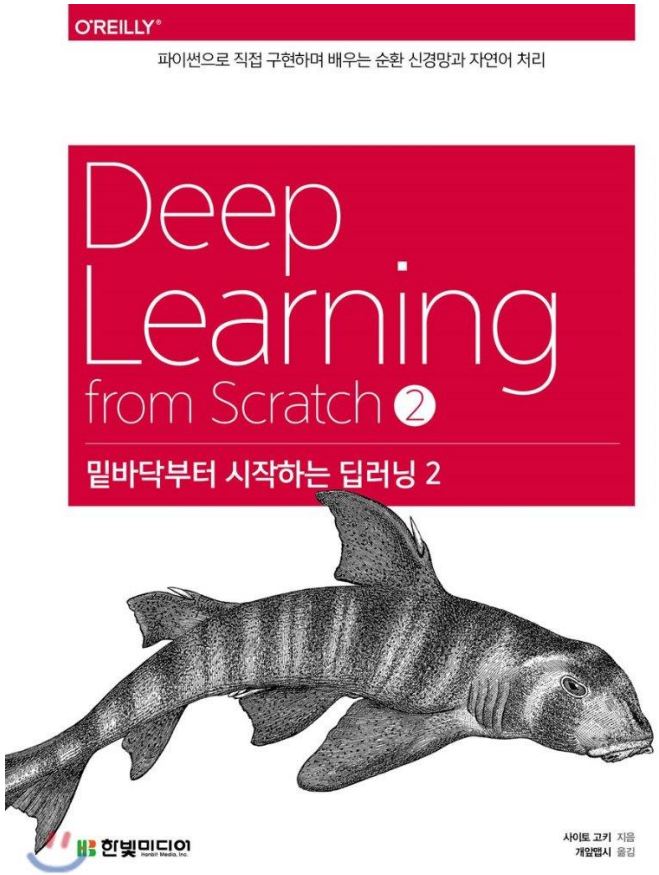
밑바닥부터 시작하는 딥러닝 ②

파이썬으로 직접 구현하며 배우는 순환신경망과 자연어처리



1장 신경망 복습

2장 자연어와 단어의 분산 표현

3장 word2vec

4장 word2vec 속도 개선

5장 순환신경망(RNN)

6장 게이트가 추가된 RNN

7장 RNN을 사용한 문장 생성

8장 어텐션

1장 신경망 복습

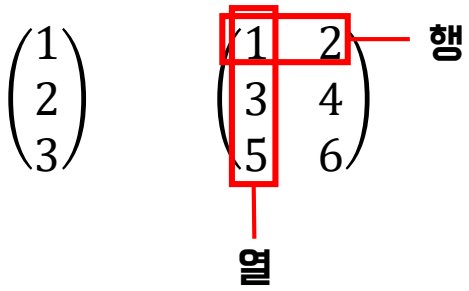
__1.1 수학과 파이썬 복습
__1.2 신경망의 추론
__1.3 신경망의 학습
__1.4 신경망으로 문제를 풀다
__1.5 계산 고속화
__1.6 정리

수학과 파이썬 복습

- **'벡터^{vector}'와 '행렬^{matrix}'**

  - **벡터 : 크기와 방향을 가진 양. 숫자가 일렬로 늘어선 집합 => 1차원 배열으로 표현 가능**
  - **행렬 : 숫자가 2차원 형태(사각형 형상)로 늘어선 것**

[그림 1-1] 벡터와 행렬의 예



[그림 1-1]의 행렬은 '3행 2열의 행렬'이라고 하고 '3X2 행렬'이라고 쓴다

NOTE_ 벡터와 행렬을 확장하여 숫자 집합을 N차원으로 표현한 것을 생각할 수 있다.
이를 일반적으로 텐서^{tensor}라고 한다.

수학과 파이썬 복습

- **행렬의 원소별$^{element-wise}$ 연산**

  NumPy는 서로 대응하는 원소끼리 (각 원소가 독립적으로) 연산이 이루어지는 element-wise 연산을 지원한다.

  Ex)

```
import numpy as np
W = np.array([[1,2,3],[4,5,6]])
X = np.array([[0,1,2],[3,4,5]])
```

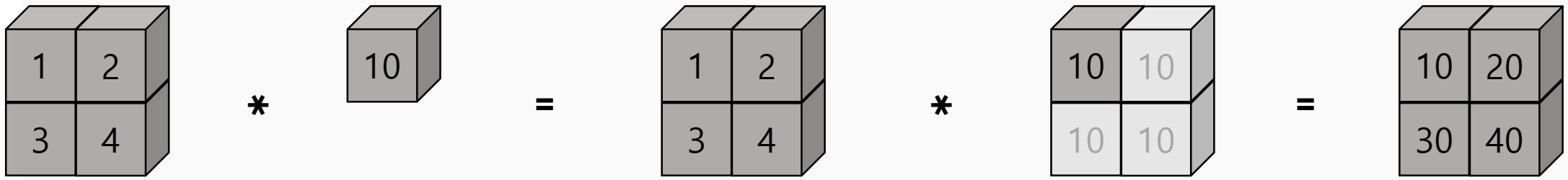
```
>>> W+X

[[ 1  3  5]
[ 7  9 11]]

>>> W*X

[[ 0  2  6]
[12 20 30]]
```

수학과 파이썬 복습

- **브로드캐스트**

  넘파이의 다차원 배열에서는 형상이 다른 배열끼리도 연산을 지원한다.
  다음과 같은 계산이 가능 ①

```
>>> A = np.array([[1,2],[3,4]])
>>> A*10
array([[10, 20],
       [30, 40]])
```

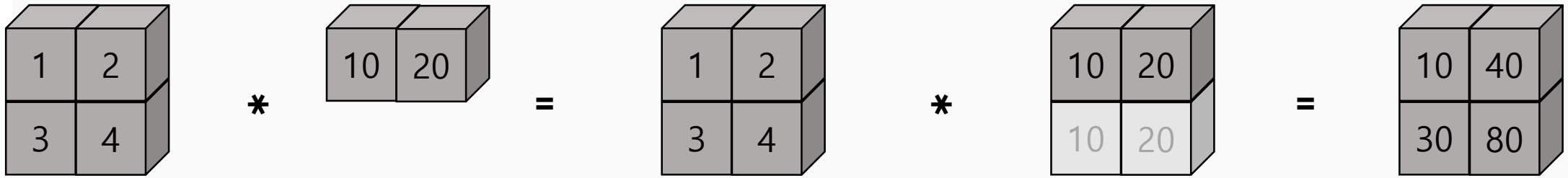


수학과 파이썬 복습

- **브로드캐스트**

  넘파이의 다차원 배열에서는 형상이 다른 배열끼리도 연산을 지원한다.
  다음과 같은 계산이 가능 ②

```
>>> A = np.array([[1,2],[3,4]])
>>> B = np.array([10,20])
>>> A*B
array([[10, 40],
       [30, 80]])
```



수학과 파이썬 복습

▪ 벡터의 내적과 행렬의 곱

• 벡터의 내적

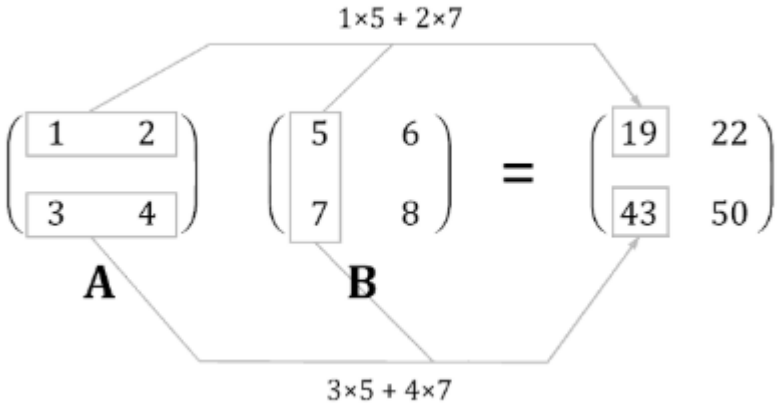
$$X \cdot Y = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

NOTE_ 벡터의 내적은 직관적으로는 '두 벡터가 얼마나 같은 방향을 향하고 있는가 ' 를 나타낸다
벡터의 길이가 1인 경우로 한정하면 완전히 같은 방향이면 1, 반대 방향이면 -1이다.

• 행렬의 곱

'왼쪽 행렬의 행벡터 (가로방향)'와 '오른쪽 행렬의 열벡터 (세로 방향)'의 내적(원소별 곱의 합)으로 계산

그림 1-5 행렬의 곱셈 방법



수학과 파이썬 복습

- **벡터의 내적과 행렬의 곱**

  numpy의 np.dot()과 np.matmul() 메서드를 이용하면 쉽게 표현 가능

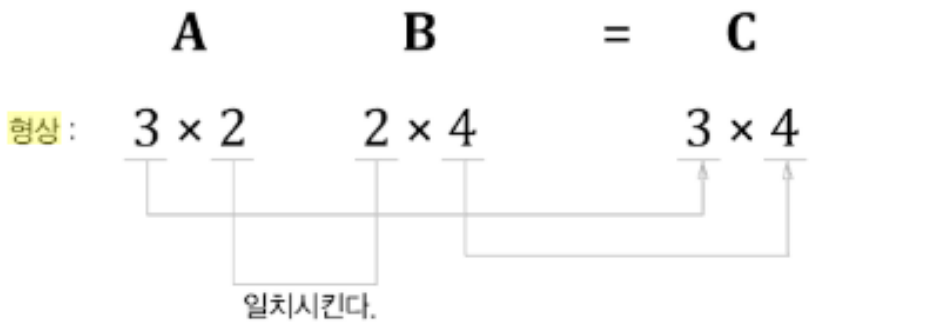
```
# 벡터의 내적
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> np.dot(a,b)
32

# 행렬의 곱
>>> A = np.array([[1,2],[3,4]]
)
>>> B = np.array([[5,6],[7,8]]
)
>>> np.matmul(A,B)
array([[19, 22],
       [43, 50]])
```

수학과 파이썬 복습

- **행렬 형상 확인**

  행렬이나 벡터를 사용해 계산할 때는 그 '형상$shape$'에 주의하여야 한다.

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



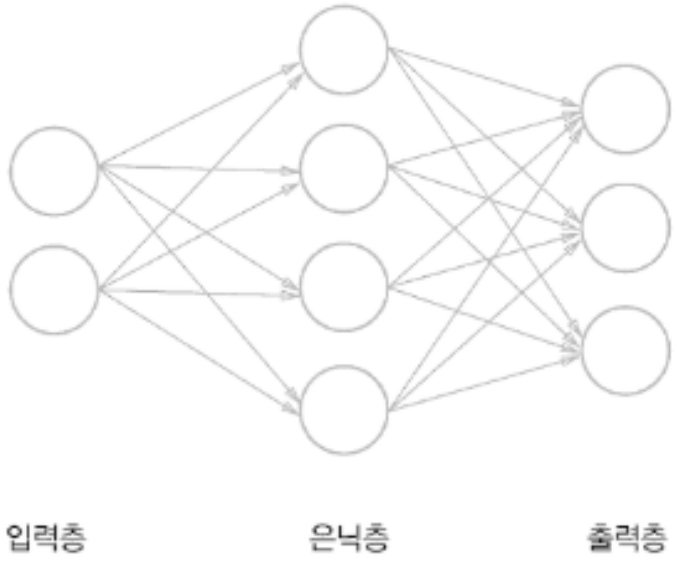
신경망의 추론

▪ 신경망 추론 전체 그림

신경망은 간단히 말하면 단순한 '함수'라 할 수 있다. 입력이 들어오면 무엇인가를 출력하는 시스템이다.
=> 신경망도 함수처럼 입력을 출력으로 변환한다.

2차원 데이터를 입력하여 3차원 데이터를 출력하는 함수를 예로 들어보자.
이 함수를 신경망으로 구현하려면 입력층$input\ layer$에는 뉴런 2개를, 출력층$output\ layer$에는 3개를 각각 준비한다.
그리고 은닉층$hidden\ layer$(혹은 중간층)에도 적당한 수의 뉴런을 배치 한다. 이번 예는 은닉층에 뉴런이 4개인 경우

그림 1-7 신경망의 예



이때 화살표에는 가중치$weight$가 존재하여, 그 가중치와 뉴런의 값을 각각 곱해서 그 합이 다음 뉴런의 입력으로 쓰인다.
(정확하게는 그 합에 활성화 함수$activation\ function$를 적용한 값이 다음 뉴런의 입력이 된다.)

이 때 각 층에는 이전 뉴런의 값에 영향받지 않는 '정수'도 더해지는데, 이 정수를 **bias**(편향)라고 한다.

[그림 1-7]과 같은 신경망은 인정하는 층의 모든 뉴런과 연결되어 있는 신경망을 완전연결계층$fully\ connected\ layer$라고 한다.

신경망의 추론

▪ 신경망 추론 전체 그림

[그림 1-7]의 신경망이 수행하는 계산을 수식으로 보면 다음과 같다. 입력은 (x_1, x_2), 가중치는 w_{11}, w_{21}으로, bias는 b_1으로 한다. 은닉층 중 첫 번째 뉴런은 다음과 같이 계산할 수 있다.

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

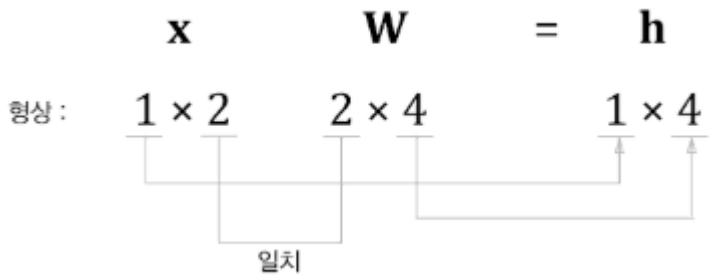
이런 식으로 위의 계산을 뉴런의 수만큼 반복하면 모든 뉴런의 값을 구할 수 있다. 이는 행렬을 이용해 다음과 같이 정리할 수 있다.

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

이를 행렬표기로 간단히 쓰면 다음과 같이 간소화할 수 있다.

$$\mathbf{h} = \mathbf{x}\mathbf{W} + \mathbf{b}$$

그림 1-8 형상 확인: 대응하는 차원의 원소 수가 일치함(편향은 생략)



신경망의 추론

▪ 신경망 추론 전체 그림

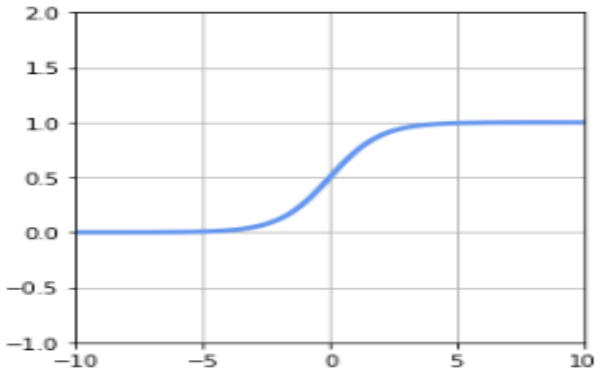
완전연결계층에 의한 변환의 미니배치 버전을 파이썬으로 구현

```
>>> W1 = np.random.randn(2,4) # 가중치
>>> b1 = np.random.randn(4)   # bias
>>> x = np.random.randn(10,2) # 입력
>>> h = np.matmul(x,W1) +
b1
```

WARNING_ bias b1의 덧셈은 브로드캐스트 된다. b1의 shape는 (4,)이지만 자동으로 (10,4)로 복제된다.

이러한 완전연결계층에 의한 변환은 '선형 ' 변환이다. 여기에 '비선형' 효과를 부여하는 것이 바로 활성화 함수이다.
비선형 활성화 함수를 이용함으로써 신경망의 표현력을 높일 수 있다.
=> 대표적인 시그모이드 함수$sigmoid\ function$

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




```
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
```

이 신경망은 3차원 데이터를 출력한다.
각 차원은 각 클래스에 대응하는 '점수 ' 가 된다.
가장 큰 값을 내뱉는 뉴런에 해당하는 클래스가 예측 결과가 된다.

신경망의 추론

▪ 계층으로 클래스화 및 순전파 구현

신경망에서 하는 처리를 계층으로 구현해보자. 완전연결계층에 의한 변환을 Affine 계층으로, 시그모이드 함수에 의한 변환을 Sigmoid 계층으로 구현한다. 각 계층은 클래스로 구현하며, 기본 변환을 수행하는 순전파는 forward()로 한다.

- 모든 계층은 forward()와 backward() 메서드를 가진다.
- 모든 계층은 인스턴스 변수인 params와 grads를 가진다.

forward()와 backward() 메서드는 각각 순전파와 역전파를 수행한다. Params는 가중치화 bias 같은 매개변수를 담는 리스트이다. grads는 params에 저장된 각 매개변수에 대응하여, 해당 매개변수의 gradient(기울기)를 저장하는 리스트이다.

NOTE_ 신경망 추론 과정에서 하는 처리는 신경망의 순전파$forward\ propagation$에 해당한다. 순전파란 말 그대로 입력층에서 출력층으로 향하는 전파이다. 뒤에 살펴볼 신경망 학습에서는 데이터(gradient)를 순전파와는 반대 방향으로 전파한다. 이를 역전파$backward\ propagation$라고 한다.

```python
import numpy as np
# Sigmoid Layer
class Sigmoid:
    def __init__(self):
        self.params = []

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

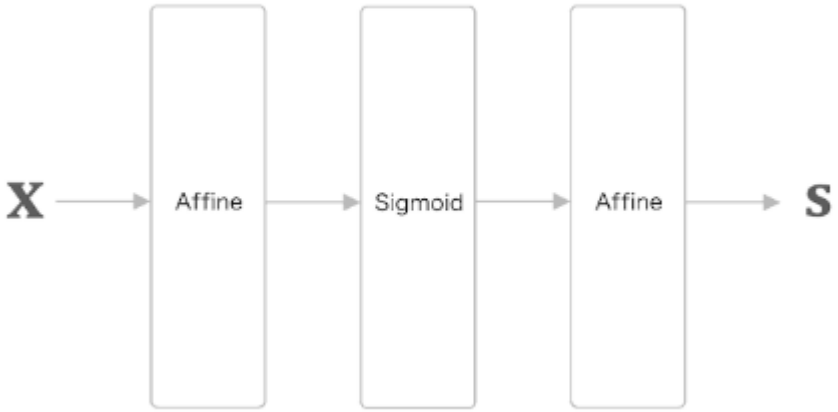
```python
import numpy as np
# Affine Layer
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]

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

신경망의 추론

- **계층으로 클래스화 및 순전파 구현**

  [그림 1-11]처럼 구성된 신경망을 구현한다.

그림 1-11 구현해볼 신경망의 계층 구성




```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

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

신경망의 학습

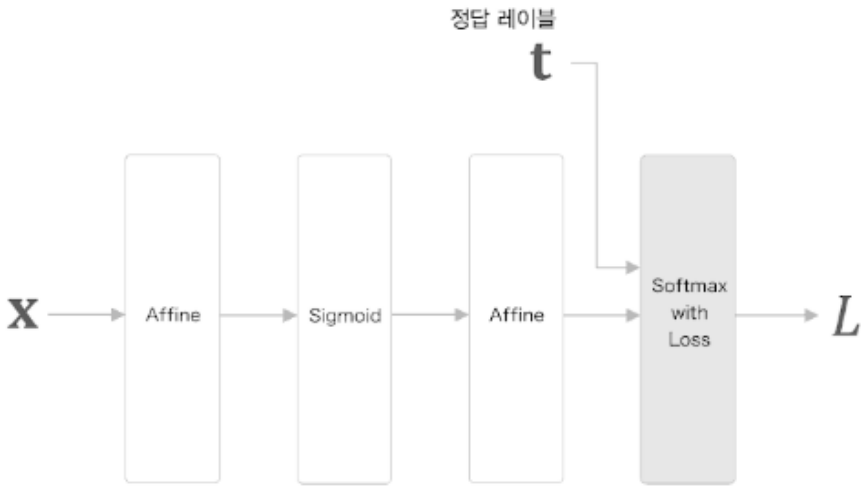
▪ 손실 함수 (Loss Function)

신경망 학습에는 학습이 얼마나 잘 되고 있는지를 알기 위한 '척도 ' 가 필요하다. 일반적으로 학습 단계의 특정 시점에서 신경망의 성능을 나타내는 척도로 손실$loss$을 사용한다. 손실은 학습 데이터 (학습 시 주어진 정답 데이터)와 신경망이 예측한 결과를 비교하여 예측이 얼마나 나쁜가를 산출한 단일 값(스칼라)이다.

이러한 손실은 손실 함수$loss\ function$을 사용해 구한다. 흔히 교차 엔트로피 오차$Cross\ Entropy\ Error$를 이용한다.
교차 엔트로피 오차는 신경망이 출력하는 각 클래스의 '확률 ' 과 '정답 레이블 ' 을 이용해 구할 수 있다.

여태까지 한 신경망에 손실을 구해보자. 우선 앞 절의 신경망에 Softmax 계층과 Cross Entropy Error 계층을 새로 추가한다.

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



이 신경망을 '계층' 관점으로 그리면 [그림 1-13]처럼 된다.
이 때 **x**는 입력 데이터, **t**는 정답레이블, **L**은 손실을 나타낸다.

신경망의 학습

▪ 미분과 기울기

신경망 학습의 목표는 손실을 최소화하는 매개변수를 찾는 것이다. 이때 중요한 것이 '미분'과 기울기'(gradient)이다.
미분은 어떠한 변수를 '조금' 변화시켰을 때 해당 함수가 얼마나 변하늦가 하는 '변화의 정도 ' 이다.

미분의 성질에 의해 여러 개의 변수(다변수)라도 마찬가지로 미분을 할 수 있다.

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

이처럼 벡터의 각 원소에 대한 미분을 정리한 것이 기울기$gradient$이다.

NOTE_ 여기서 중요한 점은 **W**가 mxn행렬이라면, $\frac{\partial L}{\partial \boldsymbol{W}}$도 mxn으로 똑같은 shape를 갖는다는 점이다.
이러한 '행렬과 그 기울기의 형상이 같다 ' 라는 성질을 이용하면 매개변수 갱신과 연쇄 법칙을 쉽게 구현할 수 있다.

신경망의 학습

▪ 연쇄 법칙(Chain Rule)

연쇄법칙이란 합성함수에 대한 미분의 법칙이다.
y = f(x)와 z = g(y)라는 두 함수가 있다고 하면, z = g(f(x))가 되어, 최종 출력 z는 두 함수를 조합해 계산 가능하다.
이때 이 합성함수의 미분(x에대한 z의 미분)은 다음과 같이 구할 수 있다.

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

이것이 바로 연쇄 법칙이다. 이 연쇄 법칙이 중요한 이유는 우리가 다루는 함수가 아무리 복잡하다 하더라도,
그 미분은 개별 함수의 미분들을 이용해 구할 수 있다는 점을 알려주기 때문이다.

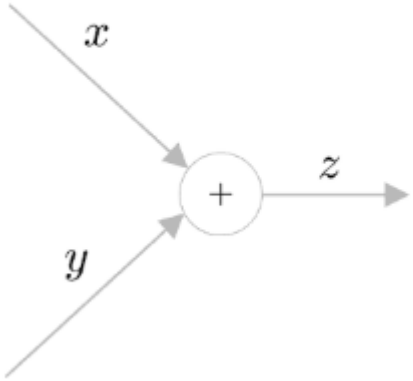
즉, 각 함수의 국소적인 미분을 계산 가능하다면, 그 값들을 곱해서 전체의 미분을 구할 수 있다.

신경망의 학습

▪ 계산 그래프

계산 그래프는 계산 과정을 시각적으로 보여준다. [그림 1-15]처럼 아주 단순한 계산 그래프를 예로 들어보자.

그림 1-15 $z = x + y$를 나타내는 계산 그래프



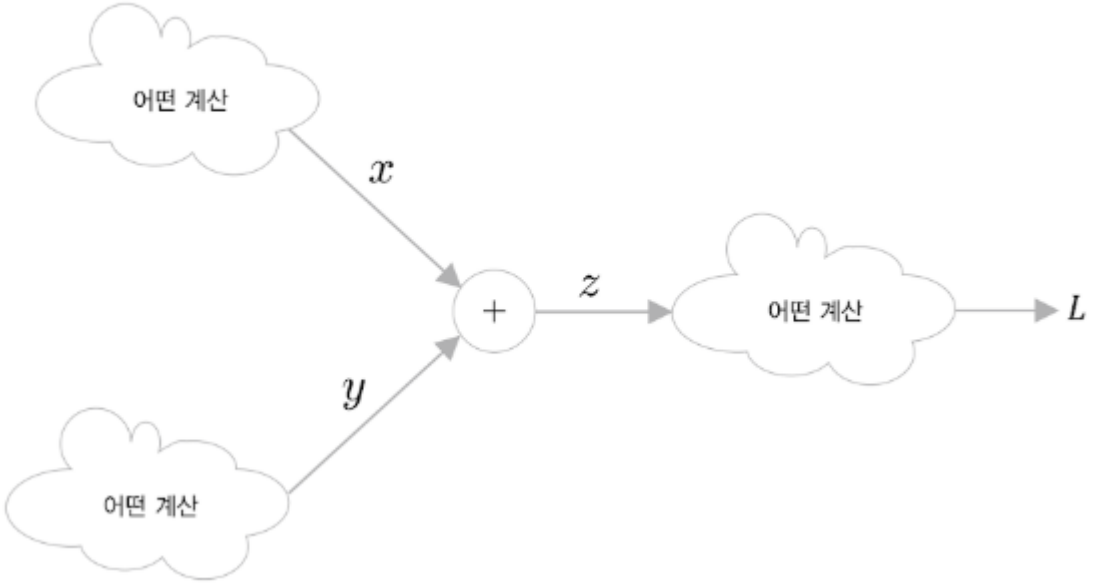
[그림 1-15]에서 보듯 계산 그래프는 노드와 화살표로 그린다. 이처럼 계산 그래프는 연산을 노드로 나타내고 그 처리 결과가 순서대로 흐른다. 이러한 계산 그래프를 이용하면 계산을 시각적으로 파악할 수 있다. 위의 예는 계산 그래프의 '순전파'이다. 여기서 중요한 점은 기울기가 순전파와 반대 방향으로 전파된다는 점인데, 이 반대 방향의 전파가 '역전파 ' 이다.

신경망의 학습

▪ 계산 그래프

역전파를 설명하기에 앞서, 역전파가 이뤄지는 전체 그림을 더 명확하게 그러보자. 앞의 그림은 z = x + y라는 계산을 다루고 있는데, 이 계산 앞뒤로도 '어떤 계산'이 있다고 가정하자. 그리고 최종적으로 스칼라 값인 L이 출력된다고 가정하자.

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

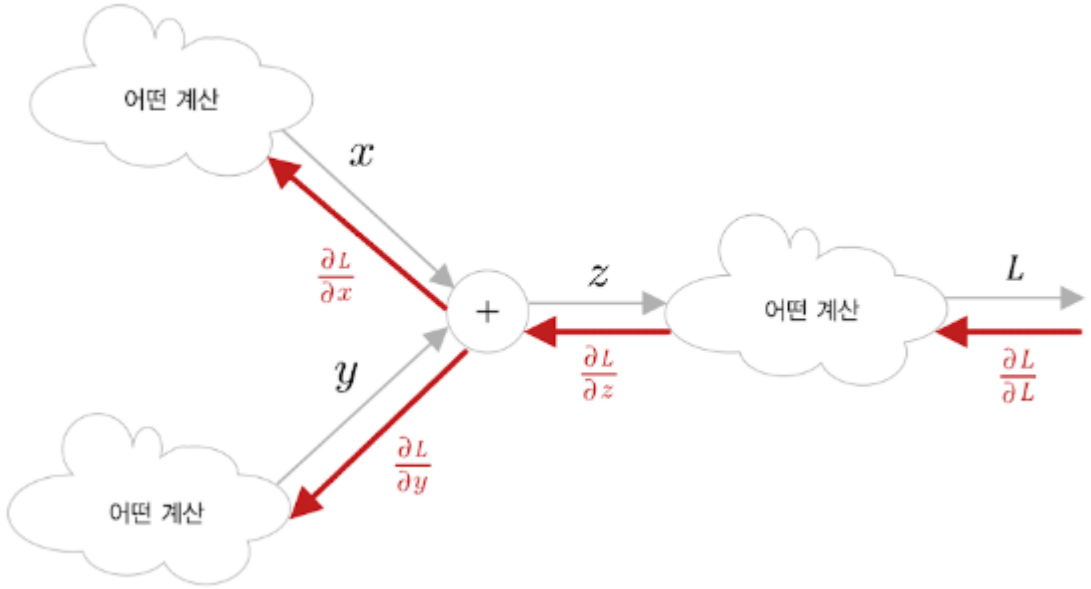


신경망의 학습

▪ 계산 그래프

우리의 목표는 L의 미분(기울기)을 각 변수에 대해 구하는 것이다. 그러면 계산 그래프는 다음과 같이 그릴 수 있다.

그림 1-17 계산 그래프의 역전파



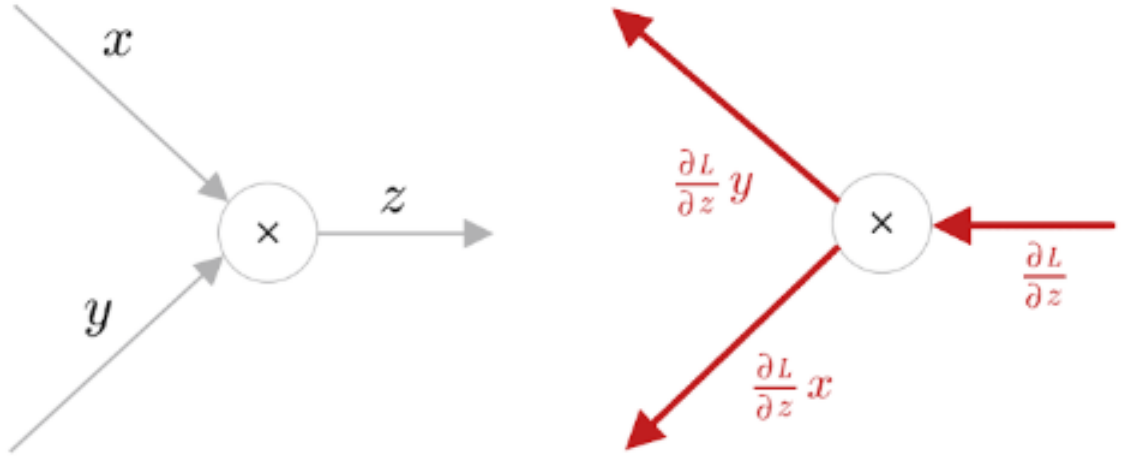
신경망의 학습

▪ 계산 그래프

• 곱셈 노드

곱셈 노드의 역전파는 [그림 1-19]처럼 '상류로부터 받은 기울기'에 '순전파 시의 입력을 서로 바꾼 값'을 곱한다.

그림 1-19 곱셈 노드의 순전파(왼쪽)와 역전파(오른쪽)



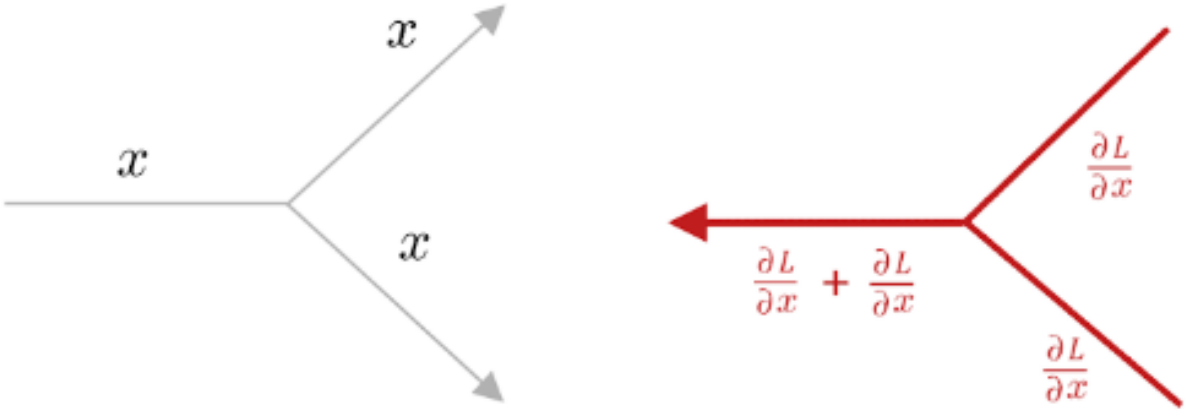
신경망의 학습

▪ 계산 그래프

• 분기 노드

분기노드는 따로 그리지 않고 단순히 선이 두 개로 나뉘도록 그리는데 ,이때 같은 값이 복제되어 분기한다.
하여 분기노드를 복제 노드라고도 한다.

그림 1-20 분기 노드의 순전파(왼쪽)와 역전파(오른쪽)



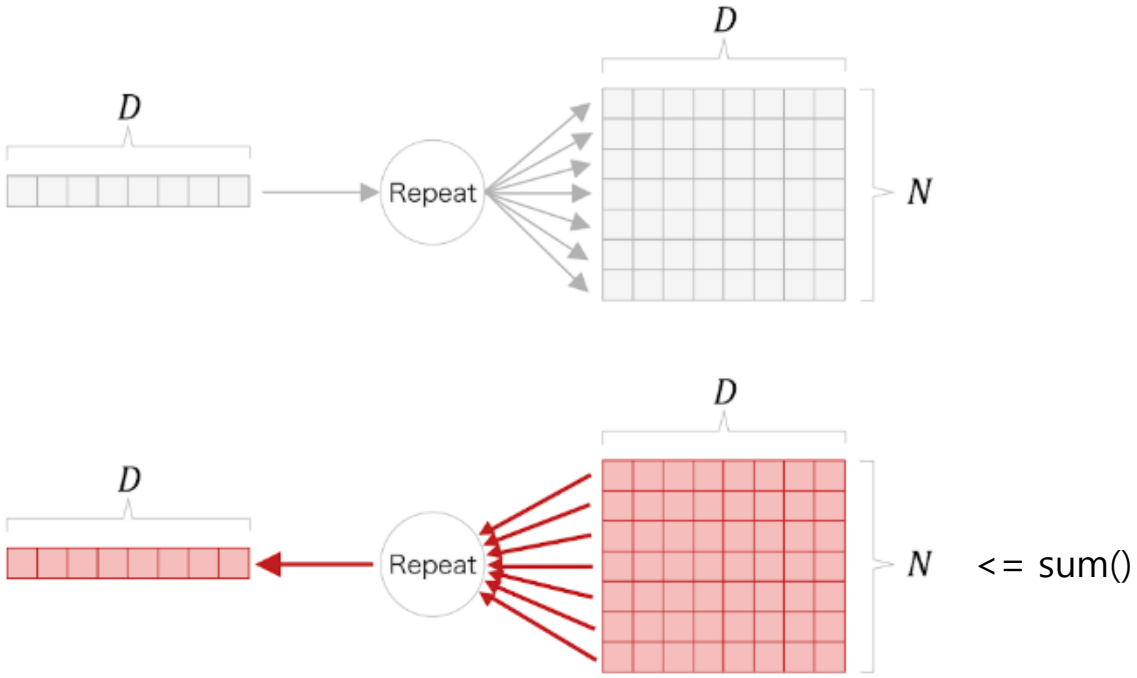
신경망의 학습

▪ 계산 그래프

• Repeat 노드

2개로 분기하는 분기 노드를 일반화하면 N개로의 분기가 된다. 이를 Repeat 노드라고 한다.

그림 1-21 Repeat 노드의 순전파(위)와 역전파(아래)



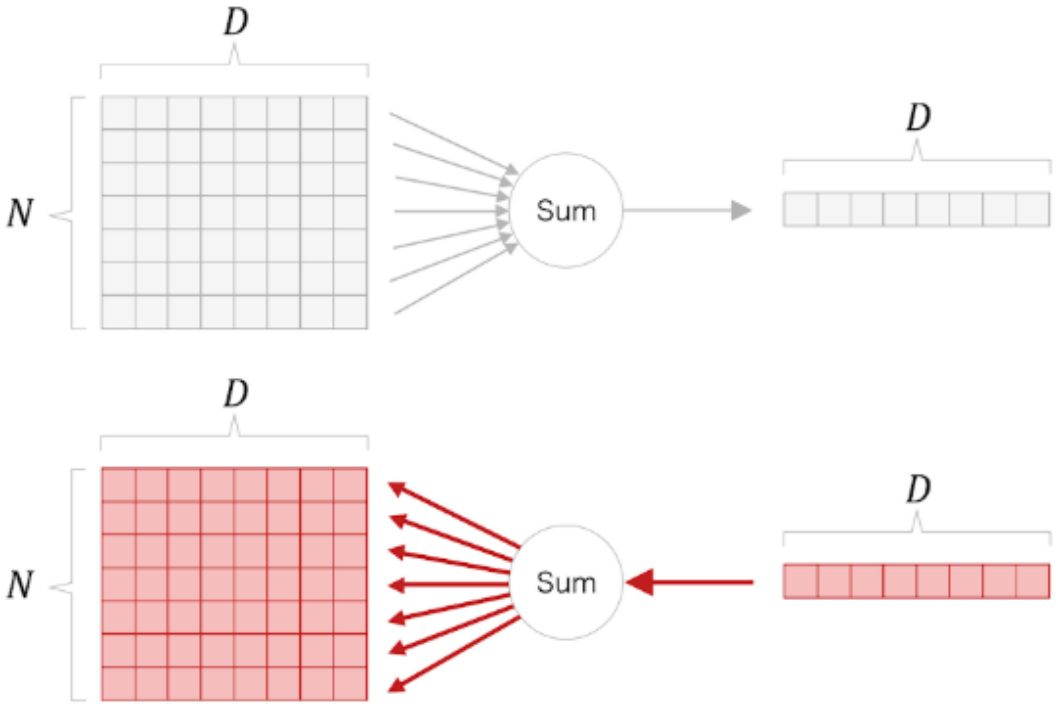
신경망의 학습

▪ 계산 그래프

• Sum 노드

범용 덧셈 노드이다. Repeat 노드의 반대라고도 볼 수 있다.

그림 1-22 Sum 노드의 순전파(위)와 역전파(아래)



신경망의 학습

▪ 계산 그래프

• Matmul 노드

행렬의 곱셈을 Matmul 노드('Matrix Multiply'의 약자)로 표현한다.

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

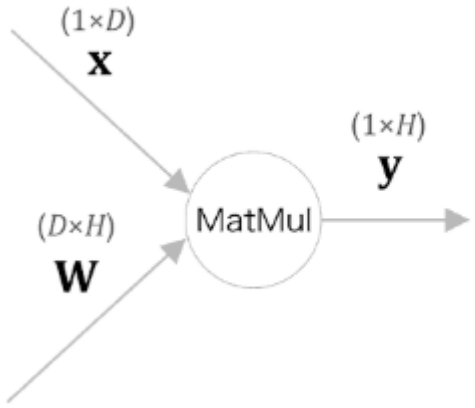
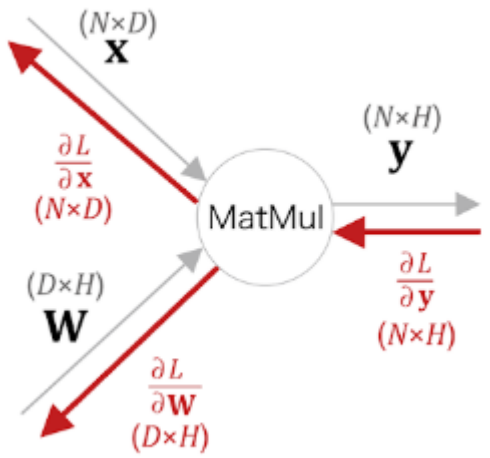


그림 1-25 MatMul 노드의 역전파



신경망의 학습

▪ 계산 그래프

• Matmul 노드 구현

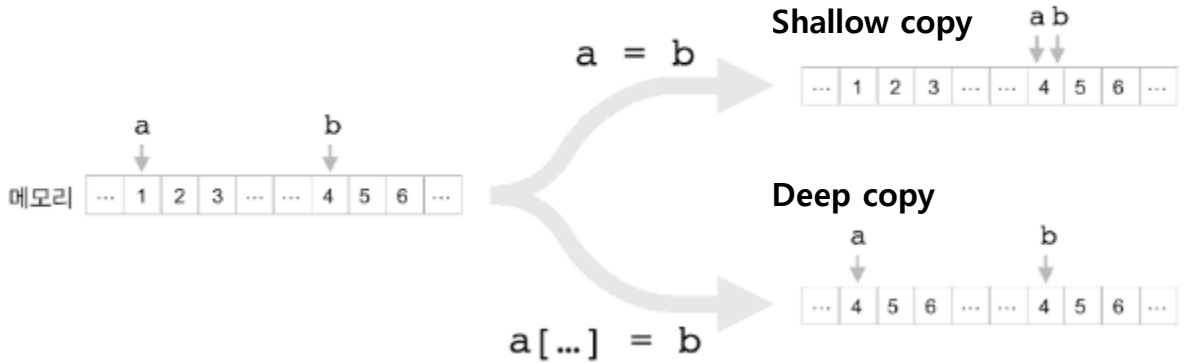
```
class Matmul:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.x = None

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

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

grads[0] = dW => Shallow Copy
Grads[0][…] = dW => Deep Copy

grads[0][...] = dW 코드에서 점 3개로 이루어진 생략 기호는 넘파이 배열이 가르키는 메모리 위치를 고정시킨 다음, 그 위치에 원소들을 덮어쓴다.

WARNING_ '생략 기호'는 넘파이 배열의 '덮어쓰기'를 수행한다.
결국 **shallow copy**냐 **deep copy**냐의 차이다.

그림 1-27 a = b와 a[...] = b의 차이: 생략 기호는 데이터를 덮어쓰기 때문에 변수가 가리키는 메모리 위치는 변하지 않는다.



신경망의 학습

▪ 기울기 도출과 역전파 구현

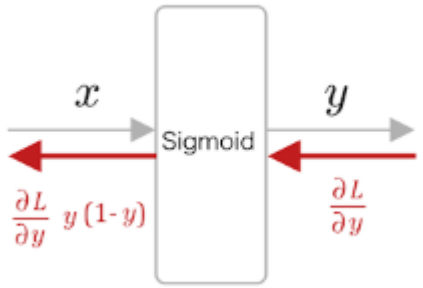
• Sigmoid 계층

시그모이드 함수를 수식으로 쓰면 $y = \frac{1}{1+e^{-x}}$ 이다. 그리고 그 미분은 다음과 같다.

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

위의 식으로부터 Sigmoid 계층의 계산 그래프를 다음처럼 그릴 수 있다.

그림 1-28 Sigmoid 계층의 계산 그래프




```
class Sigmoid:
    def __init__(self):
        self.params, self.grads = [], []
        self.out = out

    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
```

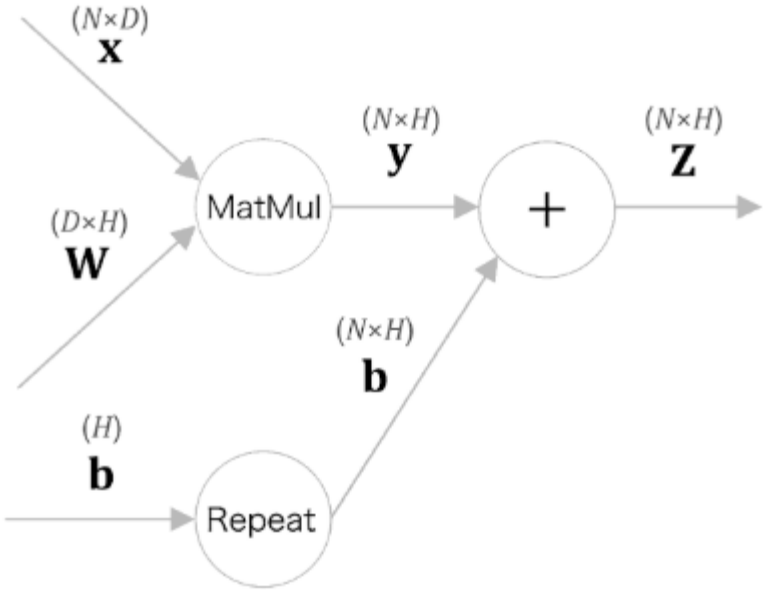
신경망의 학습

▪ 기울기 도출과 역전파 구현

• Affine 계층

Affine 계층의 순전파는 앞에서 본 것과 같이 y = np.matmul(x, W) + b와 같다.
여기서 bias를 더할 때는 브로드캐스트가 되는 점을 살펴봤었다. 그 점을 명시적으로 나타내어 계산그래프를 그리면 다음과 같다.

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.matmul(x,W) + b
        self.x = x
        return out

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

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

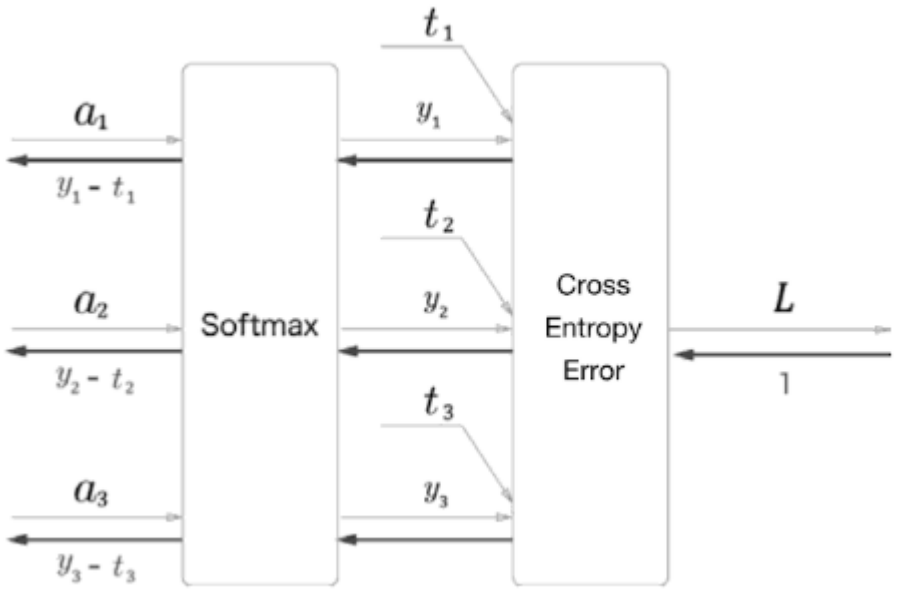
신경망의 학습

▪ 기울기 도출과 역전파 구현

• Softmax with Loss 계층

소프트맥스 함수와 교차 엔트로피 오차는 Softmax with Loss라는 하나의 계층으로 구현할 것이다.

그림 1-30 Softmax with Loss 계층의 계산 그래프



Softmax 계층은 입력 **a**를 정규화하여 **y**를 출력한다.
그리고 Cross Entropy Error 계층은 출력 **y**와 정답 레이블 **T**를 받고, 이 데이터로부터 손실 ***L***을 구해 출력한다.

신경망의 학습

▪ 가중치 갱신

오차역전파법으로 기울기를 구했으면, 그 기울기를 사용해 신경망의 매개변수를 갱신한다.
이때 신경망의 학습은 다음 순서로 수행한다.

- **1단계: 미니배치**

  훈련데이터 중에서 무작위로 다수의 데이터를 골라낸다.

- **2단계: 기울기 계산**

  오차역전파법으로 각 가중치 매개변수에 대한 손실 함수의 기울기를 구한다.

- **3단계: 매개변수 갱신**

  기울기를 사용하여 가중치 매개변수를 갱신한다.

- **4단계: 반복**

  1~3단계를 필요한 만큼 반복한다.

신경망의 학습

▪ 가중치 갱신

우선 미니배치에서 데이터를 선택하고, 이어서 오차역전파법으로 가중치의 기울기를 얻는다. 이 기울기는 현재의 가중치 매개변수에서 손실을 가장 크게 하는 방향을 가르킨다. 따라서 매개변수를 그 기울기와 반대 방향으로 갱신하면 손실을 줄일 수 있다. 이것이 바로 경사하강법$Gradient\ Descent$이다. 그런 다음, 이상의 작업을 필요한만큼 반복한다.

3단계에서 수행하는 가중치 갱신 기법의 종류 중 여기에서는 가장 단순한 **확률적경사하강법**(**SGD**)을 구현해서 사용한다. SGD는 현재의 가중치를 기울기 방향으로 일정한 거리만큼 갱신하는 방식이다.

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

여기서 α는 학습률$learning\ rate$을 나타내며, 실제로는 0.01, 0.001 같은 값을 미리 정해 사용한다.

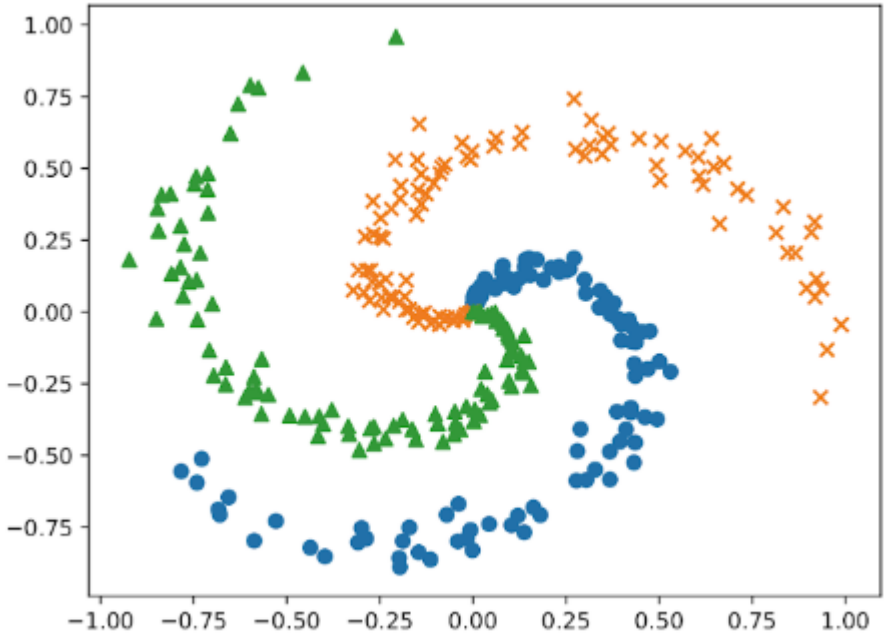
```
class SGD:
    '''
    확률적 경사하강법(Stochastic Gradient Descent)
    '''
    def __init__(self, lr=0.01):
        self.lr = lr

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

신경망으로 문제를 풀다

▪ 스파이럴 데이터셋

이 예에서는 spiral.py를 import하여 이용한다. spiral.load_data()가 데이터를 읽어온다. 이때 x가 입력데이터이고, t가 정답 레이블이다. X와 t의 shape를 출력해보면 각각 300개의 샘플 데이터를 담고있고, x는 2차원, t는 3차원 데이터이다.

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



신경망으로 문제를 풀다

- **신경망 구현**

```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
```

CS

신경망으로 문제를 풀다

- **학습용 코드**

```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)

# 학습에 사용하는 변수
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []
```

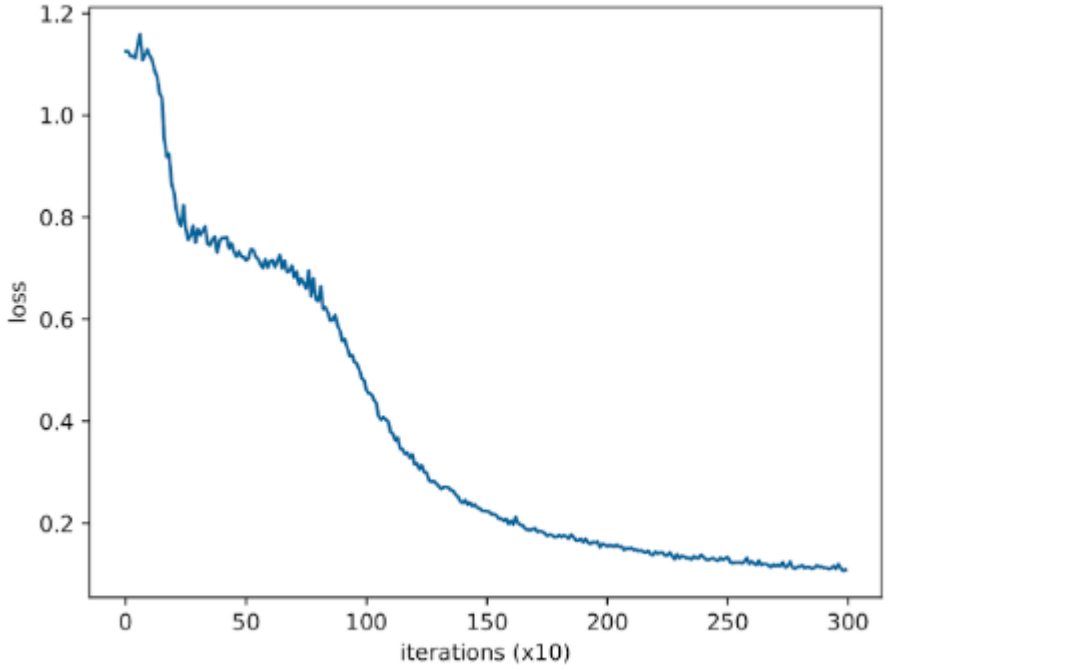
```python
for epoch in range(max_epoch):
    # 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    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
```

신경망으로 문제를 풀다

- **학습용 코드**

ch01/train_custom_loop.py를 실행하면 손실 값이 순조롭게 낮아지는 것을 알 수 있다.
그리고 그 결과를 그래프로 그린 것이 바로 [그림 1-32]이다.

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

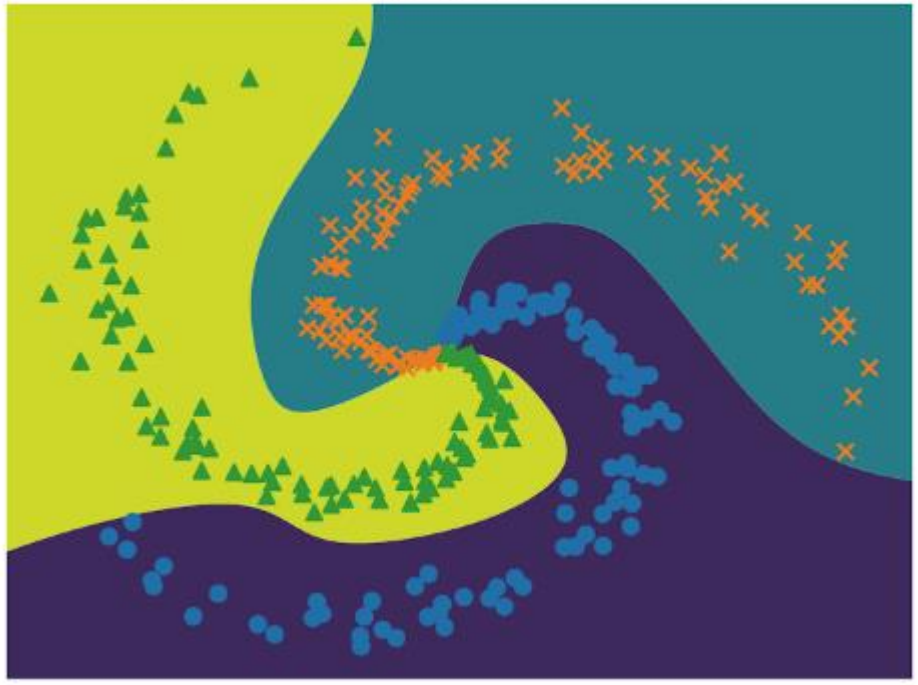


신경망으로 문제를 풀다

- **학습용 코드**

학습 후, 신경망이 영역을 어떻게 분리했는지를 시각화해보면 다음과 같다.
(이를 결정 경계$decision\ boundary$라고 한다)

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



계산 고속화

▪ 비트 정밀도

신경망의 학습과 추론데 드는 연산량은 상당하다. 그래서 신경망에서는 얼마나 빠르게 계산하느냐가 매우 중요한 주제이다.
그래서 신경망 고속화에 도움이 되는 방법 중 하나로 '비트 정밀도'가 있다.

보통 넘파이의 부동소수점 수는 기본적으로 64비트의 데이터 타입을 사용한다. (환경에 따라 다를 수 있음)
실제로 몇 비트의 부동소수점이 사용되는지는 다음 코드로 확인 가능하다.

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

넘파이는 64비트 부동소수점 수를 표준으로 사용한다. 그러나 신경망의 추론과 학습은 32비트 부동수소점으로도 문제없이 수행 가능하다. 32비트는 64비트의 절반이므로, 메모리 관점에서는 항상 32비트가 더 좋고, 또 신경망 계산 속도 측면에서도 32비트 부동소수점이 64비트 부동소수점보다 일반적으로 더 빠르다. (CPU나 GPU 아키텍처에 따라 다름)

이런 이유로 float를 32bit 형으로 바꿔서 사용 시 **계산 고속화**를 할 수 있다.

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

계산 고속화

- **GPU(쿠파이): CUDA**

딥러닝의 계산은 대량의 곱하기 연산으로 구성된다. 이 대량의 곱하기 연산 대부분은 병렬로 계산 가능한데, 바로 이 점에서는 CPU보다는 **GPU가 유리**하다.

Cupy(쿠파이)라는 파이썬 라이브러리가있다. 쿠파이는 GPU를 이용해 병렬 계산을 수행해주는 라이브러리인데, 엔비디아의 GPU에서만 동작하고, CUDA라는 GPU 전용 범용 병렬 컴퓨팅 플랫폼을 설치해야 한다.

이 쿠파이를 사용하면 간단하게 GPU를 사용해 병렬 계산이 가능한데, 더욱 중요한 점은 쿠파이는 넘파이와 호환되는 API를 제공한다는 사실이다. 여기 간단한 사용 예이다. numpy를 cupy로 대체해주기만 하면 된다!

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2,3).astype('f')
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]], dtype=float32)
>>> x.sum(axis=1)
array([ 3., 12.], dtype=float32)
```

WAINING_ 쿠파이가 넘파이의 모든 메서드를 지원하는 것은 아니다. 100% 호환되지는 않지만 많은 공통된 API를 제공한다.

정리

이번 장에서 배운 내용

- 신경망은 입력층, 은닉층(중간층), 출력층을 지닌다.
- 완전연결계층에 의해 선형 변환이 이뤄지고, 활성화 함수에 의해 비선형 변환이 이뤄진다.
- 완전연결계층이나 미니배치 처리는 행렬로 모아 한꺼번에 계산할 수 있다.
- 오차역전파법을 사용하여 신경망의 손실에 관한 기울기를 효율적으로 구할 수 있다.
- 신경망이 수행하는 처리는 계산 그래프로 시각화할 수 있으며, 순전파와 역전파를 이해하는데 도움이 된다.
- 신경망의 구성요소들을 '계층'으로 모듈화해두면, 이를 조립하여 신경망을 쉽게 구성할 수 있다.
- 신경망 고속화에는 GPU를 이용한 병렬 계산과 데이터의 비트 정밀도가 중요하다.