 1] ])

#신경망 모델 구성
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

# 첫번째 가중치의 차원 -> [2, 10], 두번째 가중치의 차원 -> [10, 4]
W1 = tf.Variable(tf.random_uniform([2, 10], -1., 1.))
W2 = tf.Variable(tf.random_uniform([10, 4], -1., 1.))

# 편향을 각각 각 레이어의 아웃풋 갯수로 설정합니다.
b1 = tf.Variable(tf.zeros([10]))
b2 = tf.Variable(tf.zeros([4]))

# 신경망의 히든 레이어에 가중치 W1과 편향 b1을 적용합니다
L1 = tf.add(tf.matmul(X, W1), b1)
L1 = tf.nn.relu(L1)

model = tf.add(tf.matmul(L1, W2), b2)  # 최종적인 아웃풋을 계산합니다.

# 텐서플로우에서 기본적으로 제공되는 크로스 엔트로피 함수를 이용
cost = tf.reduce_mean(
           tf.nn.softmax_cross_entropy_with_logits(labels=Y, logits=model))

optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
train_op = optimizer.minimize(cost)

#신경망 모델 학습
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

for step in range(1000):
    sess.run(train_op, feed_dict={X: x_data, Y: y_data})
    if (step + 1) % 10 == 0:
        print(step + 1, sess.run(cost, feed_dict={X: x_data, Y: y_data}))

#결과 확인
prediction = tf.argmax(model, 1)
target = tf.argmax(Y, 1)
print('예측값:', sess.run(prediction, feed_dict={X: x_data}))
print('실제값:', sess.run(target, feed_dict={Y: y_data}))

is_correct = tf.equal(prediction, target)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도: %.2f' % sess.run(accuracy * 100, feed_dict={X: x_data, Y: y_data}))
```

6. 모델 재사용과 텐서보드 사용법

가. 외부에 저장된 입력데이터(CSV) 파일 불러와서 사용하기

코드안에 입력데이터를 직접 입력하여 사용하는 방식은 데이터의 양이 적을 때는 상관이 없으나, 데이터의 양이 많을 경우에는 비효율적이다. 이럴 경우에는 외부에 입력데아터를 별도로 저장하고 코드에서 불러와서 사용한다. 이번 장에서는 텍스트로 기록된 입력 데이터를 CSV 파일로 저장하고, 이를 이를 읽어 사용하는 방법을 알아본다. 먼저, 앞에서 사용한 BMI 데이터를 BMI_data.csv 파일로 저장한다.

#키	몸무게	1	2	3	4
160	47	1	0	0	0
165	45	1	0	0	0
163	60	0	1	0	0
157	61	0	0	1	0
155	65	0	0	1	0
172	60	0	1	0	0
165	65	0	0	1	0
175	80	0	0	1	0
180	70	0	1	0	0
178	120	0	0	0	1

1) csv 데이터를 읽어들이고 변환

```
data = np.loadtxt('./data.csv', delimiter=',', unpack=True, dtype='float32')
# 키, 몸무게, 1, 2, 3, 4
x_data = np.transpose(data[0:2]) # x_data = [0:2] : 1열과 2열
y_data = np.transpose(data[2:])  # y_data = [2:]  : 3열부터 마지막
```

<원시 데이터>

160	47	1	0	0	0
165	45	1	0	0	0
163	60	0	1	0	0
157	61	0	0	1	0
155	65	0	0	1	0
172	60	0	1	0	0
165	65	0	0	1	0
175	80	0	0	1	0
180	70	0	1	0	0
178	120	0	0	0	1

< data[0:2] >

160	165	163	157	155	172	165	175	180	178
47	45	60	61	65	60	65	80	70	120

->

< unpack=True >

160	165	163	157	155	172	165	175	180	178
47	45	60	61	65	60	65	80	70	120
1	1	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0
0	0	0	1	1	0	1	1	0	0
0	0	0	0	0	0	0	0	0	1

x_data = np.transpose(data[0:2])

->

160	47
165	45
163	60
157	61
155	65
172	60
165	65
175	80
180	70
178	120

그림 6.1 데이터의 행과 열을 변환하기 위한 np.loadtxt(unpack)과 np.transpose 바이어스,

나. 학습모델 저장, 재사용 및 갱신

텐서플로우(TensorFlow)에서 tf.train.Saver API를 이용해서 모델과 파라미터를 저장(save)하고 불러오기(restore)오는 법을 살펴보자. tf.train.Saver API는 텐서플로우에서 모델과 파라미터를 저장하고(save)하고 불러올수(restore) 있게 만들어주는 API이다. 자세한 내용은 아래 링크의 텐서플로우 공식홈페이지의 API 문서를 참고하자.

https://www.tensorflow.org/api_docs/python/tf/train/Saver

1) 저장 관련 함수
- tf.train.Saver
- tf.train.Saver.save
- tf.train.Saver.restore

2) 세션 열고, 모델 불러들이고 저장

```
sess = tf.Session()
saver = tf.train.Saver(tf.global_variables)
```

: tf.train.Saver : 저장과 관련된 함수
: tf.global_variables : 앞서 정의한 변수들을 가져오는 함수

3) 1. 기존에 학습한 모델이 있는지 확인

```
ckpt = tf.train.get_checkpoint_state('./model')
if ckpt and tf.train.checkpoint_exists(ckpt.model_checkpoint_path):
    saver.restore(sess, ckpt.model_checkpoint_path)
else:
    sess.run(tf.global_variables_initializer())
```

: tf.train.Saver.restore : 학습된 값을 불러 온다
: 학습된 모델을 저장한 파일을 체크포인트 파일이라고 한다.

4) 최적화 후, 체크 포인트 파일에 저장하기

```
saver.save(sess,'./model/Model_Reuse_tfSaver.ckpt', global_step=global_step)
```

: tf.train.Saver.save
: './model/Model_Reuse_tfSaver.ckpt' : 체크 포인트 파일의 위치와 이름
: 상위 디렉토리 './model' 은 미리 생성해야 한다.

< Example 6-1 > 모델 저장 및 재사용

```
import tensorflow as tf
import numpy as np

data = np.loadtxt('./BMI_data.csv', delimiter=',', unpack=True, dtype='float32')

x_data = np.transpose(data[0:2])
y_data = np.transpose(data[2:])

#학습에 사용되지는 않고, 학습횟수를 카운트하기위해 사용: 'trainable=False'에 주의
global_step = tf.Variable(0, trainable=False, name='global_step')

X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

W1 = tf.Variable(tf.random_uniform([2, 10], -1., 1.))
L1 = tf.nn.relu(tf.matmul(X, W1))

W2 = tf.Variable(tf.random_uniform([10, 20], -1., 1.))
L2 = tf.nn.relu(tf.matmul(L1, W2))

W3 = tf.Variable(tf.random_uniform([20, 4], -1., 1.))
model = tf.matmul(L2, W3)

cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y, logits=model))

optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
#global_step=global_step : 최적화 함수가 최적화 될때마다 1씩 증가
train_op = optimizer.minimize(cost, global_step=global_step)

sess = tf.Session()
saver = tf.train.Saver(tf.global_variables())

ckpt = tf.train.get_checkpoint_state('./model')

if ckpt and tf.train.checkpoint_exists(ckpt.model_checkpoint_path):
    saver.restore(sess, ckpt.model_checkpoint_path)
else:
    sess.run(tf.global_variables_initializer())

# 최적화 진행
for step in range(10000):
    sess.run(train_op, feed_dict={X: x_data, Y: y_data})

    print('Step: %d, ' % sess.run(global_step),
          'Cost: %.3f' % sess.run(cost, feed_dict={X: x_data, Y: y_data}))

# 최적화가 끝난 뒤, 변수를 저장합니다.
saver.save(sess, './model/Model_Reuse_tfSaver.ckpt', global_step=global_step)
```

< Example 6-1 > 모델 저장 및 재사용 (계속)

```
# 결과 확인
prediction = tf.argmax(model, 1)
target = tf.argmax(Y, 1)
print('예측값:', sess.run(prediction, feed_dict={X: x_data}))
print('실제값:', sess.run(target, feed_dict={Y: y_data}))

is_correct = tf.equal(prediction, target)
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도: %.2f' % sess.run(accuracy * 100, feed_dict={X: x_data, Y: y_data}))

# 다른 data로 test
test_data = np.array([[170, 100]])
print('예측값:', sess.run(prediction, feed_dict={X:test_data}))
```

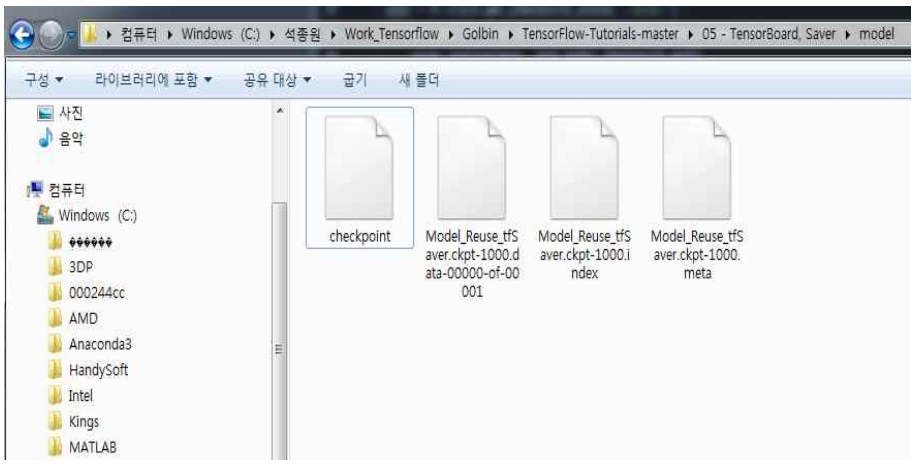


그림 6.2 model 디렉토리에 만들어진 재사용 관련 파일들

다. 텐서보드 사용하기

텐서보드란 학습하는 중간중간 손실값, 정확도, 결과물로 나온 이미지 등의 파일을 시각화하여 보여주는 텐서플로우의 도구이다. 기존의 신경망 코드에 간단하게 몇 줄의 코드만 추가하면 사용할 수 있다. 많이 사용하는 패턴은 다음 2가지이다. **1. tf.summary.scalar API를 이용한 스텝마다 손실 함수 출력, 2. 계산 그래프 시각화**

아래 코드를 실행하면 tensorboard_log라는 폴더가 생성되고, 그 안에 요약 정보들이 파일 형태로 저장된 모습을 볼 수 있다. 코드를 실행한 디렉토리에서 터미널을 띄우고, 다음 명령어를 입력하면 텐서보드를 실행한다.

```
tensorboard --logdir=./tensorboard_log
```

위 명령어를 실행하고 웹브라우저에 접속하여 아래의 URL 주소를 입력하면 텐서보드가 실행된 화면을 확인할 수 있다.

```
http://localhost:6006
```

< Example 6-2 > 텐서보드 사용

```
import tensorflow as tf

# 선형회귀 모델(Wx + b)을 정의합니다.
W = tf.Variable(tf.random_normal(shape=[1]), name="W")
b = tf.Variable(tf.random_normal(shape=[1]), name="b")
x = tf.placeholder(tf.float32, name="x")
linear_model = W*x + b

# True Value를 입력받기위한 플레이스홀더를 정의합니다.
y = tf.placeholder(tf.float32, name="y")

# 손실 함수를 정의합니다.
loss = tf.reduce_mean(tf.square(linear_model - y)) # MSE 손실함수 \mean{(y' - y)^2}
# 텐서보드를 위한 요약정보(scalar)를 정의합니다.
tf.summary.scalar('loss', loss)

# 최적화를 위한 옵티마이저를 정의합니다.
optimizer = tf.train.GradientDescentOptimizer(0.01)
train_step = optimizer.minimize(loss)

# 트레이닝을 위한 입력값과 출력값을 준비합니다.
x_train = [1, 2, 3, 4]
y_train = [2, 4, 6, 8]

# 세션을 실행하고 파라미터(W,b)를 noraml distirubtion에서 추출한 임의의 값으로 초기화
합니다.
sess = tf.Session()
sess.run(tf.global_variables_initializer())

# 텐서보드 요약정보들을 하나로 합칩니다.
merged = tf.summary.merge_all()
# 텐서보드 summary 정보들을 저장할 폴더 경로를 설정합니다.
tensorboard_writer = tf.summary.FileWriter('./tensorboard_log', sess.graph)

# 경사하강법을 1000번 수행합니다.
for i in range(1000):
  sess.run(train_step, feed_dict={x: x_train, y: y_train})

# 매스텝마다 텐서보드 요약정보값들을 계산해서 지정된 경로('./tensorboard_log')에 저장
  summary = sess.run(merged, feed_dict={x: x_train, y: y_train})
  tensorboard_writer.add_summary(summary, i)

# 테스트를 위한 입력값을 준비합니다.
x_test = [3.5, 5, 5.5, 6]
# 테스트 데이터를 이용해 학습된 선형회귀 모델이 데이터의 경향성(y=2x)을 잘 학습했는지
측정합니다.
# 예상되는 참값 : [7, 10, 11, 12]
print(sess.run(linear_model, feed_dict={x: x_test}))

sess.close()
```

7. 딥러닝으로 MNIST 데이터 식별

가. MNIST 데이터

MNIST 데이터베이스 (Modified National Institute of Standards and Technology database)는 손으로 쓴 숫자들로 이루어진 대형 데이터베이스이며, 다양한 화상 처리 시스템을 트레이닝하기 위해 일반적으로 사용된다.

이 데이터베이스는 또한 기계 학습 분야의 트레이닝 및 테스트에 널리 사용된다. NIST의 오리지널 데이터셋의 샘플을 재혼합하여 만들어졌다. 개발자들은 NIST의 트레이닝 데이터셋이 미국의 인구조사국 직원들로부터 취합한 이후로 테스팅 데이터셋이 미국의 중등학교 학생들로부터 취합되는 중에 기계 학습 실험에 딱 적합하지는 않은 것을 느꼈다. 게다가 NIST의 흑백 그림들은 28x28 픽셀의 바운딩 박스와 앤티엘리어싱(anti-aliasing) 처리되어 그레이스케일 레벨이 들어가 있도록 평준화되었다.

MNIST 데이터베이스는 60,000개의 트레이닝 이미지와 10,000개의 테스트 이미지를 포함한다. 트레이닝 세트의 절반과 테스트 세트의 절반은 NIST의 트레이닝 데이터셋에서 취합하였으며, 그 밖의 트레이닝 세트의 절반과 테스트 세트의 절반은 NIST의 테스트 데이터셋으로부터 취합되었다.

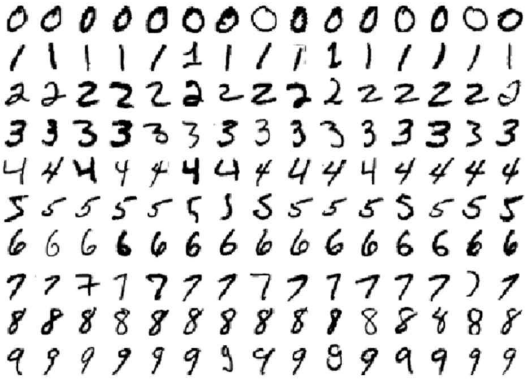


그림 7.1 MINST 테스트 데이터셋의 샘플 이미지

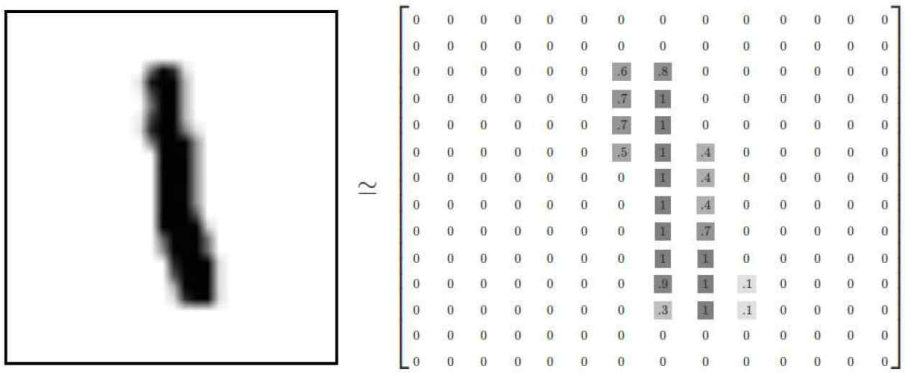


그림 7.2 MINST 데이터셋 중 “1”의 샘플 이미지 및 저장된 값 (0-1사이로 정규화된 값)

< Example 7-1 > MNIST 데이터 입력받기 및

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

# 텐서플로우에 기본 내장된 mnist 모듈을 이용하여 데이터를 로드
# 지정한 폴더에 MNIST 데이터가 없는 경우 자동으로 데이터를 다운로드
# one_hot 옵션은 레이블을 one_hot 방식의 데이터로 만들어 준다.
mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)

x_train = mnist.train.images
y_train = mnist.train.labels

x_val = mnist.test.images
y_val = mnist.test.labels

print(x_train[0].shape)   # (784, )
print(y_train[0].shape)   # (10, )
print(x_val[0].shape)     # (784, )
print(y_val[0].shape)     # (10, )

for i in range(10):

    #number = np.argmax(y_train[i])
    for index in range(10):
        if (y_train[i, index] == 1):
            number = index

    mnist_sample = np.reshape(x_train[i], [28,28])
    plt.subplot(2,5,i+1)
    plt.imshow(mnist_sample)
    plt.gray()
    plt.title(str(number))
    plt.axis('off')

plt.show()
```

그림 7.3 < Example 7-1 > 실행 결과

나. 딥러닝을 이용한 MNIST 데이터 식별

MNIST 데이터의 각 숫자 데이터는 28x28 사이즈의 이미지로 구성되어 있다. 입력 데이터는 28x28 사이즈의 이미지를 1차원으로 모양을 펴서 784개의 배열로 저장되어 있다. 앞서 언급하였듯이 훈련데이터는 60,000개 이고 테스트 데이터는 10,000개이다. 즉, 훈련데이터는 784x60,000이고, 테스트 데이터의 크기의 784x10,000이 된다.

이러한 MNIST 데이터의 구조에 따라, 신경망은 784개의 입력층을 가지게 되고, 출력층은 10개의 뉴런(0 ~9)으로 구성되게 된다. 입력과 출력증이 정해졌으므로, 은닉층은 사용자가 원하는 형태로 구성할 수 있다. 이번 예에서는 각각 256개를 가지는 두 개의 은닉층을 사용한다. 따라서 구성될 신경망은 784-256-256-10의 구조를 갖게 된다.

1) MNIST 데이터 입력

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)
```

2) 신경망 모델 구성

이미지를 한 장씩 학습시키는 것 보다는 여러 장을 묶어서 처리하는게 효율적이다. 이렇게 여러장 묶는 것을 배치(batch)라고 하며, 컴퓨터의 메모리와 성능에 따라 적절히 정한다. 아래의 코드와 같이 “None”으로 처리하면, 나중에 정해진 배치크기로 텐서플로가 알아서 처리하게 된다.

```
# 입력 값의 차원은 [배치크기, 특성값] 으로 되어 있다.
# 손글씨 이미지는 28x28 픽셀로 이루어져 있고, 이를 784개의 특성값으로 정합니다.
X = tf.placeholder(tf.float32, [None, 784])
# 결과는 0~9 의 10 가지 분류를 가집니다.
Y = tf.placeholder(tf.float32, [None, 10])
```

3) 신경망 레이어 구성

```
# 784(입력층)-256(1st 히든)-256(2nd 히든)-10(출력층 0~9 분류)
W1 = tf.Variable(tf.random_normal([784, 256], stddev=0.01))
# 입력값에 가중치를 곱하고 ReLU 함수를 이용하여 레이어를 만듭니다.
L1 = tf.nn.relu(tf.matmul(X, W1))
W2 = tf.Variable(tf.random_normal([256, 256], stddev=0.01))
# L1 레이어의 출력값에 가중치를 곱하고 ReLU 함수를 이용하여 레이어를 만듭니다.
L2 = tf.nn.relu(tf.matmul(L1, W2))
W3 = tf.Variable(tf.random_normal([256, 10], stddev=0.01))
# 최종 모델의 출력값은 W3 변수를 곱해 10개의 분류를 가지게 됩니다.
model = tf.matmul(L2, W3)
cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model,
      labels=Y))
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)
```

4) 신경망 모델 학습

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
batch_size = 100  #배치 사이즈를 100으로 정했음
total_batch = int(mnist.train.num_examples / batch_size)

for epoch in range(15):
    total_cost = 0
    for i in range(total_batch):
    # 텐서플로우의 mnist 모델의 next_batch 함수를 이용해
    # 지정한 크기만큼 학습할 데이터를 가져옵니다.
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        _, cost_val = sess.run([optimizer, cost], feed_dict={X: batch_xs, Y:
                        batch_ys})
        total_cost += cost_val

    print('Epoch:', '%04d' % (epoch + 1), 'Avg. cost =',
          '{:.3f}'.format(total_cost/ total_batch))
print('최적화 완료!')
```

5) 결과 확인

```
# model 로 예측한 값과 실제 레이블인 Y의 값을 비교합니다.
# tf.argmax 함수를 이용해 예측한 값에서 가장 큰 값을 예측한 레이블이라고 평가
# 예) [0.1 0 0 0.7 0 0.2 0 0 0 0] -> 3
is_correct = tf.equal(tf.argmax(model, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도:', sess.run( accuracy,
                 feed_dict={X:mnist.test.images, Y:mnist.test.labels}))
```

< Example 7-2 > 784-256-256-10 구조 신경망을 이용한 MNIST 식별

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)

#신경망 모델 구성
X = tf.placeholder(tf.float32, [None, 784]) #입력 데이터
Y = tf.placeholder(tf.float32, [None, 10])  # 결과는 0~9 의 10 가지

# 784(입력층)-256(1st 히든)-256(2nd 히든)-10(출력층 0~9 분류)
W1 = tf.Variable(tf.random_normal([784, 256], stddev=0.01))
L1 = tf.nn.relu(tf.matmul(X, W1))

W2 = tf.Variable(tf.random_normal([256, 256], stddev=0.01))
L2 = tf.nn.relu(tf.matmul(L1, W2))

W3 = tf.Variable(tf.random_normal([256, 10], stddev=0.01))
model = tf.matmul(L2, W3)

cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model, labels=Y))
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)

#신경망 모델 학습
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

batch_size = 100
total_batch = int(mnist.train.num_examples / batch_size)

for epoch in range(15):
    total_cost = 0

    for i in range(total_batch):
        # 텐서플로우의 mnist 모델의 next_batch 함수를 이용해
        # 지정한 크기만큼 학습할 데이터를 가져옵니다.
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)

        _, cost_val = sess.run([optimizer, cost],
                               feed_dict={X: batch_xs, Y: batch_ys})
        total_cost += cost_val

    print('Epoch:', '%04d' % (epoch + 1),
          'Avg. cost =', '{:.3f}'.format(total_cost / total_batch))
print('최적화 완료!')

#결과 확인
is_correct = tf.equal(tf.argmax(model, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도:', sess.run(accuracy,
                 feed_dict={X: mnist.test.images, Y: mnist.test.labels}))
```

다. 과적합(Overfitting)

과적합 문제(overfitting problem)란 학습데이터에 모델이 과도하게 학습되어 일반적인 데이터의 분류에 오히려 악영향을 미치는 현상을 뜻한다. 그림 7-4는 동일한 신경망을 학습시켰을 때, 학습된 신경망에서 나타나는 여러 가지 결정경계의 예를 보여준다.

먼저 'underfitting'의 경우를 보면 결정경계의 형태가 단순하여 오분류된 학습데이터가 세 가지 예시 중 가장 많은 것을 확인할 수 있다. 이러한 현상은 학습이 부족한 경우에 나타날 수 있다. 그리고 'overfitting'의 경우는 결정경계가 매우 복잡한 형태이고, 신경망이 학습데이터를 완전히 분류하도록 학습되어 있다. 이 경우에는 주어진 학습데이터에 대해서는 완벽하게 분류하지만 새로 획득한 데이터에 대해서는 오히려 분류성능이 저하될 수 있다. 'optimum'의 경우에는 결정경계를 과하게 복잡하도록 만드는 데이터에 대해서는 분류를 하지 않고 적절한 오류는 인정하는 경계선을 보여준다. 하지만 최적의 결정경계를 찾는다는 것이 쉬운 문제는 아니었고 이를 해결하기 위한 다양한 연구들이 있었다. 그림 7-5는 학습 횟수가 증가함에 따라 과적합이 발생하는 경우 오류의 변화를 보여준다. 이 경우에는 학습 횟수가 증가할수록 학습데이터에 대해서는 오류가 감소하지만 테스트 데이터에 대해서는 오류가 증가하게 되므로 적절한 학습 횟수를 결정하는 것이 중요하다.

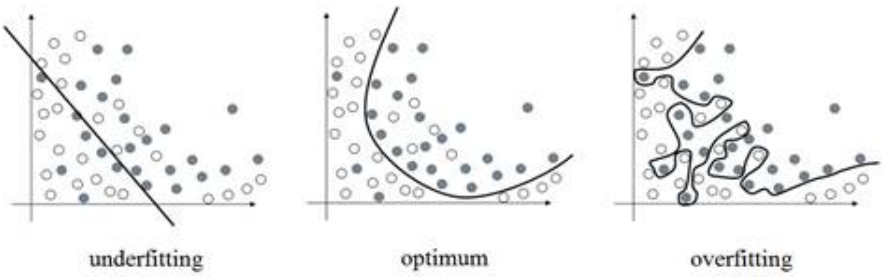


그림 7.4 학습 정도에 따라 생성된 결정경계

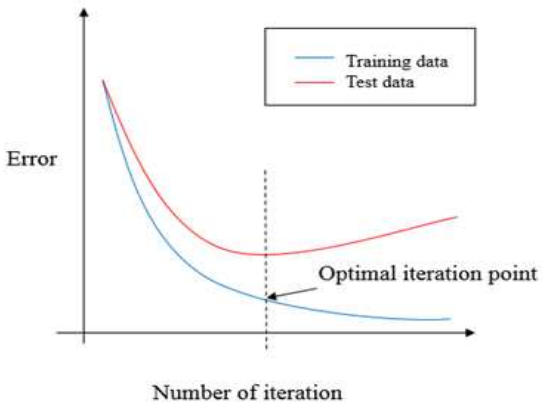


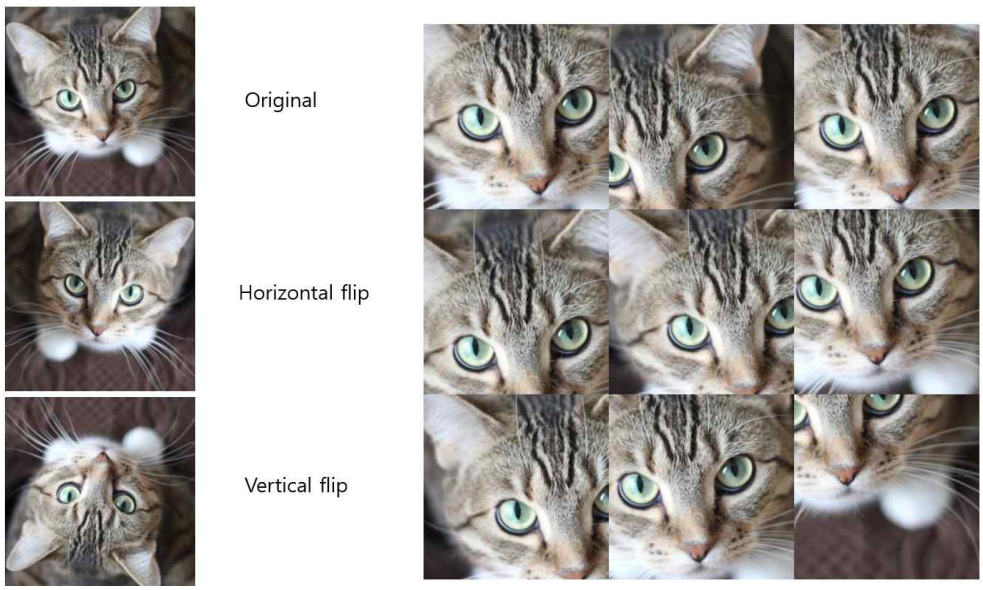
그림 7.5 과적합 문제가 발생할 경우 학습회수에 따른 오류의 변화

딥러닝 연구에서는 신경망이 복잡한 만큼 과적합 문제가 더욱 심해지는데 단순한 접근으로 큰 효과를 얻을 수 있는 세 가지 방법을 소개한다.

1) 데이터 확장(data augmentation)

먼저 소개할 방법은 데이터 확장(data augmentation) 기법이다. 과적합 문제의 원인이 학습데이터에 과하게 학습된 신경망에 있다면 학습데이터의 양을 충분히 확보한다면 과적합 현상이 발생하더라도 실제데이터를 정확하게 분류할 수 있을 것이다. 하지만 수집할 수 있는 환경이 제한되어 있는 경우에는 학습데이터의 양을 충분히 확보하기가 어렵다.

이를 보완하기 위해 기존에 보유한 데이터에 변형을 주어 학습 데이터와 유사한 데이터를 생성하여 양을 증가시키는 방법을 사용할 수 있는데 이를 데이터 확장이라고 한다. 이미지 분류의 문제에서는 원본 이미지를 뒤집기, 잘라내기, 회전, 샘플링, 스케일링, 반전, 잡음추가 등의 방법을 사용하여 학습 데이터의 양을 증가시킬 수 있고, 이는 성능 개선에 긍정적인 영향을 끼칠 수 있다. 테스트데이터에도 데이터 확장을 적용할 수 있는데, 원본 데이터와 함께 변형시킨 데이터들의 분류 결과에 투표방법을 적용하면 보다 안정적인 분류를 수행할 수 있게 된다.



(a) (b)

그림 7.6 데이터 확장(data augmentation)의 예
(a)이미지 좌우 및 상하 뒤집기 (b)임의로 잘라낸 이미지

2) 드롭 아웃(Drop-out)

두 번째 방법은 drop out 방법이다. Drop out은 신경망을 구성하는 뉴런 노드를 특정 확률로 제거한 후 학습을 진행하는 방법이다. 학습 횟수를 반복할 때 마다 노드가 제거된 신경망을 복구하고 새롭게 노드를 제거한 신경망을 생성한다. 이것은 원본 신경망을 단순화한 신경망을 다수 생성하여 각각을 학습시킨 후 가중치의 평균을 다시 원본 신경망에 적용하는 것과 같다. 그림 7-7(a)의 완전한 신경망에서 각 층의 노드를 특정 확률로 제거한 그림 7-7(b)와 같은 단순화한 모델을 다수 학습하여, 그 평균 값으로 테스트 신경망을 구성하는 방법이다.

특정한 학습데이터를 하나의 신경망에 학습시킨다면 과적합 현상이 발생할 가능성이 크겠지만 다수의 서로 다른 신경망에 학습시켜 각 신경망의 결정경계들의 평균을 구한다면 과적합 현상이 발생할 가능성을 줄일 수 있다. 서로 다른 다양한 식별모델을 학습시켜 결과를 조합하여 성능의 향상을 얻어내는 방법을 '앙상블 학습'이라고 하는데, drop out은 이러한 '앙상블 학습'을 간단한 방법으로 구현한 형태라고 할 수 있다. 실제로 drop out을 적용한 신경망을 학습시키고 평균 근사화한 신경망을 사용하여 테스트하면 일반화 오차가 현저히 낮아진다.

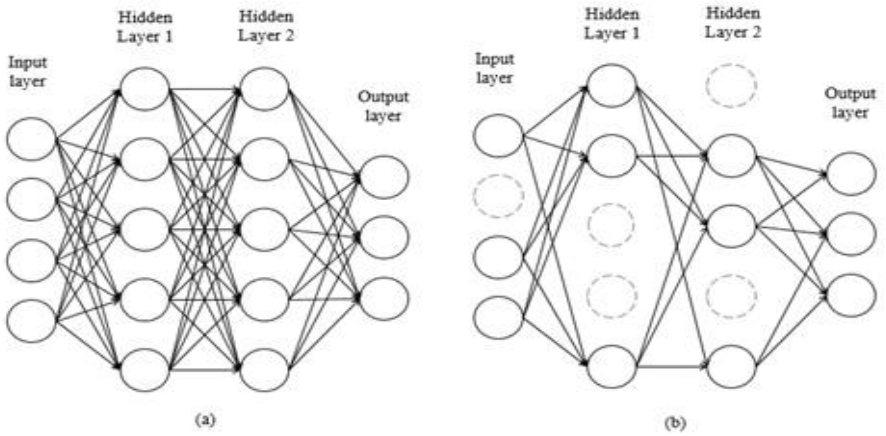


그림 7.7 Drop out
(a) 원본 신경망의 예 (b) drop out이 적용된 신경망의 예

Drop-out 기법을 적용하는 방법은 간단한다. 텐서플로에서는 tf.nn.drop이라는 drop-out을 위한 함수를 제공한다. 아래와 같이 적용하여 간단히 해결 할 수 있다. 아래 코드에서 keep_prob이 0.7이면, 70%의 노드만 사용하겠다는 뜻이다. 한 가지 주의할 점은 drop-out을 적용하더라도, 학습이 끝난 뒤 예측 시에는 0.7이 아니라 1값을 넣어서 신경망 전체를 사용해야 한다는 점이다.

```
keep_prob = tf.placeholder(tf.float32)      #drop-out 변수 설정
W1 = tf.Variable(tf.random_normal([784, 256], stddev=0.01))
L1 = tf.nn.relu(tf.matmul(X, W1))
L1 = tf.nn.dropout(L1, keep_prob)      #텐서플로에서 저공하는 drop-out 함수
```

< Example 7-3 > Drop-out 적용한 MNIST 식별

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)

#신경망 모델 구성
X = tf.placeholder(tf.float32, [None, 784])
Y = tf.placeholder(tf.float32, [None, 10])
keep_prob = tf.placeholder(tf.float32)

W1 = tf.Variable(tf.random_normal([784, 256], stddev=0.01))
L1 = tf.nn.relu(tf.matmul(X, W1))
L1 = tf.nn.dropout(L1, keep_prob)

W2 = tf.Variable(tf.random_normal([256, 256], stddev=0.01))
L2 = tf.nn.relu(tf.matmul(L1, W2))
L2 = tf.nn.dropout(L2, keep_prob)

W3 = tf.Variable(tf.random_normal([256, 10], stddev=0.01))
model = tf.matmul(L2, W3)

cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model, labels=Y))
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)

#신경망 모델 학습
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
batch_size = 100
total_batch = int(mnist.train.num_examples / batch_size)

for epoch in range(15):
    total_cost = 0

    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)

        _, cost_val = sess.run([optimizer, cost], feed_dict={X: batch_xs, Y: batch_ys,
                                                             keep_prob: 0.7})
        total_cost += cost_val

    print('Epoch:', '%04d' % (epoch + 1),
          'Avg. cost =', '{:.3f}'.format(total_cost / total_batch))

print('최적화 완료!')

#결과 확인
is_correct = tf.equal(tf.argmax(model, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도:',sess.run(accuracy,feed_dict={X: mnist.test.images, Y: mnist.test.labels,
                                             keep_prob: 1}))
```

3) 배치 정규화(Batch Normalization)

신경망의 최대 문제점은 기울기 소실 문제(Gradient Vanishing Problem)이 발생한다는 것이다. 이 문제를 해결하기 위하여 시그모이드(sigmoid)나 하이퍼탄젠트(tanh) 함수에서 렐루(ReLU)로의 활성화 함수 변화, 가중치의 초기값 설정 등의 방식을 채용하였다.

가중치의 초기값을 적절히 설정하면 각 층의 활성화 값 분포가 적당히 퍼지면서 학습이 원활하게 수행된다. 이러한 점에 착안하여 각 층의 활성화 값을 강제로 퍼뜨리는 것이 바로 배치 정규화이다. 보통의 경우 신경망 모델에 데이터를 입력하기 전에 전체 데이터 세트를 정규화하는 것이 일반적이지만, 배치 정규화는 그 이름처럼 미니-배치(mini-batch) 별로 학습 바로 전(후)에 데이터를 정규화하는 방식이다.

배치 정규화의 장점에는 학습 속도를 빠르게할 수 있다는 점, 매개변수(가중치)의 초기값에 크게 의존하지 않는다는 점 그리고 과대적합(Overfitting)을 억제한다는 점이다. 배치 정규화는 알고리즘 자체에 규제(Regularization) 효과가 있어서 가중치를 작게하여 과대적합이 생기지 않도록 해준다. 또한 배치 정규화를 적용함으로써 또 다른 가중치 규제 항이 필요가 없어지며, 드롭아웃(Dropout)을 적용할 필요가 없어 학습 속도가 빨라진다. 각 층에서의 활성화 값이 적당히 분포되도록 조정하는 것이 배치 정규화(Batch Normalization)이고, 이 배치 정규화 계층을 신경망에 적절히 삽입하여주면 된다. 배치 정규화는 활성화 함수의 앞이나 뒤쪽에 삽입할 수 있지만, 대게는 활성화 함수의 바로 앞에 삽입한다.

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{7-1}$$

$$\sigma_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{7-2}$$

$$\hat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{7-3}$$

배치 정규화는 기본적으로 데이터 분포가 평균이 0, 분산이 1이 되도록 정규화한다. 알고리즘의 수식적인 표현은 위와 같다. 사이즈가 m인 미니-배치 데이터에 대하여 평균과 분산을 구한다. 그리고 이 입력 데이터를 평균이 0, 분산이 1이 되도록(적절한 분포가 되도록) 정규화한다.

신경망은 비선형성을 가지고 있어야 표현력이 커져 복잡한 함수를 표현할 수 있는데, 위와 같이 평균이 0, 분산이 1로 고정시키는 것은, 활성화 함수의 비선형성을 없애버릴 수 있다. 예를 들어, 활성화 함수가 시그모이드 함수일 때 평균이 0, 분산이 1로 데이터를 고정시키면 활성화 함수를 통과한 값들은 거의 대부분 시그모이드 함수의 직선(선형) 부분에 머물게 된다. 비선형성이 거의 사라진 것이다. 이런 문제를 없애기 위하여 정규화된 데이터에 위와 같이 고유한 확대(scale, γ)와 이동(shift, β) 변환을 수행한다. 확대 인자(γ)와 이동 인자(β)는 역전파에 의해 학습된다. 이렇게 하면 활성화 함수에 비선형성이 부여될 수 있다.

$$y_i \leftarrow \gamma\hat{x_i} + \beta \tag{7-4}$$

이 방법은 과적합을 방지 할뿐만 아니라 학습속도도 높여주는 효과가 있다. 텐서플로의 고수준 API 라이브러리인 tf.layers에서 제공하는 tf.layers.batch_normalization함수를 이용하여 쉽게 적용할 수 있다.

아래 그림 7-8은 손글씨 인식 데이터 세트 MNIST에 배치 정규화 계층을 사용한 신경망 학습과 사용하지 않은 신경망 학습을 비교한 것이다. 배치 정규화 계층을 사용한 신경망 학습이 그렇지 않은 것보다 훨씬 더 학습 속도가 빠른 것을 볼 수 있다.

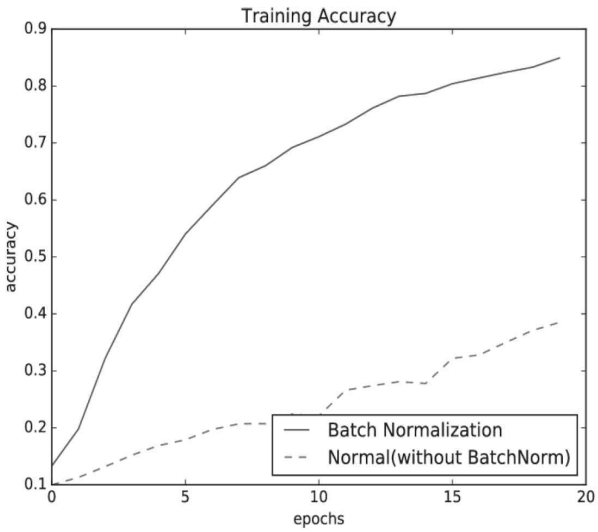


그림 7.8 MNIST에 배치 정규화 계층을 사용한 학습과 사용하지 않은 신경망 학습 비교

< Example 7-3 > Batch Normalization을 적용한 MNIST 식별

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./mnist/data/", one_hot=True)

#신경망 모델 구성
X = tf.placeholder(tf.float32, [None, 784])
Y = tf.placeholder(tf.float32, [None, 10])

W1 = tf.Variable(tf.random_normal([784, 256], stddev=0.01))
L1 = tf.nn.relu(tf.matmul(X, W1))
tf.layers.batch_normalization(L1, training=False)

W2 = tf.Variable(tf.random_normal([256, 256], stddev=0.01))
L2 = tf.nn.relu(tf.matmul(L1, W2))
tf.layers.batch_normalization(L2, training=False)

W3 = tf.Variable(tf.random_normal([256, 10], stddev=0.01))
model = tf.matmul(L2, W3)

cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=model, labels=Y))
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)

#신경망 모델 학습
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

batch_size = 100
total_batch = int(mnist.train.num_examples / batch_size)

for epoch in range(15):
    total_cost = 0

    for i in range(total_batch):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)

        _, cost_val = sess.run([optimizer, cost],
                               feed_dict={X: batch_xs, Y: batch_ys})
        total_cost += cost_val

    print('Epoch:', '%04d' % (epoch + 1),
          'Avg. cost =', '{:.3f}'.format(total_cost / total_batch))

print('최적화 완료!')

#결과 확인
is_correct = tf.equal(tf.argmax(model, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
print('정확도:',sess.run(accuracy,feed_dict={X:mnist.test.images, Y:mnist.test.labels}))
```