Ch01. 신경망 복습

Contents

- **주요 내용**
  - 수학과 파이썬 복습
  - 신경망 추론
  - 신경망 학습
  - 신경망으로 문제를 풀다
  - 계산 고속화

1.1 수학과 파이썬 복습

- 1.1.1 벡터와 행렬
  - 벡터는 크기와 방향을 가진 양
  - 파이썬에서는 벡터를 1차원 배열로 취급
  - 행렬은 2차원 배열로 표현

그림 1-1 벡터와 행렬의 예

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \qquad \begin{array}{cc} \begin{pmatrix}\boxed{1} & \boxed{2}\\ \boxed{3} & 4\\ \boxed{5} & 6\end{pmatrix} & \text{행} \\ \text{열} & \end{array}$$

그림 1-2 벡터의 표현법

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \qquad \begin{pmatrix}1 & 2 & 3\end{pmatrix}$$

열벡터 행벡터

- 실습
  - 벡터와 행렬은 np.array() 메서드로 생성
  - np.ndarray 클래스, shape은 다차원 배열의 형상, ndim은 차원 수

1.1.2 행렬의 원소별 연산

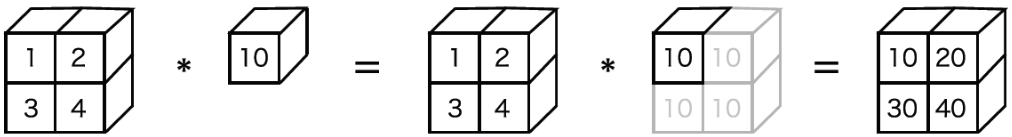
- 원소별(element-wise) 연산
  - 더하기(+)와 곱하기(*) 수행
  - 각 원소가 독립적으로 연산 수행

```
>>> import numpy as np
>>> W = np.array([[1,2,3],[4,5,6]])
>>> X = np.array([[0,1,2],[3,4,5]])
>>> W + X
array([[ 1,  3,  5],
       [ 7,  9, 11]])
>>> W * X
array([[ 0,  2,  6],
       [12, 20, 30]])
```

1.1.3 브로드캐스트

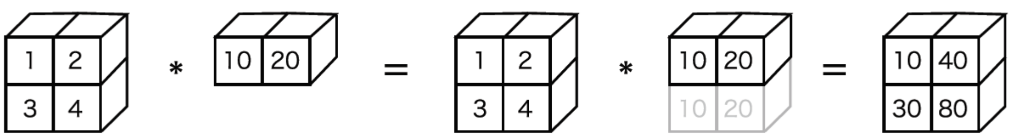
- 형상이 다른 배열끼리 연산
  - 스칼라 값이 행렬로 확장된 후에 원소별 연산 수행

그림 1-3 브로드캐스트의 예: 스칼라 값인 10이 2×2 행렬로 처리된다.



  - 1차원 배열이 2차원 배열의 형상이 같아지도록 확장 후 연산 수행

그림 1-4 브로드캐스트의 예 2



  - 넘파이의 브로드캐스트라는 기능을 제공하여 형상이 다른 배열끼리 연산 수행

1.1.4 벡터의 내적과 행렬의 곱

■ 벡터의 내적

- 두 벡터에서 대응하는 원소들의 곱을 모두 더한 것임
- 두 벡터가 얼마나 같은 방향을 향하고 있는가를 나타냄

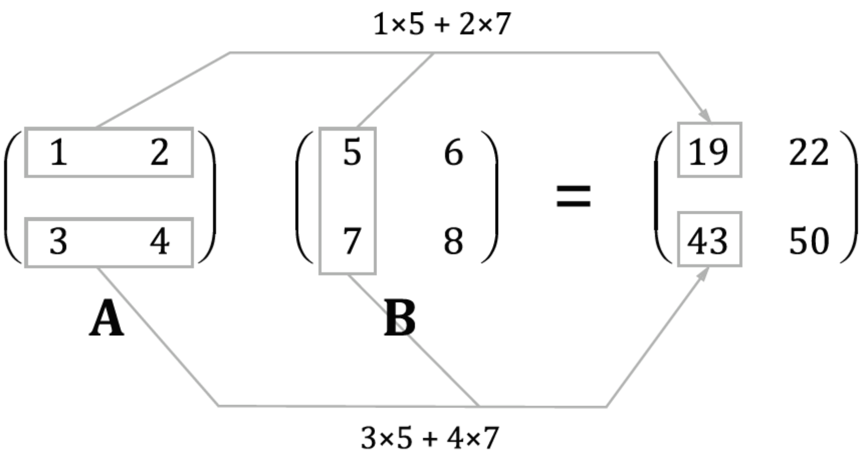
$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

- 넘파이의 np.dot() 사용

■ 행렬의 곱

- 넘파이의 np.matmul() 사용

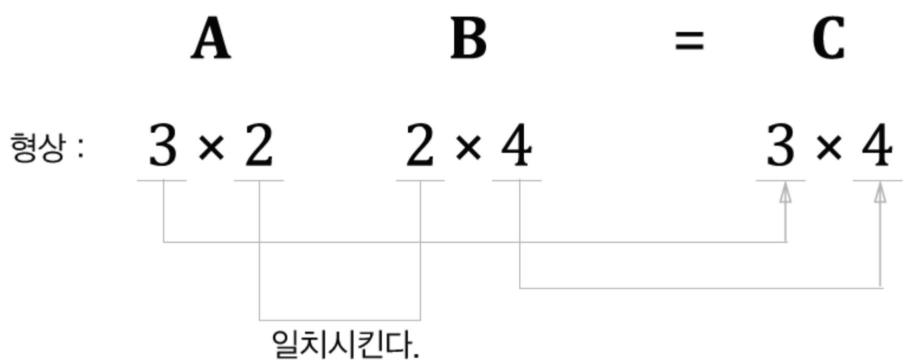
그림 1-5 행렬의 곱셈 방법



1.1.5 행렬 형상 확인

- 형상(shape)
  - 행렬이나 벡터를 사용해 계산할 때 ‘형상 확인’ 이 중요함

그림 1-6 형상 확인: 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킨다.

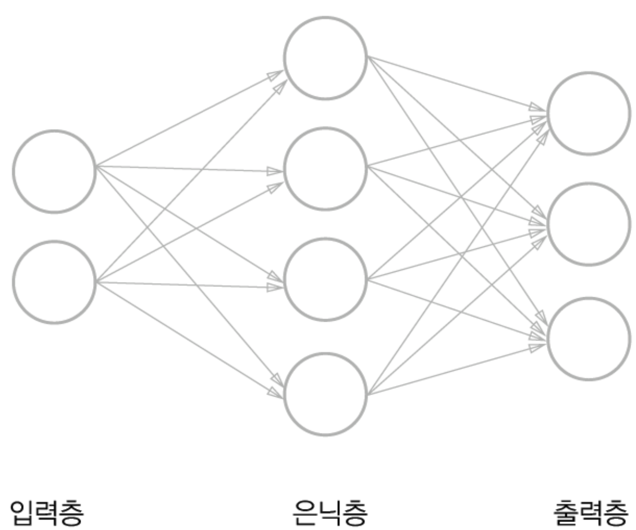


  - 행렬 A와 B가 대응하는 차원의 원소 수가 같아야 계산이 됨

1.2 신경망 추론

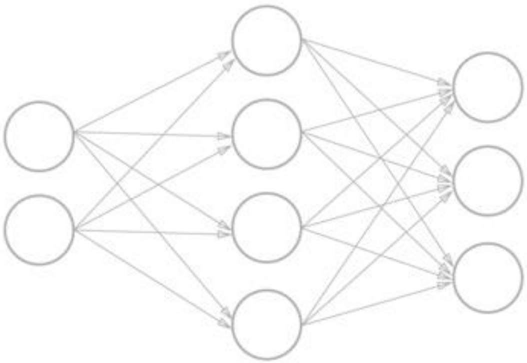
- 신경망에서 수행하는 작업은 학습과 추론임
- 신경망 추론 전체 그림
  - 신경망은 간단히 말하면 단순한 함수임 (합성함수)
  - 신경망은 함수처럼 입력을 출력으로 변환
  - 2차원 데이터를 입력하여 3차원 데이터를 출력하는 함수

그림 1-7 신경망의 예



1.2.1 신경망 추론 전체 그림

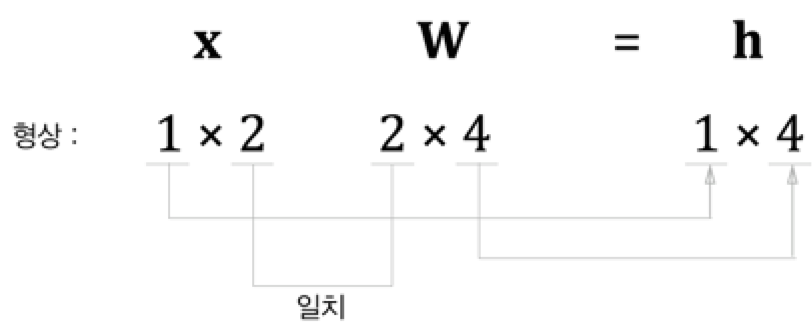
- 완전연결계층(fully connected layer)
  - 신경망이 수행하는 계산의 수식
  - 입력층 데이터 (x1, x2), 가중치 w, 편향 b



$$h_1 = x_1 w_{11} + x_2 w_{21} + b_1$$

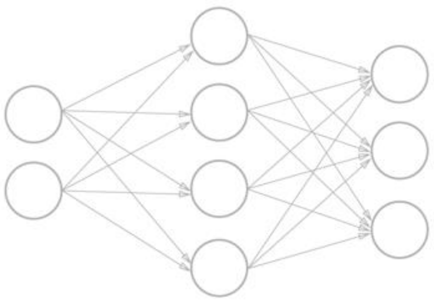
  - 완전연결계층의 수행하는 변환은 행렬의 곱을 이용해 정리

$$(h_1, h_2, h_3, h_4) = (x_1, x_2)\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + (b_1, b_2, b_3, b_4) \Rightarrow \mathbf{h} = \mathbf{xW} + \mathbf{b}$$



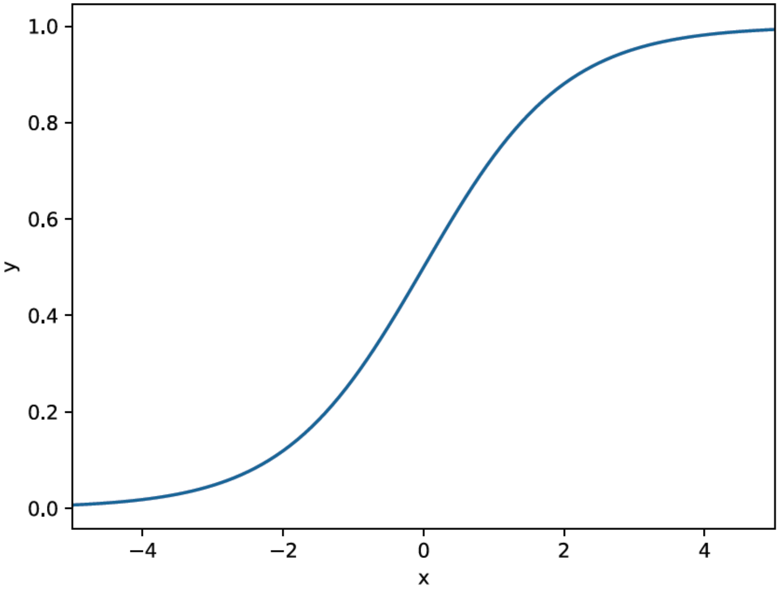
1.2.1 신경망 추론 전체 그림(1)

- 활성화 함수
  - 완전연결계층에 의한 변환은 선형 변환
  - 비선형 효과를 부여
  - 비선형 활성화 함수를 이용하여 신경망의 표현력을 높임
  - 시그모이드 함수



$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

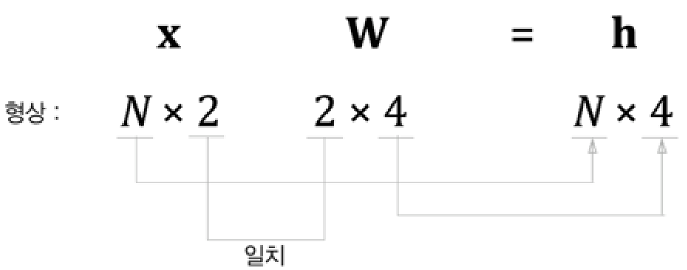
그림 1-10 시그모이드 함수의 그래프



1.2.1 신경망 추론 전체 그림(2)

- 완전연결계층 코딩 실습
  - 완전 연결계층에 의한 변환의 미니배치 구현

```
import numpy as np
W1 = np.random.randn(2, 4)   # 가중치
b1 = np.random.randn(4)      # 편향
x = np.random.randn(10, 2)   # 입력
h = np.matmul(x, W1) + b1
```



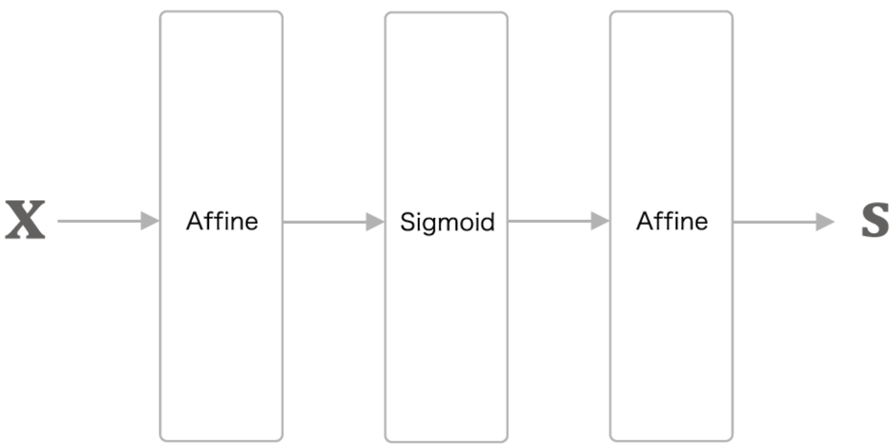
  - 비선형 활성화 함수를 이용하여 신경망의 표현력을 높임

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(10, 2)
W1 = np.random.randn(2, 4)
b1 = np.random.randn(4)
W2 = np.random.randn(4, 3)
b2 = np.random.randn(3)

h = np.matmul(x, W1) + b1
a = sigmoid(h)
s = np.matmul(a, W2) + b2
```

1.2.2 계층으로 클래스화 및 순전파 구현

- 신경망 처리를 계층(layer)로 구현
  - 완전연결계층에 의한 변환을 Affine 계층
  - 시그모이드 함수에 의한 변환을 Sigmoid 계층으로 구현
  - 모든 계층은 forward()와 backward() 메서드를 가짐
  - 모든 계층은 인스턴스 변수인 params와 grads를 가짐

그림 1-11 구현해볼 신경망의 계층 구성




```python
class Sigmoid:
    def __init__(self):
        self.params = []

    def forward(self, x):
        return 1 / (1 + np.exp(-x))
```

```python
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]

    def forward(self, x):
        W, b = self.params
        out = np.matmul(x, W) + b
        return out
```

1.2.2 계층으로 클래스화 및 순전파 구현(1)

- TwoLayerNet 클래스 신경망
  - 추론 처리는 predict() 메소드 구현
  - 매개변수 갱신과 매개변수 저장을 하나의 리스트에 보관
  - TwoLayerNet 클래스를 이용해 신경망의 추론 코드

```
x = np.random.randn(10, 2)
model = TwoLayerNet(2, 4, 3)
s = model.predict(x)
```

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 가중치와 편향 초기화
        W1 = np.random.randn(I, H)
        b1 = np.random.randn(H)
        W2 = np.random.randn(H, O)
        b2 = np.random.randn(O)

        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]

        # 모든 가중치를 리스트에 모은다.
        self.params = []
        for layer in self.layers:
            self.params += layer.params

    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x
```

1.3 신경망의 학습

- 신경망 학습
  - 최적의 매개변수 값을 찾는 작업
- 1.3.1 손실함수
  - 신경망 학습에서 학습이 얼마나 잘 되고 있는지를 알려주는 척도
  - 교차 엔트로피 오차는 신경망이 출력하는 각 클래스의 '확률'과 '정답 레이블'을 이용해 구할 수 있음

그림 1-12 손실 함수를 적용한 신경망의 계층 구성

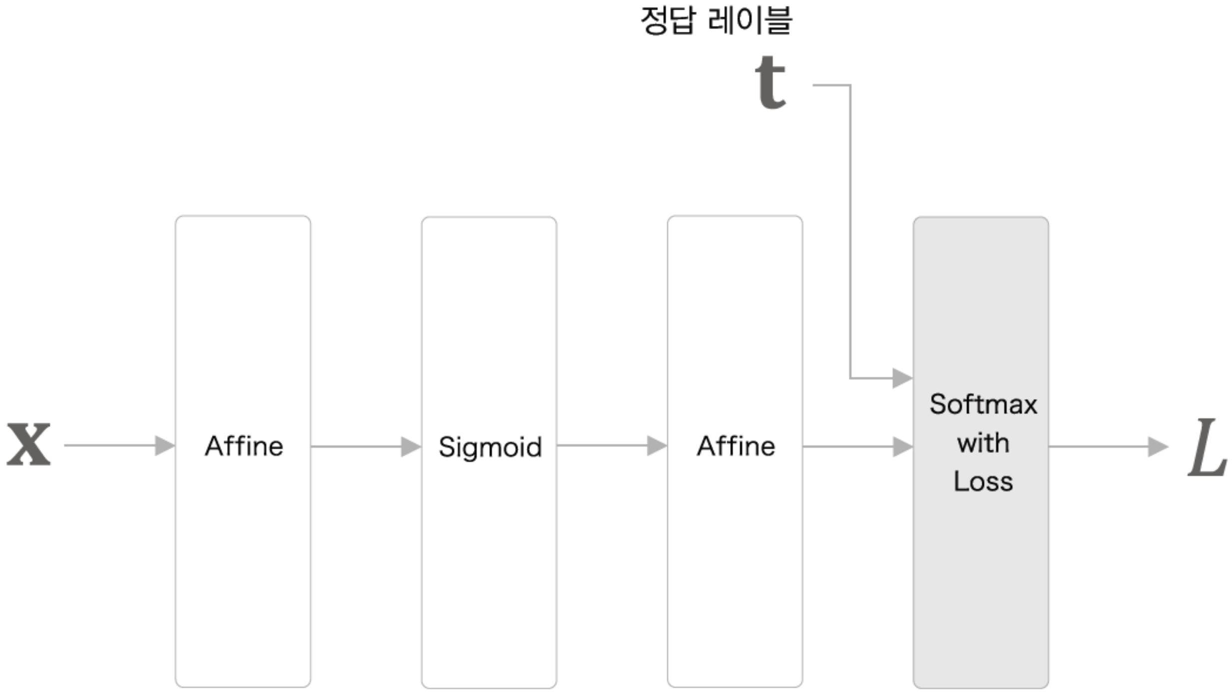
$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{n} \exp(s_i)}$$

$$L = -\sum_k t_k \log y_k$$

1.3.1 손실함수

- Softmax with Loss 계층
  - 두 계층을 통합하면 역전파 계산이 쉬워짐

그림 1-13 Softmax with Loss 계층을 이용하여 손실을 출력한다.



1.3.2 미분과 기울기

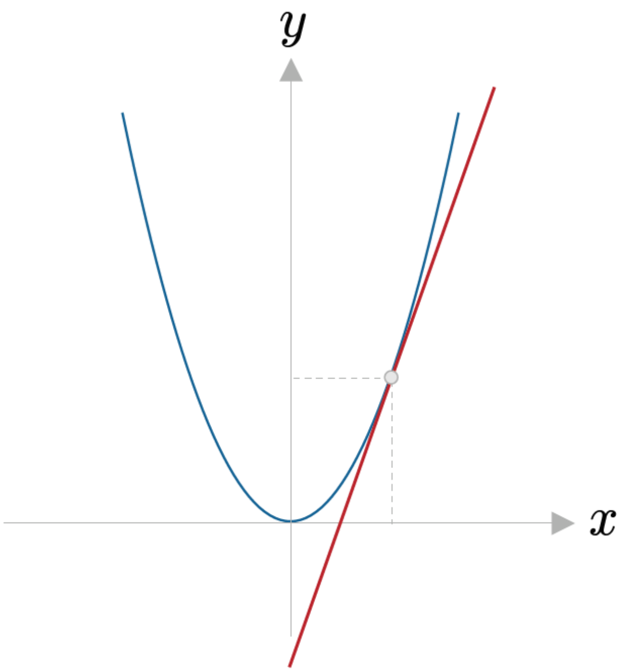
- 신경망 학습의 목표
  - 손실을 최소화하는 매개변수를 찾는 것
  - 미분과 기울기
  - 다변수라도 미분을 할 수 있음
  - 벡터의 각 원소에 대한 미분을 정리한 것이 기울기(gradient)

$$\frac{\partial L}{\partial \mathbf{x}} = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \cdots, \frac{\partial L}{\partial x_n} \right)$$

  - 행렬의 기울기

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \\ \frac{\partial L}{\partial W_{m1}} & & \frac{\partial L}{\partial W_{mn}} \end{pmatrix}$$

그림 1-14 $y = x^2$의 미분은 각 x에서의 기울기를 나타낸다.



1.3.3 연쇄 법칙

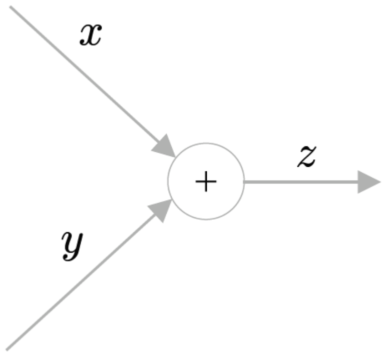
- 매개변수 갱신 방법
  - 학습 시 신경망은 데이터를 주면 손실을 출력
  - 각 매개변수에 대한 손실의 기울기
  - 기울기를 얻을 수 있다면, 그 것을 사용해 매개변수를 갱신 할 수 있음
- 오차역전파법 (back-propagation)
  - 신경망의 기울기를 구하는 방법
  - 연쇄법칙(Chain Rule) – 합성함수에 대한 미분 법칙
    - 아무리 많은 함수를 연결하더라도 그 미분은 개별 함수의 미분들을 이용해 구함
  - 합성함수의 미분

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

1.3.4 계산 그래프

- 계산 그래프
  - 계산 과정을 시각적으로 보여줌
  - 기울기도 직관적으로 구할 수 있음

그림 1-15 $z = x + y$를 나타내는 계산 그래프



- 순전파와 역전파
  - 전파되는 값은 최종 출력 L의 각 변수에 대한 미분

그림 1-16 앞뒤로 추가된 노드는 '복잡한 전체 계산'의 일부를 구성한다.

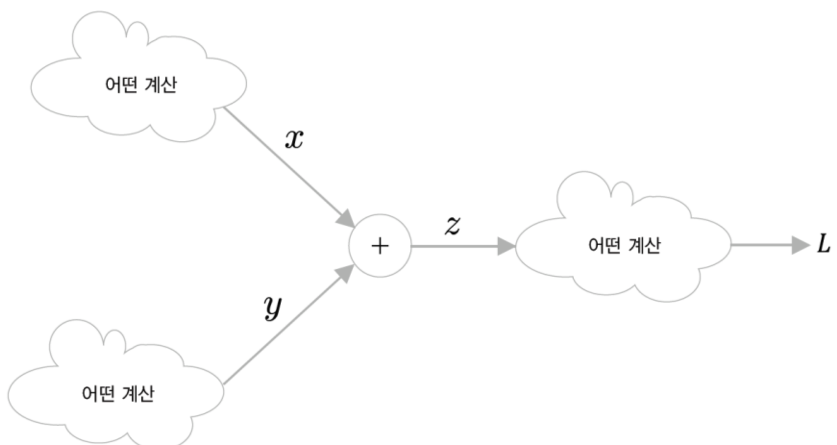
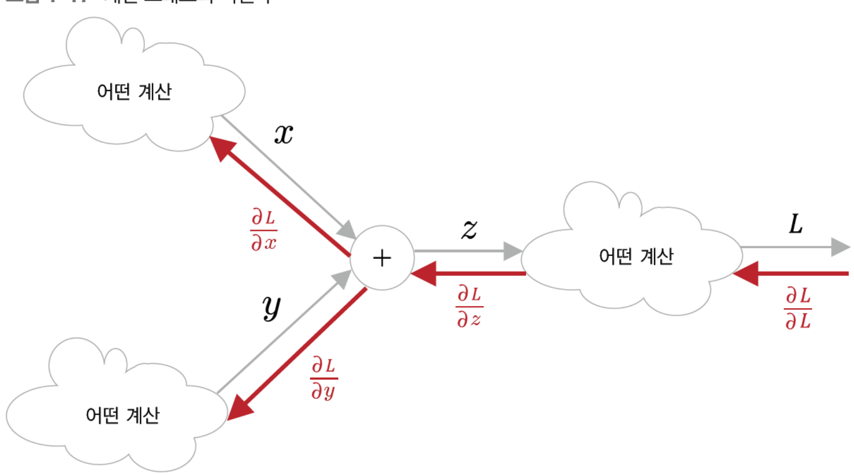


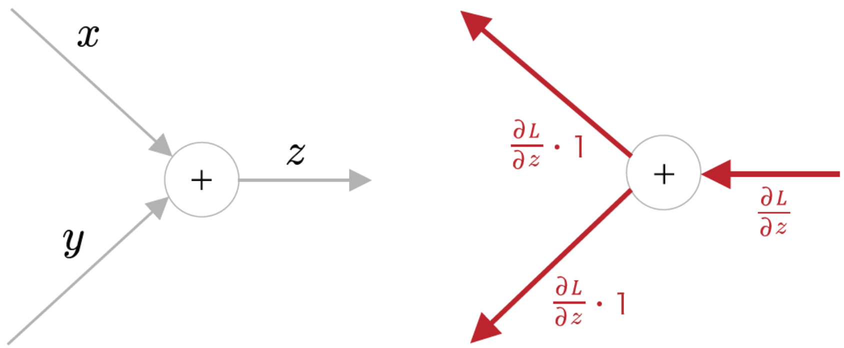
그림 1-17 계산 그래프의 역전파



1.3.4 계산 그래프(1)

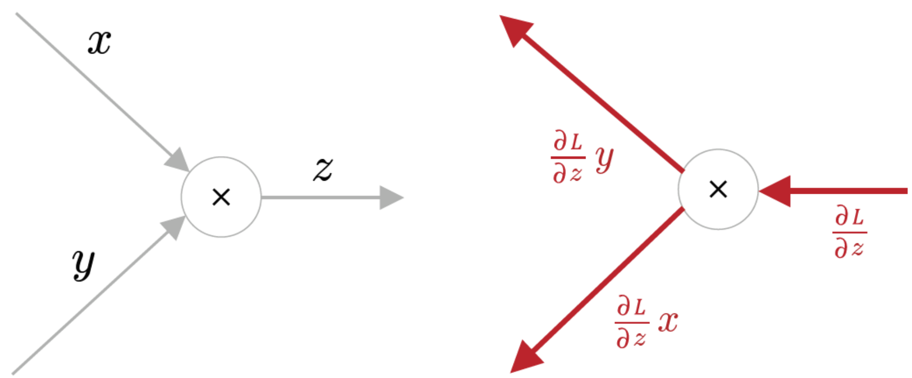
- 덧셈 노드
  - 상류로부터 받은 값에 1을 곱하여 하류로 기울기를 전파

그림 1-18 덧셈 노드의 순전파(왼쪽)와 역전파(오른쪽)



- 곱셈 노드
  - 상류로 부터 받은 기울기에 순전파시의 입력을 서로 바꾼 값을 곱함

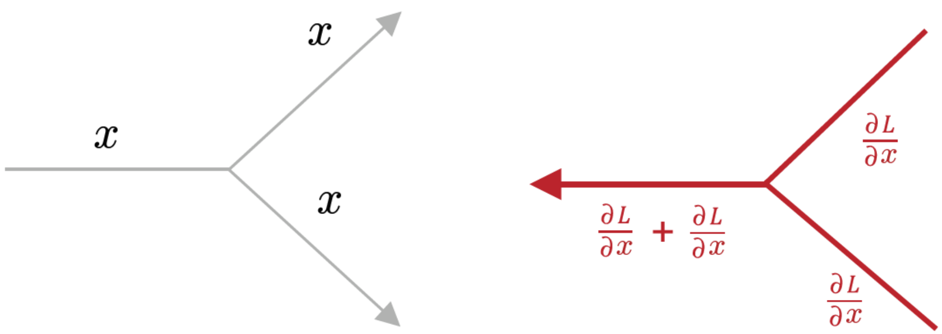
그림 1-19 곱셈 노드의 순전파(왼쪽)와 역전파(오른쪽)



1.3.4 계산 그래프(2)

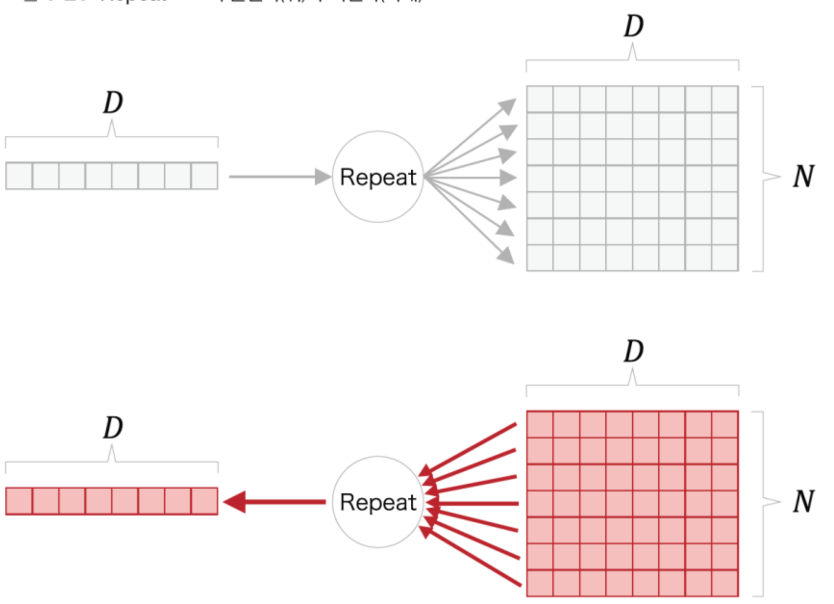
- 분기 노드
  - 같은 값이 복제되어 분기, 복제 노드라 할 수 있음

그림 1-20 분기 노드의 순전파(왼쪽)와 역전파(오른쪽)



- Repeat 노드
  - 분기 노드를 일반화하여 N개로의 분기
  - D인 배열을 N개로 복제

그림 1-21 Repeat 노드의 순전파(위)와 역전파(아래)

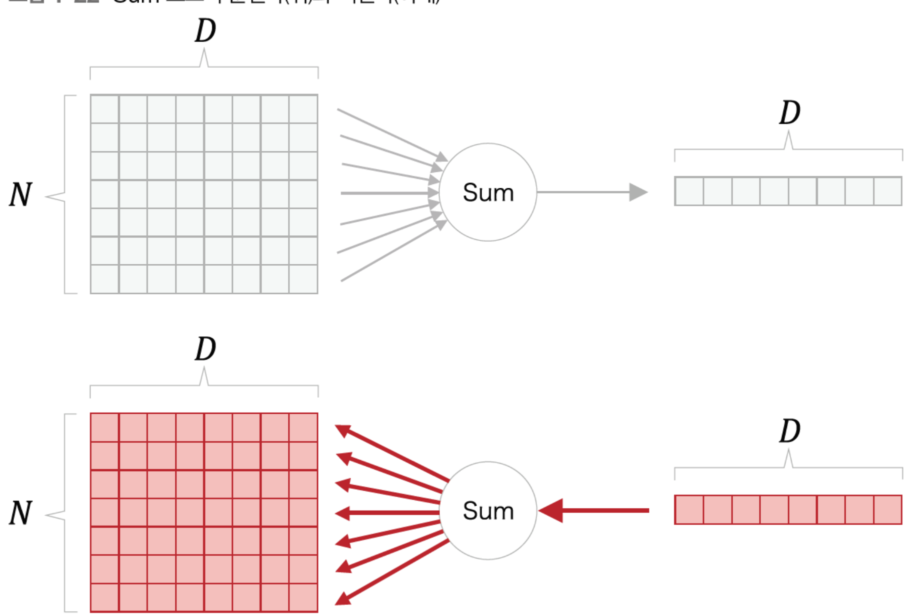



```
import numpy as np
D, N = 8, 7
x = np.random.randn(1, D)      # 입력
y = np.repeat(x, N, axis=0)    # 순전파
dy = np.random.randn(N, D)     # 무작위 기울기
dx = np.sum(dy, axis=0, keepdims=True)  # 역전파
```

1.3.4 계산 그래프(3)

- Sum 노드
  - 범용 덧셈 노드
  - Sum 노드와 Repeat 노드는 서로 반대 관계
  - N x D 배열에 대한 그 총합을 0축에 대해 구하는 계산

그림 1-22 Sum 노드의 순전파(위)와 역전파(아래)




```python
import numpy as np
D, N = 8, 7
x = np.random.randn(N, D)
y = np.sum(dy, axis=0, keepdims=True)

dy = np.random.randn(1, D)
dx = np.repeat(dy, N, axis=0)
```

1.3.4 계산 그래프(4)

- MatMul 노드
  - 행렬의 곱셈
  - y = xW 계산시

그림 1-24 행렬 곱의 형상 확인

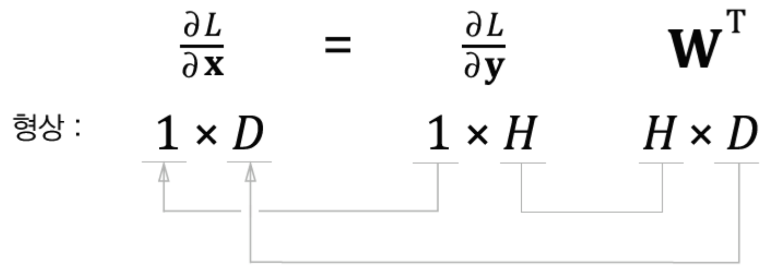


그림 1-23 MatMul 노드의 순전파: 각 변수 위에 형상을 표시함

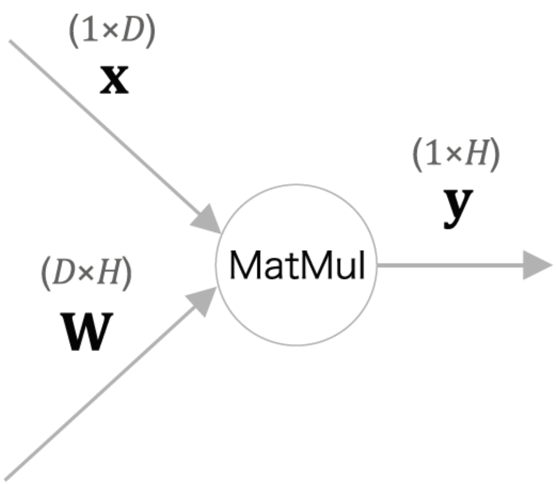
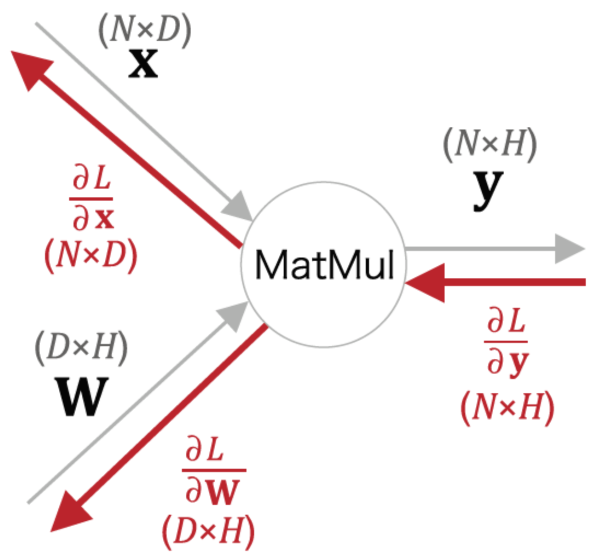


그림 1-25 MatMul 노드의 역전파



$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} W_{ij} \Rightarrow \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^{\mathrm{T}}$$

1.3.4 계산 그래프(5)

- Matmul 노드의 역전파
  - 행렬의 형상을 확인하여 행렬 곱의 역전파 식을 유도

그림 1-26 행렬의 형상을 확인하여 역전파 식을 유도한다.

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^{\mathrm{T}}$$

형상 : $N \times D$ $\quad N \times H \quad H \times D$

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{x}^{\mathrm{T}} \frac{\partial L}{\partial \mathbf{y}}$$

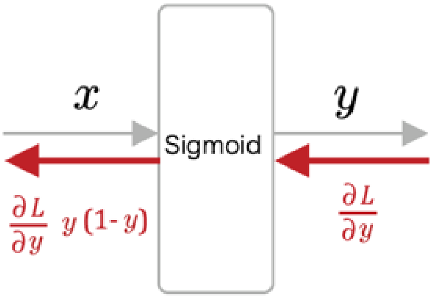
형상 : $D \times H$ $\quad D \times N \quad N \times H$

```python
class MatMul:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.x = None

    def forward(self, x):
        W, = self.params
        out = np.dot(x, W)
        self.x = x
        return out

    def backward(self, dout):
        W, = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        self.grads[0][...] = dW
        return dx
```

- Sigmoid 계층
  - Sigmoid 미분 수식

$$\frac{\partial y}{\partial x} = y(1-y)$$

  - Sigmoid 계산 그래프



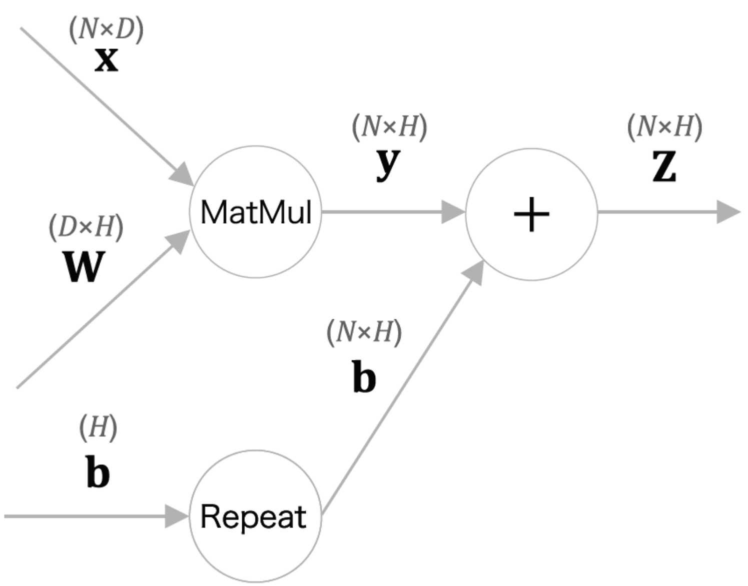
  - Sigmoid 계층 코드

```
class Sigmoid:
    def __init__(self):
        self.params, self.grads = [], []
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

1.3.5 기울기 도출과 역전파 구현(1)

- Affine 계층
  - 순전파는 y = np.matmul(x, W) + b
  - 편향을 더할 때는 넘파이의 브로드캐스트가 사용됨
  - 편향은 Repeat 노드에 의해 복제된 후 더해짐

그림 1-29 Affine 계층의 계산 그래프

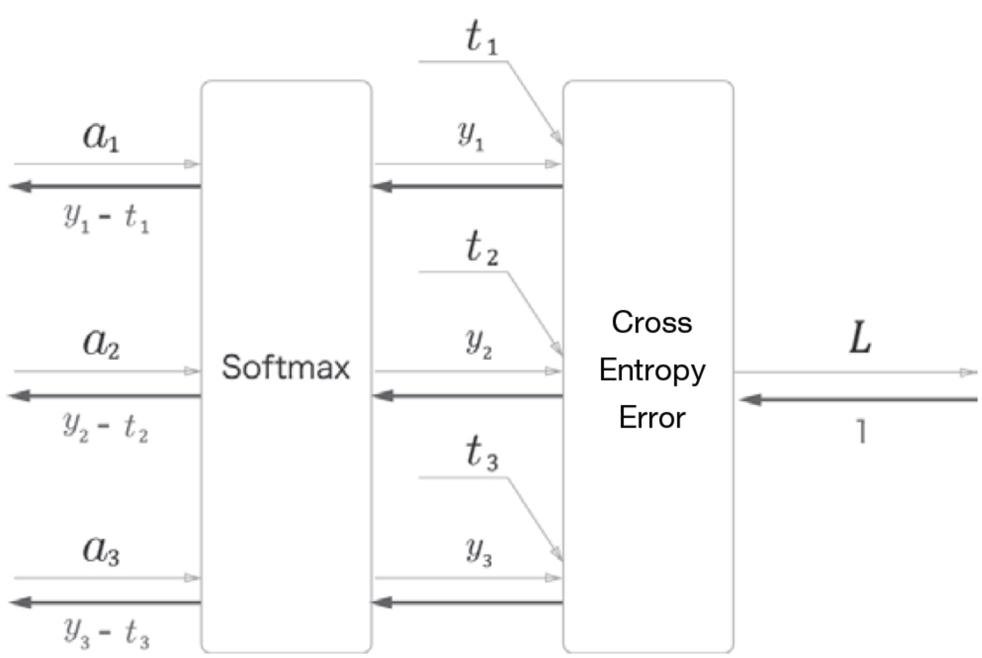



```python
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]
        self.grads = [np.zeros_like(W), np.zeros_like(b)]
        self.x = None

    def forward(self, x):
        W, b = self.params
        out = np.dot(x, W) + b
        self.x = x
        return out

    def backward(self, dout):
        W, b = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        db = np.sum(dout, axis=0)

        self.grads[0][...] = dW
        self.grads[1][...] = db
        return dx
```

1.3.5 기울기 도출과 역전파 구현(2)

- Softmax with Loss 계층
  - 소프트맥스 함수와 교차 엔트로피 오차를 하나의 계층으로 구현
  - Cross Entropy Error 계층
    - Softmax의 출력 (y1, y2, y3)와 정답 레이블(t1, t2, t3)를 받고 손실 L을 구해 출력

그림 1-30 Softmax with Loss 계층의 계산 그래프



1.3.6 가중치 갱신

- 가중치 갱신
  - 오차역전파법으로 기울기를 구한 후 그 기울기를 사용해 매개변수 갱신
- 신경망 학습 순서
  - 1단계 : 미니배치

    훈련 데이터 중에서 무작위로 다수의 데이터를 골라냄
  - 2단계 : 기울기 계산

    오차역전파법으로 각 가중치 매개변수에 대한 손실 함수의 기울기를 구함
  - 3단계 : 매개변수 갱신

    기울기를 사용하여 가중치 매개변수를 갱신
  - 4단계 : 반복

    1~3단계를 필요한 만큼 반복함

1.3.6 가중치 갱신(1)

- 경사하강법
  - 함수의 기울기(경사)를 구하고, 경사의 반대 방향으로 갱신을 반복하여 손실을 줄여 나가는 최적화 기법
- 확률적경사하강법(SGD)
  - 확률적(Stochastic)은 무작위로 선택된 데이터에 대한 기울기를 이용

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

- SGD 구현

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for i in range(len(params)):
            params[i] -= self.lr * grads[i]
```

```
model = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...)
    loss = model.forward(x_batch, t_batch)
    model.backward()
    optimizer.update(model.params, model.grads)
    ...
```

1.4 신경망으로 문제를 풀다

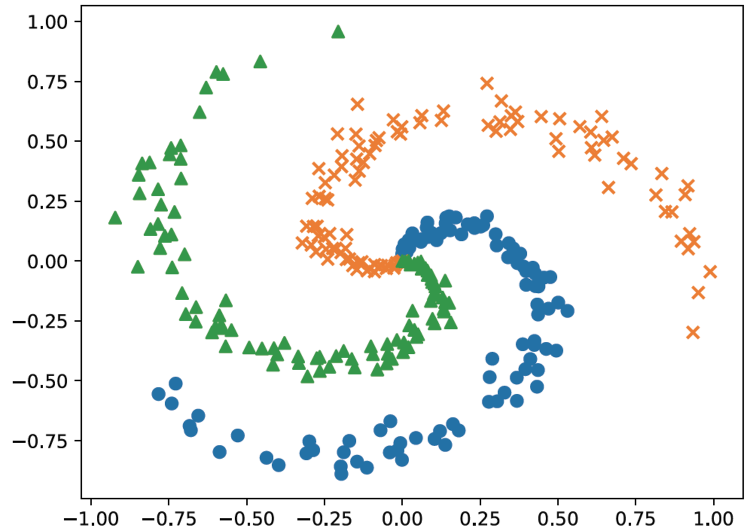
1.4.1 스파이럴 데이터셋

- 스파이럴 데이터를 읽어 들이는 클래스 구현

```
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from dataset import spiral
import matplotlib.pyplot as plt

x, t = spiral.load_data()
print('x', x.shape)  # (300, 2)
print('t', t.shape)  # (300, 3)
```

- 비선형 분리를 학습해서 클래스들을 분리
  - 입력은 2차원 데이터
  - 분류할 클래스 수는 3개

그림 1-31 학습에 이용할 스파이럴 데이터셋(3개의 클래스 각각을 X, ▲, ●로 표기)



■ 은닉층이 하나인 신경망 구현

- 가중치를 작은 무작위 값으로 설정하면 학습이 잘 진행될 가능성이 커짐

```python
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 가중치와 편향 초기화
        W1 = 0.01 * np.random.randn(I, H)
        b1 = np.zeros(H)
        W2 = 0.01 * np.random.randn(H, O)
        b2 = np.zeros(O)

        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]
        self.loss_layer = SoftmaxWithLoss()

        # 모든 가중치와 기울기를 리스트에 모은다.
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads
```

```python
    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def forward(self, x, t):
        score = self.predict(x)
        loss = self.loss_layer.forward(score, t)
        return loss

    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout
```

1.4.3 학습용 코드

- 학습을 수행하는 코드
  - 학습 데이터를 읽어 들여 신경망과 옵티마이저를 생성
  - 1.3.6에서 언급한 학습의 4 단계의 절차대로 학습 수행

```python
import sys
sys.path.append('..')  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.optimizer import SGD
from dataset import spiral
import matplotlib.pyplot as plt
from two_layer_net import TwoLayerNet


# 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

# 데이터 읽기, 모델과 옵티마이저 생성
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)
```

```python
for epoch in range(max_epoch):
    # 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
        batch_x = x[iters*batch_size:(iters+1)*batch_size]
        batch_t = t[iters*batch_size:(iters+1)*batch_size]

        # 기울기를 구해 매개변수 갱신
        loss = model.forward(batch_x, batch_t)
        model.backward()
        optimizer.update(model.params, model.grads)

        total_loss += loss
        loss_count += 1

        # 정기적으로 학습 경과 출력
        if (iters+1) % 10 == 0:
            avg_loss = total_loss / loss_count
            print('| 에폭 %d |  반복 %d / %d | 손실 %.2f'
                  % (epoch + 1, iters + 1, max_iters, avg_loss))
            loss_list.append(avg_loss)
            total_loss, loss_count = 0, 0
```

1.4.3 학습용 코드(1)

- 코드 실행
  - 손실값의 결과 그래프 (그림 1-32)
  - 결정 경계 시각화 (그림 1-33)
    - 학습된 신경망은 나선형 패턴을 올바르게 파악함

그림 1-32 손실 그래프: 가로축은 학습의 반복 수(눈금 값의 10배), 세로축은 학습 10번 반복당 손실 평균

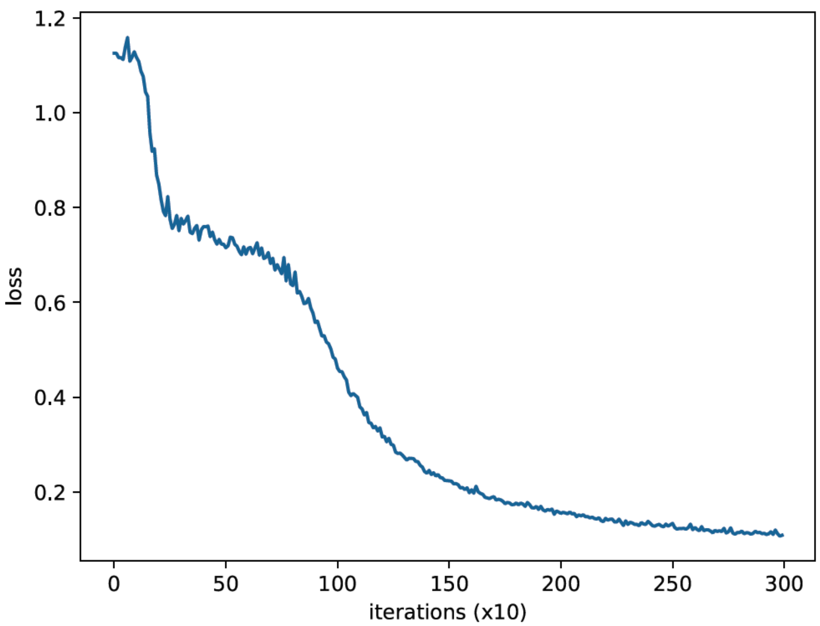
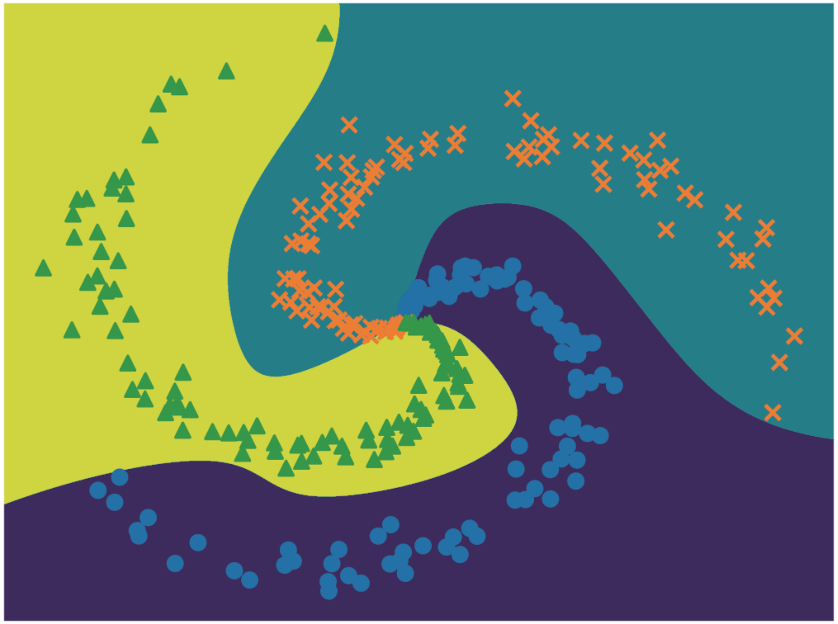


그림 1-33 학습 후 신경망의 결정 경계(신경망이 식별하는 클래스별 영역을 색으로 구분)



1.4.4 Trainer 클래스

- Trainer 클래스 역할
  - 학습 코드가 자주 필요하므로 학습을 수행하는 역할을 제공

```
model = TwoLayerNet(...)
optimizer = SGD(lr=1.0)
trainer = Trainer(model, optimizer)
```

```
import sys
sys.path.append('..')
from common.optimizer import SGD
from common.trainer import Trainer
from dataset import spiral
from two_layer_net import TwoLayerNet

# 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

trainer = Trainer(model, optimizer)
trainer.fit(x, t, max_epoch, batch_size, eval_interval=10)
trainer.plot()
```

표 1-1 Trainer 클래스의 fit() 메서드가 받는 인수: '(=XX)'는 기본값을 뜻함

인수	설명
x	입력 데이터
t	정답 레이블
max_epoch **(=10)**	학습을 수행하는 에폭 수
batch_size **(=32)**	미니배치 크기
eval_interval **(=20)**	결과(평균 손실 등)를 출력하는 간격 예컨대 eval_interval=20으로 설정하면, 20번째 반복마다 손실의 평균을 구해 화면에 출력한다.
max_grad **(=None)**	기울기 최대 노름norm 기울기 노름이 이 값을 넘어서면 기울기를 줄인다(이를 기울기 클리핑이라 하며, 자세한 설명은 '5장. 순환 신경망(RNN)' 참고).

1.5 계산 고속화

- 신경망의 학습과 추론은 대규모 컴퓨팅 리소스가 필요함
- 신경망에서는 얼마나 빠르게 계산하느냐가 중요한 주제임
- 비트 정밀도와 GPU 소개
  - 신경망 고속화에 도움이 됨

1.5.1 비트 정밀도

- 신경망의 추론과 학습 메모리 절약
  - 넘파이의 부동수수점 수는 64비트 데이터 타입이 표준
  - 신경망의 추론과 학습은 32비트 부동소수점 수로도 문제없이 수행 가능
  - 메모리 관점에서는 절반 가량 절약됨
  - 버스 대역폭 병목 현상 발생으로 데이터 타입이 작은게 유리

```
>>> import numpy as np
>>> a = np.random.randn(3)
>>> a.dtype
dtype('float64')
>>> b = np.random.randn(3).astype(np.float32)
>>> b.dtype
dtype('float32')
```

1.5.2 GPU(쿠파이)

- 딥러닝 계산
  - 대량의 곱하기 연산으로 구성
  - 대량의 곱하기 연산 대부분 병렬로 계산함으로 CPU보다 GPU가 유리함
- 쿠파이
  - GPU를 이용해 병렬 계산을 수행해주는 라이브러리
  - 파이썬 라이브러리로 엔비디아 GPU에서만 동작함
  - GPU 전용 범용 병렬 컴퓨팅 플랫폼인 CUDA를 설치해야 함
- GPU를 지원하는 코드
  - Config.GPU = True

```
import sys
sys.path.append('..')
import numpy as np
from common import config
# GPU에서 실행하려면 아래 주석을 해제하세요(CuPy 필요).
# ===============================================
# config.GPU = True
# ===============================================
```

Thank You